Contents

1 Library NormInType **2**

1.1 NormInType: Alternate Version of Normalization, for Extraction 2
1.2 Note 2
1.3 Normalization 2
1.4 Language 3
1.5 Normalization 10
1.5.1 Membership in R_T is invariant under evaluation 12
1.5.2 Closed instances of terms of type T belong to R_T 14
1.6 MoreStlc: A Typechecker for STLC 25
1.6.1 Comparing Types 26
1.6.2 The Typechecker 26
1.6.3 Properties 27

2 Library Extraction2 **29**

2.1 Extraction2: Extracting ML from Coq, Part 2 29
2.2 Extracting a Normalizer 29

3 Library Postscript **31**

3.1 Postscript 31
3.2 Looking back... 31
3.3 Looking Forward 32

4 Library PE **34**

4.1 PE: Partial Evaluation 34
4.2 Generalizing Constant Folding 34
4.2.1 Partial States 35
4.2.2 Arithmetic Expressions 36
4.2.3 Boolean Expressions 39
4.3 Partial Evaluation of Commands, Without Loops 40
4.3.1 Assignment 41
4.3.2 Conditional 42
4.3.3 The Partial Evaluation Relation 46
4.3.4 Examples 48

4.3.5 Correctness of Partial Evaluation . . . 48
4.4 Partial Evaluation of Loops . . . 51
4.4.1 Examples . . . 53
4.4.2 Correctness . . . 54
4.5 Partial Evaluation of Flowchart Programs . . . 60
4.5.1 Basic blocks . . . 61
4.5.2 Flowchart programs . . . 62
4.5.3 Partial evaluation of basic blocks and flowchart programs . . . 63

5 Library UseAuto **66**
5.1 UseAuto: Theory and Practice of Automation in Coq Proofs . . . 66
5.2 Basic Features of Proof Search . . . 66
5.2.1 Strength of Proof Search . . . 67
5.2.2 Basics . . . 67
5.2.3 Conjunctions . . . 68
5.2.4 Disjunctions . . . 69
5.2.5 Existentials . . . 70
5.2.6 Negation . . . 70
5.2.7 Equalities . . . 71
5.3 How Proof Search Works . . . 71
5.3.1 Search Depth . . . 71
5.3.2 Backtracking . . . 73
5.3.3 Adding Hints . . . 75
5.3.4 Integration of Automation in Tactics . . . 76
5.4 Examples of Use of Automation . . . 76
5.4.1 Determinism . . . 76
5.4.2 Preservation for STLC . . . 79
5.4.3 Progress for STLC . . . 80
5.4.4 BigStep and SmallStep . . . 81
5.4.5 Preservation for STLCRef . . . 83
5.4.6 Progress for STLCRef . . . 86
5.4.7 Subtyping . . . 86
5.5 Advanced Topics in Proof Search . . . 87
5.5.1 Stating Lemmas in the Right Way . . . 87
5.5.2 Unfolding of Definitions During Proof-Search . . . 88
5.5.3 Automation for Proving Absurd Goals . . . 89
5.5.4 Automation for Transitivity Lemmas . . . 91
5.6 Decision Procedures . . . 94
5.6.1 Omega . . . 94
5.6.2 Ring . . . 95
5.6.3 Congruence . . . 95
5.7 Summary . . . 96

6 Library UseTactics **98**
6.1 UseTactics: Tactic Library for Coq: A Gentle Introduction . . . 98
6.2 Tactics for introduction and case analysis . . . 99
6.2.1 The tactic *introv* . . . 99
6.2.2 The tactic *inverts* . . . 100
6.2.3 The tactics *cases* and *cases_if* . . . 102
6.3 Tactics for n-ary connectives . . . 104
6.3.1 The tactic *splits* . . . 104
6.3.2 The tactic *branch* . . . 104
6.3.3 The tactic ∃ . . . 105
6.4 Tactics for working with equality . . . 105
6.4.1 The tactics *asserts_rewrite* and *cuts_rewrite* . . . 106
6.4.2 The tactic *substs* . . . 107
6.4.3 The tactic *fequals* . . . 107
6.4.4 The tactic *applys_eq* . . . 107
6.5 Some convenient shorthands . . . 108
6.5.1 The tactic *unfolds* . . . 109
6.5.2 The tactics *false* and *tryfalse* . . . 109
6.5.3 The tactic *gen* . . . 110
6.5.4 The tactics *skip*, *skip_rewrite* and *skip_goal* . . . 110
6.5.5 The tactic *sort* . . . 112
6.6 Tactics for advanced lemma instantiation . . . 112
6.6.1 Working of *lets* . . . 113
6.6.2 Working of *applys*, *forwards* and *specializes* . . . 115
6.6.3 Example of instantiations . . . 115
6.7 Summary . . . 117

7 Library LibTactics **118**
7.1 LibTactics: A Collection of Handy General-Purpose Tactics . . . 118
7.2 Additional notations for Coq . . . 119
7.2.1 N-ary Existentials . . . 119
7.3 Tools for programming with Ltac . . . 120
7.3.1 Identity continuation . . . 120
7.3.2 Untyped arguments for tactics . . . 120
7.3.3 Optional arguments for tactics . . . 121
7.3.4 Wildcard arguments for tactics . . . 121
7.3.5 Position markers . . . 121
7.3.6 List of arguments for tactics . . . 122
7.3.7 Databases of lemmas . . . 124
7.3.8 On-the-fly removal of hypotheses . . . 125
7.3.9 Numbers as arguments . . . 126
7.3.10 Testing tactics . . . 127
7.3.11 Check no evar in goal . . . 127

7.3.12 Tagging of hypotheses 128
7.3.13 Tagging of hypotheses 128
7.3.14 Deconstructing terms 128
7.3.15 Action at occurence and action not at occurence 129
7.3.16 An alias for *eq* 130
7.4 Backward and forward chaining 130
7.4.1 Application 130
7.4.2 Assertions 132
7.4.3 Instantiation and forward-chaining 134
7.4.4 Experimental tactics for application 142
7.4.5 Adding assumptions 143
7.4.6 Application of tautologies 143
7.4.7 Application modulo equalities 144
7.5 Introduction and generalization 146
7.5.1 Introduction 146
7.5.2 Generalization 148
7.5.3 Naming 149
7.6 Rewriting 152
7.6.1 Rewriting 152
7.6.2 Replace 153
7.6.3 Renaming 154
7.6.4 Unfolding 154
7.6.5 Simplification 156
7.6.6 Evaluation 157
7.6.7 Substitution 157
7.6.8 Tactics to work with proof irrelevance 158
7.6.9 Proving equalities 158
7.7 Inversion 159
7.7.1 Basic inversion 159
7.7.2 Inversion with substitution 160
7.7.3 Injection with substitution 163
7.7.4 Inversion and injection with substitution –rough implementation . . . 164
7.7.5 Case analysis 164
7.8 Induction 168
7.9 Decidable equality 169
7.10 Equivalence 169
7.11 N-ary Conjunctions and Disjunctions 170
7.12 Tactics to prove typeclass instances 175
7.13 Tactics to invoke automation 175
7.13.1 *jauto*, a new automation tactics 175
7.13.2 Definitions of automation tactics 176
7.13.3 Definitions for parsing compatibility 177

7.13.4 Parsing for light automation . . . 177
7.13.5 Parsing for strong automation . . . 185
7.14 Tactics to sort out the proof context . . . 193
7.14.1 Hiding hypotheses . . . 193
7.14.2 Sorting hypotheses . . . 195
7.14.3 Clearing hypotheses . . . 195
7.15 Tactics for development purposes . . . 197
7.15.1 Skipping subgoals . . . 197
7.16 Compatibility with standard library . . . 199

8 Library Norm **200**
8.1 Norm: Normalization of STLC . . . 200
8.2 Language . . . 201
8.3 Normalization . . . 208
8.3.1 Membership in R_T is invariant under evaluation . . . 210
8.3.2 Closed instances of terms of type T belong to R_T . . . 212

9 Library RecordSub **221**
9.1 RecordSub: Subtyping with Records . . . 221
9.2 Core Definitions . . . 221
9.3 Subtyping . . . 224
9.3.1 Definition . . . 224
9.3.2 Subtyping Examples and Exercises . . . 225
9.3.3 Properties of Subtyping . . . 227
9.4 Typing . . . 229
9.4.1 Typing Examples . . . 230
9.4.2 Properties of Typing . . . 231
9.4.3 Exercises on Typing . . . 240

10 Library Sub **242**
10.1 Sub: Subtyping . . . 242
10.2 Concepts . . . 242
10.2.1 A Motivating Example . . . 242
10.2.2 Subtyping and Object-Oriented Languages . . . 243
10.2.3 The Subsumption Rule . . . 244
10.2.4 The Subtype Relation . . . 244
10.2.5 Exercises . . . 248
10.3 Formal Definitions . . . 251
10.3.1 Core Definitions . . . 251
10.3.2 Subtyping . . . 254
10.3.3 Typing . . . 256
10.3.4 Typing examples . . . 257
10.4 Properties . . . 257

10.4.1 Inversion Lemmas for Subtyping . . . 258
10.4.2 Canonical Forms . . . 258
10.4.3 Progress . . . 259
10.4.4 Inversion Lemmas for Typing . . . 261
10.4.5 Context Invariance . . . 264
10.4.6 Substitution . . . 265
10.4.7 Preservation . . . 267
10.4.8 Records, via Products and Top . . . 268
10.4.9 Exercises . . . 269
10.5 Exercise: Adding Products . . . 270

11 Library References 271
11.1 References: Typing Mutable References . . . 271
11.1.1 Definitions . . . 271
11.1.2 Syntax . . . 272
11.2 Pragmatics . . . 276
11.2.1 Side Effects and Sequencing . . . 276
11.2.2 References and Aliasing . . . 276
11.2.3 Shared State . . . 277
11.2.4 Objects . . . 277
11.2.5 References to Compound Types . . . 278
11.2.6 Null References . . . 279
11.2.7 Garbage Collection . . . 279
11.3 Operational Semantics . . . 280
11.3.1 Locations . . . 280
11.3.2 Stores . . . 280
11.3.3 Reduction . . . 283
11.4 Typing . . . 287
11.4.1 Store typings . . . 287
11.4.2 The Typing Relation . . . 289
11.5 Properties . . . 291
11.5.1 Well-Typed Stores . . . 291
11.5.2 Extending Store Typings . . . 292
11.5.3 Preservation, Finally . . . 293
11.5.4 Substitution lemma . . . 294
11.5.5 Assignment Preserves Store Typing . . . 297
11.5.6 Weakening for Stores . . . 298
11.5.7 Preservation! . . . 299
11.5.8 Progress . . . 302
11.6 References and Nontermination . . . 304
11.7 Additional Exercises . . . 306

12 Library Records **307**
12.1 Records: Adding Records to STLC . . . 307
12.2 Adding Records . . . 307
12.3 Formalizing Records . . . 308
12.3.1 Examples . . . 314
12.3.2 Properties of Typing . . . 315

13 Library MoreStlc **321**
13.1 MoreStlc: More on the Simply Typed Lambda-Calculus . . . 321
13.2 Simple Extensions to STLC . . . 321
13.2.1 Numbers . . . 321
13.2.2 `let`-bindings . . . 321
13.2.3 Pairs . . . 322
13.2.4 Unit . . . 324
13.2.5 Sums . . . 325
13.2.6 Lists . . . 327
13.2.7 General Recursion . . . 328
13.2.8 Records . . . 332
13.3 Exercise: Formalizing the Extensions . . . 335
13.3.1 Examples . . . 339
13.3.2 Properties of Typing . . . 343

14 Library Typechecking **347**
14.1 MoreStlc: A Typechecker for STLC . . . 347
14.1.1 Comparing Types . . . 347
14.1.2 The Typechecker . . . 348
14.1.3 Properties . . . 349

15 Library Stlc **351**
15.1 Stlc: The Simply Typed Lambda-Calculus . . . 351
15.2 The Simply Typed Lambda-Calculus . . . 351
15.2.1 Overview . . . 351
15.2.2 Syntax . . . 353
15.2.3 Operational Semantics . . . 355
15.2.4 Typing . . . 359
15.2.5 Properties . . . 364
15.2.6 Additional Exercises . . . 373
15.3 Optional Exercise: STLC with Arithmetic . . . 376
15.3.1 Syntax and Operational Semantics . . . 376

16 Library Types **378**
16.1 Types: Type Systems . . . 378
16.2 More Automation . . . 378

16.2.1 The `auto` and `eauto` Tactics . . . 378
16.2.2 The `Proof with` Tactic . . . 380
16.2.3 The `solve by inversion` Tactic . . . 380
16.2.4 The `try solve` Tactic . . . 380
16.2.5 The `f_equal` Tactic . . . 381
16.2.6 The *normalize* Tactic . . . 381
16.3 Typed Arithmetic Expressions . . . 383
16.3.1 Syntax . . . 383
16.3.2 Operational Semantics . . . 383
16.3.3 Normal Forms and Values . . . 385
16.3.4 Typing . . . 386
16.3.5 Progress . . . 388
16.3.6 Type Preservation . . . 390
16.3.7 Type Soundness . . . 391
16.3.8 Additional Exercises . . . 391

17 Library Smallstep 394
17.1 Smallstep: Small-step Operational Semantics . . . 394
17.2 Relations . . . 395
17.3 A Toy Language . . . 395
17.3.1 Values . . . 399
17.3.2 Strong Progress and Normal Forms . . . 401
17.4 Multi-Step Reduction . . . 408
17.4.1 Definitions . . . 408
17.4.2 Examples . . . 409
17.4.3 Normal Forms Again . . . 411
17.4.4 Equivalence of Big-Step and Small-Step Reduction . . . 413
17.4.5 Additional Exercises . . . 414
17.5 Small-Step Imp . . . 415
17.6 Concurrent Imp (Optional) . . . 419

18 Library Rel 424
18.1 Rel: Properties of Relations . . . 424
18.2 Basic Properties of Relations . . . 424
18.3 Reflexive, Transitive Closure . . . 428

19 Library HoareAsLogic 431
19.1 HoareAsLogic: Hoare Logic as a Logic . . . 431

20 Library HoareList 435
20.1 HoareList: Hoare Logic with Lists . . . 435
20.2 Imp Programs with Lists . . . 435
20.2.1 Repeated Definitions . . . 435

20.2.2 Extensions . . . 436
20.2.3 Repeated Definitions . . . 439
20.3 Hoare Rules . . . 442
20.3.1 Reasoning About Programs with Lists . . . 445
20.4 Formal Decorated Programs . . . 451

21 Library Hoare **458**
21.1 Hoare: Hoare Logic . . . 458
21.2 Hoare Logic . . . 459
21.2.1 Assertions . . . 459
21.2.2 Hoare Triples . . . 460
21.2.3 Weakest Preconditions . . . 462
21.2.4 Proof Rules . . . 464
21.2.5 Exercise: *HAVOC* . . . 482
21.3 Review of Hoare Logic . . . 484
21.4 Decorated Programs . . . 485
21.5 Sample Hoare Logic Proofs . . . 487
21.5.1 Example: Slow Subtraction . . . 487
21.5.2 Exercise: Reduce to Zero . . . 489
21.5.3 Exercise: Slow Addition . . . 489
21.5.4 Example: Parity . . . 490
21.5.5 Example: Finding Square Roots . . . 493
21.5.6 Exercise: Factorial . . . 495
21.6 Formalizing Decorated Programs (Optional) . . . 496
21.6.1 Syntax . . . 496
21.6.2 Extracting Verification Conditions . . . 499
21.6.3 Examples . . . 501

22 Library Equiv **505**
22.1 Equiv: Program Equivalence . . . 505
22.2 Behavioral Equivalence . . . 505
22.2.1 Definitions . . . 506
22.2.2 Examples . . . 507
22.2.3 The Functional Equivalence Axiom . . . 512
22.3 Properties of Behavioral Equivalence . . . 514
22.3.1 Behavioral Equivalence is an Equivalence . . . 514
22.3.2 Behavioral Equivalence is a Congruence . . . 515
22.4 Case Study: Constant Folding . . . 518
22.4.1 Soundness of Program Transformations . . . 518
22.4.2 The Constant-Folding Transformation . . . 519
22.4.3 Soundness of Constant Folding . . . 522
22.5 Proving That Programs Are *Not* Equivalent . . . 525
22.6 Extended exercise: Non-deterministic Imp . . . 528

22.7 Doing Without Extensionality (Optional) 532
22.8 Additional Exercises 535

23 Library Extraction **537**
23.1 Extraction: Extracting ML from Coq 537
23.2 Basic Extraction 537
23.3 Controlling Extraction of Specific Types 537
23.4 A Complete Example 538
23.5 Discussion 539

24 Library ImpCEvalFun **540**
24.1 ImpCEvalFun: Evaluation Function for Imp 540
24.1.1 Evaluation Function 540
24.1.2 Equivalence of Relational and Step-Indexed Evaluation 544
24.1.3 Determinism of Evaluation (Simpler Proof) 547

25 Library ImpParser **548**
25.1 ImpParser: Lexing and Parsing in Coq 548
25.2 Internals 548
25.2.1 Lexical Analysis 548
25.2.2 Parsing 550
25.3 Examples 555

26 Library Imp **556**
26.1 Imp: Simple Imperative Programs 556
26.2 Arithmetic and Boolean Expressions 556
26.2.1 Syntax 557
26.2.2 Evaluation 558
26.2.3 Optimization 558
26.3 Coq Automation 560
26.3.1 Tacticals 560
26.3.2 Defining New Tactic Notations 563
26.3.3 The `omega` Tactic 565
26.3.4 A Few More Handy Tactics 565
26.4 Evaluation as a Relation 566
26.4.1 Inference Rule Notation 568
26.5 Expressions With Variables 569
26.5.1 Identifiers 569
26.5.2 States 571
26.5.3 Syntax 572
26.5.4 Evaluation 573
26.6 Commands 574
26.6.1 Syntax 574

26.6.2 Examples . . . 575
26.7 Evaluation . . . 576
26.7.1 Evaluation Function (Failed Attempt) . . . 576
26.7.2 Evaluation as a Relation . . . 577
26.7.3 Determinism of Evaluation . . . 580
26.8 Reasoning About Programs . . . 581
26.8.1 Basic Examples . . . 581
26.8.2 Proving a Program Correct (Optional) . . . 582
26.9 Additional Exercises . . . 584

27 Library SfLib **588**
27.1 SfLib: Software Foundations Library . . . 588
27.2 From the Coq Standard Library . . . 588
27.3 From Basics.v . . . 588
27.4 From Props.v . . . 590
27.5 From Logic.v . . . 590
27.6 From Later Files . . . 591
27.7 Some useful tactics . . . 593

28 Library Prop **594**
28.1 Prop: Propositions and Evidence . . . 594
28.2 Inductively Defined Propositions . . . 594
28.3 Proof Objects . . . 596
28.3.1 Proof Scripts and Proof Objects . . . 597
28.3.2 Implications and Functions . . . 599
28.3.3 Induction Over Proof Objects . . . 600
28.3.4 Evenness . . . 602
28.3.5 Inverting Evidence . . . 603
28.4 Programming with Propositions . . . 606
28.5 Induction Principles . . . 608
28.5.1 Induction Principles for Inductively Defined Types . . . 608
28.5.2 Induction Hypotheses . . . 611
28.6 Optional Material . . . 612
28.6.1 Induction Principles in `Prop` . . . 612
28.6.2 More on the `induction` Tactic . . . 614
28.6.3 Building Proof Objects Incrementally . . . 616
28.7 Additional Exercises . . . 617

29 Library Logic **619**
29.1 Logic: Logic in Coq . . . 619
29.2 Quantification and Implication . . . 619
29.3 Conjunction . . . 620
29.3.1 Iff . . . 622

29.4 Disjunction 623
29.4.1 Relating $\wedge$ and $\vee$ with *andb* and *orb* 625
29.5 Falsehood 626
29.5.1 Truth 627
29.6 Negation 627
29.6.1 Inequality 629
29.7 Existential Quantification 630
29.8 Equality 632
29.8.1 Inversion, Again 634
29.9 Relations as Propositions 635
29.9.1 Digression: More Facts about $\leq$ and $<$ 639
29.10 Informal Proofs 641
29.10.1 Informal Proofs by Induction 642
29.11 Optional Material 644
29.11.1 Induction Principles for $\wedge$ and $\vee$ 644
29.11.2 Explicit Proof Objects for Induction 645
29.11.3 The Coq Trusted Computing Base 646

30 Library Gen **648**
30.1 Gen: Generalizing Induction Hypotheses 648

31 Library Poly **654**
31.1 Poly: Polymorphism and Higher-Order Functions 654
31.2 Polymorphism 654
31.2.1 Polymorphic Lists 654
31.2.2 Polymorphic Pairs 660
31.2.3 Polymorphic Options 661
31.3 Functions as Data 662
31.3.1 Higher-Order Functions 662
31.3.2 Partial Application 662
31.3.3 Digression: Currying 663
31.3.4 Filter 664
31.3.5 Anonymous Functions 665
31.3.6 Map 666
31.3.7 Fold 667
31.3.8 Functions For Constructing Functions 668
31.4 Optional Material 669
31.4.1 Non-Uniform Inductive Families (GADTs) 669
31.5 More About Coq 670
31.5.1 The `apply` Tactic 670
31.5.2 The `unfold` Tactic 672
31.5.3 Inversion 673
31.5.4 Varying the Induction Hypothesis 676

31.5.5 Using Tactics on Hypotheses 678
31.5.6 Using `destruct` on Compound Expressions 679
31.5.7 The *remember* Tactic 680
31.5.8 The `apply` ... `with` ... Tactic 682
31.6 Review 683
31.7 Additional Exercises 684

32 Library Lists **687**
32.1 Lists: Working with Structured Data 687
32.2 Pairs of Numbers 687
32.3 Lists of Numbers 689
32.3.1 Bags via Lists 693
32.4 Reasoning About Lists 695
32.4.1 Micro-Sermon 695
32.4.2 Induction on Lists 695
32.4.3 `SearchAbout` 699
32.4.4 List Exercises, Part 1 699
32.4.5 List Exercises, Part 2 700
32.5 Options 701
32.6 Extended Exercise: Dictionaries 703

33 Library Basics **705**
33.1 Basics: Functional Programming 705
33.2 Enumerated Types 705
33.2.1 Days of the Week 705
33.2.2 Booleans 706
33.2.3 Function Types 708
33.2.4 Numbers 709
33.3 Proof By Simplification 714
33.4 The `intros` Tactic 715
33.5 Proof by Rewriting 716
33.6 Case Analysis 717
33.7 Naming Cases 718
33.8 Induction 720
33.9 Formal vs. Informal Proof 722
33.10 Proofs Within Proofs 724
33.11 More Exercises 726

34 Library Preface **731**
34.1 Preface 731
34.2 Overview 731
34.3 Practicalities 732
34.3.1 Chapter Dependencies 732

34.3.2 Required Background 732
34.4 Coq 732
34.4.1 System Requirements 733
34.4.2 Access to the Coq files 733
34.5 Exercises 733
34.6 Recommended Reading 734
34.7 Translations 734
34.8 For Instructors 734

35 Library Symbols **735**
35.1 Symbols: Special symbols 735

Chapter 1

Library NormInType

1.1 NormInType: Alternate Version of Normalization, for Extraction

1.2 Note

This is an alternate version of *Norm.v* that is used as an example in *Extraction2.v*. It is not intended to be read on its own.

1.3 Normalization

```
Require Import Stlc.
```

(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?

□

1.4 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

.

Tactic Notation "T_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "TBool" | Case_aux c "TArrow" | Case_aux c "TProd" ].
Inductive tm : Type :=

  | tvar : id → tm
  | tapp : tm → tm → tm
  | tabs : id → ty → tm → tm


  | ttrue : tm
  | tfalse : tm
  | tif : tm → tm → tm → tm.
Tactic Notation "t_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp" | Case_aux c "tabs"

  | Case_aux c "ttrue" | Case_aux c "tfalse" | Case_aux c "tif" ].
```

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)

  | 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 → Type :=
  | v_abs : ∀ x T11 t12,
      value (tabs x T11 t12)

  | v_true : value ttrue
  | v_false : value tfalse
.
Hint Constructors value.
Reserved Notation "t1 '==>' t2" (at level 40).
Inductive step : tm → tm → Type :=
  | 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_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).
Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_AppAbs" | Case_aux c "ST_App1" | Case_aux c "ST_App2"

  | Case_aux c "ST_IfTrue" | Case_aux c "ST_IfFalse" | Case_aux c "ST_If" ].
Definition relation (X:Type) := X → X → Type.
Inductive multi (X:Type) (R: relation X)
                              : X → X → Type :=
  | multi_refl : ∀ (x : X),
                  multi X R x x
  | multi_step : ∀ (x y z : X),
                    R x y →
                    multi X R y z →
                    multi X R x z.
Implicit Arguments multi [[X]].
Tactic Notation "multi_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "multi_refl" | Case_aux c "multi_step" ].
Notation multistep := (multi step).
Notation "t1 '==>*' t2" := (multistep t1 t2) (at level 40).
Hint Constructors step.
Inductive ex (X:Type) (P : X→Type) : Type :=
  ex_intro : ∀ (witness:X), P witness → ex X P.
Notation "'exists' x , p" := (ex _ (fun x ⇒ p))
  (at level 200, x ident, right associativity) : type_scope.
Notation "'exists' x : X , p" := (ex _ (fun x:X ⇒ p))
  (at level 200, x ident, right associativity) : type_scope.
Definition normal_form {X:Type} (R:relation X) (t:X) : Type :=
  (∃ t', R t t') → False.
Notation step_normal_form := (normal_form step).
Definition deterministic {X: Type} (R: relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.
Hint Constructors ex.
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 → Type :=

  | T_Var : ∀ Gamma x T,
      Gamma x = Some T →
      has_type Gamma (tvar x) T
  | T_Abs : ∀ Gamma x T11 T12 t12,
      has_type (extend 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_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.

Tactic Notation "has_type_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "T_Var" | Case_aux c "T_Abs" | Case_aux c "T_App"

  | Case_aux c "T_True" | Case_aux c "T_False" | Case_aux c "T_If" ].

Hint Extern 2 (has_type _ (tapp _ _) _) ⇒ eapply T_App; auto.
Hint Extern 2 (_ = _) ⇒ compute; reflexivity.
```

Context Invariance

```
Inductive appears_free_in : id → tm → Type :=
  | 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 t2 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_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) :=
  ∀ x, appears_free_in x t → False.

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 Gamma Gamma' t S H H0. generalize dependent Gamma'.
  has_type_cases (induction H) Case;
    intros Gamma' Heqv...
  Case "T_Var".
    apply T_Var... rewrite ← Heqv...
  Case "T_Abs".
    apply T_Abs... apply IHhas_type. intros y Hafi.
    unfold extend. remember (beq_id x y) as e.
    destruct e...
```

```
  Case "T_If".
    eapply T_If...
Qed.

Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   has_type Gamma t T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  has_type_cases (induction Htyp) Case; inversion Hafi; subst...
  Case "T_Abs".
    destruct IHHtyp as [T' Hctx]... ∃ T'.
    unfold extend in Hctx.
    apply not_eq_beq_id_false in H2. rewrite H2 in Hctx...
Qed.

Corollary typable_empty__closed : ∀ t T,
    has_type empty t T →
    closed t.
Proof.
  intros t T H. unfold closed. intros x H1.
  destruct (free_in_context _ _ _ _ H1 H) as [T' C].
  inversion C. Qed.
```

Preservation

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (extend Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  t_cases (induction t) Case;
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  Case "tvar".
    simpl. rename i into y.
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      apply beq_id_eq in Heqe. subst.
      unfold extend in H1. rewrite ← beq_id_refl in H1.
      inversion H1; subst. clear H1.
      eapply context_invariance...
```

```
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra) as [T' HT']...
      inversion HT'.
    SCase "x<>y".
      apply T_Var... unfold extend in H1. rewrite <- Heqe in H1...
  Case "tabs".
    rename i into y. rename t into T11.
    apply T_Abs...
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      eapply context_invariance...
      apply beq_id_eq in Heqe. subst.
      intros x Hafi. unfold extend.
      destruct (beq_id y x)...
    SCase "x<>y".
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold extend.
      remember (beq_id y z) as e0. destruct e0...
      apply beq_id_eq in Heqe0. subst.
      rewrite <- Heqe...
Qed.
```

```
Theorem preservation : forall t t' T,
     has_type empty t T ->
     t ==> t' ->
     has_type empty t' T.
Proof with eauto.
  intros t t' T HT.
  remember (@empty ty) as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  has_type_cases (induction HT) Case;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  Case "T_App".
    inversion HE; subst...
    SCase "ST_AppAbs".
      apply substitution_preserves_typing with T1...
      inversion HT1...
Qed.
```

☐

Determinism

```
Lemma step_deterministic :
```

```
    deterministic step.
Proof with eauto.
    unfold deterministic.
    intros t t' t'' E1 E2.
    generalize dependent t''.
    step_cases (induction E1) Case; intros t'' E2; inversion E2; subst; clear E2...
    Case "ST_AppAbs"...
        inversion H2.
        apply ex_falso_quodlibet; apply value__normal in v...
    Case "ST_App1".
        inversion E1.
        f_equal...
        apply ex_falso_quodlibet; apply value__normal in H1...
    Case "ST_App2".
        apply ex_falso_quodlibet; apply value__normal in H2...
        apply ex_falso_quodlibet; apply value__normal in v...
        f_equal...
    Case "ST_IfTrue".
        inversion H3.
    Case "ST_IfFalse".
        inversion H3.
    Case "ST_If".
        inversion E1.
        inversion E1.
        f_equal...
Qed.
```

1.5 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) : Type := (∃ t' : tm, (t ==>* t') × value t') %type.
```

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:

- $R\ \mathit{bool}\ t$ iff t is a closed term of type *bool* and t halts in a value
- $R\ (T1 \to T2)\ t$ iff t is a closed term of type $T1 \to T2$ 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 predicates.*) 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 $A\to A$ *preserve* property P, all terms of type $(A\to A)$->$(A\to A)$ *preserve the property of preserving* property P, and so on. We do this by defining a family of predicates, indexed by types. For the base type A, the predicate is just P. For functional types, it says that the function should map values satisfying the predicate at the input type to values satisfying the predicate 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 $(\forall\ s,\ R\ T1\ s \to R\ TS\ (\mathit{tapp}\ t\ s))$, 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 predicates 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} : Type :=
  (has_type empty t T × (halts t ×
  (match T with
   | TBool ⇒ True
   | TArrow T1 T2 ⇒ (∀ s, R T1 s → R T2 (tapp t s))

   end))) % type.
```

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 : ∀ {T} {t}, R T t → halts t.
Proof.
  intros. destruct T; unfold R in X; inversion X; inversion X1; subst; assumption.
Qed.
```

```
Lemma R_typable_empty : ∀ {T} {t}, R T t → has_type empty t T.
Proof.
  intros. destruct T; unfold R in X; inversion X; inversion X1; 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.

1.5.1 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 determinstic. This lemma might still be true for non-deterministic languages, but the proof would be harder!

```
Definition iff (P Q : Type) := ((P → Q) × (Q → P)) % type.
```

```
Notation "P <-> Q" := (iff P Q)
                      (at level 95, no associativity) : type_scope.
```

```
Theorem ex_falso_quodlibet : ∀ (P:Type),
  False → P.
Proof.
  intros P contra.
  inversion contra. Qed.

Lemma step_preserves_halting : ∀ t t', (t ==> t') → (halts t ↔ halts t').
Proof.
 intros t t' ST. unfold halts.
 split.
 Case "->".
  intros [t'' [STM V]].
  inversion STM; subst.
   apply ex_falso_quodlibet. apply value__normal in V. unfold normal_form in V. apply
V. ∃ t'. auto.
   rewrite (step_deterministic _ _ _ ST H). ∃ t''. split; assumption.
 Case "<-".
  intros [t'0 [STM V]].
  ∃ t'0. split; eauto.
  eapply multi_step. eassumption. assumption.
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 finiteness 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 : ∀ T t t', (t ==> t') → R T t → 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.
Qed.
```

The generalization to multiple steps is trivial:

```
Lemma multistep_preserves_R : ∀ T t t',
```

```
  (t ==>* t') → R T t → R T t'.
Proof.
  intros T t t' STM; induction STM; intros.
  assumption.
  apply IHSTM. eapply step_preserves_R. apply r. 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' : ∀ T t t',
  has_type empty t T → (t ==> t') → R T t' → R T t.
Proof.
  induction T; intros t t' typable_empty_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. assumption.
  split. apply (step_preserves_halting _ _ E); eauto.
  auto.
  split. assumption.
  split. apply (step_preserves_halting _ _ E); eauto.
  intros.
  eapply IHT2. eapply T_App. apply typable_empty_t.
  eapply R_typable_empty. assumption.
  apply ST_App1. apply E.
  apply RRt'; auto.
Qed.

Lemma multistep_preserves_R' : ∀ T t t',
  has_type empty t T → (t ==>* t') → R T t' → R T t.
Proof.
  intros T t t' HT STM.
  induction STM; intros.
    assumption.
    eapply step_preserves_R'. assumption. apply r. apply IHSTM.
    eapply preservation; eauto. auto.
Qed.
```

1.5.2 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 *tabs x T1 t2* belongs to $R\_(T1 \to T2)$ 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 $x1{:}T1,..xn{:}Tn \vdash t : T$ and $v1,...,vn$ are values such that $R\ T1\ v1$, $R\ T2\ v2$, ..., $R\ Tn\ vn$, then $R\ T\ ([x1{:=}v1][x2{:=}v2]...[xn{:=}vn]t)$.

The proof will proceed by induction on the typing derivation $x1{:}T1,..xn{:}Tn \vdash t : T$; 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 $x1,...xn$ 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 $...,y{:}bool,...,y{:}nat,...$ and a corresponding term substitution written as $...[y{:=}(tbool\ true)]...[y{:=}(tnat\ 3)]...t$. Since environments are extended from left to right, the binding $y{:}nat$ hides the binding $y{:}bool$; since substitutions are performed right to left, we do the substitution $y{:=}(tnat\ 3)$ first, so that the substitution $y{:=}(tbool\ true)$ 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 × 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 mextend (Gamma : context) (xts : tass) :=
  match xts with
  | nil ⇒ Gamma
  | ((x,v)::xts') ⇒ extend (mextend Gamma xts') x v
  end.
```

We will need some simple operations that work uniformly on environments and type assigments

```
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 nxs} : list (id × X) :=
  match nxs with
    | nil ⇒ nil
    | ((n',x)::nxs') ⇒ if beq_id n' n then drop n nxs' else (n',x)::(drop n nxs')
  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 → Type :=
| 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 → False) →
     ∀ t', [x:=t']t = t.
Proof with eauto.
  t_cases (induction t) Case; intros; simpl.
  Case "tvar".
    remember (beq_id x i) as e. destruct e...
      destruct H. apply beq_id_eq in Heqe. subst...
  Case "tapp".
    rewrite IHt1... rewrite IHt2...
  Case "tabs".
    remember (beq_id x i) as e. destruct e...
```

```
      symmetry in Heqe; apply beq_id_false_not_eq in Heqe.
        rewrite IHt...
  Case "ttrue"...
  Case "tfalse"...
  Case "tif".
    rewrite IHt1... rewrite IHt2... rewrite IHt3...
Qed.

Lemma subst_closed: ∀ t,
      closed t →
      ∀ x t', [x:=t']t = t.
Proof.
  intros. apply vacuous_substitution. apply H. Qed.

Lemma subst_not_afi : ∀ t x v, closed v → (appears_free_in x ([x:=v]t) → False).
Proof with eauto.   unfold closed, not.
  t_cases (induction t) Case; intros x v P A; simpl in A.
    Case "tvar".
     remember (beq_id x i) as e; destruct e...
       inversion A; subst. rewrite ← beq_id_refl in Heqe; inversion Heqe.
    Case "tapp".
     inversion A; subst...
    Case "tabs".
     remember (beq_id x i) as e; destruct e...
       apply beq_id_eq in Heqe; subst. inversion A; subst...
       inversion A; subst...

    Case "ttrue".
     inversion A.
    Case "tfalse".
     inversion A.
    Case "tif".
     inversion A; subst...
Qed.

Lemma duplicate_subst : ∀ t' x t v,
  closed v → [x:=t]([x:=v]t') = [x:=v]t'.
Proof.
  intros. eapply vacuous_substitution. apply subst_not_afi. auto.
Qed.

Lemma swap_subst : ∀ t x x1 v v1, x ≠ x1 → closed v → closed v1 →
                   [x1:=v1]([x:=v]t) = [x:=v]([x1:=v1]t).
Proof with eauto.
 t_cases (induction t) Case; intros; simpl.
  Case "tvar".
```

```
    remember (beq_id x i) as e; destruct e; remember (beq_id x1 i) as e; destruct e.
      apply beq_id_eq in Heqe. apply beq_id_eq in Heqe0. subst.
        apply ex_falso_quodlibet...
      apply beq_id_eq in Heqe; subst. simpl.
        rewrite ← beq_id_refl. apply subst_closed...
      apply beq_id_eq in Heqe0; subst. simpl.
        rewrite ← beq_id_refl. rewrite subst_closed...
      simpl. rewrite ← Heqe. rewrite ← Heqe0...
  Case "tapp".
    rewrite IHt1... rewrite IHt2...
  Case "tabs".
    destruct (beq_id x i); destruct (beq_id x1 i)...
      rewrite IHt...
  Case "ttrue"...
  Case "tfalse"...
  Case "tif".
    f_equal...
Qed.
```

Properties of multi-substitutions

```
Lemma msubst_closed: ∀ t, closed t → ∀ ss, 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 charcterizing how *msubst* of closed terms distributes over `subst` and over each term form

```
Lemma subst_msubst: ∀ env x v t, closed v → closed_env env →
  msubst env ([x:=v]t) = [x:=v](msubst (drop x env) t).
Proof.
  induction env0; intros.
    auto.
    destruct a. simpl.
```

```
    inversion H0. fold closed_env in H2.
    remember (beq_id i x) as e; destruct e.
      apply beq_id_eq in Heqe; subst.
        rewrite duplicate_subst; auto.
      symmetry in Heqe. apply beq_id_false_not_eq in Heqe.
      simpl. rewrite swap_subst; eauto.
Qed.
```

```
Lemma msubst_var: ∀ ss x, closed_env ss →
   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: ∀ 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 : ∀ ss t1 t2, msubst ss (tapp t1 t2) = tapp (msubst ss t1) (msubst ss t2).
Proof.
 induction ss; intros.
   reflexivity.
   destruct a.
    simpl. rewrite ← IHss. auto.
Qed.
```

You'll need similar functions for the other term constructors.

```
Lemma msubst_true : ∀ ss, msubst ss ttrue = ttrue.
Proof.
  induction ss; intros.
    auto.
```

```
    destruct a.
       simpl. auto.
Qed.

Lemma msubst_false : ∀ ss, msubst ss tfalse = tfalse.
Proof.
  induction ss; intros.
    auto.
    destruct a.
       simpl. auto.
Qed.

Lemma msubst_if : ∀ ss t0 t1 t2,
  msubst ss (tif t0 t1 t2) = tif (msubst ss t0) (msubst ss t1) (msubst ss t2).
Proof.
   induction ss; intros.
     auto.
     destruct a. simpl. auto.
Qed.
```

Properties of multi-extensions

We need to connect the behavior of type assignments with that of their corresponding contexts.

```
Lemma mextend_lookup : ∀ (c : tass) (x:id), lookup x c = (mextend empty c) x.
Proof.
  induction c; intros.
    auto.
    destruct a. unfold lookup, mextend, extend. destruct (beq_id i x); auto.
Qed.

Lemma mextend_drop : ∀ (c: tass) Gamma x x',
       mextend Gamma (drop x c) x' = if beq_id x x' then Gamma x' else mextend
Gamma c x'.
   induction c; intros.
      destruct (beq_id x x'); auto.
      destruct a. simpl.
      remember (beq_id i x) as e; destruct e.
        apply beq_id_eq in Heqe; subst. rewrite IHc.
            remember (beq_id x x') as e; destruct e. auto. unfold extend. rewrite ←
Heqe. auto.
        simpl. unfold extend. remember (beq_id i x') as e; destruct e.
            apply beq_id_eq in Heqe0; subst.
                                   remember (beq_id x x') as e; destruct e.
```

```
                 apply beq_id_eq in Heqe0; subst. rewrite ← beq_id_refl in Heqe.
inversion Heqe.
                 auto.
            auto.
Qed.
```

Properties of Instantiations

These are strightforward.

```
Lemma instantiation_domains_match: ∀ {c} {e},
  instantiation c e → ∀ {x} {T}, lookup x c = Some T → ∃ t, lookup x e = 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 : ∀ c e, instantiation c e → 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 : ∀ c e, instantiation c e →
                          ∀ x t T, lookup x c = Some T →
                                       lookup x e = Some t → 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 : ∀ c env,
  instantiation c env → ∀ x, instantiation (drop x c) (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 : ∀ v t t',
  value v → (t ==>* t') → (tapp v t) ==>* (tapp v t').
Proof.
  intros v t t' V STM. induction STM.
   apply multi_refl.
   eapply multi_step.
     apply ST_App2; eauto. auto.
Qed.
```

```
Lemma multistep_If : ∀ t1 t1' t2 t3,
  (t1 ==>* t1') → (tif t1 t2 t3) ==>* (tif t1' t2 t3).
Proof.
  intros t1 t1' t2 t3 STM. induction STM.
  apply multi_refl.
  eapply multi_step.
  apply ST_If; 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 : ∀ c e,
     instantiation c e →
     ∀ Gamma t S, has_type (mextend Gamma c) t S →
     has_type Gamma (msubst e t) S.
Proof.
  induction 1; intros.
    simpl in X. simpl. auto.
    simpl in X0. simpl.
    apply IHX.
    eapply substitution_preserves_typing; eauto.
    apply (R_typable_empty r).
Qed.
```

```
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.
  multi_cases (induction G) Case.
    Case "multi_refl". assumption.
    Case "multi_step".
      apply multi_step with y. assumption.
      apply IHG. assumption. Qed.

Theorem multi_R : ∀ (X:Type) (R:relation X) (x y : X),
       R x y → multi R x y.
Proof.
  intros X R x y r.
  apply multi_step with y. apply r. apply multi_refl. Qed.
```

And at long last, the main lemma.

```
Lemma msubst_R : ∀ c env t T,
  has_type (mextend empty c) t T → instantiation c env → R T (msubst env t).
Proof.
  intros c env0 t T HT V.
  generalize dependent env0.
  remember (mextend empty c) as Gamma.
  assert (∀ x, Gamma x = lookup x c).
    intros. rewrite HeqGamma. rewrite mextend_lookup. auto.
  clear HeqGamma.
  generalize dependent c.
  has_type_cases (induction HT) Case; intros.

  Case "T_Var".
   rewrite H in e. destruct (instantiation_domains_match V e) as [t P].
   eapply instantiation_R; eauto.
   rewrite msubst_var. rewrite P. auto. eapply instantiation_env_closed; eauto.

  Case "T_Abs".
    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 extend. rewrite mextend_drop. remember (beq_id x x0) as e; destruct e.
auto.
        rewrite H.
          clear - c Heqe. induction c.
              simpl. rewrite ← Heqe. auto.
              simpl. destruct a. unfold extend. destruct (beq_id i x0); auto.
```

```
    unfold R. fold R. split.
      auto.
     split. apply value_halts. apply v_abs.
     intros.
     destruct (R_halts X) as [v [P Q]].
     pose proof (multistep_preserves_R _ _ _ P X).
     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 X0).
       eapply instantiation_env_closed; eauto.
       eapply (IHHT ((x,T11)::c)).
          intros. unfold extend, lookup. destruct (beq_id x x0); auto.
       constructor; auto.
  Case "T_App".
    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.
  Case "T_True".
    rewrite msubst_true.
    unfold R. split.
    apply T_True.
    split. unfold halts. ∃ ttrue. split. apply multi_refl. apply v_true. auto.
  Case "T_False".
    rewrite msubst_false.
    unfold R. split.
    apply T_False.
    split. unfold halts. ∃ tfalse. split. apply multi_refl. apply v_false. auto.
  Case "T_If".
    rewrite msubst_if.
    assert (WT: has_type empty (tif (msubst env0 t0) (msubst env0 t1) (msubst env0
t2)) T).
      apply T_If; eapply msubst_preserves_typing; eauto;
        eapply context_invariance; eauto; intros; rewrite ← mextend_lookup; auto.
    pose proof (IHHT1 c H env0 V) as IH1.
    destruct (R_halts IH1) as [v [P Q]].
```

```
  assert (R TBool v).
    eapply multistep_preserves_R. apply P. apply IH1.
 pose proof (R_typable_empty X).
 inversion Q; subst.
   inversion X0.

   inversion X0.

   eapply multistep_preserves_R' with (msubst env0 t1).
   assumption.
   eapply multi_trans. apply multistep_If. eapply P.
   eapply multi_R. apply ST_IfTrue.
   apply (IHHT2 c H env0 V).

   eapply multistep_preserves_R' with (msubst env0 t2).
   assumption.
   eapply multi_trans. apply multistep_If. eapply P.
   eapply multi_R. apply ST_IfFalse.
   apply (IHHT3 c H env0 V).
Qed.
```

Normalization Theorem

```
Theorem normalization : ∀ t T, has_type empty t T → halts t.
Proof.
  intros.
  replace t with (msubst nil t).
  eapply R_halts.
  eapply msubst_R; eauto. instantiate (2:= nil). eauto.
  eapply V_nil.
  auto.
Qed.
```

1.6 MoreStlc: A Typechecker for 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.

1.6.1 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 : ∀ 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 : ∀ T1 T2,
  beq_ty T1 T2 = true → T1 = T2.
Proof with auto.
  intros T1. induction T1; intros T2 Hbeq; destruct T2; inversion Hbeq.
  Case "T1=TBool".
    reflexivity.
  Case "T1=TArrow T1_1 T1_2".
    apply andb_true in H0. inversion H0 as [Hbeq1 Hbeq2].
    apply IHT1_1 in Hbeq1. apply IHT1_2 in Hbeq2. subst...
Qed.
```

1.6.2 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 *T1* and *T2*).

```
Fixpoint type_check (Gamma:context) (t:tm) : option ty :=
  match t with
  | tvar x ⇒ Gamma x
  | tabs x T11 t12 ⇒ match type_check (extend Gamma x T11) t12 with
                          | Some T12 ⇒ Some (TArrow T11 T12)
                          | _ ⇒ None
                        end
  | tapp t1 t2 ⇒ match type_check Gamma t1, type_check Gamma t2 with
                      | Some (TArrow T11 T12),Some T2 ⇒
                        if beq_ty T11 T2 then Some T12 else None
                      | _,_ ⇒ None
                    end
  | ttrue ⇒ Some TBool
  | tfalse ⇒ Some TBool
  | tif x t f ⇒ match type_check Gamma x with
                     | Some TBool ⇒
                       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.
```

1.6.3 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 : ∀ Gamma t T,
  type_check Gamma t = Some T → has_type Gamma t T.
Proof with eauto.
  intros Gamma t. generalize dependent Gamma.
  t_cases (induction t) Case; intros Gamma T Htc; inversion Htc; clear Htc.
  Case "tvar"...
  Case "tapp".
    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.
    remember (beq_ty T11 T2) as b.
    destruct b; try solve by inversion.
    symmetry in Heqb. apply beq_ty__eq in Heqb.
    inversion H0; subst...
  Case "tabs".
    rename i into y. rename t into T1.
    remember (extend Gamma y T1) as G'.
    remember (type_check G' t0) as TO2.
    destruct TO2; try solve by inversion.
    inversion H0; subst...
  Case "ttrue"...
  Case "tfalse"...
  Case "tif".
    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.
    remember (beq_ty T1 T2) as b.
    destruct b; try solve by inversion.
    symmetry in Heqb. apply beq_ty__eq in Heqb.
    inversion H0. subst. subst...
Qed.

Theorem type_checking_complete : ∀ Gamma t T,
  has_type Gamma t T → type_check Gamma t = Some T.
Proof with auto.
  intros Gamma t T Hty.
  has_type_cases (induction Hty) Case; simpl.
  Case "T_Var"...
  Case "T_Abs". rewrite IHHty...
  Case "T_App".
    rewrite IHHty1. rewrite IHHty2.
    rewrite (beq_ty_refl T1)...
  Case "T_True"...
  Case "T_False"...
  Case "T_If". rewrite IHHty1. rewrite IHHty2.
    rewrite IHHty3. rewrite (beq_ty_refl T)...
Qed.
```

Chapter 2

Library Extraction2

2.1 Extraction2: Extracting ML from Coq, Part 2

Let's take a look at a slightly fancier way of using Coq's extraction facilities. Instead of translating a functional program in Coq into a functional program in OCaml, we'll extract the computational content of an interesting *proof* – the proof that the STLC is normalizing. The result will be an evaluator that takes well-typed STLC terms to their final normal forms.

Note that this is a slightly unusual way of using Coq. In the words of Adam Chlipala...

Many fans of the Curry-Howard correspondence support the idea of extracting programs from proofs. In reality, few users of Coq and related tools do any such thing. Instead, extraction is better thought of as an optimization that reduces the runtime costs of expressive typing. (Chlipala, CPDT)

However, it's interesting to see that it's possible at all! (And to think about what the extracted program does.)

2.2 Extracting a Normalizer

```
Require Import Extraction.
Require Import NormInType.
```

The *NormInType* module is a variant of the *Norm* module with a few significant differences. The essential point is that, during extraction, everything to do with `Prop` is *erased*. So, to extract a normalizer from a proof of normalization, we need to carry out the essential bits of the normalization proof in `Type` rather than `Prop`.

NormInType also incorporates a copy of the STLC typechecking function from the *Typechecking* module and its proof of correctness. (The function itself is no different from before, and the correctness proof has just a few small differences because of the changes we made to the basic definitions of STLC.) We need these things because the proof of normalization proceeds by induction on a typing derivation, so the extracted normalization function must be passed a data structure representing a typing derivation. In *normdriver.ml*, we obtain this derivation from the proof that the typechecking algorithm is sound.

`Extraction` "norm.ml" *normalization type_check type_checking_sound*.

Take a look at *normdriver.ml* to see how this plumbing works in detail.

Finally, we can compile and execute our normalizer in the same way as we did with our evaluator for *Imp*.

```
ocamlc -w -20 -o normdriver norm.mli norm.ml normdriver.ml
./normdriver
```

Chapter 3

Library Postscript

3.1 Postscript

3.2 Looking back...

- *Functional programming* - "declarative" programming (recursion over persistent data structures)
  - higher-order functions
  - polymorphism
- *Coq*, an industrial-strength proof assistant
- *Logic*, the mathematical basis for software engineering:

```
        logic                        calculus
--------------------   =   ----------------------------
software engineering       mechanical/civil engineering
```

  - inductively defined sets and relations
  - inductive proofs
  - proof objects
- *Foundations of programming languages*
  - notations and definitional techniques for precisely specifying
    * abstract syntax
    * operational semantics
      · big-step style

            - small-step style
        - type systems
    - Hoare logic
    - program equivalence
    - fundamental metatheory of type systems
        - progress and preservation
    - theory of subtyping

3.3 Looking Forward

Some good places to go for more...

- Optional chapters of *Software Foundations* :-)
- Cutting-edge conferences on programming languages:
    - POPL
    - PLDI
    - OOPSLA
    - ICFP
    - (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 Castéran. Springer-Verlag, 2004.

Chapter 4

Library PE

4.1 PE: Partial Evaluation

Equiv.v introduced constant folding as an example of a program transformation and proved that it preserves the meaning of the program. 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 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 Export Imp.
Require Import FunctionalExtensionality.
```

4.2 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.

4.2.1 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' SCase means to reason by cases whether *beq_id V V'* is *true* or *false*. In the case where *beq_id V V'* = *true*, the tactic substitutes *V* for *V'* throughout.

```
Tactic Notation "compare" ident(i) ident(j) ident(c) :=
  let H := fresh "Heq" i j in
  destruct (beq_id i j) as [|]eqn:H;
  [ Case_aux c "equal"; symmetry in H; apply beq_id_eq in H; subst j
  | Case_aux c "not equal" ].
```

```
Theorem pe_domain: ∀ pe_st V n,
  pe_lookup pe_st V = Some n →
  true = existsb (beq_id V) (map (@fst _ _) pe_st).
Proof. intros pe_st V n H. induction pe_st as [| [V' n'] pe_st].
  Case "[]". inversion H.
  Case "::". simpl in H. simpl. compare V V' SCase; auto. Qed.
```

4.2.2 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
  | AId i ⇒ match pe_lookup pe_st i with
               | Some n ⇒ ANum n
               | None ⇒ AId i
               end
  | APlus a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 + n2)
      | (a1', a2') ⇒ APlus a1' a2'
      end
  | AMinus a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 - n2)
      | (a1', a2') ⇒ AMinus a1' a2'
      end
  | AMult a1 a2 ⇒
      match (pe_aexp pe_st a1, pe_aexp pe_st a2) with
      | (ANum n1, ANum n2) ⇒ ANum (n1 × n2)
      | (a1', a2') ⇒ 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 (AId X) (ANum 1)) (AId Y))
  = APlus (ANum 4) (AId Y).
Proof. reflexivity. Qed.
```

```
Example text_pe_aexp2:
  pe_aexp [(Y,3)] (APlus (APlus (AId X) (ANum 1)) (AId Y))
  = APlus (APlus (AId 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) :=
  ∀ V n, Some n = pe_lookup pe_st V → st V = n.

Theorem pe_aexp_correct_weak: ∀ st pe_st, pe_consistent st pe_st →
  ∀ a, aeval st a = aeval st (pe_aexp pe_st a).
Proof. unfold pe_consistent. intros st pe_st H a.
  aexp_cases (induction a) Case; simpl;
    try reflexivity;
    try (destruct (pe_aexp pe_st a1);
         destruct (pe_aexp pe_st a2);
         rewrite IHa1; rewrite IHa2; reflexivity).
  Case "AId".
    remember (pe_lookup pe_st i) as l. destruct l.
    SCase "Some". rewrite H with (n:=n) by apply Heql. reflexivity.
    SCase "None". reflexivity.
Qed.
```

However, we will soon want our partial evaluator to remove assignments. For example, it will simplify X ::= ANum 3; Y ::= AMinus (AId X) (AId Y); X ::= ANum 4 to just Y ::= AMinus (ANum 3) (AId Y); X ::= ANum 4 by delaying the assignment to X until the end. To accomplish this simplification, we need the result of partial evaluating pe_aexp (X,3) (AMinus (AId X) (AId Y)) to be equal to *AMinus* (*ANum* 3) (*AId Y*) and *not* the original expression *AMinus* (*AId X*) (*AId Y*). After all, it would be incorrect, not just inefficient, to transform X ::= ANum 3; Y ::= AMinus (AId X) (AId Y); X ::= ANum 4 to Y ::= AMinus (AId X) (AId Y); X ::= ANum 4 even though the output expressions *AMinus* (*ANum* 3) (*AId Y*) and *AMinus* (*AId X*) (*AId Y*) both satisfy the correctness criterion that we just proved. Indeed, if we were to just define *pe_aexp pe_st a* = *a* then the theorem *pe_aexp_correct'* would already trivially hold.

Instead, we want to prove that the *pe_aexp* is correct in a stronger sense: evaluating the expression produced by partial evaluation (*aeval st* (*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_override (st:state) (pe_st:pe_state) : state :=
  match pe_st with
  | [] ⇒ st
  | (V,n)::pe_st ⇒ update (pe_override st pe_st) V n
  end.

Example test_pe_override:
  pe_override (update empty_state Y 1) [(X,3),(Z,2)]
  = update (update (update empty_state Y 1) Z 2) X 3.
Proof. reflexivity. Qed.
```

Although *pe_override* operates on a concrete *list* representing a *pe_state*, its behavior is defined entirely by the *pe_lookup* interpretation of the *pe_state*.

```
Theorem pe_override_correct: ∀ st pe_st V0,
  pe_override 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 update. rewrite beq_id_sym.
  compare V0 V Case; auto. Qed.
```

We can relate *pe_consistent* to *pe_override* 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_override_consistent: ∀ st pe_st,
  pe_consistent (pe_override st pe_st) pe_st.
Proof. intros st pe_st V n H. rewrite pe_override_correct.
  destruct (pe_lookup pe_st V); inversion H. reflexivity. Qed.
```

```
Theorem pe_consistent_override: ∀ st pe_st,
  pe_consistent st pe_st → ∀ V, st V = pe_override st pe_st V.
Proof. intros st pe_st H V. rewrite pe_override_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: ∀ (pe_st:pe_state) (a:aexp) (st:state),
  aeval (pe_override st pe_st) a = aeval st (pe_aexp pe_st a).
Proof.
  intros pe_st a st.
  aexp_cases (induction a) Case; simpl;
    try reflexivity;
    try (destruct (pe_aexp pe_st a1);
         destruct (pe_aexp pe_st a2);
         rewrite IHa1; rewrite IHa2; reflexivity).
  rewrite pe_override_correct. destruct (pe_lookup pe_st i); reflexivity.
Qed.
```

4.2.3 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 ble_nat 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 (AId X) (ANum 3)))
  = BFalse.
Proof. reflexivity. Qed.

Example test_pe_bexp2: ∀ b,
  b = BNot (BLe (AId X) (APlus (AId 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: ∀ (pe_st:pe_state) (b:bexp) (st:state),
  beval (pe_override st pe_st) b = beval st (pe_bexp pe_st b).
Proof.
  intros pe_st b st.
  bexp_cases (induction b) Case; simpl;
    try reflexivity;
    try (remember (pe_aexp pe_st a) as a';
         remember (pe_aexp pe_st a0) as a0';
         assert (Ha: aeval (pe_override st pe_st) a = aeval st a');
         assert (Ha0: aeval (pe_override 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 (ble_nat 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.
```

4.3 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 || 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 ::= ANum 3 ; Y ::= AMult (AId Z) (APlus (AId X) (AId X))) / ☐ || (Y ::= AMult (AId Z) (ANum 6)) / (*X*,3) to hold. The assignment to *X* appears in the final partial state, not the residual command.

4.3.1 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* ::= *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* ::= *AMult* (*AId Z*) (*APlus* (*AId X*) (*AId 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_override* 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].
  Case "[]". destruct (beq_id V V0); reflexivity.
  Case "::". simpl. compare V V' SCase.
    SCase "equal". rewrite IHpe_st.
      replace (beq_id V0 V) with (beq_id V V0) by apply beq_id_sym.
      destruct (beq_id V V0); reflexivity.
    SCase "not equal". simpl. compare V0 V' SSCase.
      SSCase "equal". rewrite HeqVV'. reflexivity.
      SSCase "not equal". 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. rewrite beq_id_sym.
```

```
  compare V V0 Case.
  Case "equal". reflexivity.
  Case "not equal". rewrite pe_remove_correct. rewrite HeqVV0. reflexivity.
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_override_update_remove: ∀ st pe_st V n,
  update (pe_override st pe_st) V n =
  pe_override (update st V n) (pe_remove pe_st V).
Proof. intros st pe_st V n. apply functional_extensionality. intros V0.
  unfold update. rewrite !pe_override_correct. rewrite pe_remove_correct.
  destruct (beq_id V V0); reflexivity. Qed.

Theorem pe_override_update_add: ∀ st pe_st V n,
  update (pe_override st pe_st) V n =
  pe_override st (pe_add pe_st V n).
Proof. intros st pe_st V n. apply functional_extensionality. intros V0.
  unfold update. rewrite !pe_override_correct. rewrite pe_add_correct.
  destruct (beq_id V V0); reflexivity. Qed.
```

4.3.2 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.

Lemma existsb_app: ∀ X (f:X→bool) l1 l2,
  existsb f (l1 ++ l2) = orb (existsb f l1) (existsb f l2).
Proof. intros X f l1 l2. induction l1. reflexivity.
  simpl. rewrite IHl1. rewrite orb_assoc. reflexivity. Qed.

Theorem pe_disagree_domain: ∀ (pe_st1 pe_st2 : pe_state) (V:id),
  true = pe_disagree_at pe_st1 pe_st2 V →
  true = existsb (beq_id V) (map (@fst _ _) pe_st1 ++
                                  map (@fst _ _) pe_st2).
Proof. unfold pe_disagree_at. intros pe_st1 pe_st2 V H.
  rewrite existsb_app. symmetry. apply orb_true_intro.
  remember (pe_lookup pe_st1 V) as lookup1.
  destruct lookup1 as [n1|]. left. symmetry. apply pe_domain with n1. auto.
  remember (pe_lookup pe_st2 V) as lookup2.
  destruct lookup2 as [n2|]. right. symmetry. 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
  | [] ⇒ []
  | x::l ⇒ x :: filter (fun y ⇒ negb (beq_id x y)) (pe_unique l)
  end.

Lemma existsb_beq_id_filter: ∀ V f l,
  existsb (beq_id V) (filter f l) = andb (existsb (beq_id V) l) (f V).
Proof. intros V f l. induction l as [| h l].
  Case "[]". reflexivity.
  Case "h::l". simpl. remember (f h) as fh. destruct fh.
    SCase "true = f h". simpl. rewrite IHl. compare V h SSCase.
      rewrite ← Heqfh. reflexivity. reflexivity.
    SCase "false = f h". rewrite IHl. compare V h SSCase.
      rewrite ← Heqfh. rewrite !andb_false_r. reflexivity. reflexivity.
Qed.
```

```
Theorem pe_unique_correct: ∀ l x,
  existsb (beq_id x) l = existsb (beq_id x) (pe_unique l).
Proof. intros l x. induction l as [| h t]. reflexivity.
  simpl in *. compare x h Case.
  Case "equal". reflexivity.
  Case "not equal".
    rewrite → existsb_beq_id_filter, ← IHt, → beq_id_sym, → Heqxh,
            → andb_true_r. reflexivity. 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: ∀ pe_st1 pe_st2 V,
  pe_lookup pe_st1 V = pe_lookup pe_st2 V ↔
  false = existsb (beq_id V) (pe_compare pe_st1 pe_st2).
Proof. intros pe_st1 pe_st2 V.
  unfold pe_compare. rewrite ← pe_unique_correct, → existsb_beq_id_filter.
  split; intros Heq.
  Case "->".
    symmetry. apply andb_false_intro2. unfold pe_disagree_at. rewrite Heq.
    destruct (pe_lookup pe_st2 V).
    rewrite ← beq_nat_refl. reflexivity.
    reflexivity.
  Case "<-".
    assert (Hagree: pe_disagree_at pe_st1 pe_st2 V = false).
      SCase "Proof of assertion".
      remember (pe_disagree_at pe_st1 pe_st2 V) as disagree.
      destruct disagree; [| reflexivity].
      rewrite → andb_true_r, ← pe_disagree_domain in Heq.
      inversion Heq.
      apply Heqdisagree.
    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 beq_nat_eq with n1 n2. reflexivity.
    rewrite ← negb_involutive. rewrite Hagree. 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_override* only depends on the *pe_lookup* interpretation of partial states, *pe_override* also does not care which of the two partial states we remove the variables from; that theorem *pe_compare_override* 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: ∀ pe_st ids V,
  pe_lookup (pe_removes pe_st ids) V =
  if existsb (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.
  replace (beq_id V' V) with (beq_id V V') by apply beq_id_sym.
  destruct (beq_id V V'); destruct (existsb (beq_id V) ids); reflexivity.
Qed.

Theorem pe_compare_removes: ∀ 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.
  remember (existsb (beq_id V) (pe_compare pe_st1 pe_st2)) as b.
  destruct b. reflexivity.
  apply pe_compare_correct in Heqb. apply Heqb. Qed.

Theorem pe_compare_override: ∀ pe_st1 pe_st2 st,
  pe_override st (pe_removes pe_st1 (pe_compare pe_st1 pe_st2)) =
  pe_override st (pe_removes pe_st2 (pe_compare pe_st1 pe_st2)).
Proof. intros. apply functional_extensionality. intros V.
  rewrite !pe_override_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 ⇒ match existsb (beq_id V) ids, pe_lookup pe_st V with
            | true, Some n ⇒ n
            | _, _ ⇒ st V
            end.

Theorem assign_removes: ∀ pe_st ids st,
  pe_override st pe_st =
  pe_override (assigned pe_st ids st) (pe_removes pe_st ids).
Proof. intros pe_st ids st. apply functional_extensionality. intros V.
  rewrite !pe_override_correct. rewrite pe_removes_correct. unfold assigned.
  destruct (existsb (beq_id V)); destruct (pe_lookup pe_st V); reflexivity.
Qed.

Lemma ceval_extensionality: ∀ c st st1 st2,
  c / st || st1 → (∀ V, st1 V = st2 V) → c / st || st2.
Proof. intros c st st1 st2 H Heq.
  apply functional_extensionality in Heq. rewrite ← Heq. apply H. Qed.

Theorem eval_assign: ∀ pe_st ids st,
  assign pe_st ids / st || assigned pe_st ids st.
Proof. intros pe_st ids st. induction ids as [| V ids]; simpl.
  Case "[]". eapply ceval_extensionality. apply E_Skip. reflexivity.
  Case "V::ids".
    remember (pe_lookup pe_st V) as lookup. destruct lookup.
    SCase "Some". eapply E_Seq. apply IHids. unfold assigned. simpl.
      eapply ceval_extensionality. apply E_Ass. simpl. reflexivity.
      intros V0. unfold update. rewrite beq_id_sym. compare V0 V SSCase.
      SSCase "equal". rewrite ← Heqlookup. reflexivity.
      SSCase "not equal". reflexivity.
    SCase "None". eapply ceval_extensionality. apply IHids.
      unfold assigned. intros V0. simpl. compare V0 V SSCase.
      SSCase "equal". rewrite ← Heqlookup.
        destruct (existsb (beq_id V0) ids); reflexivity.
      SSCase "not equal". reflexivity. Qed.
```

4.3.3 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 '||' c1' '/' st'"
```

```
  (at level 40, st at level 39, c1' at level 39).
Inductive pe_com : com → pe_state → com → pe_state → Prop :=
  | PE_Skip : ∀ pe_st,
      SKIP / pe_st || SKIP / pe_st
  | PE_AssStatic : ∀ pe_st a1 n1 l,
      pe_aexp pe_st a1 = ANum n1 →
      (l ::= a1) / pe_st || SKIP / pe_add pe_st l n1
  | PE_AssDynamic : ∀ 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').
Tactic Notation "pe_com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "PE_Skip"
  | Case_aux c "PE_AssStatic" | Case_aux c "PE_AssDynamic"
  | Case_aux c "PE_Seq"
  | Case_aux c "PE_IfTrue" | Case_aux c "PE_IfFalse" | Case_aux c "PE_If" ].
Hint Constructors pe_com.
```

```
Hint Constructors ceval.
```

4.3.4 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 (AId Z) (APlus (AId X) (AId X)))
  / [] || (SKIP; Y ::= AMult (AId 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 (AId 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 (AId Y) (ANum 4) THEN
     Y ::= ANum 4;
     IFB BEq (AId X) (AId Y) THEN Y ::= ANum 999 ELSE SKIP FI
   ELSE SKIP FI) / []
  || (SKIP;
       IFB BLe (AId 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.
```

4.3.5 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_override 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_override st pe_st || st'') →
  (c' / pe_st' / st || st'').
Proof. intros c pe_st pe_st' c' Hpe.
  pe_com_cases (induction Hpe) Case; intros st st'' Heval;
  try (inversion Heval; subst;
       try (rewrite → pe_bexp_correct, → H in *; solve by inversion);
       []);
  eauto.
  Case "PE_AssStatic". econstructor. econstructor.
    rewrite → pe_aexp_correct. rewrite ← pe_override_update_add.
    rewrite → H. reflexivity.
  Case "PE_AssDynamic". econstructor. econstructor. reflexivity.
    rewrite → pe_aexp_correct. rewrite ← pe_override_update_remove.
    reflexivity.
  Case "PE_Seq".
    edestruct IHHpe1. eassumption. subst.
    edestruct IHHpe2. eassumption.
    eauto.
  Case "PE_If". inversion Heval; subst.
    SCase "E'IfTrue". edestruct IHHpe1. eassumption.
      econstructor. apply E_IfTrue. rewrite ← pe_bexp_correct. assumption.
      eapply E_Seq. eassumption. apply eval_assign.
      rewrite ← assign_removes. eassumption.
    SCase "E_IfFalse". edestruct IHHpe2. eassumption.
      econstructor. apply E_IfFalse. rewrite ← pe_bexp_correct. assumption.
      eapply E_Seq. eassumption. apply eval_assign.
      rewrite → pe_compare_override.
      rewrite ← assign_removes. eassumption.
Qed.

Theorem pe_com_sound:
  ∀ c pe_st pe_st' c', c / pe_st || c' / pe_st' →
```

```
  ∀ st st'',
  (c' / pe_st' / st || st'') →
  (c / pe_override st pe_st || st'').
Proof. intros c pe_st pe_st' c' Hpe.
  pe_com_cases (induction Hpe) Case;
    intros st st'' [st' Heval Heq];
    try (inversion Heval; []; subst); auto.
  Case "PE_AssStatic". rewrite ← pe_override_update_add. apply E_Ass.
    rewrite → pe_aexp_correct. rewrite → H. reflexivity.
  Case "PE_AssDynamic". rewrite ← pe_override_update_remove. apply E_Ass.
    rewrite ← pe_aexp_correct. reflexivity.
  Case "PE_Seq". eapply E_Seq; eauto.
  Case "PE_IfTrue". apply E_IfTrue.
    rewrite → pe_bexp_correct. rewrite → H. reflexivity. eauto.
  Case "PE_IfFalse". apply E_IfFalse.
    rewrite → pe_bexp_correct. rewrite → H. reflexivity. eauto.
  Case "PE_If".
    inversion Heval; subst; inversion H7;
      (eapply ceval_deterministic in H8; [| apply eval_assign]); subst.
    SCase "E_IfTrue".
      apply E_IfTrue. rewrite → pe_bexp_correct. assumption.
      rewrite ← assign_removes. eauto.
    SCase "E_IfFalse".
      rewrite → pe_compare_override.
      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:
  ∀ c pe_st pe_st' c', c / pe_st || c' / pe_st' →
  ∀ st st'',
  (c / pe_override st pe_st || st'') ↔
  (c' / pe_st' / st || st'').
Proof. intros c pe_st pe_st' c' H st st''. split.
  Case "->". apply pe_com_complete. apply H.
  Case "<-". apply pe_com_sound. apply H.
Qed.
```

4.4 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 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 if 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 '/' st '||' c1' '/' st' '/' 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 : ∀ pe_st,
      SKIP / pe_st || SKIP / pe_st / SKIP
  | PE_AssStatic : ∀ pe_st a1 n1 l,
      pe_aexp pe_st a1 = ANum n1 →
      (l ::= a1) / pe_st || SKIP / pe_add pe_st l n1 / SKIP
  | PE_AssDynamic : ∀ 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 / SKIP
  | PE_Seq : ∀ pe_st pe_st' pe_st'' c1 c2 c1' c2' c'',
      c1 / pe_st || c1' / pe_st' / SKIP →
      c2 / pe_st' || c2' / pe_st'' / c'' →
      (c1 ; c2) / pe_st || (c1' ; c2') / pe_st'' / c''
  | PE_IfTrue : ∀ pe_st pe_st' b1 c1 c2 c1' c'',
```

```
      pe_bexp pe_st b1 = BTrue →
      c1 / pe_st || c1' / pe_st' / c'' →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st || c1' / pe_st' / c''
  | PE_IfFalse : ∀ pe_st pe_st' b1 c1 c2 c2' c'',
      pe_bexp pe_st b1 = BFalse →
      c2 / pe_st || c2' / pe_st' / c'' →
      (IFB b1 THEN c1 ELSE c2 FI) / pe_st || c2' / pe_st' / c''
  | PE_If : ∀ pe_st pe_st1 pe_st2 b1 c1 c2 c1' c2' c'',
      pe_bexp pe_st b1 ≠ BTrue →
      pe_bexp pe_st b1 ≠ BFalse →
      c1 / pe_st || c1' / pe_st1 / c'' →
      c2 / pe_st || c2' / pe_st2 / c'' →
      (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)
             / c''
  | PE_WhileEnd : ∀ pe_st b1 c1,
      pe_bexp pe_st b1 = BFalse →
      (WHILE b1 DO c1 END) / pe_st || SKIP / pe_st / SKIP
  | PE_WhileLoop : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2' c2'',
      pe_bexp pe_st b1 = BTrue →
      c1 / pe_st || c1' / pe_st' / SKIP →
      (WHILE b1 DO c1 END) / pe_st' || c2' / pe_st'' / c2'' →
      pe_compare pe_st pe_st'' ≠ [] →
      (WHILE b1 DO c1 END) / pe_st || (c1';c2') / pe_st'' / c2''
  | PE_While : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2' c2'',
      pe_bexp pe_st b1 ≠ BFalse →
      pe_bexp pe_st b1 ≠ BTrue →
      c1 / pe_st || c1' / pe_st' / SKIP →
      (WHILE b1 DO c1 END) / pe_st' || c2' / pe_st'' / c2'' →
      pe_compare pe_st pe_st'' ≠ [] →
      (c2'' = SKIP ∨ c2'' = WHILE b1 DO c1 END) →
      (WHILE b1 DO c1 END) / pe_st
        || (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 : ∀ pe_st b1 c1,
      pe_bexp pe_st b1 ≠ BFalse →
```

```
    (WHILE b1 DO c1 END) / pe_st || SKIP / pe_st / (WHILE b1 DO c1 END)
  | PE_WhileFixedLoop : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2',
      pe_bexp pe_st b1 = BTrue →
      c1 / pe_st || c1' / pe_st' / SKIP →
      (WHILE b1 DO c1 END) / pe_st'
        || c2' / pe_st'' / (WHILE b1 DO c1 END) →
      pe_compare pe_st pe_st'' = [] →
      (WHILE b1 DO c1 END) / pe_st
        || (WHILE BTrue DO SKIP END) / pe_st / SKIP

  | PE_WhileFixed : ∀ pe_st pe_st' pe_st'' b1 c1 c1' c2',
      pe_bexp pe_st b1 ≠ BFalse →
      pe_bexp pe_st b1 ≠ BTrue →
      c1 / pe_st || c1' / pe_st' / SKIP →
      (WHILE b1 DO c1 END) / pe_st'
        || c2' / pe_st'' / (WHILE b1 DO c1 END) →
      pe_compare pe_st pe_st'' = [] →
      (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'').
Tactic Notation "pe_com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "PE_Skip"
  | Case_aux c "PE_AssStatic" | Case_aux c "PE_AssDynamic"
  | Case_aux c "PE_Seq"
  | Case_aux c "PE_IfTrue" | Case_aux c "PE_IfFalse" | Case_aux c "PE_If"
  | Case_aux c "PE_WhileEnd" | Case_aux c "PE_WhileLoop"
  | Case_aux c "PE_While" | Case_aux c "PE_WhileFixedEnd"
  | Case_aux c "PE_WhileFixedLoop" | Case_aux c "PE_WhileFixed" ].
Hint Constructors pe_com.
```

4.4.1 Examples

```
Tactic Notation "step" ident(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) (AId X) DO
  Y ::= AMult (AId Y) (AId Y);
  X ::= AMinus (AId X) (ANum 1)
END.
```

Example pe_loop_example1: square_loop / □ || (WHILE BLe (ANum 1) (AId X) DO (Y ::= AMult (AId Y) (AId Y); X ::= AMinus (AId 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 (AId Y) (AId Y); SKIP); (Y ::= AMult (AId Y) (AId Y); SKIP); (Y ::= AMult (AId Y) (AId 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 (AId X) (ANum 0)) THEN (SKIP; X ::= AMinus (AId X) (ANum 1)); IFB BNot (BEq (AId X) (ANum 0)) THEN (SKIP; X ::= AMinus (AId X) (ANum 1)); IFB BNot (BEq (AId X) (ANum 0)) THEN (SKIP; X ::= AMinus (AId X) (ANum 1)); WHILE BNot (BEq (AId X) (ANum 0)) DO (SKIP; X ::= AMinus (AId 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 (AId X) (ANum 2) DO X ::= AMinus (ANum 1) (AId 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.

4.4.2 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 '/' st '||' st' '#' n"
  (at level 40, st at level 39, st' at level 39).

Inductive ceval_count : com → state → state → nat → Prop :=
  | E'Skip : ∀ st,
      SKIP / st || st # 0
  | E'Ass : ∀ st a1 n l,
      aeval st a1 = n →
```

```
      (l ::= a1) / st || (update st l n) # 0
  | E'Seq : ∀ c1 c2 st st' st'' n1 n2,
      c1 / st || st' # n1 →
      c2 / st' || st'' # n2 →
      (c1 ; c2) / st || st'' # (n1 + n2)
  | E'IfTrue : ∀ st st' b1 c1 c2 n,
      beval st b1 = true →
      c1 / st || st' # n →
      (IFB b1 THEN c1 ELSE c2 FI) / st || st' # n
  | E'IfFalse : ∀ st st' b1 c1 c2 n,
      beval st b1 = false →
      c2 / st || st' # n →
      (IFB b1 THEN c1 ELSE c2 FI) / st || st' # n
  | E'WhileEnd : ∀ b1 st c1,
      beval st b1 = false →
      (WHILE b1 DO c1 END) / st || st # 0
  | E'WhileLoop : ∀ st st' st'' b1 c1 n1 n2,
      beval st b1 = true →
      c1 / st || st' # n1 →
      (WHILE b1 DO c1 END) / st' || st'' # n2 →
      (WHILE b1 DO c1 END) / st || st'' # S (n1 + n2)

  where "c1 '/' st '||' st' # n" := (ceval_count c1 st st' n).

Tactic Notation "ceval_count_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E'Skip" | Case_aux c "E'Ass" | Case_aux c "E'Seq"
  | Case_aux c "E'IfTrue" | Case_aux c "E'IfFalse"
  | Case_aux c "E'WhileEnd" | Case_aux c "E'WhileLoop" ].

Hint Constructors ceval_count.

Theorem ceval_count_complete: ∀ c st st',
  c / st || st' → ∃ n, c / st || 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: ∀ c st st' n,
  c / st || st' # n → c / st || st'.
Proof. intros c st st' n Heval. induction Heval; eauto. Qed.

Theorem pe_compare_nil_lookup: ∀ pe_st1 pe_st2,
```

```
  pe_compare pe_st1 pe_st2 = [] →
  ∀ 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. reflexivity. Qed.

Theorem pe_compare_nil_override: ∀ pe_st1 pe_st2,
  pe_compare pe_st1 pe_st2 = [] →
  ∀ st, pe_override st pe_st1 = pe_override st pe_st2.
Proof. intros pe_st1 pe_st2 H st.
  apply functional_extensionality. intros V.
  rewrite !pe_override_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 : ∀ st' n',
     c' / st || st' →
     c'' / pe_override st' pe_st' || st'' # n' →
     n' ≤ n →
     c' / pe_st' / c'' / st || 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: ∀ c' pe_st' c'' st st'' n n',
  n' ≤ n →
  c' / pe_st' / c'' / st || st'' # n' →
  c' / pe_st' / c'' / st || 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:
  ∀ c pe_st pe_st' c' c'', c / pe_st || c' / pe_st' / c'' →
  ∀ st st'' n,
  (c / pe_override st pe_st || st'' # n) →
  (c' / pe_st' / c'' / st || st'' # n).
Proof. intros c pe_st pe_st' c' c'' Hpe.
  pe_com_cases (induction Hpe) Case; intros st st'' n Heval;
  try (inversion Heval; subst;
       try (rewrite → pe_bexp_correct, → H in *; solve by inversion);
```

```
      []);
eauto.
Case "PE_AssStatic". econstructor. econstructor.
  rewrite → pe_aexp_correct. rewrite ← pe_override_update_add.
  rewrite → H. apply E'Skip. auto.
Case "PE_AssDynamic". econstructor. econstructor. reflexivity.
  rewrite → pe_aexp_correct. rewrite ← pe_override_update_remove.
  apply E'Skip. auto.
Case "PE_Seq".
  edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
  inversion Hskip. subst.
  edestruct IHHpe2. eassumption.
  econstructor; eauto. omega.
Case "PE_If". inversion Heval; subst.
  SCase "E'IfTrue". edestruct IHHpe1. eassumption.
    econstructor. apply E_IfTrue. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite ← assign_removes. eassumption. eassumption.
  SCase "E_IfFalse". edestruct IHHpe2. eassumption.
    econstructor. apply E_IfFalse. rewrite ← pe_bexp_correct. assumption.
    eapply E_Seq. eassumption. apply eval_assign.
    rewrite → pe_compare_override.
    rewrite ← assign_removes. eassumption. eassumption.
Case "PE_WhileLoop".
  edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
  inversion Hskip. subst.
  edestruct IHHpe2. eassumption.
  econstructor; eauto. omega.
Case "PE_While". inversion Heval; subst.
  SCase "E_WhileEnd". econstructor. apply E_IfFalse.
    rewrite ← pe_bexp_correct. assumption.
    apply eval_assign.
    rewrite ← assign_removes. inversion H2; subst; auto.
    auto.
  SCase "E_WhileLoop".
    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_override, ← assign_removes. eassumption.
```

```
      omega.
  Case "PE_WhileFixedLoop". apply ex_falso_quodlibet.
    generalize dependent (S (n1 + n2)). intros n.
    clear - Case H H0 IHHpe1 IHHpe2. generalize dependent st.
    induction n using lt_wf_ind; intros st Heval. inversion Heval; subst.
    SCase "E'WhileEnd". rewrite pe_bexp_correct, H in H7. inversion H7.
    SCase "E'WhileLoop".
      edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
      inversion Hskip. subst.
      edestruct IHHpe2. eassumption.
      rewrite ← (pe_compare_nil_override _ _ H0) in H7.
      apply H1 in H7; [| omega]. inversion H7.
  Case "PE_WhileFixed". generalize dependent st.
    induction n using lt_wf_ind; intros st Heval. inversion Heval; subst.
    SCase "E'WhileEnd". rewrite pe_bexp_correct in H8. eauto.
    SCase "E'WhileLoop". rewrite pe_bexp_correct in H5.
      edestruct IHHpe1 as [? ? ? Hskip ?]. eassumption.
      inversion Hskip. subst.
      edestruct IHHpe2. eassumption.
      rewrite ← (pe_compare_nil_override _ _ H1) in H8.
      apply H2 in H8; [| omega]. inversion H8.
      econstructor; [ eapply E_WhileLoop; eauto | eassumption | omega].
Qed.

Theorem pe_com_sound:
  ∀ c pe_st pe_st' c' c'', c / pe_st || c' / pe_st' / c'' →
  ∀ st st'' n,
  (c' / pe_st' / c'' / st || st'' # n) →
  (c / pe_override st pe_st || st'').
Proof. intros c pe_st pe_st' c' c'' Hpe.
  pe_com_cases (induction Hpe) Case;
    intros st st'' n [st' n' Heval Heval' Hle];
    try (inversion Heval; []; subst);
    try (inversion Heval'; []; subst); eauto.
  Case "PE_AssStatic". rewrite ← pe_override_update_add. apply E_Ass.
    rewrite → pe_aexp_correct. rewrite → H. reflexivity.
  Case "PE_AssDynamic". rewrite ← pe_override_update_remove. apply E_Ass.
    rewrite ← pe_aexp_correct. reflexivity.
  Case "PE_Seq". eapply E_Seq; eauto.
  Case "PE_IfTrue". apply E_IfTrue.
    rewrite → pe_bexp_correct. rewrite → H. reflexivity.
    eapply IHHpe. eauto.
  Case "PE_IfFalse". apply E_IfFalse.
```

```
    rewrite -> pe_bexp_correct. rewrite -> H. reflexivity.
    eapply IHHpe. eauto.
  Case "PE_If". inversion Heval; subst; inversion H7; subst; clear H7.
    SCase "E_IfTrue".
      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.
    SCase "E_IfFalse".
      eapply ceval_deterministic in H8; [| apply eval_assign]. subst.
      rewrite -> pe_compare_override in Heval'.
      rewrite <- assign_removes in Heval'.
      apply E_IfFalse. rewrite -> pe_bexp_correct. assumption.
      eapply IHHpe2. eauto.
  Case "PE_WhileEnd". apply E_WhileEnd.
    rewrite -> pe_bexp_correct. rewrite -> H. reflexivity.
  Case "PE_WhileLoop". eapply E_WhileLoop.
    rewrite -> pe_bexp_correct. rewrite -> H. reflexivity.
    eapply IHHpe1. eauto. eapply IHHpe2. eauto.
  Case "PE_While". inversion Heval; subst.
    SCase "E_IfTrue".
      inversion H9. subst. clear H9.
      inversion H10. subst. clear H10.
      eapply ceval_deterministic in H11; [| apply eval_assign]. subst.
      rewrite -> pe_compare_override in Heval'.
      rewrite <- assign_removes in Heval'.
      eapply E_WhileLoop. rewrite -> pe_bexp_correct. assumption.
      eapply IHHpe1. eauto.
      eapply IHHpe2. eauto.
    SCase "E_IfFalse". apply ceval_count_sound in Heval'.
      eapply ceval_deterministic in H9; [| apply eval_assign]. subst.
      rewrite <- assign_removes in Heval'.
      inversion H2; subst.
      SSCase "c2'' = SKIP". inversion Heval'. subst. apply E_WhileEnd.
        rewrite -> pe_bexp_correct. assumption.
      SSCase "c2'' = WHILE b1 DO c1 END". assumption.
  Case "PE_WhileFixedEnd". eapply ceval_count_sound. apply Heval'.
  Case "PE_WhileFixedLoop".
    apply loop_never_stops in Heval. inversion Heval.
  Case "PE_WhileFixed".
    clear - Case H1 IHHpe1 IHHpe2 Heval.
    remember (WHILE pe_bexp pe_st b1 DO c1'; c2' END) as c'.
```

```
ceval_cases (induction Heval) SCase;
  inversion Heqc'; subst; clear Heqc'.
SCase "E_WhileEnd". apply E_WhileEnd.
  rewrite pe_bexp_correct. assumption.
SCase "E_WhileLoop".
  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_override _ _ H1). eassumption. apply le_n.
Qed.

Corollary pe_com_correct:
 ∀ c pe_st pe_st' c', c / pe_st || c' / pe_st' / SKIP →
 ∀ st st'',
 (c / pe_override st pe_st || st'') ↔
 (∃ st', c' / st || st' ∧ pe_override st' pe_st' = st'').
Proof. intros c pe_st pe_st' c' H st st''. split.
 Case "->". 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.
 Case "<-". 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.
```

4.5 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.

4.5.1 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.

Tactic Notation "block_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "Goto" | Case_aux c "If" | Case_aux c "Assign" ].

Implicit Arguments Goto [[Label]].
Implicit Arguments If [[Label]].
Implicit 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 (AId Y) (ANum 1))
   (Assign X (AMinus (ANum 1) (AId 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 (update st i (aeval st a)) k
  end.

Example keval_example:
```

```
  keval empty_state parity_body
  = (update (update empty_state Y 0) X 1, loop).
Proof. reflexivity. Qed.
```

4.5.2 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 → option (block L).

Definition parity : program parity_label := fun l ⇒
  match l with
  | entry ⇒ Some (Assign X (ANum 0) (Goto loop))
  | loop ⇒ Some (If (BLe (ANum 1) (AId Y)) body done)
  | body ⇒ Some parity_body
  | done ⇒ 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 → L → state → L → Prop :=
  | E_None: ∀ st l,
    p l = None →
    peval p st l st l
  | E_Some: ∀ st l k st' l' st'' l'',
    p l = Some k →
    keval st k = (st', l') →
    peval p st' l' st'' l'' →
    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 ⇒ peval _ _ _ st _).
  eapply E_Some. reflexivity. reflexivity.
  eapply E_Some. reflexivity. reflexivity.
  apply E_None. reflexivity.
  apply functional_extensionality. intros i. rewrite update_same; auto.
Qed.

Tactic Notation "peval_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E_None" | Case_aux c "E_Some" ].
```

4.5.3 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 *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)], *loop*) and ([(X,1)], *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 × L) :=
  match k with
  | Goto l ⇒ Goto (pe_st , l)
  | If b l1 l2 ⇒
    match pe_bexp pe_st b with
    | BTrue ⇒ Goto (pe_st , l1)
    | BFalse ⇒ Goto (pe_st , l2)
    | b' ⇒ If b' (pe_st , l1) (pe_st , l2)
    end
  | Assign i a k ⇒
    match pe_aexp pe_st a with
    | ANum n ⇒ pe_block (pe_add pe_st i n) k
    | a' ⇒ 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 (AId Y) (ANum 1)) (Goto ([(X,1)] , loop)).
Proof. reflexivity. Qed.

Theorem pe_block_correct: ∀ (L:Type) st pe_st k st' pe_st' (l':L),
  keval st (pe_block pe_st k) = (st' , (pe_st' , l')) →
  keval (pe_override st pe_st) k = (pe_override st' pe_st' , l').
Proof. intros. generalize dependent pe_st. generalize dependent st.
  block_cases (induction k as [l | b l1 l2 | i a k]) Case;
    intros st pe_st H.
  Case "Goto". inversion H; reflexivity.
  Case "If".
    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.
  Case "Assign".
```

```
    simpl in *. rewrite pe_aexp_correct.
    destruct (pe_aexp pe_st a); simpl;
      try solve [rewrite pe_override_update_add; apply IHk; apply H];
      solve [rewrite pe_override_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_override 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_override st pe_st) l st'o l' ↔
  pe_peval p st pe_st l st'o l'.
Proof. intros.
  split; [Case "->" | Case "<-"].
  Case "->". intros Heval.
    remember (pe_override st pe_st) as sto.
    generalize dependent pe_st. generalize dependent st.
    peval_cases (induction Heval as
      [ sto l Hlookup | sto l k st'o l' st''o l'' Hlookup Hkeval Heval ])
      SCase; intros st pe_st Heqsto; subst sto.
    SCase "E_None". eapply pe_peval_intro. apply E_None.
      simpl. rewrite Hlookup. reflexivity. reflexivity.
    SCase "E_Some".
      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.
  Case "<-". 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.
```

```
peval_cases (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 ])
  SCase; intros l pe_st Heqpe_st_l;
  inversion Heqpe_st_l; inversion Heqpe_st'_l'; repeat subst.
SCase "E_None". apply E_None. simpl in Hlookup.
  destruct (p l'); [ solve [ inversion Hlookup ] | reflexivity ].
SCase "E_Some".
  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.
```

Chapter 5

Library UseAuto

5.1 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 LibTactics.
```

5.2 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.

5.2.1 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 predicate (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.

5.2.2 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.
```

We can ask `auto` to tell us what proof it came up with, by invoking `info auto` in place of `auto`.

```
Lemma solving_by_apply' : ∀ (P Q : nat→Prop),
  (∀ n, Q n → P n) →
  (∀ n, Q n) →
  P 2.
Proof. info auto. Qed.
```

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.
```

Remark: Again, we can use `info eauto` to see what proof `eauto` comes up with.

5.2.3 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 : ∀ (F F' : Prop),
  F ∧ F' → 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' : ∀ (F F' : 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 : ∀ (P Q R : **nat**→Prop) (F : 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 : ∀ (P Q : **nat**→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 : ∀ (P Q : **nat**→Prop) (F : Prop),
  $(\forall\ n, P\ n) \wedge (\forall\ n, Q\ n) \rightarrow P\ 2$.
Proof. *jauto*. Qed.

5.2.4 Disjunctions

The tactics auto and eauto can handle disjunctions that occur in the goal.

Lemma solving_disj_goal : ∀ (F F' : 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 : ∀ (F F' : Prop),
  F ∨ F' → F' ∨ 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 : ∀ (F1 F2 F3 : Prop),
  ((¬F1 ∧ F3) ∨ (F2 ∧ ¬F3)) →
  (F2 → F1) →
  (F2 → F3) →
  ¬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.

5.2.5 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 : ∀ (f : nat→Prop),
  f 2 → ∃ 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 : ∀ (f g : nat→Prop),
  (∀ x, f x → g x) →
  (∃ a, f a) →
  (∃ a, g a).
Proof.
  auto. eauto. iauto.   jauto. Qed.
```

5.2.6 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 $P \to False$ but that the unfolding of this definition is not performed automatically. Consider the following example.

```
Lemma negation_study_1 : ∀ (P : nat→Prop),
  P 0 → (∀ x, ¬ P x) → 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 : ∀ (P : nat→Prop),
  P 0 → (∀ x, ¬ P x) → False.
Proof. jauto. Qed.
```

We will come back later on to the behavior of proof search with respect to the unfolding of definitions.

5.2.7 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 : ∀ (f g : nat→Prop),
  (∀ x, f x = g x) → 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.

5.3 How Proof Search Works

5.3.1 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 and error. This application produces subgoals. There are two possible cases. If the sugboals 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.
Admitted.
```

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 sugoals. 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 : ∀ (P : nat→Prop),
  P 0 →
  (P 0 → P 1) →
  (P 1 → P 2) →
  (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 $P\ (k\text{-}1)$ 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 : ∀ (P : nat→Prop),
  (P 0) →
  (∀ k, P (k-1) → P k) →
```

(P 4).
`Proof. auto. Qed.`

However, to prove P 5, one needs to call at least `auto` 6.

```
Lemma search_depth_4 : ∀ (P : nat→Prop),
  (P 0) →
  (∀ k, P (k-1) → P k) →
  (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 : ∀ (P : nat→Prop),
  (P 0) →
  (∀ k, P (k-1) → P k) →
  (P 4 ∧ P 4).
Proof. auto. auto 6. Qed.
```

5.3.2 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 : ∀ (P : nat→Prop),
  (P 0) →
  (∀ k, P (k+1) → P k) →
  (∀ k, P (k-1) → P k) →
  (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 H3
```

```
depth=3 apply H3
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 *H3*. The effect of applying *H3* is to replace the goal P 2 with the goal P 1. Then, again, *H3* 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 *H2* at any time. The reason is that `auto` always tries to use the most recently introduced hypothesis first, and *H3* is a more recent hypothesis than *H2* in the goal. So, let's permute the hypotheses *H2* and *H3* and see what happens.

Lemma working_of_auto_2 : $\forall$ (P : **nat**$\rightarrow$`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 H3 H2*. `eauto`. `Qed`.

This time, the output message suggests that the proof search investigates many possibilities. Replacing `debug eauto` with `info eauto`, we observe that the proof that `eauto` comes up with is actually not the simplest one. `apply` *H2*; `apply` *H3*; `apply` *H3*; `apply` *H3*; `exact` *H1* This proof goes through the proof obligation P 3, even though it is not any useful. The following tree drawing describes all the goals that automation has been 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 *H2*, producing the subgoal P 3. To solve it, `eauto` 4 has tried again to apply *H2*, 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 *H3*, 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 *H2* 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 starting with `apply` *H2*.

5.3.3 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 than any number is less than or equal to itself, $\forall$ x, $x \le x$, 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.

5.3.4 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 (×) to the name of a tactic. For example, `apply`× *H* is equivalent to `apply` *H*; *auto_star*, where *auto_star* is a tactic that can be defined as needed. By default, *auto_star* first tries to solve the goal using `auto`, and if this does not succeed then it tries to call *jauto*. Even though *jauto* is strictly stronger than `auto`, it makes sense to call `auto` first: when `auto` succeeds it may save a lot of time, and when `auto` fails to prove the goal, it fails very quickly.

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` [`auto` | *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`×, `destruct`× *H*, *inverts*× *H*, *lets*× *I*: *H x*, *specializes*× *H x*, and so on... There are two notable exceptions. The tactic `auto`× is just another name for the tactic *auto_star*. And the tactic `apply`× *H* calls `eapply` *H* (or the more powerful *applys H* if needed), and then calls *auto_star*. Note that there is no `eapply`× *H* tactic, use `apply`× *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`¬ *H*, `destruct`¬ *H*, `subst`¬, `auto`¬ 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.

5.4 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.

5.4.1 Determinism

`Module` DeterministicImp.
  `Require Import` Imp.

Recall the original proof of the determinism lemma for the IMP language, shown below.

`Theorem` ceval_deterministic: ∀ *c st st1 st2*,

```
  c / st || st1 →
  c / st || st2 →
  st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2.
  (ceval_cases (induction E1) Case); intros st2 E2; inversion E2; subst.
  Case "E_Skip". reflexivity.
  Case "E_Ass". reflexivity.
  Case "E_Seq".
    assert (st' = st'0) as EQ1.
      SCase "Proof of assertion". apply IHE1_1; assumption.
    subst st'0.
    apply IHE1_2. assumption.
  Case "E_IfTrue".
    SCase "b1 evaluates to true".
      apply IHE1. assumption.
    SCase "b1 evaluates to false (contradiction)".
      rewrite H in H5. inversion H5.
  Case "E_IfFalse".
    SCase "b1 evaluates to true (contradiction)".
      rewrite H in H5. inversion H5.
    SCase "b1 evaluates to false".
      apply IHE1. assumption.
  Case "E_WhileEnd".
    SCase "b1 evaluates to true".
      reflexivity.
    SCase "b1 evaluates to false (contradiction)".
      rewrite H in H2. inversion H2.
  Case "E_WhileLoop".
    SCase "b1 evaluates to true (contradiction)".
      rewrite H in H4. inversion H4.
    SCase "b1 evaluates to false".
      assert (st' = st'0) as EQ1.
        SSCase "Proof of assertion". 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': ∀ c st st1 st2,
  c / st || st1 →
```

```
  c / st || st2 →
  st1 = st2.
Proof.
   admit.
Qed.
```

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'': ∀ 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.
  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 invokation 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.
Admitted.
```

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.
```

5.4.2 Preservation for STLC

```
Module PreservationProgressStlc.
  Require Import Stlc.
  Import STLC.
```

Consider the proof of perservation of STLC, shown below. This proof already uses `eauto` through the triple-dot mechanism.

```
Theorem preservation : ∀ t t' T,
  has_type empty t T →
  t ==> t' →
  has_type empty t' T.
Proof with eauto.
  remember (@empty ty) as Gamma.
  intros t t' T HT. generalize dependent t'.
  (has_type_cases (induction HT) Case); intros t' HE; subst Gamma.
  Case "T_Var".
    inversion HE.
  Case "T_Abs".
    inversion HE.
  Case "T_App".
    inversion HE; subst...
    SCase "ST_AppAbs".
      apply substitution_preserves_typing with T11...
      inversion HT1...
  Case "T_True".
    inversion HE.
  Case "T_False".
    inversion HE.
  Case "T_If".
    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*× and *applys*× to call `auto`× after a call to *inverts* or to *applys*. The solution is three lines long.

```
Theorem preservation' : ∀ t t' T,
  has_type empty t T →
  t ==> t' →
  has_type empty t' T.
Proof.
   admit.
Qed.
```

5.4.3 Progress for STLC

Consider the proof of the progress theorem.

```
Theorem progress : ∀ t T,
```

```
  has_type empty t T →
  value t ∨ ∃ t', t ==> t'.
Proof with eauto.
  intros t T Ht.
  remember (@empty ty) as Gamma.
  (has_type_cases (induction Ht) Case); subst Gamma...
  Case "T_Var".
    inversion H.
  Case "T_App".
    right. destruct IHHt1...
    SCase "t1 is a value".
      destruct IHHt2...
      SSCase "t2 is a value".
        inversion H; subst; try solve by inversion.
        ∃ ([x0:=t2]t)...
      SSCase "t2 steps".
       destruct H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
    SCase "t1 steps".
      destruct H as [t1' Hstp]. ∃ (tapp t1' t2)...
  Case "T_If".
    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' : ∀ t T,
  has_type empty t T →
  value t ∨ ∃ t', t ==> t'.
Proof.
   admit.
Qed.
```

```
End PreservationProgressStlc.
```

5.4.4 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 : ∀ t v,
```

```
  normal_form_of t v → ∃ n, v = C n ∧ t || 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.
  ∃ n. split. reflexivity.
  multi_cases (induction Hs) Case; subst.
  Case "multi_refl".
    apply E_Const.
  Case "multi_step".
    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 : ∀ t v,
  t ==>* v → ∀ n, C n = v → t || n.
Proof.
   admit.
Qed.
```

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' : ∀ t v,
  normal_form_of t v → ∃ n, v = C n ∧ t || n.
Proof.
   admit.
Qed.
```

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 looses information when applied to a predicate whose arguments are not reduced to variables, such as t ==>* $(C\ n)$. 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'' : ∀ t v,
```

```
  normal_form_of t v → ∃ n, v = C n ∧ t || n.
Proof.
   admit.
Qed.
End Semantics.
```

5.4.5 Preservation for STLCRef

```
Module PreservationProgressReferences.
  Require Import References.
  Import STLCRef.
  Hint Resolve store_weakening extends_refl.
```

The proof of preservation for *STLCRef* can be found in chapter *References*. It contains 58 lines (not counting the labelling of cases). 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 : ∀ ST t t' T st st',
  has_type empty ST t T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
    (extends ST' ST ∧
     has_type empty ST' t' T ∧
     store_well_typed ST' st').
Proof.
  remember (@empty ty) as Gamma. introv Ht. gen t'.
  (has_type_cases (induction Ht) Case); introv HST Hstep;

   subst Gamma; inverts Hstep; eauto.
  Case "T_App".
  SCase "ST_AppAbs".
  ∃ ST. inverts Ht1. splits×. applys× substitution_preserves_typing.
  SCase "ST_App1".
  forwards: IHHt1. eauto. eauto. eauto.
  jauto_set_hyps; intros.
  jauto_set_goal; intros.
  eauto. eauto. eauto.
  SCase "ST_App2".
  forwards*: IHHt2.
  forwards*: IHHt.
```

```
  forwards*: IHHt.
  forwards*: IHHt1.
  forwards*: IHHt2.
  forwards*: IHHt1.
  Case "T_Ref".
  SCase "ST_RefValue".
  ∃ (snoc ST T1). inverts keep HST. splits.
    apply extends_snoc.
    applys_eq T_Loc 1.
      rewrite length_snoc. omega.
    unfold store_Tlookup. rewrite ← H. rewrite× nth_eq_snoc.
    apply× store_well_typed_snoc.
  forwards*: IHHt.
  Case "T_Deref".
  SCase "ST_DerefLoc".
  ∃ ST. splits×.
  lets [_ Hsty]: HST.
  applys_eq× Hsty 1.
  inverts× Ht.
  forwards*: IHHt.
  Case "T_Assign".
  SCase "ST_Assign".
  ∃ ST. splits×. applys× assign_pres_store_typing. inverts× 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_snoc*, which takes the form *nth* (*length* l) (*snoc* l x) $d = x$. 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 occuring 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_snoc* 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_snoc' : ∀ (A : Type) (l : list A) (x d : A) (n : nat),
  n = length l → nth n (snoc l x) d = x.
Proof. intros. subst. apply nth_eq_snoc. Qed.
```

The proof case for *ref* from the preservation theorem then becomes much easier to prove, because `rewrite` *nth_eq_snoc'* now succeeds.

```
Lemma preservation_ref : ∀ (st:store) (ST : store_ty) T1,
```

```
  length ST = length st →
  TRef T1 = TRef (store_Tlookup (length st) (snoc ST T1)).
Proof.
  intros. dup.
  unfold store_Tlookup. rewrite× nth_eq_snoc'.
  fequal. symmetry. apply× nth_eq_snoc'.
Qed.
```

The optimized proof of preservation is summarized next.

```
Theorem preservation' : ∀ ST t t' T st st',
  has_type empty ST t T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
    (extends ST' ST ∧
     has_type empty ST' t' T ∧
     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.
  ∃ ST. inverts Ht1. splits×. applys× substitution_preserves_typing.
  forwards*: IHHt1.
  forwards*: IHHt2.
  forwards*: IHHt.
  forwards*: IHHt.
  forwards*: IHHt1.
  forwards*: IHHt2.
  forwards*: IHHt1.
  ∃ (snoc ST T1). inverts keep HST. splits.
    apply extends_snoc.
    applys_eq T_Loc 1.
      rewrite length_snoc. omega.
      unfold store_Tlookup. rewrite× nth_eq_snoc'.
    apply× store_well_typed_snoc.
  forwards*: IHHt.
  ∃ ST. splits×. lets [_ Hsty]: HST.
   applys_eq× Hsty 1. inverts× Ht.
  forwards*: IHHt.
  ∃ ST. splits×. applys× assign_pres_store_typing. inverts× Ht1.
  forwards*: IHHt1.
  forwards*: IHHt2.
Qed.
```

5.4.6 Progress for STLCRef

The proof of progress for *STLCRef* can be found in chapter *References*. It contains 53 lines and the optimized proof script is, here again, half the length.

```
Theorem progress : ∀ ST t T st,
  has_type empty ST t T →
  store_well_typed ST st →
  (value t ∨ ∃ t', ∃ st', t / st ==> t' / st').
Proof.
  introv Ht HST. remember (@empty ty) as Gamma.
  induction Ht; subst Gamma; tryfalse; try solve [left*].
  right. destruct× IHHt1 as [K||].
    inverts K; inverts Ht1.
     destruct× IHHt2.
  right. destruct× IHHt as [K||].
    inverts K; try solve [inverts Ht]. eauto.
  right. destruct× IHHt as [K||].
    inverts K; try solve [inverts Ht]. eauto.
  right. destruct× 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.
```

5.4.7 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 (extend 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` *I* with *lets I*: *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 (extend empty x S1) s2 T2.
Proof.
   admit.
Qed.
```

The lemma *substitution_preserves_typing* has already been used to illustrate the working of *lets* and *applys* in chapter *UseTactics*. Optimize further this proof using automation (with the star symbol), and using the tactic *cases_if'*. The solution is 33 lines, including the *Case* instructions (21 lines without them).

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
  has_type (extend Gamma x U) t S →
  has_type empty v U →
  has_type Gamma ([x:=v]t) S.
Proof.
   admit.
Qed.

End SubtypingInversion.
```

5.5 Advanced Topics in Proof Search

5.5.1 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 predicate on natural numbers. This predicate 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 $\forall\ n\ m,\ P\ m \to m \neq 0 \to P\ n$, then `eauto` works. However, with $\forall\ n\ m,\ m \neq 0 \to P\ m \to P\ n$, 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.
Admitted.
```

It is very important to understand that the hypothesis $\forall\ n\ m,\ P\ m \to m \neq 0 \to P\ n$ is eauto-friendly, whereas $\forall\ n\ m,\ m \neq 0 \to P\ m \to P\ n$ 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 $P\ m$.

5.5.2 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 predicate 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 $True \to myFact$.

```
Lemma demo_hint_unfold_context_1 :
  (True → myFact) → P 3.
Proof.
  intros.
  auto.   unfold myFact in *. auto. Qed.
```

There is actually one exception to the previous rule: a constant occuring in an hypothesis is automatically unfolded if the hypothesis can be directly applied to the current goal. For example, `auto` can prove $myFact \to P\ 3$, as illustrated below.

```
Lemma demo_hint_unfold_context_2 :
  myFact → P 3.
Proof. auto. Qed.
```

5.5.3 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 : ∀ x,
```

$(x \le 3) \to \neg (x > 3)$.

Equivalently, one could state that a number greater than three is not less than or equal to 3.

Parameter *gt_not_le* : $\forall\ x$,
$(x > 3) \to \neg (x \le 3)$.

In fact, both statements are equivalent to a third one stating that $x \le 3$ and $x > 3$ are contradictory, in the sense that they imply *False*.

Parameter *le_gt_false* : $\forall\ x$,
$(x \le 3) \to (x > 3) \to$ **False**.

The following investigation aim at figuring out which of the three statments 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 $\exists\ x,\ x \le 3 \land x > 3$ is absurd.

Hint Resolve *le_not_gt*.

Lemma demo_auto_absurd_1 :
$(\exists\ x\ ,\ x \le 3 \land x > 3) \to$ **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 :
$(\exists\ x\ ,\ x \le 3 \land x > 3) \to$ **False**.
Proof.
dup.

intros. *jauto_set*. eauto.

jauto.
Qed.

In summary, a lemma of the form *H1* $\to$ *H2* $\to$ *False* is a much more effective hint than *H1* $\to \neg$ *H2*, even though the two statments 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 : ∀ x,
  (x ≤ 3) → (x > 3) → 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).

5.5.4 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 $\rightarrow$ *subtype* T U $\rightarrow$ *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, *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. Admitted.
```

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 sugoal, *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* $?T\ U \vdash \_ \Rightarrow$ `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: *subtype* $S\ ?T \vdash \_$. It indices 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: *subtype* $S\ ?T \vdash \_$ indicates that we do not impose further condition on how the proof obligation has to look like.
- The branch $\Rightarrow$ `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 : $\forall\ T1\ T2\ T3\ T4$,
  subtype $T1\ T2 \rightarrow$ *subtype* $T2\ T3 \rightarrow$ *subtype* $T3\ T4 \rightarrow$ *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 : $\forall\ S\ T$,
  subtype $S\ T$.
`Proof`.
  `intros`. `eauto`. *Admitted*.

5.6 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.

5.6.1 Omega

The tactic `omega` supports natural numbers (type *nat*) as well as integers (type *Z*, available by including the module *ZArith*). It supports addition, substraction, 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 proof *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.
```

5.6.2 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 Z) and not natural numbers (type nat). 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.
```

5.6.3 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 :
   ∀ (f : nat→nat→nat) (g h : nat→nat) (x y z : nat),
   f (g x) (g y) = z →
   2 = g x →
   g y = h z →
   f 2 (h z) = z.
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 :
   ∀ (f : nat→nat→nat) (g h : nat→nat) (x y z : nat),
   (∀ a, g a = h a) →
   f (g x) (g y) = z →
   g x = 2 →
   f 2 (h y) = z.
Proof. congruence. Qed.
```

Next is an example where `congruence` is very useful.

```
Lemma congruence_demo_4 : ∀ (f g : nat→nat),
```

$(\forall\ a, f\ \ a = g\ \ a) \rightarrow$
$f\ (g\ (g\ 2)) = g\ (f\ (f\ 2))$.
```
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 :
$\forall\ (f\ g\ h : \textbf{nat}\rightarrow\textbf{nat})\ (x : \textbf{nat})$,
$(\forall\ a, f\ \ a = h\ \ a) \rightarrow$
$g\ x = f\ x \rightarrow$
$g\ x \neq h\ x \rightarrow$
False.
```
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.

5.7 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.

Chapter 6

Library UseTactics

6.1 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 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 specification 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.

6.2 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`.

6.2.1 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 where 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: ∀ c st st1 st2,
  c / st || st1 →
  c / st || st2 →
  st1 = st2.
Proof.
  introv E1 E2. Admitted.
```

When there is no hypothesis to be named, one can call *introv* without any argument.

```
Theorem dist_exists_or : ∀ (X:Type) (P Q : X → Prop),
  (∃ x, P x ∨ Q x) ↔ (∃ x, P x) ∨ (∃ x, Q x).
Proof.
  introv. Admitted.
```

The tactic *introv* also applies to statements in which ∀ and → are interleaved.

```
Theorem ceval_deterministic': ∀ c st st1,
  (c / st || st1) → ∀ st2, (c / st || st2) → st1 = st2.
Proof.
  introv E1 E2. Admitted.
```

Like the arguments of `intros`, the arguments of *introv* can be structured patterns.

Theorem exists_impl: forall X (P : X -> Prop) (Q : Prop) (R : Prop), (forall x, P x -> Q) -> ((exists x : X, P x) -> 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.

6.2.2 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 intented 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: ∀ c,
  cequiv (SKIP; c) c.
Proof.
  introv. split; intros H.
  dup.   inversion H. subst. inversion H2. subst. assumption.
  inverts H. inverts H2. assumption.
Admitted.
```

A slightly more interesting example appears next.

```
Theorem ceval_deterministic: ∀ c st st1 st2,
  c / st || st1 →
  c / st || st2 →
  st1 = st2.
Proof.
  introv E1 E2. generalize dependent st2.
  (ceval_cases (induction E1) Case); intros st2 E2.
  admit. admit.   dup. inversion E2. subst. admit.
   inverts E2. admit.
Admitted.
```

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 *introv*.

```
Theorem ceval_deterministic': ∀ c st st1 st2,
```

```
  c / st || st1 →
  c / st || st2 →
  st1 = st2.
Proof.
  introv E1 E2. generalize dependent st2.
  (ceval_cases (induction E1) Case); intros st2 E2;
    inverts E2 as.
  Case "E_Skip". reflexivity.
  Case "E_Ass".
   subst n.
    reflexivity.
  Case "E_Seq".
   intros st3 Red1 Red2.
    assert (st' = st3) as EQ1.
      SCase "Proof of assertion". apply IHE1_1; assumption.
    subst st3.
    apply IHE1_2. assumption.
  Case "E_IfTrue".
    SCase "b1 evaluates to true".
   intros.
      apply IHE1. assumption.
    SCase "b1 evaluates to false (contradiction)".
   intros.
      rewrite H in H5. inversion H5.
Admitted.
```

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`; *introv H1 H2 H3*. An example follows.

```
Theorem skip_left': ∀ c,
  cequiv (SKIP; c) c.
Proof.
  introv. split; intros H.
  inverts H as U V.    inverts U. assumption.
Admitted.
```

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 :
  ¬ ∃ 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.

6.2.3 The tactics *cases* and *cases_if*

```
Module CasesExample.
  Require Import Stlc.
  Import STLC.
```

As you probably have learned, the tactic `destruct` can be used to perform a case analysis. However, this tactic sometimes destroys useful information. The tactic *remember* is intended to introduce an equality that avoids `destruct` loosing such useful information. The tactic *cases* provided by *LibTactics* packages *remember* and `destruct` together in order to shorten proof scripts.

The tactic *cases E* behaves like *remember E* `as` *x*; `destruct` *x*, only with the difference that it generates the symmetric of the equality produced by *remember*. For example, *cases* would produce the equality *beq_id k1 k2* = *true* rather than the equality *true* = *beq_id k1 k2*. Indeed, the former reads much more naturally than the latter. Moreover, the syntax *cases E* `as` *H* allows to involve the *cases* tactic by specifying how to name the equality generated.

Remark: *cases* is quite similar to *case_eq*. For the sake of compatibility with *remember* and *case_eq*, the library "LibTactics" provides a tactic called *cases'* that generates exactly the same equalities as *remember* or *case_eq* would, i.e., producing an equality in the form *true* = *beq_id k1 k2* rather than *beq_id k1 k2* = *true*. The following examples illustrate the behavior of the tactic *cases' E* `as` *H*.

```
Theorem update_same : ∀ x1 k1 k2 (f : state),
  f k1 = x1 →
  (update f k1 x1) k2 = f k2.
Proof.
  intros x1 k1 k2 f Heq.
  unfold update. subst.
  dup.

  remember (beq_id k1 k2) as b. destruct b.
    apply beq_id_eq in Heqb. subst. reflexivity.
    reflexivity.

  cases' (beq_id k1 k2) as E.
    apply beq_id_eq in E. subst. reflexivity.
    reflexivity.
Qed.
```

The tactic *cases_if* is a tactic that allows performing a case analysis without having to explicitly specify which value the case analysis should be upon. More precisely, the tactic *cases_if* looks in the goal or in the context for an expression of the form `if` *E* `then` .. `else` .., and it invokes *cases E*. Remark: if there are several possibilities, *cases_if* only consider the first one.

The tactic *cases_if* thus saves the need to copy-past an expression that occurs in the current proof obligation, leading to shorter and more robust proof scripts.

Here again, for compatibility reasons, the library provides a tactic called *cases_if'*. Moreover, one may write *cases_if* `as` *H* or *cases_if'* `as` *H* for specifying the name to use for the generated equality.

```
Theorem update_same' : ∀ x1 k1 k2 (f : state),
  f k1 = x1 →
  (update f k1 x1) k2 = f k2.
Proof.
  intros x1 k1 k2 f Heq.
  unfold update. subst.
```

```
  cases_if' as E.
    apply beq_id_eq in E. subst. reflexivity.
    reflexivity.
Qed.
End CasesExample.
```

6.3 Tactics for n-ary connectives

Because Coq encodes conjunctions and disjunctions using binary constructors ∧ and ∨, working with a conjunction or a disjunction of N facts can sometimes be quite cumbursome. For this reason, "LibTactics" provides tactics offering direct support for n-ary conjunctions and disjunctions. It also provides direct support for n-ary existententials.

This section presents the following tactics:

- *splits* for decomposing n-ary conjunctions,
- *branch* for decomposing n-ary disjunctions,
- ∃ for proving n-ary existentials.

```
Module NaryExamples.
  Require Import References SfLib.
  Import STLCRef.
```

6.3.1 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* ∧ *G2* ∧ *G3* into the three subgoals *G1*, *G2* and *G3*.

```
Lemma demo_splits : ∀ n m,
  n > 0 ∧ n < m ∧ m < n+10 ∧ m ≠ 3.
Proof.
  intros. splits.
Admitted.
```

6.3.2 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* ∨ *G2* ∨ *G3*, the tactic *branch* 2 leaves only *G2* as subgoal. The following example illustrates the behavior of the *branch* tactic.

```
Lemma demo_branch : ∀ n m,
  n < m ∨ n = m ∨ 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.
```

6.3.3 The tactic ∃

The library "LibTactics" introduces a notation for n-ary existentials. For example, one can write ∃ *x* *y* *z*, *H* instead of ∃ *x*, ∃ *y*, ∃ *z*, *H*. Similarly, the library provides a n-ary tactic ∃ *a* *b* *c*, which is a shorthand for ∃ *a*; ∃ *b*; ∃ *c*. The following example illustrates both the notation and the tactic for dealing with n-ary existentials.

```
Theorem progress : ∀ ST t T st,
  has_type empty ST t T →
  store_well_typed ST st →
  value t ∨ ∃ t' st', t / st ==> t' / st'.
Proof with eauto.
  intros ST t T st Ht HST. remember (@empty ty) as Gamma.
  (has_type_cases (induction Ht) Case); subst; try solve by inversion...
  Case "T_App".
    right. destruct IHHt1 as [Ht1p | Ht1p]...
    SCase "t1 is a value".
      inversion Ht1p; subst; try solve by inversion.
      destruct IHHt2 as [Ht2p | Ht2p]...
      SSCase "t2 steps".
        inversion Ht2p as [t2' [st' Hstep]].
        ∃ (tapp (tabs x T t) t2') st'...
Admitted.
```

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.
```

6.4 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,
- *substs* for improving the `subst` tactic,
- *fequals* for improving the `f_equal` tactic,
- *applys_eq* for proving $P\ x\ y$ using an hypothesis $P\ x\ z$, automatically producing an equality $y = z$ as subgoal.

Module EqualityExamples.

6.4.1 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 : ∀ n m : nat,
  (0 + n) × m = n × 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' : ∀ n m : nat,
  (0 + n) × m = n × 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'' : ∀ u v w x y z: nat,
  (u + v) × (S (w × x + y)) = z.
Proof.
```

```
  intros. asserts_rewrite (∀ a b, a*(S b) = a×b+a).
Admitted.
```

6.4.2 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 : ∀ x y (f:nat→nat),
  x = f x → y = x → y = f x.
Proof.
  intros. substs.   assumption.
Qed.
```

6.4.3 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.
Admitted.
```

6.4.4 The tactic *applys_eq*

The tactic *applys_eq* is a variant of `eapply` that introduces equalities for subterms that do not unify. For example, assume the goal is the proposition $P\ x\ y$ and assume we have the assumption H asserting that $P\ x\ z$ 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 $P\ x\ z$, but this would require copy-pasting the values of y and z. With the tactic *applys_eq*, we can call *applys_eq* H 1, which proves the goal and leaves only the subgoal $y = z$. The value 1 given as argument to *applys_eq* indicates that we want an equality to be introduced for the first argument of $P\ x\ y$ counting from the right. The three following examples illustrate the behavior of a call to *applys_eq* H 1, a call to *applys_eq* H 2, and a call to *applys_eq* H 1 2.

```
Axiom big_expression_using : nat→nat.

Lemma demo_applys_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.
  applys_eq H 1.
    admit. Qed.
```

If the mismatch was on the first argument of P instead of the second, we would have written *applys_eq H 2*. Recall that the occurences are counted from the right.

```
Lemma demo_applys_eq_2 : ∀ (P:nat→nat→Prop) x y z,
  P (big_expression_using z) x →
  P (big_expression_using y) x.
Proof.
  introv H. applys_eq H 2.
Admitted.
```

When we have a mismatch on two arguments, we want to produce two equalities. To achieve this, we may call *applys_eq H 1 2*. More generally, the tactic *applys_eq* expects a lemma and a sequence of natural numbers as arguments.

```
Lemma demo_applys_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. applys_eq H 1 2.
Admitted.

End EqualityExamples.
```

6.5 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.

6.5.1 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 UnfoldsExample.
```

6.5.2 The tactics *false* and *tryfalse*

The tactic *false* can be used to replace any goal with *False*. In short, it is a shorthand for `apply` *ex_falso_quodlibet*. 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 $x = true$ and $x = false$.

```
Lemma demo_false :
  ∀ n, S n = 1 → 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 :
  (∀ n, n < 0 → False) → (3 < 0) → 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 :
```

```
  ∀ n, S n = 1 → n = 0.
Proof.
  intros. destruct n; tryfalse. reflexivity.
Qed.
```

6.5.3 The tactic *gen*

The tactic *gen* is a shortand for `generalize dependent` that accepts several arguments at once. An invokation of this tactic takes the form *gen x y z*.

```
Module GenExample.
  Require Import Stlc.
  Import STLC.

Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (extend Gamma x U) t S →
     has_type empty v U →
     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.
Qed.

End GenExample.
```

6.5.4 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.
Admitted.
```

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).
Admitted.
```

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 invokations 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.
  (ceval_cases (induction E1) Case); introv E2; inverts E2 as.
  Case "E_Skip". reflexivity.
  Case "E_Ass".
```

```
    subst n.
    reflexivity.
  Case "E_Seq".
    intros st3 Red1 Red2.
    assert (st' = st3) as EQ1.
      SCase "Proof of assertion".
   eapply IH. eapply E1_1. eapply Red1.
    subst st3.
   eapply IH. eapply E1_2. eapply Red2.
Admitted.

End SkipExample.
```

6.5.5 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: ∀ c st st1 st2,
  c / st || st1 →
  c / st || st2 →
  st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2.
  (ceval_cases (induction E1) Case); intros st2 E2; inverts E2.
  admit. admit.    sort. Admitted.

End SortExamples.
```

6.6 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 existentential variables, and facts whose instantiation is not provided are turned into subgoals.

Remark: this instantion 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*).

6.6.1 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* *I*: *E0* *E1* .. *EN*, for building an hypothesis named *I* 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 the to instantiate the lemma with the arguments provided.

```
Module ExamplesLets.
  Require Import Sub.

Axiom typing_inversion_var : ∀ (G:context) (x:id) (T:ty),
  has_type G (tvar x) T →
  ∃ S, G x = Some S ∧ 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 : ∀ (G:context) (x:id) (T:ty),
  has_type G (tvar x) T → 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.

Qed.
```

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 : ∀ (G:context) (x:id) (T:ty), True.
Proof.
  intros G x T.
  lets (S & Eq & Sub): typing_inversion_var G x T ___.
Admitted.
```

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 : ∀ (x:id), True.
Proof.
  intros x.
  lets (S & Eq & Sub): typing_inversion_var x ___.
Admitted.
```

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 ___.
Admitted.
```

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.
Admitted.
```

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 writting *lets* K: H __ 3.

```
Lemma demo_lets_underscore :
  (∀ n m, n ≤ m → n < m+1) → True.
Proof.
  intros H.
  lets K: H 3.     clear K.
  lets K: H __ 3.     clear K.
```

Admitted.

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.

6.6.2 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.

- *applys* allows building a lemma using the advanced instantion

mode of *lets*, and then apply that lemma right away. So, *applys 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 *applys* appear further on. Several examples of use of *forwards* can be found in the tutorial chapter *UseAuto*.

6.6.3 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 *applys* is used instead of `apply`. The proof also contains some holes that you need to fill in as an exercise.

Lemma substitution_preserves_typing : ∀ *Gamma x U v t S*,
has_type (extend *Gamma x U*) *t S* →
has_type empty *v U* →
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.
(t_cases (induction t) Case); intros; simpl.
Case "tvar".
  rename i into y.
 lets (T&Hctx&Hsub): typing_inversion_var Htypt.
  unfold extend in Hctx.
  remember (beq_id x y) as e. destruct e...
  SCase "x=y".
    apply beq_id_eq in Heqe. subst.
    inversion Hctx; subst. clear Hctx.
    apply context_invariance with empty...
    intros x Hcontra.
      lets [T' HT']: free_in_context S empty Hcontra...
      inversion HT'.
Case "tapp".
 admit.
Case "tabs".
  rename i into y. rename t into T1.
 lets (T2&Hsub&Htypt2): typing_inversion_abs Htypt.
 applys T_Sub (TArrow T1 T2)...
   apply T_Abs...
  remember (beq_id x y) as e. destruct e.
  SCase "x=y".
    eapply context_invariance...
    apply beq_id_eq in Heqe. subst.
    intros x Hafi. unfold extend.
    destruct (beq_id y x)...
  SCase "x<>y".
    apply IHt. eapply context_invariance...
    intros z Hafi. unfold extend.
    remember (beq_id y z) as e0. destruct e0...
    apply beq_id_eq in Heqe0. subst.
    rewrite ← Heqe...
Case "ttrue".
  lets: typing_inversion_true Htypt...
Case "tfalse".
  lets: typing_inversion_false Htypt...
Case "tif".
  lets (Htyp1&Htyp2&Htyp3): typing_inversion_if Htypt...
Case "tunit".
```

lets: typing_inversion_unit *Htypt*...

Qed.

End ExamplesInstantiations.

6.7 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 ∃ to deal with n-ary constructs.
- *asserts_rewrite*, *cuts_rewrite*, *substs* and *fequals* help working with equalities.
- *lets*, *forwards*, *specializes* and *applys* provide means of very conveniently instantiating lemmas.
- *applys_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 lastest version from: http://www.chargueraud.org/softs/tlc/ .

Chapter 7

Library LibTactics

7.1 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, hidding the fact that the underlying implementation is based on binary predicates.
- 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 mecanism. 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.
```

7.2 Additional notations for Coq

7.2.1 N-ary Existentials

∃ *T1* ... *TN*, *P* is a shorthand for ∃ *T1*, ..., ∃ *TN*, *P*. Note that *Coq.Program.Syntax* already defines exists for arity up to 4.

```
Notation "'exists' x1 ',' P" :=
  (∃ x1 , P)
  (at level 200, x1 ident,
   right associativity) : type_scope.
Notation "'exists' x1 x2 ',' P" :=
  (∃ x1 , ∃ x2 , P)
  (at level 200, x1 ident, x2 ident,
   right associativity) : type_scope.
Notation "'exists' x1 x2 x3 ',' P" :=
  (∃ x1 , ∃ x2 , ∃ x3 , P)
  (at level 200, x1 ident, x2 ident, x3 ident,
   right associativity) : type_scope.
Notation "'exists' x1 x2 x3 x4 ',' P" :=
  (∃ x1 , ∃ x2 , ∃ x3 , ∃ x4 , P)
  (at level 200, x1 ident, x2 ident, x3 ident, x4 ident,
   right associativity) : type_scope.
Notation "'exists' x1 x2 x3 x4 x5 ',' P" :=
  (∃ x1 , ∃ x2 , ∃ x3 , ∃ x4 , ∃ x5 , P)
  (at level 200, x1 ident, x2 ident, x3 ident, x4 ident, x5 ident,
   right associativity) : type_scope.
Notation "'exists' x1 x2 x3 x4 x5 x6 ',' P" :=
  (∃ x1 , ∃ x2 , ∃ x3 , ∃ x4 , ∃ x5 , ∃ x6 , P)
```

```
  (at level 200, x1 ident, x2 ident, x3 ident, x4 ident, x5 ident,
   x6 ident,
   right associativity) : type_scope.
Notation "'exists' x1 x2 x3 x4 x5 x6 x7 ',' P" :=
  (∃ x1 , ∃ x2 , ∃ x3 , ∃ x4 , ∃ x5 , ∃ x6 ,
   ∃ x7 , P)
  (at level 200, x1 ident, x2 ident, x3 ident, x4 ident, x5 ident,
   x6 ident, x7 ident,
   right associativity) : type_scope.
Notation "'exists' x1 x2 x3 x4 x5 x6 x7 x8 ',' P" :=
  (∃ x1 , ∃ x2 , ∃ x3 , ∃ x4 , ∃ x5 , ∃ x6 ,
   ∃ x7 , ∃ x8 , P)
  (at level 200, x1 ident, x2 ident, x3 ident, x4 ident, x5 ident,
   x6 ident, x7 ident, x8 ident,
   right associativity) : type_scope.
Notation "'exists' x1 x2 x3 x4 x5 x6 x7 x8 x9 ',' P" :=
  (∃ x1 , ∃ x2 , ∃ x3 , ∃ x4 , ∃ x5 , ∃ x6 ,
   ∃ x7 , ∃ x8 , ∃ x9 , P)
  (at level 200, x1 ident, x2 ident, x3 ident, x4 ident, x5 ident,
   x6 ident, x7 ident, x8 ident, x9 ident,
   right associativity) : type_scope.
Notation "'exists' x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 ',' P" :=
  (∃ x1 , ∃ x2 , ∃ x3 , ∃ x4 , ∃ x5 , ∃ x6 ,
   ∃ x7 , ∃ x8 , ∃ x9 , ∃ x10 , P)
  (at level 200, x1 ident, x2 ident, x3 ident, x4 ident, x5 ident,
   x6 ident, x7 ident, x8 ident, x9 ident, x10 ident,
   right associativity) : type_scope.
```

7.3 Tools for programming with Ltac

7.3.1 Identity continuation

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

7.3.2 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.
```

7.3.3 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 invokation, 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.
```

7.3.4 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.
```

7.3.5 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*. If 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.
```

7.3.6 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*.

```
Require Import List.

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.
```

7.3.7 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`.

```
Definition ltac_database (D:Boxer) (T:Boxer) (A:Boxer) := True.

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. 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 ].
```

7.3.8 On-the-fly removal of hypotheses

In a list of arguments » *H1 H2 .. HN* passed to a tactic such as *lets* or *applys* 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.
```

7.3.9 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 ⇒ 0%nat
  | BinInt.Zpos p ⇒ BinPos.nat_of_P p
  | BinInt.Zneg p ⇒ 0%nat
  end.

Ltac nat_from_number N :=
  match type of N with
  | nat ⇒ constr:(N)
  | BinInt.Z ⇒ 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 ⇒ pattern E at 1
  | 2 ⇒ pattern E at 2
  | 3 ⇒ pattern E at 3
  | 4 ⇒ pattern E at 4
  | 5 ⇒ pattern E at 5
  | 6 ⇒ pattern E at 6
  | 7 ⇒ pattern E at 7
  | 8 ⇒ pattern E at 8
  end.

Tactic Notation "ltac_pattern" constr(E) "at" constr(K) "in" hyp(H) :=
  match nat_from_number K with
  | 1 ⇒ pattern E at 1 in H
  | 2 ⇒ pattern E at 2 in H
```

```
| 3 ⇒ pattern E at 3 in H
| 4 ⇒ pattern E at 4 in H
| 5 ⇒ pattern E at 5 in H
| 6 ⇒ pattern E at 6 in H
| 7 ⇒ pattern E at 7 in H
| 8 ⇒ pattern E at 8 in H
end.
```

7.3.10 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 : ∀ P, P → P → P.
Proof. auto. Qed.
```

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

```
Tactic Notation "dup" constr(N) :=
  dup_tactic N.
Tactic Notation "dup" :=
  dup 2.
```

7.3.11 Check no evar in goal

```
Ltac check_noevar M :=
  match M with M ⇒ idtac end.
```

```
Ltac check_noevar_hyp H :=
  let T := type of H in
  match type of H with T ⇒ idtac end.
```

```
Ltac check_noevar_goal :=
  match goal with ⊢ ?G ⇒ match G with G ⇒ idtac end end.
```

7.3.12 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.
```

7.3.13 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.
```

7.3.14 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 _ ⇒ constr:(P)
  | ?P ⇒ 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 ⇒ constr:((X1 X2 X3 X4 X5 X6,X))
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X ⇒ constr:((X1 X2 X3 X4 X5,X))
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X ⇒ constr:((X1 X2 X3 X4,X))
  | ?X1 ?X2 ?X3 ?X4 ?X ⇒ constr:((X1 X2 X3,X))
  | ?X1 ?X2 ?X3 ?X ⇒ constr:((X1 X2,X))
  | ?X1 ?X2 ?X ⇒ constr:((X1,X))
  | ?X1 ?X ⇒ constr:((X1,X))
  end.
```

7.3.15 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 ⊢ ?P _ ⇒ set (p:=P) end;
  Tac; unfold p; clear p.
```

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

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

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

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

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

7.3.16 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, arguments at next level).
```

7.4 Backward and forward chaining

7.4.1 Application

rapply 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 *applys_N T*, where *N* is a natural number, provides a more efficient way of using *applys T*. It avoids trying out all possible arities, by specifying explicitely 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.
```

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

constructors calls `constructor` or `econstructor`.

```
Tactic Notation "constructors" :=
```

```
  first [ constructor | econstructor ]; unfold eq'.
```

7.4.2 Assertions

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 .. xN = D y1 .. 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 ].
```

tryfalse `by` *tac* / is that same as *tryfalse* except that it tries to solve the goal using tactic *tac* if `assumption` and `discriminate` do not apply. It is equivalent to `try solve` \[*false*; *tac* \]. Example: *tryfalse* `by congruence`/

```
Tactic Notation "tryfalse" "by" tactic(tac) "/" :=
  try solve [ false; instantiate; tac ].
```

false T tries *false*; `apply` *T*, or otherwise adds *T* as an assumption and calls *false*.

```
Tactic Notation "false" constr(T) "by" tactic(tac) "/" :=
  false_goal; first
    [ first [ apply T | eapply T | rapply T]; instantiate; tac
    | let H := fresh in lets_base H T;
      first [ discriminate H
            | false; instantiate; tac ] ].

Tactic Notation "false" constr(T) :=
  false T by idtac/.
```

false_invert proves any goal provided there is at least one hypothesis *H* in the context that can be proved absurd by calling `inversion` *H*.

```
Ltac false_invert_tactic :=
  match goal with H:_ ⊢ _ ⇒
```

```
    solve [ inversion H
          | clear H; false_invert_tactic
          | fail 2 ] end.
Tactic Notation "false_invert" :=
  false_invert_tactic.
```

tryfalse_invert tries to prove the goal using *false* or *false_invert*, and leaves the goal unchanged if it does not succeed.

```
Tactic Notation "tryfalse_invert" :=
  try solve [ false | false_invert ].
```

asserts H: T is another syntax for assert (H : T), which also works with introduction patterns. For instance, one can write: *asserts* \[x P\] ($\exists$ n, n = 3), or *asserts* \[H|H\] (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 .. HN: T is a shorthand for *asserts* \[*H1* \[*H2* \[.. *HN*\]\]\]\: T].

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

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

```
Tactic Notation "asserts" ":" constr(T) :=
  let H := fresh 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) ":" constr(T) :=
  cut (T); [ intros I | idtac ].
```

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

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

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

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

7.4.3 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. $\forall\ x,\ P\ x \to \forall\ y\ z,\ Q\ x\ y\ z \to R\ z$.

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 often saves a lot of time to give only the dependent variables, (here x, y and z), and have the hypotheses generated as subgoals. In this "Vars" mode, only variables are to be provided. For instance, lemma E applied to 3 and 4 is a term of type $\forall\ z,\ Q\ 3\ 4\ z \to R\ z$, and $P\ 3$ is a new subgoal. It is possible to use wildcards to introduce existential variables.

However, there are situations where some of the hypotheses already exists, and it saves time to instantiate the lemma E using the hypotheses. For instance, suppose F is a term of type P 2. Then the application of E to F in this "Hyps" mode is a term of type $\forall$ y z, Q 2 y $z \rightarrow R$ z. Each wildcard use will generate an assertion instead, for instance if G has type Q 2 3 4, then the application of E to a wildcard and to G in mode-h is a term of type R 4, and P 2 is a new subgoal.

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

The general syntax is *tactic* (» *E1* .. *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* .. *EN* as short for *tactic* (»*Hnts E1* .. *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.

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

Ltac boxerlist_next_type vs :=
  match vs with
  | nil ⇒ constr:(ltac_wild)
```

```
  | (boxer ltac_wild)::?vs' ⇒ boxerlist_next_type vs'
  | (boxer ltac_wilds)::_ ⇒ constr:(ltac_wild)
  | (@boxer ?T _)::_ ⇒ constr:(T)
  end.

Ltac build_app_hnts t vs final :=
  let rec go t vs :=
    match vs with
    | nil ⇒ first [ final t | fail 1 ]
    | (boxer ltac_wilds)::_ ⇒ first [ build_app_alls t final | fail 1 ]
    | (boxer ?v)::?vs' ⇒
      let cont t' := go t' vs in
      let cont' t' := go t' vs' in
      let T := type of t in
      let T := eval hnf in T in
      match v with
      | ltac_wild ⇒
         first [ let U := boxerlist_next_type vs' in
           match U with
           | ltac_wild ⇒
             match T with
             | ?P → ?Q ⇒ first [ app_assert t P cont' | fail 3 ]
             | ∀ _:?A, _ ⇒ first [ app_evar t A cont' | fail 3 ]
             end
           | _ ⇒
             match T with
             | U → ?Q ⇒ first [ app_assert t U cont' | fail 3 ]
             | ∀ _:U, _ ⇒ first [ app_evar t U cont' | fail 3 ]
             | ?P → ?Q ⇒ first [ app_assert t P cont | fail 3 ]
             | ∀ _:?A, _ ⇒ first [ app_evar t A cont | fail 3 ]
             end
           end
         | fail 2 ]
      | _ ⇒
          match T with
          | ?P → ?Q ⇒ first [ app_arg t P v cont'
                              | app_assert t P cont
                              | fail 3 ]
          | ∀ _:?A, _ ⇒ first [ cont' (t v)
                                  | app_evar t A cont
                                  | fail 3 ]
          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
    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: (» $E0$ $E1$ .. 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 $I1$ $I2$ IN, or empty. Syntax *lets* H: $E0$ $E1$ .. 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; fast_rm_inside E.
Tactic Notation "lets" ":" constr(E) :=
  let H := fresh in lets H: E.
Tactic Notation "lets" ":" constr(E0)
 constr(A1) :=
  lets: (» E0 A1).
Tactic Notation "lets" ":" constr(E0)
 constr(A1) constr(A2) :=
```

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

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

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

Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
 constr(A1) :=
  lets [I1 I2]: E0 A1.
Tactic Notation "lets" simple_intropattern(I1) simple_intropattern(I2) ":" constr(E0)
```

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

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

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

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

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

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

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

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

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

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);
  fast_rm_inside Ei.

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;
  fast_rm_inside Ei.
```

applys (» *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 *applys* defined earlier on. *applys E0 E1 E2* .. *EN* is also available.

```
Ltac applys_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 applys_base E :=
  match type of E with
  | list Boxer ⇒ applys_build E
  | _ ⇒ first [ rapply E | applys_build E ]
  end; fast_rm_inside E.

Tactic Notation "applys" constr(E) :=
  applys_base E.
Tactic Notation "applys" constr(E0) constr(A1) :=
  applys (» E0 A1).
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) :=
  applys (» E0 A1 A2).
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) constr(A3) :=
  applys (» E0 A1 A2 A3).
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  applys (» E0 A1 A2 A3 A4).
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  applys (» E0 A1 A2 A3 A4 A5).
```

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

```
Ltac fapplys_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 "fapplys" constr(E0) :=
  match type of E0 with
  | list Boxer ⇒ fapplys_build E0
  | _ ⇒ fapplys_build (» E0)
  end.
Tactic Notation "fapplys" constr(E0) constr(A1) :=
```

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

7.4.4 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 | applys H ].
```

sapply stands for "super apply". It tries `apply`, `eapply`, *applys* and *fapply*, and also tries to head-normalize the goal first.

```
Tactic Notation "sapply" constr(H) :=
  first [ apply H | eapply H | rapply H | applys H
        | hnf; apply H | hnf; eapply H | hnf; applys H
        | fapply H ].
```

7.4.5 Adding assumptions

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

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

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

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

lets_simpl: E is the same as *lets_simpl* H: E with the name H being choosed automatically.

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

lets_hnf: E is the same as *lets_hnf* H: E with the name H being choosed automatically.

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

put X: E is a synonymous for `pose` ($X := E$). Other syntaxes are *put*: E.

```
Tactic Notation "put" ident(X) ":" constr(E) :=
  pose (X := E).
Tactic Notation "put" ":" constr(E) :=
  let X := fresh "X" in pose (X := E).
```

7.4.6 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* $[A \land B]$ `by` *showing* `first` $[A]$ *and* `then` $[A \to B]$, *through the command* [*logic* (*foral* A B, $A \to (A \to B) \to A \land B$)]

```
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).
```

7.4.7 Application modulo equalities

The tactic *equates* replaces a goal of the form $P\ x\ y\ z$ with a goal of the form $P\ x\ ?a\ z$ and a subgoal $?a = y$. The introduction of the evar $?a$ makes it possible to apply lemmas that would not apply to the original goal, for example a lemma of the form $\forall\ n\ m,\ P\ n\ n\ m$, 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)
  (A2 : ∀ (x1 : A1), Type)
  (A3 : ∀ (x1 : A1) (x2 : A2 x1), Type)
  (A4 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), Type)
  (A5 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3), Type)
  (A6 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3) (x5 : A5 x4), Type).
Lemma equates_0 : ∀ (P Q:Prop),
  P → P = Q → Q.
Proof. intros. subst. auto. Qed.
Lemma equates_1 :
  ∀ (P:A0→Prop) x1 y1,
  P y1 → x1 = y1 → P x1.
Proof. intros. subst. auto. Qed.
Lemma equates_2 :
  ∀ y1 (P:A0→∀(x1:A1),Prop) x1 x2,
  P y1 x2 → x1 = y1 → P x1 x2.
Proof. intros. subst. auto. Qed.
Lemma equates_3 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1),Prop) x1 x2 x3,
  P y1 x2 x3 → x1 = y1 → P x1 x2 x3.
Proof. intros. subst. auto. Qed.
Lemma equates_4 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1)(x3:A3 x2),Prop) x1 x2 x3 x4,
  P y1 x2 x3 x4 → x1 = y1 → P x1 x2 x3 x4.
Proof. intros. subst. auto. Qed.
```

```
Lemma equates_5 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1)(x3:A3 x2)(x4:A4 x3),Prop) x1 x2 x3 x4 x5,
  P y1 x2 x3 x4 x5 → x1 = y1 → P x1 x2 x3 x4 x5.
Proof. intros. subst. auto. Qed.

Lemma equates_6 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1)(x3:A3 x2)(x4:A4 x3)(x5:A5 x4),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).
```

7.5 Introduction and generalization

7.5.1 Introduction

introv is used to name only non-dependent hypothesis.

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

```
Ltac introv_rec :=
  match goal with
  | ⊢ ?P → ?Q ⇒ idtac
  | ⊢ ∀ _, _ ⇒ intro; introv_rec
  | ⊢ _ ⇒ 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 $x \to \neg Q$ 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`.

7.5.2 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*.

7.5.3 Naming

sets X: *E* is the same as set (*X* := *E*) in *, that is, it replaces all occurences 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* ← *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 ⊢ performs no substitution at all.

Tactic Notation "set_eq" *ident*(*X*) *ident*(*HX*) ":" constr(*E*) :=
set (*X* := *E*); *def_to_eq X HX E*.
Tactic Notation "set_eq" *ident*(*X*) ":" constr(*E*) :=
let *HX* := fresh "EQ" *X* in *set_eq X HX*: *E*.
Tactic Notation "set_eq" ":" constr(*E*) :=
let *X* := fresh "X" in *set_eq X*: *E*.

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

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

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

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

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

Tactic Notation "set_eq" ident(X) ident(HX) ":" constr(E) "in" "|-" :=
  set (X := E) in |-; def_to_eq 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 occurences 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 $(X = E) \to C$. *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).
```

7.6 Rewriting

7.6.1 Rewriting

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).
```

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.

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

7.6.2 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" constr(E) "with" constr(F) "in" hyp(H) :=
  let T := fresh in assert (T: E = F); [ | replace E with F in H; clear T ].
```

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

```
Tactic Notation "replaces" constr(E) "at" constr(K) "with" constr(F) :=
  let T := fresh in assert (T: E = F); [ | rewrites T at K; clear T ].
```

```
Tactic Notation "replaces" constr(E) "at" constr(K) "with" constr(F) "in" hyp(H)
:=
  let T := fresh in assert (T: E = F); [ | rewrites T at K in H; clear T ].
```

7.6.3 Renaming

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

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

7.6.4 Unfolding

unfolds unfolds the head definition in the goal, i.e. if the goal has form *P x1 ... 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.

```
Ltac apply_to_head_of E cont :=
  let go E :=
    let P := get_head E in cont P in
  match E with
  | ∀ _,_ ⇒ intros; apply_to_head_of E cont
  | ?A = ?B ⇒ first [ go A | go B ]
```

```
  | ?A ⇒ go A
  end.

Ltac unfolds_base :=
  match goal with ⊢ ?G ⇒
   apply_to_head_of G ltac:(fun P ⇒ 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 ... xN* then it calls `unfold` *P* `in` *H*.

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

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

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

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

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

```
Tactic Notation "folds" constr(H) :=
  fold H in *.
Tactic Notation "folds" constr(H1) "," constr(H2) :=
  folds H1; folds H2.
Tactic Notation "folds" constr(H1) "," constr(H2) "," constr(H3) :=
  folds H1; folds H2; folds H3.
Tactic Notation "folds" constr(H1) "," constr(H2) "," constr(H3)
 "," constr(H4) :=
  folds H1; folds H2; folds H3; folds H4.
Tactic Notation "folds" constr(H1) "," constr(H2) "," constr(H3)
 "," constr(H4) "," constr(H5) :=
  folds H1; folds H2; folds H3; folds H4; folds H5.
```

7.6.5 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" reference(F1) :=
  simpl F1 in *.
Tactic Notation "simpls" reference(F1) "," reference(F2) :=
  simpls F1; simpls F2.
Tactic Notation "simpls" reference(F1) "," reference(F2)
 "," reference(F3) :=
  simpls F1; simpls F2; simpls F3.
Tactic Notation "simpls" reference(F1) "," reference(F2)
 "," reference(F3) "," reference(F4) :=
  simpls F1; simpls F2; simpls F3; simpls F4.
```

unsimpl E replaces all occurence 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).
```

7.6.6 Evaluation

```
Tactic Notation "hnfs" := hnf in *.
```

7.6.7 Substitution

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

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

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

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

substs 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 *substs below H* will substitute equalities below hypothesis *H*.

```
Tactic Notation "substs" "below" "body" constr(M) :=
  substs_below M.
```

substs 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 "substs" "below" hyp(H) :=
  match type of H with ?M ⇒ substs below body M end.
```

subst_hyp H substitutes the equality contained in H. The behaviour is extended in LibData –TODO

```
Ltac subst_hyp_base H :=
  match type of H with
  | ?x = ?y ⇒ first [ subst x | subst y ]
  end.
```

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

7.6.8 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).
```

7.6.9 Proving equalities

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 .. xN = f y1 .. 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).
```

7.7 Inversion

7.7.1 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 explicitely, 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).
```

7.7.2 Inversion with substitution

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

```
Axiom inj_pair2 : ∀ (U : Type) (P : U → Type) (p : U) (x y : P p),
        existT P p x = existT P p y → 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.
```

7.7.3 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 .. aN = C b1 .. bN* and substitute all equalities *ai = bi* that have been generated.

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

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

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

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

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

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

7.7.4 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.
```

7.7.5 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 x := fresh "Eq" in cases E as H.
```

case_if_post is to be defined later as a tactic to clean up goals.

```
Ltac case_if_post := idtac.
```

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`. It looks in the goal first, and otherwise in the first hypothesis that contains and `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 E Eq :=
  match type of E with
  | {_}+{_} ⇒ destruct E as [Eq | Eq]
  | _ ⇒ let X := fresh in
         sets_eq ← X Eq: E;
         destruct X
  end; 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 $x = y$ or $x == y$, it substitutes it in the goal

```
Ltac cases_if_on_tactic 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_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 _] ⇒ case_if_on B as Eq
  | K: context [if ?B then _ else _] ⊢ _ ⇒ case_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.
```

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 and `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.
```

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.
```

7.8 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 (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.
```

7.9 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.
```

7.10 Equivalence

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

```
Lemma iff_intro_swap : ∀ (P Q : Prop),
  (Q → P) → (P → Q) → (P ↔ Q).
Proof. intuition. Qed.

Tactic Notation "iff" simple_intropattern(H1) simple_intropattern(H2) :=
  split; [ intros H1 | intros H2 ].
Tactic Notation "iff" simple_intropattern(H) :=
  iff H H.
Tactic Notation "iff" :=
  let H := fresh "H" in iff H.

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

```
  let H := fresh "H" in iff ← H.
```

7.11 N-ary Conjunctions and Disjunctions

N-ary Conjunctions Splitting in Goals
Underlying implementation of *splits*.

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

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

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

  end.

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

splits applies to a goal of the form (*T1* ∧ .. ∧ *TN*) and destruct it into *N* subgoals *T1* .. *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.

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* ∨ ... ∨ *PK* ∨ ... ∨ *PN* and leaves the goal *PK*. It 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, applies to a goal of the form $P1 \lor ... \lor PK \lor ... \lor PN$ 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.
```

∃ *T1* ... *TN* is a shorthand for ∃ *T1*; ...; ∃ *TN*. It is intended to prove goals of the form *exist X1 .. XN*, *P*. If an argument provided is __ (double underscore), then an evar is introduced. ∃ *T1* .. *TN* ___ is equivalent to ∃ *T1* .. *TN* __ __ __ with as many __ as possible.

```
Tactic Notation "exists_original" constr(T1) :=
  ∃ T1.
Tactic Notation "exists" constr(T1) :=
  match T1 with
  | ltac_wild ⇒ esplit
  | ltac_wilds ⇒ repeat esplit
  | _ ⇒ ∃ T1
  end.
Tactic Notation "exists" constr(T1) constr(T2) :=
  ∃ T1; ∃ T2.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) :=
  ∃ T1; ∃ T2; ∃ T3.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
 constr(T5) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
 constr(T5) constr(T6) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5; ∃ T6.
```

```
Tactic Notation "exists___" constr(N) :=
  let rec aux N :=
    match N with
    | 0 ⇒ idtac
    | S ?N' ⇒ 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.

Existentials and conjunctions in hypotheses
todo: doc

```
Ltac intuit_core :=
  repeat match goal with
  | H: _ ∧ _ ⊢ _ ⇒ destruct H
  | H: ∃ a , _ ⊢ _ ⇒ destruct H
  end.

Ltac intuit_from H :=
  first [ progress (intuit_core)
        | destruct H; intuit_core ].

Tactic Notation "intuit" :=
  intuit_core.
Tactic Notation "intuit" constr(H) :=
  intuit_from H.
```

7.12 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 ].
```

7.13 Tactics to invoke automation

7.13.1 *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.

```
Ltac jauto_set_hyps :=
  repeat match goal with H: ?T ⊢ _ ⇒
    match T with
    | _ ∧ _ ⇒ destruct H
    | ∃ a , _ ⇒ destruct H
    | _ ⇒ generalizes 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 *.

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].
```

7.13.2 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 ¬ 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 × is used after a tactic.

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

`auto`¬ 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.

```
Tactic Notation "auto" "~" :=
  auto_tilde.
Tactic Notation "auto" "~" constr(E1) :=
```

lets: *E1*; *auto_tilde*.
Tactic Notation "auto" "~" constr(*E1*) constr(*E2*) :=
lets: *E1*; *lets*: *E2*; *auto_tilde*.
Tactic Notation "auto" "~" constr(*E1*) constr(*E2*) constr(*E3*) :=
lets: *E1*; *lets*: *E2*; *lets*: *E3*; *auto_tilde*.

auto× 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 "auto" "*" :=
auto_star.
Tactic Notation "auto" "*" constr(*E1*) :=
lets: *E1*; *auto_star*.
Tactic Notation "auto" "*" constr(*E1*) constr(*E2*) :=
lets: *E1*; *lets*: *E2*; *auto_star*.
Tactic Notation "auto" "*" 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. *auto_false*¬ and *auto_false*× are also available.

Ltac *auto_false_base cont* :=
try solve [*cont tt* | *tryfalse* by congruence/
| try split; *intros_all*; *tryfalse* by congruence/].

Tactic Notation "auto_false" :=
auto_false_base ltac:(fun *tt* ⇒ auto).
Tactic Notation "auto_false" "~" :=
auto_false_base ltac:(fun *tt* ⇒ auto~).
Tactic Notation "auto_false" "*" :=
auto_false_base ltac:(fun *tt* ⇒ auto*).

7.13.3 Definitions for parsing compatibility

Tactic Notation "f_equal" :=
f_equal.
Tactic Notation "constructor" :=
constructor.
Tactic Notation "simple" :=
simpl.

7.13.4 Parsing for light automation

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

- *cuts* and *asserts* only call `auto` on their first subgoal,
- `apply¬` relies on *sapply* rather than `apply`,
- *tryfalse¬* 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¬` will not be available.

```
Tactic Notation "equates" "~" constr(E) :=
    equates E; auto¬.
Tactic Notation "equates" "~" constr(n1) constr(n2) :=
  equates n1 n2; auto¬.
Tactic Notation "equates" "~" constr(n1) constr(n2) constr(n3) :=
  equates n1 n2 n3; auto¬.
Tactic Notation "equates" "~" constr(n1) constr(n2) constr(n3) constr(n4) :=
  equates n1 n2 n3 n4; auto¬.
Tactic Notation "applys_eq" "~" constr(H) constr(E) :=
  applys_eq H E; auto_tilde.
Tactic Notation "applys_eq" "~" constr(H) constr(n1) constr(n2) :=
  applys_eq H n1 n2; auto_tilde.
Tactic Notation "applys_eq" "~" constr(H) constr(n1) constr(n2) constr(n3) :=
  applys_eq H n1 n2 n3; auto_tilde.
Tactic Notation "applys_eq" "~" constr(H) constr(n1) constr(n2) constr(n3) constr(n4)
:=
  applys_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(T) :=
  false T by auto_tilde/.
Tactic Notation "tryfalse" "~" :=
  tryfalse by auto_tilde/.
Tactic Notation "tryfalse_invert" "~" :=
  first [ tryfalse¬ | false_invert ].

Tactic Notation "asserts" "~" simple_intropattern(H) ":" constr(E) :=
  asserts H: E; [ auto_tilde | idtac ].
Tactic Notation "cuts" "~" simple_intropattern(H) ":" constr(E) :=
  cuts H: E; [ auto_tilde | idtac ].
Tactic Notation "cuts" "~" ":" constr(E) :=
  cuts: E; [ auto_tilde | idtac ].

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

Tactic Notation "lets" "~" ":" constr(E) :=
  lets: E; auto_tilde.
Tactic Notation "lets" "~" ":" constr(E0)
```

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

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

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

```
Tactic Notation "forwards" "~" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "forwards" "~" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards: E0 A1 A2 A3 A4 A5; auto_tilde.
Tactic Notation "applys" "~" constr(H) :=
  sapply H; auto_tilde. Tactic Notation "applys" "~" constr(E0) constr(A1) :=
  applys E0 A1; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) :=
  applys E0 A1; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) constr(A2) :=
  applys E0 A1 A2; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) constr(A2) constr(A3) :=
  applys E0 A1 A2 A3; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  applys E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "applys" "~" constr(E0) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  applys 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" "~" reference(F1) :=
  unfolds F1; auto_tilde.
Tactic Notation "unfolds" "~" reference(F1) "," reference(F2) :=
  unfolds F1, F2; auto_tilde.
Tactic Notation "unfolds" "~" reference(F1) "," reference(F2) "," reference(F3) :=
  unfolds F1, F2, F3; auto_tilde.
Tactic Notation "unfolds" "~" reference(F1) "," reference(F2) "," reference(F3) ","
 reference(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 "substs" "~" :=
  substs; 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 "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 "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 "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 "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_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" "~" constr(T1) :=
  ∃ 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.
```

7.13.5 Parsing for strong automation

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

Exception: use *subs*× instead of `subst`× 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 "applys_eq" "*" constr(H) constr(E) :=
  applys_eq H E; auto_star.
Tactic Notation "applys_eq" "*" constr(H) constr(n1) constr(n2) :=
  applys_eq H n1 n2; auto_star.
Tactic Notation "applys_eq" "*" constr(H) constr(n1) constr(n2) constr(n3) :=
```

```
  applys_eq H n1 n2 n3; auto_star.
Tactic Notation "applys_eq" "*" constr(H) constr(n1) constr(n2) constr(n3) constr(n4)
:=
  applys_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(T) :=
  false T by auto_star/.
Tactic Notation "tryfalse" "*" :=
  tryfalse by auto_star/.
Tactic Notation "tryfalse_invert" "*" :=
  first [ tryfalse× | false_invert ].

Tactic Notation "asserts" "*" simple_intropattern(H) ":" constr(E) :=
  asserts H: E; [ auto_star | idtac ].
Tactic Notation "cuts" "*" simple_intropattern(H) ":" constr(E) :=
```

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

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

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

Tactic Notation "applys" "*" constr(*H*) :=
sapply H; *auto_star*. Tactic Notation "applys" "*" constr(*E0*) constr(*A1*) :=
applys E0 A1; *auto_star*.
Tactic Notation "applys" "*" constr(*E0*) constr(*A1*) :=
applys E0 A1; *auto_star*.
Tactic Notation "applys" "*" constr(*E0*) constr(*A1*) constr(*A2*) :=
applys E0 A1 A2; *auto_star*.
Tactic Notation "applys" "*" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) :=
applys E0 A1 A2 A3; *auto_star*.
Tactic Notation "applys" "*" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) constr(*A4*)
:=
applys E0 A1 A2 A3 A4; *auto_star*.
Tactic Notation "applys" "*" constr(*E0*) constr(*A1*) constr(*A2*) constr(*A3*) constr(*A4*)

```
constr(A5) :=
  applys E0 A1 A2 A3 A4 A5; auto_star.
Tactic Notation "specializes" "*" hyp(H) :=
  specializes H; auto_star.
Tactic Notation "specializes" "~" hyp(H) constr(A1) :=
  specializes H A1; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H A1 A2; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) :=
  specializes H A1 A2 A3; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
:=
  specializes H A1 A2 A3 A4; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) constr(A3) constr(A4)
constr(A5) :=
  specializes H A1 A2 A3 A4 A5; auto_star.
Tactic Notation "fapply" "*" constr(E) :=
  fapply E; auto_star.
Tactic Notation "sapply" "*" constr(E) :=
  sapply E; auto_star.
Tactic Notation "logic" constr(E) :=
  logic_base E ltac:(fun _ ⇒ auto_star).
Tactic Notation "intros_all" "*" :=
  intros_all; auto_star.
Tactic Notation "unfolds" "*" :=
  unfolds; auto_star.
Tactic Notation "unfolds" "*" reference(F1) :=
  unfolds F1; auto_star.
Tactic Notation "unfolds" "*" reference(F1) "," reference(F2) :=
  unfolds F1, F2; auto_star.
Tactic Notation "unfolds" "*" reference(F1) "," reference(F2) "," reference(F3) :=
  unfolds F1, F2, F3; auto_star.
Tactic Notation "unfolds" "*" reference(F1) "," reference(F2) "," reference(F3) ","
 reference(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 "substs" "*" :=
  substs; auto_star.
Tactic Notation "intro_hyp" "*" hyp(H) :=
  subst_hyp H; auto_star.
Tactic Notation "intro_subst" "*" :=
  intro_subst; auto_star.
Tactic Notation "subst_eq" "*" constr(E) :=
  subst_eq E; auto_star.

Tactic Notation "rewrite" "*" constr(E) :=
  rewrite E; auto_star.
Tactic Notation "rewrite" "*" "<-" constr(E) :=
  rewrite ← E; auto_star.
Tactic Notation "rewrite" "*" constr(E) "in" hyp(H) :=
  rewrite E in H; auto_star.
Tactic Notation "rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  rewrite ← E in H; auto_star.

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

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

Tactic Notation "cuts_rewrite" "*" constr(E) :=
  cuts_rewrite E; auto_star.
Tactic Notation "cuts_rewrite" "*" "<-" constr(E) :=
```

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

`Tactic Notation` "fequal" "*" :=
fequal; *auto_star*.
`Tactic Notation` "fequals" "*" :=
fequals; *auto_star*.
`Tactic Notation` "pi_rewrite" "*" `constr`(E) :=
pi_rewrite E; *auto_star*.
`Tactic Notation` "pi_rewrite" "*" `constr`(E) "in" *hyp*(H) :=
pi_rewrite E `in` H; *auto_star*.

`Tactic Notation` "invert" "*" *hyp*(H) :=
invert H; *auto_star*.
`Tactic Notation` "inverts" "*" *hyp*(H) :=
inverts H; *auto_star*.
`Tactic Notation` "injects" "*" *hyp*(H) :=
injects H; *auto_star*.
`Tactic Notation` "inversions" "*" *hyp*(H) :=
inversions H; *auto_star*.

`Tactic Notation` "cases" "*" `constr`(E) "as" *ident*(H) :=
cases E `as` H; *auto_star*.
`Tactic Notation` "cases" "*" `constr`(E) :=
cases E; *auto_star*.
`Tactic Notation` "case_if" "*" :=
case_if; *auto_star*.
`Tactic Notation` "case_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 "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" "*" constr($T1$) :=
  ∃ *T1*; *auto_star*.
Tactic Notation "exists" "*" constr($T1$) constr($T2$) :=
  ∃ *T1 T2*; *auto_star*.
Tactic Notation "exists" "*" constr($T1$) constr($T2$) constr($T3$) :=
  ∃ *T1 T2 T3*; *auto_star*.
Tactic Notation "exists" "*" constr($T1$) constr($T2$) constr($T3$) constr($T4$) :=
  ∃ *T1 T2 T3 T4*; *auto_star*.
Tactic Notation "exists" "*" constr($T1$) constr($T2$) constr($T3$) constr($T4$)
 constr($T5$) :=
  ∃ *T1 T2 T3 T4 T5*; *auto_star*.

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

7.14 Tactics to sort out the proof context

7.14.1 Hiding hypotheses

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

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

Lemma ltac_something_eq : ∀ (e:Type),
  e = (@ltac_something _ e).
Proof. auto. Qed.

Lemma ltac_something_hide : ∀ (e:Type),
  e → (@ltac_something _ e).
Proof. auto. Qed.

Lemma ltac_something_show : ∀ (e:Type),
  (@ltac_something _ e) → e.
Proof. 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" "*" :=
  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 _ _ ⊢ _ ⇒ show_hyp H end.
Tactic Notation "hide_hyps" :=
  repeat match goal with H: ?T ⊢ _ ⇒
    match type of T with
    | Prop ⇒
      match T with
      | @ltac_something _ _ ⇒ fail 2
      | _ ⇒ hide_hyp H
      end
    | _ ⇒ fail 1
    end
  end.
```

hide H and *show* H automatically select between *hide_hyp* or *hide_def*, and *show_hyp* or *show_def*. Similarly *hide_all* and *show_all* apply to all.

```
Tactic Notation "hide" hyp(H) :=
  first [hide_def H | hide_hyp H].
Tactic Notation "show" hyp(H) :=
  first [show_def H | show_hyp H].
Tactic Notation "hide_all" :=
  hide_hyps; hide_defs.
Tactic Notation "show_all" :=
```

```
unfold ltac_something in *.
```

hide_term E can be used to hide a term from the goal. *show_term* or *show_term E* can be used to reveal it. *hide_term E* in *H* can be used to specify an hypothesis.

```
Tactic Notation "hide_term" constr(E) :=
  change E with (@ltac_something _ E).
Tactic Notation "show_term" constr(E) :=
  change (@ltac_something _ E) with E.
Tactic Notation "show_term" :=
  unfold ltac_something.

Tactic Notation "hide_term" constr(E) "in" hyp(H) :=
  change E with (@ltac_something _ E) in H.
Tactic Notation "show_term" constr(E) "in" hyp(H) :=
  change (@ltac_something _ E) with E in H.
Tactic Notation "show_term" "in" hyp(H) :=
  unfold ltac_something in H.
```

7.14.2 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.
```

7.14.3 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; clear X3.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4) :=
  clears X1; clears X2; clear X3; clear X4.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
 ident(X5) :=
  clears X1; clears X2; clear X3; clear X4; clear X5.
Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
 ident(X5) ident(X6) :=
  clears X1; clears X2; clear X3; clear X4; clear X5; clear 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.

```
Tactic Notation "clears_all" :=
  repeat match goal with H: _ ⊢ _ ⇒ clear H end.
```

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.
```

7.15 Tactics for development purposes

7.15.1 Skipping subgoals

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 ⊢ ?G ⇒
    let H := 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.
```

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 := fresh in skip H: T.
```

```
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 ⊢ ?*G* ⇒ *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.

```
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.
```

7.16 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: `Require Import` *LibTacticsCompatibility*.

```
Module LibTacticsCompatibility.
  Tactic Notation "apply" "*" constr(H) :=
    sapply H; auto_star.
  Tactic Notation "subst" "*" :=
    subst; auto_star.
End LibTacticsCompatibility.
```

```
Open Scope nat_scope.
```

Chapter 8

Library Norm

8.1 Norm: Normalization of STLC

```
Require Import Stlc.
```

(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 nonterminating 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?

☐

8.2 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
.
```

```
Tactic Notation "T_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "TBool" | Case_aux c "TArrow" | Case_aux c "TProd" ].
```

```
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.
```

```
Tactic Notation "t_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp" | Case_aux c "tabs"
  | Case_aux c "tpair" | Case_aux c "tfst" | Case_aux c "tsnd"
  | Case_aux c "ttrue" | Case_aux c "tfalse" | Case_aux c "tif" ].
```

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).

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_AppAbs" | Case_aux c "ST_App1" | Case_aux c "ST_App2"
  | Case_aux c "ST_Pair1" | Case_aux c "ST_Pair2"
    | Case_aux c "ST_Fst" | Case_aux c "ST_FstPair"
    | Case_aux c "ST_Snd" | Case_aux c "ST_SndPair"
  | Case_aux c "ST_IfTrue" | Case_aux c "ST_IfFalse" | Case_aux c "ST_If" ].

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 (extend 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.

Tactic Notation "has_type_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "T_Var" | Case_aux c "T_Abs" | Case_aux c "T_App"
  | Case_aux c "T_Pair" | Case_aux c "T_Fst" | Case_aux c "T_Snd"
```

```
  | Case_aux c "T_True" | Case_aux c "T_False" | Case_aux c "T_If" ].
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) :=
  ∀ x, ¬ appears_free_in x t.

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'.
  has_type_cases (induction H) Case;
    intros Gamma' Heqv...
  Case "T_Var".
    apply T_Var... rewrite ← Heqv...
  Case "T_Abs".
    apply T_Abs... apply IHhas_type. intros y Hafi.
    unfold extend. remember (beq_id x y) as e.
    destruct e...
  Case "T_Pair".
    apply T_Pair...
  Case "T_If".
    eapply T_If...
Qed.

Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   has_type Gamma t T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  has_type_cases (induction Htyp) Case; inversion Hafi; subst...
  Case "T_Abs".
    destruct IHHtyp as [T' Hctx]... ∃ T'.
    unfold extend in Hctx.
    apply not_eq_beq_id_false in H2. rewrite H2 in Hctx...
Qed.

Corollary typable_empty__closed : ∀ t T,
    has_type empty t T →
    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 : ∀ Gamma x U v t S,
     has_type (extend Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  t_cases (induction t) Case;
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  Case "tvar".
    simpl. rename i into y.
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      apply beq_id_eq in Heqe. subst.
      unfold extend in H1. rewrite ← beq_id_refl in H1.
      inversion H1; subst. clear H1.
      eapply context_invariance...
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra) as [T' HT']...
      inversion HT'.
    SCase "x<>y".
      apply T_Var... unfold extend in H1. rewrite ← Heqe in H1...
  Case "tabs".
    rename i into y. rename t into T11.
    apply T_Abs...
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      eapply context_invariance...
      apply beq_id_eq in Heqe. subst.
      intros x Hafi. unfold extend.
      destruct (beq_id y x)...
    SCase "x<>y".
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold extend.
      remember (beq_id y z) as e0. destruct e0...
      apply beq_id_eq in Heqe0. subst.
      rewrite ← Heqe...
Qed.

Theorem preservation : ∀ t t' T,
     has_type empty t T →
```

```
    t ==> t' →
    has_type empty t' T.
Proof with eauto.
  intros t t' T HT.
  remember (@empty ty) as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  has_type_cases (induction HT) Case;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  Case "T_App".
    inversion HE; subst...
    SCase "ST_AppAbs".
      apply substitution_preserves_typing with T1...
      inversion HT1...
  Case "T_Fst".
    inversion HT...
  Case "T_Snd".
    inversion HT...
Qed.
```

☐

Determinism

```
Lemma step_deterministic :
   deterministic step.
Proof with eauto.
   unfold deterministic.
   Admitted.
```

8.3 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 *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 predicates*.) 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 $(A \rightarrow A)$->$(A \rightarrow A)$ *preserve the property of preserving* property P, and so on. We do this by defining a family of predicates, indexed by types. For the base type A, the predicate is just P. For functional types, it says that the function should map values satisfying the predicate at the input type to values satisfying the predicate 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 $(\forall\ s,\ R\ T1\ s \rightarrow R\ TS\ (tapp\ t\ s))$, 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 predicates 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 ⇒ True
   | TArrow T1 T2 ⇒ (∀ s, R T1 s → R T2 (tapp t s))

   | TProd T1 T2 ⇒ 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 : ∀ {T} {t}, R T t → halts t.
Proof.
  intros. destruct T; unfold R in H; inversion H; inversion H1; assumption.
Qed.
```

```
Lemma R_typable_empty : ∀ {T} {t}, R T t → 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.

8.3.1 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 determinstic. This lemma might still be true for non-deterministic languages, but the proof would be harder!

```
Lemma step_preserves_halting : ∀ t t', (t ==> t') → (halts t ↔ halts t').
Proof.
```

```
 intros t t' ST. unfold halts.
 split.
 Case "->".
  intros [t'' [STM V]].
  inversion STM; subst.
   apply ex_falso_quodlibet. apply value__normal in V. unfold normal_form in V. apply
V. ∃ t'. auto.
   rewrite (step_deterministic _ _ _ ST H). ∃ t''. split; assumption.
 Case "<-".
  intros [t'0 [STM V]].
  ∃ 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 finiteness 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 : ∀ T t t', (t ==> t') → R T t → 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 : ∀ T t t',
  (t ==>* t') → R T t → 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' : ∀ T t t',
  has_type empty t T → (t ==> t') → R T t' → R T t.
Proof.
   Admitted.

Lemma multistep_preserves_R' : ∀ T t t',
  has_type empty t T → (t ==>* t') → R T t' → 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.
```

8.3.2 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 *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 $x1{:}T1,..xn{:}Tn \vdash t : T$ and *v1*,...,*vn* are values such that $R\ T1\ v1$, $R\ T2\ v2$, ..., $R\ Tn\ vn$, then $R\ T\ ([x1{:=}v1][x2{:=}v2]...[xn{:=}vn]t)$.

The proof will proceed by induction on the typing derivation $x1{:}T1,..xn{:}Tn \vdash t : T$; 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*,...*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 ...,y:*bool*,...,y:*nat*,... and a corresponding term substitution written as ...[y:=(*tbool true*)]...[y:=(*tnat* 3)]...t. Since environments are extended from left to right, the binding y:*nat* hides the binding y:*bool*; since substitutions are performed right to left, we do the substitution y:=(*tnat* 3) first, so that the substitution y:=(*tbool 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 × 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 mextend (Gamma : context) (xts : tass) :=
  match xts with
  | nil ⇒ Gamma
  | ((x,v)::xts') ⇒ extend (mextend Gamma xts') x v
  end.
```

We will need some simple operations that work uniformly on environments and type assigments

```
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 nxs} : list (id × X) :=
  match nxs with
    | nil ⇒ nil
    | ((n',x)::nxs') ⇒ if beq_id n' n then drop n nxs' else (n',x)::(drop n nxs')
  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: ∀ t,
     closed t →
     ∀ x t', [x:=t'] t = t.
Proof.
  intros. apply vacuous_substitution. apply H. Qed.

Lemma subst_not_afi : ∀ t x v, closed v → ¬ appears_free_in x ([x:=v] t).
Proof with eauto.   unfold closed, not.
  t_cases (induction t) Case; intros x v P A; simpl in A.
    Case "tvar".
     remember (beq_id x i) as e; destruct e...
       inversion A; subst. rewrite ← beq_id_refl in Heqe; inversion Heqe.
    Case "tapp".
     inversion A; subst...
    Case "tabs".
     remember (beq_id x i) as e; destruct e...
       apply beq_id_eq in Heqe; subst. inversion A; subst...
       inversion A; subst...
    Case "tpair".
     inversion A; subst...
    Case "tfst".
     inversion A; subst...
    Case "tsnd".
     inversion A; subst...
    Case "ttrue".
     inversion A.
    Case "tfalse".
     inversion A.
    Case "tif".
     inversion A; subst...
```

```
Qed.

Lemma duplicate_subst : ∀ t' x t v,
  closed v → [x:=t]([x:=v]t') = [x:=v]t'.
Proof.
  intros. eapply vacuous_substitution. apply subst_not_afi. auto.
Qed.

Lemma swap_subst : ∀ t x x1 v v1, x ≠ x1 → closed v → closed v1 →
                     [x1:=v1]([x:=v]t) = [x:=v]([x1:=v1]t).
Proof with eauto.
 t_cases (induction t) Case; intros; simpl.
  Case "tvar".
   remember (beq_id x i) as e; destruct e; remember (beq_id x1 i) as e; destruct e.
      apply beq_id_eq in Heqe. apply beq_id_eq in Heqe0. subst.
         apply ex_falso_quodlibet...
      apply beq_id_eq in Heqe; subst. simpl.
         rewrite ← beq_id_refl. apply subst_closed...
      apply beq_id_eq in Heqe0; subst. simpl.
         rewrite ← beq_id_refl. rewrite subst_closed...
      simpl. rewrite ← Heqe. rewrite ← Heqe0...
   Admitted.
```

Properties of multi-substitutions

```
Lemma msubst_closed: ∀ t, closed t → ∀ ss, 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 charcterizing how *msubst* of closed terms distributes over `subst` and over each term form

```
Lemma subst_msubst: ∀ env x v t, closed v → closed_env env →
  msubst env ([x:=v]t) = [x:=v](msubst (drop x env) t).
Proof.
  induction env0; intros.
```

```
    auto.
    destruct a. simpl.
    inversion H0. fold closed_env in H2.
    remember (beq_id i x) as e; destruct e.
      apply beq_id_eq in Heqe; subst.
        rewrite duplicate_subst; auto.
      symmetry in Heqe. apply beq_id_false_not_eq in Heqe.
      simpl. rewrite swap_subst; eauto.
Qed.
```

```
Lemma msubst_var: ∀ ss x, closed_env ss →
   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: ∀ 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 : ∀ ss t1 t2, msubst ss (tapp t1 t2) = tapp (msubst ss t1) (msubst ss t2).
Proof.
 induction ss; intros.
   reflexivity.
   destruct a.
    simpl. rewrite ← 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 mextend_lookup : ∀ (c : tass) (x:id), lookup x c = (mextend empty c) x.
Proof.
  induction c; intros.
    auto.
    destruct a. unfold lookup, mextend, extend. destruct (beq_id i x); auto.
Qed.
```

```
Lemma mextend_drop : ∀ (c: tass) Gamma x x',
       mextend Gamma (drop x c) x' = if beq_id x x' then Gamma x' else mextend
Gamma c x'.
   induction c; intros.
      destruct (beq_id x x'); auto.
      destruct a. simpl.
      remember (beq_id i x) as e; destruct e.
        apply beq_id_eq in Heqe; subst. rewrite IHc.
            remember (beq_id x x') as e; destruct e. auto. unfold extend. rewrite ←
Heqe. auto.
        simpl. unfold extend. remember (beq_id i x') as e; destruct e.
            apply beq_id_eq in Heqe0; subst.
                                  remember (beq_id x x') as e; destruct e.
                  apply beq_id_eq in Heqe0; subst. rewrite ← beq_id_refl in Heqe.
inversion Heqe.
                  auto.
            auto.
Qed.
```

Properties of Instantiations

These are strightforward.

```
Lemma instantiation_domains_match: ∀ {c} {e},
  instantiation c e → ∀ {x} {T}, lookup x c = Some T → ∃ t, lookup x e = 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 : ∀ c e, instantiation c e → 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 : ∀ c e, instantiation c e →
                        ∀ x t T, lookup x c = Some T →
                                 lookup x e = Some t → 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 : ∀ c env,
  instantiation c env → ∀ x, instantiation (drop x c) (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 : ∀ v t t',
  value v → (t ==>* t') → (tapp v t) ==>* (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 : ∀ c e,
```

```
    instantiation c e →
    ∀ Gamma t S, has_type (mextend Gamma c) t S →
    has_type Gamma (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 : ∀ c env t T,
  has_type (mextend empty c) t T → instantiation c env → R T (msubst env t).
Proof.
  intros c env0 t T HT V.
  generalize dependent env0.
  remember (mextend empty c) as Gamma.
  assert (∀ x, Gamma x = lookup x c).
    intros. rewrite HeqGamma. rewrite mextend_lookup. auto.
  clear HeqGamma.
  generalize dependent c.
  has_type_cases (induction HT) Case; intros.
  Case "T_Var".
   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.
  Case "T_Abs".
    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 extend. rewrite mextend_drop. remember (beq_id x x0) as e; destruct e.
auto.
        rewrite H.
          clear - c Heqe. induction c.
              simpl. rewrite ← Heqe. auto.
              simpl. destruct a. unfold extend. 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 extend, lookup. destruct (beq_id x x0); auto.
      constructor; auto.
  Case "T_App".
    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 : ∀ t T, has_type empty t T → halts t.
Proof.
  intros.
  replace t with (msubst nil t).
  eapply R_halts.
  eapply msubst_R; eauto. instantiate (2:= nil). eauto.
  eapply V_nil.
  auto.
Qed.
```

Chapter 9

Library RecordSub

9.1 RecordSub: Subtyping with Records

```
Require Export MoreStlc.
```

9.2 Core Definitions

Syntax

```
Inductive ty : Type :=

  | TTop : ty
  | TBase : id → ty
  | TArrow : ty → ty → ty

  | TRNil : ty
  | TRCons : id → ty → ty → ty.
Tactic Notation "T_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "TTop" | Case_aux c "TBase" | Case_aux c "TArrow"
  | Case_aux c "TRNil" | Case_aux c "TRCons" ].
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.

Tactic Notation "t_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp" | Case_aux c "tabs"
  | Case_aux c "tproj" | Case_aux c "trnil" | Case_aux c "trcons" ].
```

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).

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_AppAbs" | Case_aux c "ST_App1" | Case_aux c "ST_App2"
  | Case_aux c "ST_Proj1" | Case_aux c "ST_ProjRcd" | Case_aux c "ST_Rcd"
  | Case_aux c "ST_Rcd_Head" | Case_aux c "ST_Rcd_Tail" ].

Hint Constructors step.
```

9.3 Subtyping

Now we come to the interesting part. We begin by defining the subtyping relation and developing some of its important technical properties.

9.3.1 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.
```

```
Tactic Notation "subtype_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "S_Refl" | Case_aux c "S_Trans" | Case_aux c "S_Top"
  | Case_aux c "S_Arrow" | Case_aux c "S_RcdWidth"
  | Case_aux c "S_RcdDepth" | Case_aux c "S_RcdPerm" ].
```

9.3.2 Subtyping Examples and Exercises

```
Module Examples.
```

```
Notation x := (Id 0).
Notation y := (Id 1).
Notation z := (Id 2).
Notation j := (Id 3).
Notation k := (Id 4).
Notation i := (Id 5).
Notation A := (TBase (Id 6)).
Notation B := (TBase (Id 7)).
Notation C := (TBase (Id 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.
```

9.3.3 Properties of Subtyping

Well-Formedness

```
Lemma subtype__wf : ∀ S T,
  subtype S T →
  well_formed_ty T ∧ well_formed_ty S.
Proof with eauto.
  intros S T Hsub.
  subtype_cases (induction Hsub) Case;
    intros; try (destruct IHHsub1; destruct IHHsub2)...
  Case "S_RcdPerm".
    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.
  T_cases (induction T) Case; intros; try solve by inversion.
  Case "TRCons".
    inversion H. subst. unfold Tlookup in H0.
    remember (beq_id i i0) as b. destruct b; subst...
    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.
```

```
subtype_cases (induction Hsub) Case; intros Ti Hget;
  try solve by inversion.
Case "S_Refl".
  ∃ Ti...
Case "S_Trans".
  destruct (IHHsub2 Ti) as [Ui Hui]... destruct Hui.
  destruct (IHHsub1 Ui) as [Si Hsi]... destruct Hsi.
  ∃ Si...
Case "S_RcdDepth".
  rename i0 into k.
  unfold Tlookup. unfold Tlookup in Hget.
  remember (beq_id i k) as b. destruct b...
  SCase "i = k – we're looking up the first field".
    inversion Hget. subst. ∃ S1...
Case "S_RcdPerm".
  ∃ Ti. split.
  SCase "lookup".
    unfold Tlookup. unfold Tlookup in Hget.
    remember (beq_id i i1) as b. destruct b...
    SSCase "i = i1 – we're looking up the first field".
      remember (beq_id i i2) as b. destruct b...
      SSSCase "i = i2 - -contradictory".
        destruct H0.
        apply beq_id_eq in Heqb. apply beq_id_eq in Heqb0.
        subst...
  SCase "subtype".
    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 : ∀ *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.
```

9.4 Typing

```
Definition context := id → (option ty).
Definition empty : context := (fun _ ⇒ None).
Definition extend (Gamma : context) (x:id) (T : ty) :=
  fun x' ⇒ if beq_id x x' then Some T else Gamma x'.

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 (extend 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).

Hint Constructors has_type.
```

```
Tactic Notation "has_type_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "T_Var" | Case_aux c "T_Abs" | Case_aux c "T_App"
  | Case_aux c "T_Proj" | Case_aux c "T_Sub"
  | Case_aux c "T_RNil" | Case_aux c "T_RCons" ].
```

9.4.1 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.
```

9.4.2 Properties of Typing

Well-Formedness

```
Lemma has_type__wf : ∀ Gamma t T,
  has_type Gamma t T → well_formed_ty T.
Proof with eauto.
  intros Gamma t T Htyp.
  has_type_cases (induction Htyp) Case...
  Case "T_App".
    inversion IHHtyp1...
  Case "T_Proj".
    eapply wf_rcd_lookup...
  Case "T_Sub".
    apply subtype__wf in H.
    destruct H...
Qed.
```

```
Lemma step_preserves_record_tm : ∀ tr tr',
  record_tm tr →
  tr ==> tr' →
  record_tm tr'.
Proof.
  intros tr tr' Hrt Hstp.
  inversion Hrt; subst; inversion Hstp; subst; eauto.
Qed.
```

Field Lookup

```
Lemma lookup_field_in_value : ∀ v T i Ti,
  value v →
  has_type empty v T →
  Tlookup i T = Some Ti →
  ∃ vi, tlookup i v = Some vi ∧ has_type empty vi Ti.
Proof with eauto.
  remember empty as Gamma.
  intros t T i Ti Hval Htyp. revert Ti HeqGamma Hval.
  has_type_cases (induction Htyp) Case; intros; subst; try solve by inversion.
  Case "T_Sub".
    apply (rcd_types_match S) in H0... destruct H0 as [Si [HgetSi Hsub]].
    destruct (IHHtyp Si) as [vi [Hget Htyvi]]...
  Case "T_RCons".
    simpl in H0. simpl. simpl in H1.
    remember (beq_id i i0) as b. destruct b.
```

```
    SCase "i is first".
      inversion H1. subst. ∃ t...
    SCase "i in tail".
      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

```
: ∀ Gamma s T1 T2,
     has_type Gamma s (TArrow T1 T2) →
     value s →
     ∃ x, ∃ S1, ∃ s2,
        s = tabs x S1 s2.
Proof with eauto.
   Admitted.
   □
```

```
Theorem progress : ∀ t T,
     has_type empty t T →
     value t ∨ ∃ t', t ==> t'.
Proof with eauto.
  intros t T Ht.
  remember empty as Gamma.
  revert HeqGamma.
  has_type_cases (induction Ht) Case;
    intros HeqGamma; subst...
  Case "T_Var".
    inversion H.
  Case "T_App".
    right.
    destruct IHHt1; subst...
    SCase "t1 is a value".
      destruct IHHt2; subst...
      SSCase "t2 is a value".
        destruct (canonical_forms_of_arrow_types empty t1 T1 T2)
          as [x [S1 [t12 Heqt1]]]...
        subst. ∃ ([x:=t2] t12)...
      SSCase "t2 steps".
        destruct H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
    SCase "t1 steps".
      destruct H as [t1' Hstp]. ∃ (tapp t1' t2)...
  Case "T_Proj".
    right. destruct IHHt...
```

SCase "rcd is value".
  `destruct` (lookup_field_in_value *t T i Ti*) `as` [*t' [Hget Ht'*]]...
SCase "rcd_steps".
  `destruct` *H0* `as` [*t' Hstp*]. ∃ (tproj *t' i*)...
Case "T_RCons".
`destruct` *IHHt1*...
SCase "head is a value".
  `destruct` *IHHt2*...
  SSCase "tail steps".
    `right`. `destruct` *H2* `as` [*tr' Hstp*].
    ∃ (trcons *i t tr'*)...
SCase "head steps".
  `right`. `destruct` *H1* `as` [*t' Hstp*].
  ∃ (trcons *i t' tr*)... `Qed`.

Informal proof of progress:

Theorem : For any term *t* and type *T*, if *empty* ⊢ *t* : *T* then *t* is a value or *t* ==> *t'* for some term *t'*.

Proof : Let *t* and *T* be given such that *empty* ⊢ *t* : *T*. We go by induction on the typing derivation. Cases *T_Abs* and *T_RNil* are immediate because abstractions and {} are always values. Case *T_Var* is vacuous because variables cannot be typed in the empty context.

- If the last step in the typing derivation is by *T_App*, then there are terms *t1 t2* and types *T1 T2* such that *t* = *t1 t2*, *T* = *T2*, *empty* ⊢ *t1* : *T1* → *T2* and *empty* ⊢ *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* ==> *t1'* for some term *t1'*. Then *t1 t2* ==> *t1' t2* by *ST_App1*.
  - Otherwise *t1* is a value.
    - Suppose *t2* ==> *t2'* for some term *t2'*. Then *t1 t2* ==> *t1 t2'* by rule *ST_App2* because *t1* is a value.
    - Otherwise, *t2* is a value. By lemma *canonical_forms_for_arrow_types*, *t1* = *x*:*S1*.*s2* for some *x*, *S1*, and *s2*. And (*x*:*S1*.*s2*) *t2* ==> [*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*, *empty* ⊢ *tr* : *Tr*, and *Tlookup i Tr* = *Some T*.

  The IH for the typing subderivation gives us that either *tr* is a value or it steps. If *tr* ==> *tr'* for some term *tr'*, then *tr.i* ==> *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 *tlookup i tr* = *Some ti*. It follows that *tr.i* ==> *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 *empty* ⊢ *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*, *empty* ⊢ *t1* : *T1* and *empty* ⊢ *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* ==> *t1'* for some term *t1'*. Then {*i*=*t1*, *tr*} ==> {*i*=*t1'*, *tr*} by rule *ST_Rcd_Head*.
  - Otherwise *t1* is a value.
    * Suppose *tr* ==> *tr'* for some term *tr'*. Then {*i*=*t1*, *tr*} ==> {*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 : ∀ Gamma x T,
  has_type Gamma (tvar x) T →
  ∃ S,
    Gamma x = Some S ∧ subtype S T.
Proof with eauto.
  intros Gamma x T Hty.
  remember (tvar x) as t.
  has_type_cases (induction Hty) Case; intros;
    inversion Heqt; subst; try solve by inversion.
  Case "T_Var".
    ∃ T...
  Case "T_Sub".
    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.
  has_type_cases (induction Hty) Case; intros;
```

```
    inversion Heqt; subst; try solve by inversion.
  Case "T_App".
    ∃ T1...
  Case "T_Sub".
    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 (extend Gamma x S1) t2 S2).
Proof with eauto.
  intros Gamma x S1 t2 T H.
  remember (tabs x S1 t2) as t.
  has_type_cases (induction H) Case;
    inversion Heqt; subst; intros; try solve by inversion.
  Case "T_Abs".
    assert (Hwf := has_type__wf _ _ _ H0).
    ∃ T12...
  Case "T_Sub".
    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,
    Tlookup i T = Some Si ∧ subtype Si Ti ∧ has_type Gamma t1 T.
Proof with eauto.
  intros Gamma i t1 Ti H.
  remember (tproj t1 i) as t.
  has_type_cases (induction H) Case;
    inversion Heqt; subst; intros; try solve by inversion.
  Case "T_Proj".
    assert (well_formed_ty Ti) as Hwf.
      SCase "pf of assertion".
        apply (wf_rcd_lookup i T Ti)...
        apply has_type__wf in H...
    ∃ T. ∃ Ti...
  Case "T_Sub".
    destruct IHhas_type as [U [Ui [Hget [Hsub Hty]]]]...
    ∃ U. ∃ Ui... Qed.

Lemma typing_inversion_rcons : ∀ Gamma i ti tr T,
  has_type Gamma (trcons i ti tr) T →
```

```
  ∃ Si, ∃ Sr,
    subtype (TRCons i Si Sr) T ∧ has_type Gamma ti Si ∧
    record_tm tr ∧ has_type Gamma tr Sr.
Proof with eauto.
  intros Gamma i ti tr T Hty.
  remember (trcons i ti tr) as t.
  has_type_cases (induction Hty) Case;
    inversion Heqt; subst...
  Case "T_Sub".
    apply IHHty in H0.
    destruct H0 as [Ri [Rr [HsubRS [HtypRi HtypRr]]]].
    ∃ Ri. ∃ Rr...
  Case "T_RCons".
    assert (well_formed_ty (TRCons i T Tr)) as Hwf.
      SCase "pf of assertion".
        apply has_type__wf in Hty1.
        apply has_type__wf in Hty2...
    ∃ T. ∃ Tr... Qed.

Lemma abs_arrow : ∀ x S1 s2 T1 T2,
  has_type empty (tabs x S1 s2) (TArrow T1 T2) →
     subtype T1 S1
  ∧ has_type (extend 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'.
  has_type_cases (induction H) Case;
    intros Gamma' Heqv...
  Case "T_Var".
    apply T_Var... rewrite ← Heqv...
  Case "T_Abs".
    apply T_Abs... apply IHhas_type. intros x0 Hafi.
    unfold extend. remember (beq_id x x0) as e.
    destruct e...
  Case "T_App".
    apply T_App with T1...
  Case "T_RCons".
    apply T_RCons... Qed.

Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   has_type Gamma t T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  has_type_cases (induction Htyp) Case; subst; inversion Hafi; subst...
  Case "T_Abs".
    destruct (IHHtyp H5) as [T Hctx]. ∃ T.
    unfold extend in Hctx. apply not_eq_beq_id_false in H3.
    rewrite H3 in Hctx... Qed.
```

Preservation

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (extend Gamma x U) t S →
     has_type empty v U →
     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.
  t_cases (induction t) Case; intros; simpl.
  Case "tvar".
    rename i into y.
    destruct (typing_inversion_var _ _ _ Htypt) as [T [Hctx Hsub]].
    unfold extend in Hctx.
    remember (beq_id x y) as e. destruct e...
    SCase "x=y".
      apply beq_id_eq in Heqe. 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'.
    SCase "x<>y".
      destruct (subtype__wf _ _ Hsub)...
  Case "tapp".
    destruct (typing_inversion_app _ _ _ _ Htypt) as [T1 [Htypt1 Htypt2]].
    eapply T_App...
  Case "tabs".
    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...
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      eapply context_invariance...
      apply beq_id_eq in Heqe. subst.
      intros x Hafi. unfold extend.
      destruct (beq_id y x)...
    SCase "x<>y".
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold extend.
```

```
      remember (beq_id y z) as e0. destruct e0...
      apply beq_id_eq in Heqe0. subst.
      rewrite ← Heqe...
  Case "tproj".
    destruct (typing_inversion_proj _ _ _ _ Htypt)
      as [T [Ti [Hget [Hsub Htypt1]]]]...
  Case "trnil".
    eapply context_invariance...
    intros y Hcontra. inversion Hcontra.
  Case "trcons".
    destruct (typing_inversion_rcons _ _ _ _ _ Htypt) as
      [Ti [Tr [Hsub [HtypTi [Hrcdt2 HtypTr]]]]].
    apply T_Sub with (TRCons i Ti Tr)...
    apply T_RCons...
    SCase "record_ty Tr".
      apply subtype__wf in Hsub. destruct Hsub. inversion H0...
    SCase "record_tm ([x:=v]t2)".
      inversion Hrcdt2; subst; simpl... Qed.

Theorem preservation : ∀ t t' T,
     has_type empty t T →
     t ==> t' →
     has_type empty t' T.
Proof with eauto.
  intros t t' T HT.
  remember empty as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  has_type_cases (induction HT) Case;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  Case "T_App".
    inversion HE; subst...
    SCase "ST_AppAbs".
      destruct (abs_arrow _ _ _ _ _ HT1) as [HA1 HA2].
      apply substitution_preserves_typing with T...
  Case "T_Proj".
    destruct (lookup_field_in_value _ _ _ _ H2 HT H)
      as [vi [Hget Hty]].
    rewrite H4 in Hget. inversion Hget. subst...
  Case "T_RCons".
    eauto using step_preserves_record_tm. Qed.
```

Informal proof of *preservation*:

Theorem: If *t*, *t'* are terms and T is a type such that $empty \vdash t : T$ and $t ==> t'$, then $empty \vdash t' : T$.

Proof: Let t and T be given such that $empty \vdash t : T$. 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 $t = t1\ t2$, $T = T2$, $empty \vdash t1 : T1 \rightarrow T2$ and $empty \vdash t2 : T1$.

  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 $empty \vdash [x{:=}t2]\ t12 : T2$ 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 $t = tr.i$, $empty \vdash tr : Tr$, and $Tlookup\ i\ Tr = Some\ T$.

  The IH for the typing derivation gives us that, for any term tr', if $tr ==> tr'$ then $empty \vdash tr'\ Tr$. 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 $tlookup\ i\ tr = Some\ vi$ and $t' = vi$. But by lemma *lookup_field_in_value*, $empty \vdash vi : Ti$ as desired.

- If the final step of the derivation is by *T_Sub*, then there is a type S such that $S <: T$ and $empty \vdash t : S$. 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 $t = \{i{=}t1,\ tr\}$, $T = \{i{:}T1,\ Tr\}$, $record\_tm\ tr$, $record\_tm\ Tr$, $empty \vdash t1 : T1$ and $empty \vdash tr : Tr$.

  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.

9.4.3 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 |- t : S1->S2 S1 <: T1 T1 <: S1 S2 <: T2
    - ——————————————- (T_Funny1) Gamma |- t : T1->T2
- Suppose we add the following reduction rule:
    - —————————— (ST_Funny21)

  {} ==> (\x:Top. x)
- Suppose we add the following subtyping rule:
    - ———————- (S_Funny3)

  {} <: Top->Top
- Suppose we add the following subtyping rule:
    - ———————- (S_Funny4)

  Top->Top <: {}
- Suppose we add the following evaluation rule:
    - —————————- (ST_Funny5)

  ({} t) ==> (t {})
- Suppose we add the same evaluation rule *and* a new typing rule:
    - —————————- (ST_Funny5)

  ({} t) ==> (t {})
    - ————————————— (T_Funny6)

  empty |- {} : Top->Top
- Suppose we *change* the arrow subtyping rule to: S1 <: T1 S2 <: T2
    - —————————————- (S_Arrow') S1->S2 <: T1->T2

□

Chapter 10

Library Sub

10.1 Sub: Subtyping

```
Require Export MoreStlc.
```

10.2 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.

10.2.1 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.

10.2.2 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 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.

10.2.3 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 |- t : S S <: T

(T_Sub) Gamma |- 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., $S = \{x{:}A{\to}A,\ y{:}B{\to}B\}$), but choose to forget about one of the fields ($T = \{y{:}B{\to}B\}$) so that we can pass t to a function that requires just a single-field record.

10.2.4 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 U <: T

(S_Trans) S <: T ... and a rule of *reflexivity*, since any type T is always just 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 -> Student
g : (C -> Person) -> 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->S2 <: S1->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->S2 <: T1->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 : Person -> C
g : (Student -> C) -> 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 $Person \rightarrow C$ is a subtype of $Student \rightarrow C$ 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 $T1 \rightarrow T2$.

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 S2 <: T2

(S_Arrow_wrong) S1->S2 <: T1->T2 Give a concrete example of functions f and g with types...

```
f : Student -> Nat
g : (Person -> Nat) -> Nat
```

... such that the application $g\ f$ will get stuck during execution.

☐

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,

```
{name:String, age:Nat, gpa:Nat} <: {name:String, age:Nat}
{name:String, age:Nat} <: {name:String}
{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:Student} <: {x: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,

```
{name:String,age:Nat} <: {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 jk in j1..jn, exists ip in i1..im, such that jk=ip and Sp <: Tk

---

(S_Rcd) {i1:S1...im:Sm} <: {j1:T1...jn:Tn} 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) {i1:T1...in:Tn} <: {i1:T1...im:Tm} We can use *S_RcdWidth* to drop later fields of a multi-field record while keeping earlier fields, showing for example that {*age*:*Nat*,*name*:*String*} <: {*name*:*String*,*age*:*Nat*}.

Second, we can apply subtyping inside the components of a compound record type: S1 <: T1 ... Sn <: Tn

---

(S_RcdDepth) {i1:S1...in:Sn} <: {i1:T1...in:Tn} For example, we can use *S_RcdDepth* and *S_RcdWidth* together to show that {*y*:*Student*, *x*:*Nat*} <: {*y*:*Person*}.

Third, we need to be able to reorder fields. For example, we might expect that {*name*:*String*, *gpa*:*Nat*, *age*:*Nat*} <: *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: {i1:S1...in:Sn} is a permutation of {i1:T1...in:Tn}

(S_RcdPerm) {i1:S1...in:Sn} <: {i1:T1...in:Tn}

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).

Top

Finally, it is natural to give the subtype relation a maximal 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 <: 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 |- t : S S <: T

  - ———————————— (T_Sub) Gamma |- t : T

  to the typing relation, and

- defining a subtype relation as follows: S <: U U <: T

  - ——————— (S_Trans) S <: T

    - —— (S_Refl)

    T <: T

    - ——- (S_Top)

    S <: Top

  S1 <: T1 S2 <: T2

  - —————————- (S_Prod) S1*S2 <: T1*T2

  T1 <: S1 S2 <: T2

  - —————————- (S_Arrow) S1->S2 <: T1->T2

    n > m

    (S_RcdWidth) {i1:T1...in:Tn} <: {i1:T1...im:Tm}

    S1 <: T1 ... Sn <: Tn

    (S_RcdDepth) {i1:S1...in:Sn} <: {i1:T1...in:Tn}

    {i1:S1...in:Sn} is a permutation of {i1:T1...in:Tn}

    (S_RcdPerm) {i1:S1...in:Sn} <: {i1:T1...in:Tn}

10.2.5 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.)

- $T{\rightarrow}S <: T{\rightarrow}S$
- $Top{\rightarrow}U <: S{\rightarrow}Top$

- $(C{\rightarrow}C) \rightarrow (A{\times}B)$ <: $(C{\rightarrow}C) \rightarrow (Top{\times}B)$

- $T{\rightarrow}T{\rightarrow}U$ <: $S{\rightarrow}S{\rightarrow}V$

- $(T{\rightarrow}T)$->U <: $(S{\rightarrow}S)$->V

- $((T{\rightarrow}S)$->$T)$->U <: $((S{\rightarrow}T)$->$S)$->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* → *Student*

- *Student* → *Person*

- *Student* → *Top*

- *Person* → *Student*

Write these types in order from the most specific to the most general.
Where does the type *Top*→*Top*→*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 T1, (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 T2 T2, S <: T1*T2 -> exists S1 S2, S = S1*S2 /\ S1 <: T1 /\ S2 <: T2 □

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 *S0*, *S1*, etc., such that all the *Si*'s are different and each $S(i+1)$ is a subtype of *Si*.
- There is an infinite *ascending* chain of distinct types in the subtype relation—that is, an infinite sequence of types *S0*, *S1*, etc., such that all the *Si*'s are different and each $S(i+1)$ is a supertype of *Si*.

☐

Exercise: 2 stars (proper_subtypes) Is the following statement true or false? Briefly explain your answer. forall T, ~(exists n, T = TBase n) -> exists S, S <: T /\ S <> T]] ☐

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.) empty |- (\p:T*Top. p.fst) ((\z:A.z), unit) : A->A
- What is the *largest* type *T* that makes the same assertion true?

☐

Exercise: 2 stars (small_large_2)

- What is the *smallest* type *T* that makes the following assertion true? empty |- (\p:(A->A * B->B). p) ((\z:A.z), (\z:B.z)) : T
- What is the *largest* type *T* that makes the same assertion true?

☐

Exercise: 2 stars, optional (small_large_3)

- What is the *smallest* type *T* that makes the following assertion true? a:A |- (\p:(A*T). (p.snd) (p.fst)) (a , \z:A.z) : A
- What is the *largest* type *T* that makes the same assertion true?

☐

Exercise: 2 stars (small_large_4)

- What is the *smallest* type T that makes the following assertion true? exists S, empty |- (\p:(A*T). (p.snd) (p.fst)) : S
- What is the *largest* type T that makes the same assertion true?

☐

Exercise: 2 stars (smallest_1) What is the *smallest* type T that makes the following assertion true? exists S, exists t, empty |- (\x:T. x x) t : S]] ☐

Exercise: 2 stars (smallest_2) What is the *smallest* type T that makes the following assertion true? empty |- (\x:Top. x) ((\z:A.z) , (\z:B.z)) : T]] ☐

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.
☐

10.3 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.

10.3.1 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
.

Tactic Notation "T_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "TTop" | Case_aux c "TBool"
  | Case_aux c "TBase" | Case_aux c "TArrow"
  | Case_aux c "TUnit" |
  ].

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
.

Tactic Notation "t_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp"
  | Case_aux c "tabs" | Case_aux c "ttrue"
  | Case_aux c "tfalse" | Case_aux c "tif"
  | Case_aux c "tunit"
  ].
```

Substitution

The definition of substitution remains the same as for the ordinary 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)
  | t_true :
      value ttrue
  | t_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).

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_AppAbs" | Case_aux c "ST_App1"
  | Case_aux c "ST_App2" | Case_aux c "ST_IfTrue"
  | Case_aux c "ST_IfFalse" | Case_aux c "ST_If"
  ].

Hint Constructors step.
```

10.3.2 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.

```
Inductive subtype : ty → ty → Prop :=
  | S_Refl : ∀ T,
    subtype T T
  | S_Trans : ∀ S U T,
    subtype S U →
    subtype U T →
    subtype S T
  | S_Top : ∀ S,
    subtype S TTop
  | S_Arrow : ∀ S1 S2 T1 T2,
    subtype T1 S1 →
    subtype S2 T2 →
    subtype (TArrow S1 S2) (TArrow T1 T2)
.
```

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.

Tactic Notation "subtype_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "S_Refl" | Case_aux c "S_Trans"
  | Case_aux c "S_Top" | Case_aux c "S_Arrow"
  ].
```

Module EXAMPLES.

```
Notation x := (Id 0).
Notation y := (Id 1).
Notation z := (Id 2).

Notation A := (TBase (Id 6)).
Notation B := (TBase (Id 7)).
Notation C := (TBase (Id 8)).

Notation String := (TBase (Id 9)).
Notation Float := (TBase (Id 10)).
Notation Integer := (TBase (Id 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 }

```
Definition Person : ty :=
admit.
Definition Student : ty :=
admit.
Definition Employee : ty :=
admit.

Example sub_student_person :
  subtype Student Person.
Proof.
   Admitted.

Example sub_employee_person :
  subtype Employee Person.
Proof.
   Admitted.
```

☐

```
Example subtyping_example_0 :
  subtype (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 :

```
  subtype (TArrow TTop Student)
          (TArrow (TArrow C C) Person).
Proof with eauto.
  Admitted.
```

☐

Exercise: 1 star, optional (subtyping_example_2) Example subtyping_example_2 :

```
  subtype (TArrow TTop Person)
          (TArrow Person TTop).
Proof with eauto.
  Admitted.
```

☐

```
End Examples.
```

10.3.3 Typing

The only change to the typing relation is the addition of the rule of subsumption, *T_Sub*.

```
Definition context := id → (option ty).
Definition empty : context := (fun _ ⇒ None).
Definition extend (Gamma : context) (x:id) (T : ty) :=
  fun x' ⇒ if beq_id x x' then Some T else Gamma x'.

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 (extend 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_True : ∀ Gamma,
       has_type Gamma ttrue TBool
  | T_False : ∀ Gamma,
       has_type Gamma tfalse TBool
  | T_If : ∀ t1 t2 t3 T Gamma,
       has_type Gamma t1 TBool →
       has_type Gamma t2 T →
       has_type Gamma t3 T →
```

```
      has_type Gamma (tif t1 t2 t3) T
  | T_Unit : ∀ Gamma,
      has_type Gamma tunit TUnit

  | T_Sub : ∀ Gamma t S T,
      has_type Gamma t S →
      subtype S T →
      has_type Gamma t T.
Hint Constructors has_type.
Tactic Notation "has_type_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "T_Var" | Case_aux c "T_Abs"
  | Case_aux c "T_App" | Case_aux c "T_True"
  | Case_aux c "T_False" | Case_aux c "T_If"
  | Case_aux c "T_Unit"
  | Case_aux c "T_Sub" ].
```

10.3.4 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.
```

10.4 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.

10.4.1 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* 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 : ∀ *U*,

```
    subtype U TBool →
      U = TBool.
Proof with auto.
  intros U Hs.
  remember TBool as V.
   Admitted.
```

Exercise: 3 stars, optional (sub_inversion_arrow) Lemma sub_inversion_arrow : ∀ *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.
```

□

10.4.2 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*→*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* ⊢ *t1* : *T11*→*T12* 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_types : ∀ Gamma s T1 T2,
  has_type Gamma s (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 : ∀ Gamma s,
  has_type Gamma s TBool →
  value s →
  (s = ttrue ∨ s = tfalse).
Proof with eauto.
  intros Gamma s Hty Hv.
  remember TBool as T.
  has_type_cases (induction Hty) Case; try solve by inversion...
  Case "T_Sub".
    subst. apply sub_inversion_Bool in H. subst...
Qed.
```

10.4.3 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 *empty* ⊢ *t* : *T* then *t* is a value or *t* ==> *t'* for some term *t'*.

Proof: Let *t* and *T* be given, with *empty* ⊢ *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 $t = t1\ t2$, $T = T2$, *empty* $\vdash$ *t1* : *T1* $\rightarrow$ *T2*, and *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 \x:*S1*.*s2* for some *x*, *S1*, and *s2*. But then (\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 = `if` *t1* `then` *t2* `else` *t3*, with *empty* $\vdash$ *t1* : *Bool* and with *empty* $\vdash$ *t2* : *T* and *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* = *true* or *t1* = *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 *empty* $\vdash$ *t* : *S*. The desired result is exactly the induction hypothesis for the typing subderivation.

```
Theorem progress : ∀ t T,
     has_type empty t T →
     value t ∨ ∃ t', t ==> t'.
Proof with eauto.
  intros t T Ht.
  remember empty as Gamma.
  revert HeqGamma.
  has_type_cases (induction Ht) Case;
    intros HeqGamma; subst...
  Case "T_Var".
    inversion H.
  Case "T_App".
    right.
    destruct IHHt1; subst...
    SCase "t1 is a value".
      destruct IHHt2; subst...
      SSCase "t2 is a value".
```

```
      destruct (canonical_forms_of_arrow_types empty t1 T1 T2)
        as [x [S1 [t12 Heqt1]]]...
      subst. ∃ ([x:=t2]t12)...
    SSCase "t2 steps".
      inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
  SCase "t1 steps".
    inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...
Case "T_If".
  right.
  destruct IHHt1.
  SCase "t1 is a value"...
    assert (t1 = ttrue ∨ t1 = tfalse)
      by (eapply canonical_forms_of_Bool; eauto).
    inversion H0; subst...
    inversion H. rename x into t1'. eauto.
Qed.
```

10.4.4 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* ⊢ *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* ⊢ *x*:*S1*.*t2* : *T*, then there is a type *S2* such that *Gamma*, *x*:*S1* ⊢ *t2* : *S2* and *S1* → *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* ⊢ *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* → *T12* and *Gamma*, *x*:*S1* ⊢ *t2* : *T12*. Picking *T12* for *S2* gives us what we need: *S1* → *T12* <: *S1* → *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* ⊢ *x*:*S1*.*t2* : *S*. The IH for the typing subderivation tell us that there is some type $S2$ with $S1 \to S2$ <: S and $Gamma$, $x{:}S1 \vdash t2 : S2$. Picking type $S2$ gives us what we need, since $S1 \to S2$ <: T then follows by *S_Trans*.

```
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 (extend Gamma x S1) t2 S2).
Proof with eauto.
  intros Gamma x S1 t2 T H.
  remember (tabs x S1 t2) as t.
  has_type_cases (induction H) Case;
    inversion Heqt; subst; intros; try solve by inversion.
  Case "T_Abs".
    ∃ T12...
  Case "T_Sub".
    destruct IHhas_type as [S2 [Hsub Hty]]...
  Qed.
```

Similarly...

```
Lemma typing_inversion_var : ∀ Gamma x T,
  has_type Gamma (tvar x) T →
  ∃ S ,
    Gamma x = Some S ∧ subtype S T.
Proof with eauto.
  intros Gamma x T Hty.
  remember (tvar x) as t.
  has_type_cases (induction Hty) Case; intros;
    inversion Heqt; subst; try solve by inversion.
  Case "T_Var".
    ∃ T...
  Case "T_Sub".
    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.
  has_type_cases (induction Hty) Case; intros;
    inversion Heqt; subst; try solve by inversion.
  Case "T_App".
```

```
    ∃ T1...
  Case "T_Sub".
    destruct IHHty as [U1 [Hty1 Hty2]]...
Qed.

Lemma typing_inversion_true : ∀ Gamma T,
  has_type Gamma ttrue T →
  subtype TBool T.
Proof with eauto.
  intros Gamma T Htyp. remember ttrue as tu.
  has_type_cases (induction Htyp) Case;
    inversion Heqtu; subst; intros...
Qed.

Lemma typing_inversion_false : ∀ Gamma T,
  has_type Gamma tfalse T →
  subtype TBool T.
Proof with eauto.
  intros Gamma T Htyp. remember tfalse as tu.
  has_type_cases (induction Htyp) Case;
    inversion Heqtu; subst; intros...
Qed.

Lemma typing_inversion_if : ∀ Gamma t1 t2 t3 T,
  has_type Gamma (tif t1 t2 t3) T →
  has_type Gamma t1 TBool
  ∧ has_type Gamma t2 T
  ∧ has_type Gamma t3 T.
Proof with eauto.
  intros Gamma t1 t2 t3 T Hty.
  remember (tif t1 t2 t3) as t.
  has_type_cases (induction Hty) Case; intros;
    inversion Heqt; subst; try solve by inversion.
  Case "T_If".
    auto.
  Case "T_Sub".
    destruct (IHHty H0) as [H1 [H2 H3]]...
Qed.

Lemma typing_inversion_unit : ∀ Gamma T,
  has_type Gamma tunit T →
    subtype TUnit T.
Proof with eauto.
  intros Gamma T Htyp. remember tunit as tu.
  has_type_cases (induction Htyp) Case;
```

```
    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 : ∀ x S1 s2 T1 T2,
  has_type empty (tabs x S1 s2) (TArrow T1 T2) →
     subtype T1 S1
  ∧ has_type (extend empty x S1) s2 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.
```

10.4.5 Context Invariance

The context invariance lemma follows the same pattern as in the pure 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_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.
```

```
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'.
  has_type_cases (induction H) Case;
    intros Gamma' Heqv...
  Case "T_Var".
    apply T_Var... rewrite ← Heqv...
  Case "T_Abs".
    apply T_Abs... apply IHhas_type. intros x0 Hafi.
    unfold extend. remember (beq_id x x0) as e.
    destruct e...
  Case "T_App".
    apply T_App with T1...
  Case "T_If".
    apply T_If...
Qed.

Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   has_type Gamma t T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  has_type_cases (induction Htyp) Case;
      subst; inversion Hafi; subst...
  Case "T_Abs".
    destruct (IHHtyp H4) as [T Hctx]. ∃ T.
    unfold extend in Hctx. apply not_eq_beq_id_false in H2.
    rewrite H2 in Hctx... Qed.
```

10.4.6 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 : ∀ Gamma x U v t S,
     has_type (extend Gamma x U) t S →
     has_type empty v U →
     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.
  t_cases (induction t) Case; intros; simpl.
  Case "tvar".
    rename i into y.
    destruct (typing_inversion_var _ _ _ Htypt)
        as [T [Hctx Hsub]].
    unfold extend in Hctx.
    remember (beq_id x y) as e. destruct e...
    SCase "x=y".
      apply beq_id_eq in Heqe. 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'.
  Case "tapp".
    destruct (typing_inversion_app _ _ _ _ Htypt)
        as [T1 [Htypt1 Htypt2]].
    eapply T_App...
  Case "tabs".
    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...
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      eapply context_invariance...
      apply beq_id_eq in Heqe. subst.
      intros x Hafi. unfold extend.
      destruct (beq_id y x)...
    SCase "x<>y".
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold extend.
      remember (beq_id y z) as e0. destruct e0...
      apply beq_id_eq in Heqe0. subst.
      rewrite ← Heqe...
  Case "ttrue".
      assert (subtype TBool S)
        by apply (typing_inversion_true _ _ Htypt)...
```

```
  Case "tfalse".
      assert (subtype TBool S)
        by apply (typing_inversion_false _ _ Htypt)...
  Case "tif".
    assert (has_type (extend Gamma x U) t1 TBool
            ∧ has_type (extend Gamma x U) t2 S
            ∧ has_type (extend Gamma x U) t3 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.
  Case "tunit".
    assert (subtype TUnit S)
      by apply (typing_inversion_unit _ _ Htypt)...
Qed.
```

10.4.7 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 *empty* $\vdash$ *t* : *T* and *t* ==> *t'*, then *empty* $\vdash$ *t'* : *T*.

Proof: Let *t* and *T* be given such that *empty* $\vdash$ *t* : *T*. We go by induction on the structure of this typing derivation, leaving *t'* general. The cases *T_Abs*, *T_Unit*, *T_True*, and *T_False* cases are vacuous because abstractions and constants 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*, *empty* $\vdash$ *t1* : *T1* $\to$ *T2* and *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* = \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 *empty* $\vdash$ [*x*:=*t2*] *t12* : *T2* as desired.

  - If the final step of the derivation uses rule *T_If*, then there are terms *t1*, *t2*, and *t3* such that *t* = `if` *t1* `then` *t2* `else` *t3*, with *empty* $\vdash$ *t1* : *Bool* and with *empty* $\vdash$ *t2* : *T* and *empty* $\vdash$ *t3* : *T*. Moreover, by the induction hypothesis, if *t1* steps to *t1'* then *empty* ⊢ *t1'* : *Bool*. There are three cases to consider, depending on which rule was used to show *t* ==> *t'*.

  * If *t* ==> *t'* by rule *ST_If*, then *t'* = `if` *t1'* `then` *t2* `else` *t3* with *t1* ==> *t1'*. By the induction hypothesis, *empty* ⊢ *t1'* : *Bool*, and so *empty* ⊢ *t'* : *T* by *T_If*.
  * If *t* ==> *t'* by rule *ST_IfTrue* or *ST_IfFalse*, then either *t'* = *t2* or *t'* = *t3*, and *empty* ⊢ *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 *empty* ⊢ *t* : *S*. The result is immediate by the induction hypothesis for the typing subderivation and an application of *T_Sub*. □

```
Theorem preservation : ∀ t t' T,
     has_type empty t T →
     t ==> t' →
     has_type empty t' T.
Proof with eauto.
  intros t t' T HT.
  remember empty as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  has_type_cases (induction HT) Case;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  Case "T_App".
    inversion HE; subst...
    SCase "ST_AppAbs".
      destruct (abs_arrow _ _ _ _ _ HT1) as [HA1 HA2].
      apply substitution_preserves_typing with T...
Qed.
```

10.4.8 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:Nat, b:Nat} ----> {Nat,Nat}       i.e. (Nat,(Nat,Top))
{c:Nat, a:Nat} ----> {Nat,Top,Nat}   i.e. (Nat,(Top,(Nat,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.

10.4.9 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 |- t : S1->S2 S1 <: T1 T1 <: S1 S2 <: T2

  – ———————————————- (T_Funny1) Gamma |- t : T1->T2

- Suppose we add the following reduction rule:

  – ——————— (ST_Funny21)

  unit ==> (\x:Top. x)

- Suppose we add the following subtyping rule:

  – —————- (S_Funny3)

  Unit <: Top->Top

- Suppose we add the following subtyping rule:

  – —————- (S_Funny4)

  Top->Top <: Unit

- Suppose we add the following evaluation rule:

  – ——————- (ST_Funny5)

  (unit t) ==> (t unit)

- Suppose we add the same evaluation rule *and* a new typing rule:

- ————————- (ST_Funny5)

(unit t) ==> (t unit)

- ——————————— (T_Funny6)

empty |- Unit : Top->Top

- Suppose we *change* the arrow subtyping rule to: S1 <: T1 S2 <: T2

  - ———————————- (S_Arrow') S1->S2 <: T1->T2

☐

10.5 Exercise: Adding Products

Exercise: 4 stars, optional (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 well-formedness relation in the obvious way.
- Extend the operational semantics with the same reduction rules as in the last chapter.
- Extend the subtyping relation with this rule: S1 <: T1 S2 <: T2

  - —————————— (Sub_Prod) S1 * S2 <: T1 * T2

- Extend the typing relation with the same rules for pairs and projections as in the last chapter.
- 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.) ☐

Chapter 11

Library References

11.1 References: Typing Mutable References

```
Require Export 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.

11.1.1 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.

11.1.2 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 *Ref* T is the type of references which point to a cell holding values of type T.

```
T ::= Nat
    | Unit
    | T -> T
    | 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 ::= ...              Terms
    | 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...

- *ref t* (formally, *tref t*) allocates a new reference cell with the value *t* and evaluates to the location of the newly allocated cell;
- !*t* (formally, *tderef t*) evaluates to the contents of the cell referenced by *t*;
- *t1* := *t2* (formally, *tassign t1 t2*) assigns *t2* to the cell referenced by *t1*; and
- *l* (formally, *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.

```
Tactic Notation "t_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp"
  | Case_aux c "tabs" | Case_aux c "tzero"
  | Case_aux c "tsucc" | Case_aux c "tpred"
  | Case_aux c "tmult" | Case_aux c "tif0"
  | Case_aux c "tunit" | Case_aux c "tref"
  | Case_aux c "tderef" | Case_aux c "tassign"
  | Case_aux c "tloc" ].
```

```
Module ExampleVariables.
```

```
Definition x := Id 0.
Definition y := Id 1.
Definition r := Id 2.
Definition s := Id 3.
```

```
End ExampleVariables.
```

Typing (Preview)

Informally, the typing rules for allocation, dereferencing, and assignment will look like this: Gamma |- t1 : T1

(T_Ref) Gamma |- ref t1 : Ref T1
Gamma |- t1 : Ref T11

(T_Deref) Gamma |- !t1 : T11
Gamma |- t1 : Ref T11 Gamma |- t2 : T11

(T_Assign) Gamma |- t1 := t2 : 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 → Prop :=
  | v_abs : ∀ x T t,
      value (tabs x T t)
  | v_nat : ∀ n,
```

```
      value (tnat n)
  | v_unit :
      value tunit
  | v_loc : ∀ 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).
```

11.2 Pragmatics

11.2.1 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 (Id 0) TUnit t2) t1.
```

11.2.2 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!

11.2.3 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.

11.2.4 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 `let`s have finished and the third one is about to begin.

☐

11.2.5 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?

☐

11.2.6 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*).

11.2.7 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.

□

11.3 Operational Semantics

11.3.1 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*?" 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 *Float*s, 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.

11.3.2 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 *tref 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 add a new reference cell to the store, we use *snoc*.

```
Fixpoint snoc {A:Type} (l:list A) (x:A) : list A :=
  match l with
  | nil ⇒ x :: nil
  | h :: t ⇒ h :: snoc t x
  end.
```

We will need some boring lemmas about *snoc*. The proofs are routine inductions.

```
Lemma length_snoc : ∀ A (l:list A) x,
  length (snoc l x) = S (length l).
Proof.
  induction l; intros; [ auto | simpl; rewrite IHl; auto ]. Qed.

Lemma nth_lt_snoc : ∀ A (l:list A) x d n,
  n < length l →
  nth n l d = nth n (snoc l x) d.
Proof.
  induction l as [|a l']; intros; try solve by inversion.
  Case "l = a :: l'".
    destruct n; auto.
    simpl. apply IHl'.
    simpl in H. apply lt_S_n in H. assumption.
Qed.

Lemma nth_eq_snoc : ∀ A (l:list A) x d,
  nth (length l) (snoc l x) d = x.
Proof.
  induction l; intros; [ auto | simpl; rewrite IHl; auto ].
Qed.
```

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
    | O ⇒ 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 : ∀ A n (x:A),
  replace n x nil = nil.
Proof.
  destruct n; auto.
Qed.

Lemma length_replace : ∀ 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 : ∀ l t st,
  l < length st →
  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.
  Case "st = []".
   inversion Hlen.
  Case "st = t' :: st'".
    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.
  Case "l1 = 0".
    destruct st.
    SCase "st = []". rewrite replace_nil...
    SCase "st = _:: _". destruct l2... contradict Hneq...
  Case "l1 = S l1'".
    destruct st as [|t2 st2].
    SCase "st = []". destruct l2...
    SCase "st = t2 :: st2".
      destruct l2...
      simpl; apply IHl1'...
Qed.
```

11.3.3 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

(ST_AppAbs) (\x:T.t12) v2 / st ==> *x:=v2*t12 / st
t1 / st ==> t1' / st'

(ST_App1) t1 t2 / st ==> t1' t2 / st'
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 ==> t1' / st'

(ST_Deref) !t1 / st ==> !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 derefence 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 ==> 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 ==> t1' / st'

(ST_Assign1) t1 := t2 / st ==> t1' := t2 / st'

t2 / st ==> t2' / st'

(ST_Assign2) v1 := t2 / st ==> 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 ==> 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 ==> t1' / st'

(ST_Ref) ref t1 / st ==> 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) ref v1 / st ==> loc |st| / st,v1 The value resulting from this step is the newly allocated location itself. (Formally, *st,v1* means *snoc st v1*.)

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 '/' st1 '==>' t2 '/' st2"
  (at level 40, st1 at level 39, t2 at level 39).

Inductive step : tm × store → tm × store → Prop :=
  | ST_AppAbs : ∀ x T t12 v2 st,
          value v2 →
          tapp (tabs x T t12) v2 / st ==> [x:=v2]t12 / st
  | ST_App1 : ∀ t1 t1' t2 st st',
          t1 / st ==> t1' / st' →
          tapp t1 t2 / st ==> tapp t1' t2 / st'
  | ST_App2 : ∀ v1 t2 t2' st st',
          value v1 →
          t2 / st ==> t2' / st' →
          tapp v1 t2 / st ==> tapp v1 t2'/ st'
  | ST_SuccNat : ∀ n st,
          tsucc (tnat n) / st ==> tnat (S n) / st
  | ST_Succ : ∀ t1 t1' st st',
          t1 / st ==> t1' / st' →
          tsucc t1 / st ==> tsucc t1' / st'
  | ST_PredNat : ∀ n st,
          tpred (tnat n) / st ==> tnat (pred n) / st
  | ST_Pred : ∀ t1 t1' st st',
          t1 / st ==> t1' / st' →
          tpred t1 / st ==> tpred t1' / st'
  | ST_MultNats : ∀ n1 n2 st,
          tmult (tnat n1) (tnat n2) / st ==> tnat (mult n1 n2) / st
  | ST_Mult1 : ∀ t1 t2 t1' st st',
          t1 / st ==> t1' / st' →
          tmult t1 t2 / st ==> tmult t1' t2 / st'
  | ST_Mult2 : ∀ v1 t2 t2' st st',
          value v1 →
          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) / snoc st v1
  | 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)).

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_AppAbs" | Case_aux c "ST_App1"
  | Case_aux c "ST_App2" | Case_aux c "ST_SuccNat"
  | Case_aux c "ST_Succ" | Case_aux c "ST_PredNat"
  | Case_aux c "ST_Pred" | Case_aux c "ST_MultNats"
  | Case_aux c "ST_Mult1" | Case_aux c "ST_Mult2"
  | Case_aux c "ST_If0" | Case_aux c "ST_If0_Zero"
  | Case_aux c "ST_If0_Nonzero" | Case_aux c "ST_RefValue"
  | Case_aux c "ST_Ref" | Case_aux c "ST_DerefLoc"
  | Case_aux c "ST_Deref" | Case_aux c "ST_Assign"
  | Case_aux c "ST_Assign1" | Case_aux c "ST_Assign2" ].

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).
```

11.4 Typing

Our contexts for free variables will be exactly the same as for the STLC, partial maps from identifiers to types.

```
Definition context := partial_map ty.
```

11.4.1 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 !(*loc* 1) in the store [*unit*, *unit*], the result is *unit*; if we evaluate the same term in the store [*unit*, \\$x$:*Unit.x*], the result is \\$x$:*Unit.x*. With respect to the former store, the location 1 has type *Unit*, and with respect to the latter it has type *Unit*→*Unit*. This observation leads us immediately to a first attempt at a typing rule for locations: Gamma |- lookup l st : T1

Gamma |- loc l : 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 *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 |- lookup l st : T1

Gamma; st |- loc l : 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:

```
[\x:Nat. (!(loc 1)) x, \x:Nat. (!(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 store_ty := list ty.
```

The *store_Tlookup* function retrieves the type at a particular index.

```
Definition store_Tlookup (n:nat) (ST:store_ty) :=
  nth n ST 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 !(*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 |- loc l : Ref (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.

11.4.2 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*):

l < |ST|

(T_Loc) Gamma; ST |- loc l : Ref (lookup l ST)

Gamma; ST |- t1 : T1

(T_Ref) Gamma; ST |- ref t1 : Ref T1

Gamma; ST |- t1 : Ref T11

(T_Deref) Gamma; ST |- !t1 : T11

Gamma; ST |- t1 : Ref T11 Gamma; ST |- t2 : T11

(T_Assign) Gamma; ST |- t1 := t2 : Unit

```
Inductive has_type : context → store_ty → tm → ty → Prop :=
  | T_Var : ∀ Gamma ST x T,
      Gamma x = Some T →
      has_type Gamma ST (tvar x) T
  | T_Abs : ∀ Gamma ST x T11 T12 t12,
      has_type (extend Gamma x T11) ST t12 T12 →
      has_type Gamma ST (tabs x T11 t12) (TArrow T11 T12)
  | T_App : ∀ T1 T2 Gamma ST t1 t2,
      has_type Gamma ST t1 (TArrow T1 T2) →
      has_type Gamma ST t2 T1 →
      has_type Gamma ST (tapp t1 t2) T2
  | T_Nat : ∀ Gamma ST n,
      has_type Gamma ST (tnat n) TNat
  | T_Succ : ∀ Gamma ST t1,
      has_type Gamma ST t1 TNat →
      has_type Gamma ST (tsucc t1) TNat
  | T_Pred : ∀ Gamma ST t1,
      has_type Gamma ST t1 TNat →
      has_type Gamma ST (tpred t1) TNat
  | T_Mult : ∀ Gamma ST t1 t2,
      has_type Gamma ST t1 TNat →
```

```
      has_type Gamma ST t2 TNat →
      has_type Gamma ST (tmult t1 t2) TNat
  | T_If0 : ∀ Gamma ST t1 t2 t3 T,
      has_type Gamma ST t1 TNat →
      has_type Gamma ST t2 T →
      has_type Gamma ST t3 T →
      has_type Gamma ST (tif0 t1 t2 t3) T
  | T_Unit : ∀ Gamma ST,
      has_type Gamma ST tunit TUnit
  | T_Loc : ∀ Gamma ST l,
      l < length ST →
      has_type Gamma ST (tloc l) (TRef (store_Tlookup l ST))
  | T_Ref : ∀ Gamma ST t1 T1,
      has_type Gamma ST t1 T1 →
      has_type Gamma ST (tref t1) (TRef T1)
  | T_Deref : ∀ Gamma ST t1 T11,
      has_type Gamma ST t1 (TRef T11) →
      has_type Gamma ST (tderef t1) T11
  | T_Assign : ∀ Gamma ST t1 t2 T11,
      has_type Gamma ST t1 (TRef T11) →
      has_type Gamma ST t2 T11 →
      has_type Gamma ST (tassign t1 t2) TUnit.

Hint Constructors has_type.

Tactic Notation "has_type_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "T_Var" | Case_aux c "T_Abs" | Case_aux c "T_App"
  | Case_aux c "T_Nat" | Case_aux c "T_Succ" | Case_aux c "T_Pred"
  | Case_aux c "T_Mult" | Case_aux c "T_If0"
  | Case_aux c "T_Unit" | Case_aux c "T_Loc"
  | Case_aux c "T_Ref" | Case_aux c "T_Deref"
  | Case_aux c "T_Assign" ].
```

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 $Gamma \vdash t : T$, then we can replace the free variables in t with values of the types listed in $Gamma$ 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.

11.5 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.

11.5.1 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 : ∀ ST T t st t' st',
  has_type empty ST t T →
  t / st ==> t' / st' →
  has_type empty ST t' T.
Admitted.
```

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) :=
  length ST = length st ∧
  (∀ l, l < length st →
     has_type empty ST (store_lookup l st) (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* ⊢ *st* and *ST2* ⊢ *st*?

□

We can now state something closer to the desired preservation property:

```
Theorem preservation_wrong2 : ∀ ST T t st t' st',
  has_type empty ST t T →
  t / st ==> t' / st' →
  store_well_typed ST st →
  has_type empty ST t' T.
Admitted.
```

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*.

11.5.2 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 → store_ty → Prop :=
  | extends_nil : ∀ ST',
      extends ST' nil
  | extends_cons : ∀ x ST' ST,
      extends ST' ST →
      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 : ∀ l ST ST',
  l < length ST →
  extends ST' ST →
  store_Tlookup l ST' = 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'.
  Case "nil". inversion Hlen.
  Case "cons". unfold store_Tlookup in *.
```

```
    destruct ST'.
    SCase "ST' = nil". inversion HST'.
    SCase "ST' = a' :: ST'2".
      inversion HST'; subst.
      destruct l as [|l'].
      SSCase "l = 0"...
      SSCase "l = S l'". 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, *snoc ST T* extends *ST*, and *extends* is reflexive.

```
Lemma extends_snoc : ∀ ST T,
  extends (snoc ST T) ST.
Proof with auto.
  induction ST; intros T...
  simpl...
Qed.

Lemma extends_refl : ∀ ST,
  extends ST ST.
Proof.
  induction ST; auto.
Qed.
```

11.5.3 Preservation, Finally

We can now give the final, correct statement of the type preservation property:

```
Definition preservation_theorem := ∀ ST t t' T st st',
  has_type empty ST t T →
  store_well_typed ST st →
  t / st ==> t' / st' →
```

```
∃ ST',
  (extends ST' ST ∧
   has_type empty ST' t' 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 *snoc ST T1*, where *T1* is the type of the value *v1* in the extended store *snoc st v1*, 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.

11.5.4 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).

Tactic Notation "afi_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "afi_var"
  | Case_aux c "afi_app1" | Case_aux c "afi_app2" | Case_aux c "afi_abs"
  | Case_aux c "afi_succ" | Case_aux c "afi_pred"
  | Case_aux c "afi_mult1" | Case_aux c "afi_mult2"
  | Case_aux c "afi_if0_1" | Case_aux c "afi_if0_2" | Case_aux c "afi_if0_3"
  | Case_aux c "afi_ref" | Case_aux c "afi_deref"
  | Case_aux c "afi_assign1" | Case_aux c "afi_assign2" ].

Hint Constructors appears_free_in.

Lemma free_in_context : ∀ x t T Gamma ST,
   appears_free_in x t →
   has_type Gamma ST t T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros. generalize dependent Gamma. generalize dependent T.
  afi_cases (induction H) Case;
        intros; (try solve [ inversion H0; subst; eauto ]).
  Case "afi_abs".
    inversion H1; subst.
    apply IHappears_free_in in H8.
    apply not_eq_beq_id_false in H.
    rewrite extend_neq in H8; assumption.
```

```
Qed.

Lemma context_invariance : ∀ Gamma Gamma' ST t T,
  has_type Gamma ST t T →
  (∀ x, appears_free_in x t → Gamma x = Gamma' x) →
  has_type Gamma' ST t T.
Proof with eauto.
  intros.
  generalize dependent Gamma'.
  has_type_cases (induction H) Case; intros...
  Case "T_Var".
    apply T_Var. symmetry. rewrite ← H...
  Case "T_Abs".
    apply T_Abs. apply IHhas_type; intros.
    unfold extend.
    remember (beq_id x x0) as e; destruct e...
    apply H0. apply afi_abs. apply beq_id_false_not_eq... auto.
  Case "T_App".
    eapply T_App.
      apply IHhas_type1...
      apply IHhas_type2...
  Case "T_Mult".
    eapply T_Mult.
      apply IHhas_type1...
      apply IHhas_type2...
  Case "T_If0".
    eapply T_If0.
      apply IHhas_type1...
      apply IHhas_type2...
      apply IHhas_type3...
  Case "T_Assign".
    eapply T_Assign.
      apply IHhas_type1...
      apply IHhas_type2...
Qed.

Lemma substitution_preserves_typing : ∀ Gamma ST x s S t T,
  has_type empty ST s S →
  has_type (extend Gamma x S) ST t T →
  has_type Gamma ST ([x:=s]t) T.
Proof with eauto.
  intros Gamma ST x s S t T Hs Ht.
  generalize dependent Gamma. generalize dependent T.
  t_cases (induction t) Case; intros T Gamma H;
```

```
      inversion H; subst; simpl...
    Case "tvar".
      rename i into y.
      remember (beq_id x y) as eq; destruct eq; subst.
      SCase "x = y".
        apply beq_id_eq in Heqeq; subst.
        rewrite extend_eq in H3.
        inversion H3; subst.
        eapply context_invariance...
        intros x Hcontra.
        destruct (free_in_context _ _ _ _ _ _ Hcontra Hs) as [T' HT'].
        inversion HT'.
      SCase "x <> y".
        apply T_Var.
        rewrite extend_neq in H3...
    Case "tabs". subst.
      rename i into y.
      remember (beq_id x y) as eq; destruct eq.
      SCase "x = y".
        apply beq_id_eq in Heqeq; subst.
        apply T_Abs. eapply context_invariance...
        intros. apply extend_shadow.
      SCase "x <> x0".
        apply T_Abs. apply IHt.
        eapply context_invariance...
        intros. unfold extend.
        remember (beq_id y x0) as e. destruct e...
        apply beq_id_eq in Heqe; subst.
        rewrite <- Heqeq...
Qed.
```

11.5.5 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 : ∀ ST st l t,
  l < length st →
  store_well_typed ST st →
  has_type empty ST t (store_Tlookup l ST) →
  store_well_typed ST (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'.
  remember (beq_nat l' l) as ll'; destruct ll'.
  Case "l' = l".
    apply beq_nat_eq in Heqll'; subst.
    rewrite lookup_replace_eq...
  Case "l' <> l".
    symmetry in Heqll'; apply beq_nat_false in Heqll'.
    rewrite lookup_replace_neq...
    rewrite length_replace in Hl'.
    apply H0...
Qed.
```

11.5.6 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 : ∀ Gamma ST ST' t T,
  extends ST' ST →
  has_type Gamma ST t T →
  has_type Gamma ST' t T.
Proof with eauto.
  intros. has_type_cases (induction H0) Case; eauto.
  Case "T_Loc".
    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_snoc : ∀ ST st t1 T1,
  store_well_typed ST st →
  has_type empty ST t1 T1 →
  store_well_typed (snoc ST T1) (snoc st t1).
Proof with auto.
```

```
  intros.
  unfold store_well_typed in *.
  inversion H as [Hlen Hmatch]; clear H.
  rewrite !length_snoc.
  split...
  Case "types match.".
    intros l Hl.
    unfold store_lookup, store_Tlookup.
    apply le_lt_eq_dec in Hl; inversion Hl as [Hlt | Heq].
    SCase "l < length st".
      apply lt_S_n in Hlt.
      rewrite <- !nth_lt_snoc...
      apply store_weakening with ST. apply extends_snoc.
      apply Hmatch...
      rewrite Hlen...
    SCase "l = length st".
      inversion Heq.
      rewrite nth_eq_snoc.
      rewrite <- Hlen. rewrite nth_eq_snoc...
      apply store_weakening with ST... apply extends_snoc.
Qed.
```

11.5.7 Preservation!

Now that we've got everything set up right, the proof of preservation is actually quite straightforward.

```
Theorem preservation : ∀ ST t t' T st st',
  has_type empty ST t T →
  store_well_typed ST st →
  t / st ==> t' / st' →
  ∃ ST',
    (extends ST' ST ∧
     has_type empty ST' t' 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'.
  has_type_cases (induction Ht) Case; intros t' HST Hstep;
    subst; try (solve by inversion); inversion Hstep; subst;
    try (eauto using store_weakening, extends_refl).
  Case "T_App".
```

```
  SCase "ST_AppAbs". ∃ ST.
    inversion Ht1; subst.
    split; try split... eapply substitution_preserves_typing...
  SCase "ST_App1".
    eapply IHHt1 in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  SCase "ST_App2".
    eapply IHHt2 in H5...
    inversion H5 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
Case "T_Succ".
  SCase "ST_Succ".
    eapply IHHt in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
Case "T_Pred".
  SCase "ST_Pred".
    eapply IHHt in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
Case "T_Mult".
  SCase "ST_Mult1".
    eapply IHHt1 in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
  SCase "ST_Mult2".
    eapply IHHt2 in H5...
    inversion H5 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'...
Case "T_If0".
  SCase "ST_If0_1".
    eapply IHHt1 in H0...
    inversion H0 as [ST' [Hext [Hty Hsty]]].
    ∃ ST'... split...
Case "T_Ref".
  SCase "ST_RefValue".
    ∃ (snoc ST T1).
    inversion HST; subst.
    split.
      apply extends_snoc.
    split.
```

```
        replace (TRef T1)
          with (TRef (store_Tlookup (length st) (snoc ST T1))).
        apply T_Loc.
        rewrite ← H. rewrite length_snoc. omega.
        unfold store_Tlookup. rewrite ← H. rewrite nth_eq_snoc...
        apply store_well_typed_snoc; assumption.
    SCase "ST_Ref".
      eapply IHHt in H0...
      inversion H0 as [ST' [Hext [Hty Hsty]]].
      ∃ ST'...
  Case "T_Deref".
    SCase "ST_DerefLoc".
      ∃ ST. split; try split...
      inversion HST as [_ Hsty].
      replace T11 with (store_Tlookup l ST).
      apply Hsty...
      inversion Ht; subst...
    SCase "ST_Deref".
      eapply IHHt in H0...
      inversion H0 as [ST' [Hext [Hty Hsty]]].
      ∃ ST'...
  Case "T_Assign".
    SCase "ST_Assign".
      ∃ ST. split; try split...
      eapply assign_pres_store_typing...
      inversion Ht1; subst...
    SCase "ST_Assign1".
      eapply IHHt1 in H0...
      inversion H0 as [ST' [Hext [Hty Hsty]]].
      ∃ ST'...
    SCase "ST_Assign2".
      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.

☐

11.5.8 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 : ∀ ST t T st,
  has_type empty ST t T →
  store_well_typed ST st →
  (value t ∨ ∃ t', ∃ st', t / st ==> t' / st').
Proof with eauto.
    intros ST t T st Ht HST. remember (@empty ty) as Gamma.
  has_type_cases (induction Ht) Case; subst; try solve by inversion...
  Case "T_App".
    right. destruct IHHt1 as [Ht1p | Ht1p]...
    SCase "t1 is a value".
      inversion Ht1p; subst; try solve by inversion.
      destruct IHHt2 as [Ht2p | Ht2p]...
      SSCase "t2 steps".
        inversion Ht2p as [t2' [st' Hstep]].
        ∃ (tapp (tabs x T t) t2'). ∃ st'...
    SCase "t1 steps".
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tapp t1' t2). ∃ st'...
  Case "T_Succ".
    right. destruct IHHt as [Ht1p | Ht1p]...
    SCase "t1 is a value".
      inversion Ht1p; subst; try solve [ inversion Ht ].
      SSCase "t1 is a tnat".
        ∃ (tnat (S n)). ∃ st...
    SCase "t1 steps".
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tsucc t1'). ∃ st'...
  Case "T_Pred".
    right. destruct IHHt as [Ht1p | Ht1p]...
    SCase "t1 is a value".
      inversion Ht1p; subst; try solve [inversion Ht ].
      SSCase "t1 is a tnat".
        ∃ (tnat (pred n)). ∃ st...
    SCase "t1 steps".
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tpred t1'). ∃ st'...
  Case "T_Mult".
    right. destruct IHHt1 as [Ht1p | Ht1p]...
    SCase "t1 is a value".
```

```
      inversion Ht1p; subst; try solve [inversion Ht1].
      destruct IHHt2 as [Ht2p | Ht2p]...
      SSCase "t2 is a value".
        inversion Ht2p; subst; try solve [inversion Ht2].
        ∃ (tnat (mult n n0)). ∃ st...
      SSCase "t2 steps".
        inversion Ht2p as [t2' [st' Hstep]].
        ∃ (tmult (tnat n) t2'). ∃ st'...
    SCase "t1 steps".
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tmult t1' t2). ∃ st'...
  Case "T_If0".
    right. destruct IHHt1 as [Ht1p | Ht1p]...
    SCase "t1 is a value".
      inversion Ht1p; subst; try solve [inversion Ht1].
      destruct n.
      SSCase "n = 0". ∃ t2. ∃ st...
      SSCase "n = S n'". ∃ t3. ∃ st...
    SCase "t1 steps".
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tif0 t1' t2 t3). ∃ st'...
  Case "T_Ref".
    right. destruct IHHt as [Ht1p | Ht1p]...
    SCase "t1 steps".
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tref t1'). ∃ st'...
  Case "T_Deref".
    right. destruct IHHt as [Ht1p | Ht1p]...
    SCase "t1 is a value".
      inversion Ht1p; subst; try solve by inversion.
      eexists. eexists. apply ST_DerefLoc...
      inversion Ht; subst. inversion HST; subst.
      rewrite ← H...
    SCase "t1 steps".
      inversion Ht1p as [t1' [st' Hstep]].
      ∃ (tderef t1'). ∃ st'...
  Case "T_Assign".
    right. destruct IHHt1 as [Ht1p|Ht1p]...
    SCase "t1 is a value".
      destruct IHHt2 as [Ht2p|Ht2p]...
      SSCase "t2 is a value".
        inversion Ht1p; subst; try solve by inversion.
```

```
      eexists. eexists. apply ST_Assign...
      inversion HST; subst. inversion Ht1; subst.
      rewrite H in H5...
    SSCase "t2 steps".
      inversion Ht2p as [t2' [st' Hstep]].
      ∃ (tassign t1 t2'). ∃ st'...
  SCase "t1 steps".
    inversion Ht1p as [t1' [st' Hstep]].
    ∃ (tassign t1' t2). ∃ st'...
Qed.
```

11.6 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* (*x*:*Unit.unit*) creates a reference to a cell of type *Unit* → *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 (\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 : ∃ T, has_type empty [] loop 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 extend. simpl. reflexivity. auto.
  eapply T_Assign.
    eapply T_Var. unfold extend. 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 ==>+. This is just like the reflexive step closure of single-step reduction (which we're been writing ==>*), except that it is not reflexive: *t* ==>+ *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 : has_type empty nil factorial (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.

☐

11.7 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.
```

Chapter 12

Library Records

12.1 Records: Adding Records to STLC

```
Require Export Stlc.
```

12.2 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, ..., in=tn}         record
    | t.i                         projection

v ::=                          Values:
    | ...
    | {i1=v1, ..., in=vn}         record value

T ::=                          Types:
    | ...
    | {i1:T1, ..., in:Tn}         record type
```

Reduction: ti ==> ti' (ST_Rcd)

{i1=v1, ..., im=vm, in=tn, ...} ==> {i1=v1, ..., im=vm, in=tn', ...}
t1 ==> t1'

(ST_Proj1) t1.i ==> t1'.i

(ST_ProjRcd) {..., i=vi, ...}.i ==> vi Typing: Gamma |- t1 : T1 ... Gamma |- tn : Tn

---

(T_Rcd) Gamma |- {i1=t1, ..., in=tn} : {i1:T1, ..., in:Tn}
Gamma |- t : {..., i:Ti, ...}

(T_Proj) Gamma |- t.i : Ti

12.3 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 : `Type`) := **list** (**id** × X).

`Inductive` **ty** : `Type` :=
  | TBase : **id** → **ty**
  | TArrow : **ty** → **ty** → **ty**
  | TRcd : (alist **ty**) → **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**.

```
Tactic Notation "T_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "TBase" | Case_aux c "TArrow"
  | Case_aux c "TRNil" | Case_aux c "TRCons" ].
```

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.
```

```
Tactic Notation "t_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp" | Case_aux c "tabs"
  | Case_aux c "tproj" | Case_aux c "trnil" | Case_aux c "trcons" ].
```

Some variables, for examples...

```
Notation a := (Id 0).
Notation f := (Id 1).
Notation g := (Id 2).
Notation l := (Id 3).
Notation A := (TBase (Id 4)).
Notation B := (TBase (Id 5)).
Notation k := (Id 6).
Notation i1 := (Id 7).
Notation i2 := (Id 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 rules 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).

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_AppAbs" | Case_aux c "ST_App1" | Case_aux c "ST_App2"
  | Case_aux c "ST_Proj1" | Case_aux c "ST_ProjRcd"
  | Case_aux c "ST_Rcd_Head" | Case_aux c "ST_Rcd_Tail" ].

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.

```
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 (extend 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 Ti Tr,
      has_type Gamma t Tr →
      Tlookup i Tr = Some Ti →
      has_type Gamma (tproj t i) Ti
  | 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).

Hint Constructors has_type.

Tactic Notation "has_type_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "T_Var" | Case_aux c "T_Abs" | Case_aux c "T_App"
```

| *Case_aux* *c* "T_Proj" | *Case_aux* *c* "T_RNil" | *Case_aux* *c* "T_RCons"].

12.3.1 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 :
  has_type empty
    (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)))
    (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,
      has_type (extend empty a (TRCons i2 (TArrow A A)
                                    TRNil))
               (trcons i1 (tabs a B (tvar a)) (tvar a))
               T.
Proof.
  Admitted.
```

```
Example typing_nonexample_2 : ∀ y,
  ¬ ∃ T,
    has_type (extend empty y A)
           (tapp (tabs a (TRCons i1 A TRNil)
                     (tproj (tvar a) i1))
                   (trcons i1 (tvar y) (trcons i2 (tvar y) trnil)))
           T.
Proof.
  Admitted.
```

12.3.2 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.
  T_cases (induction T) Case; intros; try solve by inversion.
  Case "TRCons".
    inversion H. subst. unfold Tlookup in H0.
    remember (beq_id i i0) as b. destruct b; subst...
    inversion H0. subst... Qed.

Lemma step_preserves_record_tm : ∀ tr tr',
  record_tm tr →
  tr ==> tr' →
  record_tm tr'.
Proof.
  intros tr tr' Hrt Hstp.
  inversion Hrt; subst; inversion Hstp; subst; auto.
Qed.

Lemma has_type__wf : ∀ Gamma t T,
  has_type Gamma t T → well_formed_ty T.
Proof with eauto.
  intros Gamma t T Htyp.
  has_type_cases (induction Htyp) Case...
  Case "T_App".
    inversion IHHtyp1...
  Case "T_Proj".
    eapply wf_rcd_lookup...
Qed.
```

Field Lookup

Lemma: If *empty* $\vdash v : T$ and *Tlookup i T* returns *Some Ti*, then *tlookup i v* returns *Some ti* for some term *ti* such that *has_type empty ti Ti*.

Proof: By induction on the typing derivation *Htyp*. Since *Tlookup* i T = *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 = *trcons i0 t tr* and T = *TRCons i0 T Tr* for some *i0*, t, *tr*, T and *Tr*.

This leaves two possiblities to consider - either *i0* = i or not.

- If i = *i0*, then since *Tlookup* i (*TRCons i0 T Tr*) = *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 = Tlookup i Tr and tlookup i t = tlookup i tr, so the result follows from the induction hypothesis. □

```
Lemma lookup_field_in_value : ∀ v T i Ti,
  value v →
  has_type empty v T →
  Tlookup i T = Some Ti →
  ∃ ti, tlookup i v = Some ti ∧ has_type empty ti Ti.
Proof with eauto.
  intros v T i Ti Hval Htyp Hget.
  remember (@empty ty) as Gamma.
  has_type_cases (induction Htyp) Case; subst; try solve by inversion...
  Case "T_RCons".
    simpl in Hget. simpl. destruct (beq_id i i0).
    SCase "i is first".
      simpl. inversion Hget. subst.
      ∃ t...
    SCase "get tail".
      destruct IHHtyp2 as [vi [Hgeti Htypi]]...
      inversion Hval... Qed.
```

Progress

```
Theorem progress : ∀ t T,
     has_type empty t T →
     value t ∨ ∃ t', t ==> t'.
Proof with eauto.
  intros t T Ht.
  remember (@empty ty) as Gamma.
  generalize dependent HeqGamma.
  has_type_cases (induction Ht) Case; intros HeqGamma; subst.
  Case "T_Var".
    inversion H.
```

```
Case "T_Abs".
  left...
Case "T_App".
  right.
  destruct IHHt1; subst...
  SCase "t1 is a value".
    destruct IHHt2; subst...
    SSCase "t2 is a value".
      inversion H; subst; try (solve by inversion).
      ∃ ([x:=t2]t12)...
    SSCase "t2 steps".
      destruct H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
  SCase "t1 steps".
    destruct H as [t1' Hstp]. ∃ (tapp t1' t2)...
Case "T_Proj".
  right. destruct IHHt...
  SCase "rcd is value".
    destruct (lookup_field_in_value _ _ _ _ H0 Ht H) as [ti [Hlkup _]].
    ∃ ti...
  SCase "rcd_steps".
    destruct H0 as [t' Hstp]. ∃ (tproj t' i)...
Case "T_RNil".
  left...
Case "T_RCons".
  destruct IHHt1...
  SCase "head is a value".
    destruct IHHt2; try reflexivity.
    SSCase "tail is a value".
      left...
    SSCase "tail steps".
      right. destruct H2 as [tr' Hstp].
      ∃ (trcons i t tr')...
  SCase "head steps".
    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,
     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'.
  has_type_cases (induction H) Case;
    intros Gamma' Heqv...
  Case "T_Var".
    apply T_Var... rewrite ← Heqv...
  Case "T_Abs".
    apply T_Abs... apply IHhas_type. intros y Hafi.
    unfold extend. remember (beq_id x y) as e.
    destruct e...
  Case "T_App".
    apply T_App with T1...
  Case "T_RCons".
    apply T_RCons... Qed.

Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   has_type Gamma t T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  has_type_cases (induction Htyp) Case; inversion Hafi; subst...
  Case "T_Abs".
```

```
    destruct IHHtyp as [T' Hctx]... ∃ T'.
    unfold extend in Hctx.
    apply not_eq_beq_id_false in H3. rewrite H3 in Hctx...
Qed.
```

Preservation

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (extend Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  t_cases (induction t) Case;
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  Case "tvar".
    simpl. rename i into y.
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      apply beq_id_eq in Heqe. subst.
      unfold extend in H0. rewrite ← beq_id_refl in H0.
      inversion H0; subst. clear H0.
      eapply context_invariance...
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra) as [T' HT']...
      inversion HT'.
    SCase "x<>y".
      apply T_Var... unfold extend in H0. rewrite ← Heqe in H0...
  Case "tabs".
    rename i into y. rename t into T11.
    apply T_Abs...
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      eapply context_invariance...
      apply beq_id_eq in Heqe. subst.
      intros x Hafi. unfold extend.
      destruct (beq_id y x)...
    SCase "x<>y".
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold extend.
      remember (beq_id y z) as e0. destruct e0...
      apply beq_id_eq in Heqe0. subst.
```

```
      rewrite ← Heqe...
  Case "trcons".
    apply T_RCons... inversion H7; subst; simpl...
Qed.
```

```
Theorem preservation : ∀ t t' T,
      has_type empty t T →
      t ==> t' →
      has_type empty t' T.
Proof with eauto.
  intros t t' T HT.
  remember (@empty ty) as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  has_type_cases (induction HT) Case;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  Case "T_App".
    inversion HE; subst...
    SCase "ST_AppAbs".
      apply substitution_preserves_typing with T1...
      inversion HT1...
  Case "T_Proj".
    destruct (lookup_field_in_value _ _ _ _ H2 HT H)
      as [vi [Hget Htyp]].
    rewrite H4 in Hget. inversion Hget. subst...
  Case "T_RCons".
    apply T_RCons... eapply step_preserves_record_tm...
Qed.
```

□

End STLCExtendedRecords.

Chapter 13

Library MoreStlc

13.1 MoreStlc: More on the Simply Typed Lambda-Calculus

```
Require Export Stlc.
```

13.2 Simple Extensions to STLC

The simply typed lambda-calculus has enough structure to make its theoretical properties interesting, but it is not yet 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.

13.2.1 Numbers

Adding types, constants, and primitive operations for numbers is easy – just a matter of combining the `Types` and *Stlc* chapters.

13.2.2 `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)
    | let x=t in t       let-binding
```

Reduction: t1 ==> t1'

(ST_Let1) let x=t1 in t2 ==> let x=t1' in t2

(ST_LetValue) let x=v1 in t2 ==> *x:=v1*t2 Typing: Gamma |- t1 : T1 Gamma , x:T1 |- t2 : T2

(T_Let) Gamma |- let x=t1 in t2 : T2

13.2.3 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 ==> t1'

(ST_Pair1) (t1,t2) ==> (t1',t2)
t2 ==> t2'

(ST_Pair2) (v1,t2) ==> (v1,t2')
t1 ==> t1'

(ST_Fst1) t1.fst ==> t1'.fst

(ST_FstPair) (v1,v2).fst ==> v1
t1 ==> t1'

(ST_Snd1) t1.snd ==> t1'.snd

(ST_SndPair) (v1,v2).snd ==> 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 |- t1 : T1 Gamma |- t2 : T2

(T_Pair) Gamma |- (t1,t2) : T1*T2
Gamma |- t1 : T11*T12

(T_Fst) Gamma |- t1.fst : T11
Gamma |- t1 : T11*T12

(T_Snd) Gamma |- 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*.

13.2.4 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:

```
t ::=                 Terms
    | ...
    | unit              unit value

v ::=                 Values
    | ...
    | unit              unit

T ::=                 Types
    | ...
    | Unit              Unit type
```

Typing:

(T_Unit) Gamma |- unit : 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.

13.2.5 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 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* v 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 Nat unit
    else
      inl String ...
```

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 ::=                Terms
    | ...
    | inl T t           tagging (left)
    | inr T t           tagging (right)
    | case t of         case
        inl x => t
      | inr x => t

v ::=                Values
    | ...
    | inl T v           tagged value (left)
    | inr T v           tagged value (right)

T ::=                Types
    | ...
    | T + T             sum type
```

Evaluation:
t1 ==> t1'

(ST_Inl) inl T t1 ==> inl T t1'
t1 ==> t1'

(ST_Inr) inr T t1 ==> inr T t1'
t0 ==> t0'

(ST_Case) case t0 of inl x1 => t1 | inr x2 => t2 ==> case t0' of inl x1 => t1 | inr x2 => t2

(ST_CaseInl) case (inl T v0) of inl x1 => t1 | inr x2 => t2 ==> *x1:=v0*t1

(ST_CaseInr) case (inr T v0) of inl x1 => t1 | inr x2 => t2 ==> *x2:=v0*t2
Typing: Gamma |- t1 : T1

(T_Inl) Gamma |- inl T2 t1 : T1 + T2
Gamma |- t1 : T2

(T_Inr) Gamma |- inr T1 t1 : T1 + T2
Gamma |- t0 : T1+T2 Gamma , x1:T1 |- t1 : T Gamma , x2:T2 |- t2 : T

(T_Case) Gamma |- case t0 of inl x1 => t1 | inr x2 => t2 : T

The reason for the type annotation on the *inl* and *inr* forms has to do with uniqueness of types. 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.

13.2.6 Lists

The typing features we have seen can be classified into *base types* like *Bool*, and *type constructors* like → and × 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
```

While we say that *cons v1 v2* is a value, we really mean that *v2* should also be a list – we'll have to enforce this in the formal definition of value.

Syntax:

```
t ::=                Terms
    | ...
    | nil T
    | 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) cons t1 t2 ==> cons t1' t2
t2 ==> t2'

(ST_Cons2) cons v1 t2 ==> cons v1 t2'
t1 ==> t1'

(ST_Lcase1) (lcase t1 of nil -> t2 | xh::xt -> t3) ==> (lcase t1' of nil -> t2 | xh::xt -> t3)

(ST_LcaseNil) (lcase nil T of nil -> t2 | xh::xt -> t3) ==> t2

(ST_LcaseCons) (lcase (cons vh vt) of nil -> t2 | xh::xt -> t3) ==> $xh{:=}vh,xt{:=}vt$t3
Typing:

(T_Nil) Gamma |- nil T : List T
Gamma |- t1 : T Gamma |- t2 : List T

(T_Cons) Gamma |- cons t1 t2: List T
Gamma |- t1 : List T1 Gamma |- t2 : T Gamma , h:T1, t:List T1 |- t3 : T

(T_Lcase) Gamma |- (lcase t1 of nil -> t2 | h::t -> t3) : T

13.2.7 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 *fact* 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) ($Nat \rightarrow Nat$)->($Nat \rightarrow Nat$) is a function f such that F f is behaviorally equivalent to f.)

Syntax:

```
    t ::=                Terms
        | ...
        | fix t              fixed-point operator
```

Reduction: t1 ==> t1'

(ST_Fix1) fix t1 ==> fix t1'
F = \xf:T1.t2

(ST_FixAbs) fix F ==> xf:=`fix` Ft2 Typing: Gamma |- t1 : T1->T1

(T_Fix) Gamma |- fix t1 : T1

Let's see how *ST_FixAbs* works by reducing *fact* 3 = `fix` F 3, where F = (\\$f$. \\$x$. `if` x=0 `then` 1 `else` $x \times (f\ (pred\ x)))$ (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 (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, recommended (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 $T \to T$ 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
```

13.2.8 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).

This extension is conceptually 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*). Therefore records are not included in the extended exercise below, but they are used to motivate the *Subtyping* chapter.

Syntax:

```
t ::=                          Terms
    | ...
    | {i1=t1, ..., in=tn}         record
    | t.i                         projection

v ::=                          Values
    | ...
    | {i1=v1, ..., in=vn}         record value

T ::=                          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 ==> ti'

(ST_Rcd) {i1=v1, ..., im=vm, in=ti, ...} ==> {i1=v1, ..., im=vm, in=ti', ...}
t1 ==> t1'

(ST_Proj1) t1.i ==> t1'.i

(ST_ProjRcd) {..., i=vi, ...}.i ==> 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 |- t1 : T1 ... Gamma |- tn : Tn

(T_Rcd) Gamma |- {i1=t1, ..., in=tn} : {i1:T1, ..., in:Tn}

Gamma |- t : {..., i:Ti, ...}

(T_Proj) Gamma |- 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 is 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:

```
{a:Nat, b:Nat}      ---->   {Nat,Nat}
{c:Nat, a:Nat}      ---->   {Nat,Unit,Nat}
{f:Nat,c:Nat}       ---->   {Unit,Unit,Nat,Unit,Unit,Nat}
```

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 *<l1:T1,l2:T2,...ln:Tn>* where *l1,l2,...* 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.

13.3 Exercise: Formalizing the Extensions

Exercise: 4 stars, recommended (STLC_extensions) Formalizing the extensions (not including the optional ones) is left to you. We've provided the necessary extensions 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.

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.

A good order for attempting the extensions is:

- numbers first (since they are both familiar and simple)
- products and units
- let (which involves binding)
- sums (which involve slightly more complex binding)
- lists (which involve yet more complex binding)
- `fix`

`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.

Tactic Notation "T_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "TArrow" | Case_aux c "TNat"
  | Case_aux c "TProd" | Case_aux c "TUnit"
  | Case_aux c "TSum" | Case_aux c "TList" ].

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
  | tlcase : 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 ...
```

```
Tactic Notation "t_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp" | Case_aux c "tabs"
  | Case_aux c "tnat" | Case_aux c "tsucc" | Case_aux c "tpred"
  | Case_aux c "tmult" | Case_aux c "tif0"
  | Case_aux c "tpair" | Case_aux c "tfst" | Case_aux c "tsnd"
  | Case_aux c "tunit" | Case_aux c "tlet"
  | Case_aux c "tinl" | Case_aux c "tinr" | Case_aux c "tcase"
  | Case_aux c "tnil" | Case_aux c "tcons" | Case_aux c "tlcase"
  | Case_aux c "tfix" ].
```

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)

  | _ ⇒ 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)

  .

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')


where "t1 '==>' t2" := (step t1 t2).

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_AppAbs" | Case_aux c "ST_App1" | Case_aux c "ST_App2"

  ].
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.

```
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 (extend 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
```

.

Hint Constructors **has_type**.

Tactic Notation "has_type_cases" *tactic*(first) *ident*(*c*) :=
  first;
  [*Case_aux* *c* "T_Var" | *Case_aux* *c* "T_Abs" | *Case_aux* *c* "T_App"

].

13.3.1 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 := (Id 0).
Notation f := (Id 1).
Notation g := (Id 2).
Notation l := (Id 3).
Notation k := (Id 6).
Notation i1 := (Id 7).
Notation i2 := (Id 8).
Notation x := (Id 9).
Notation y := (Id 10).
Notation processSum := (Id 11).
Notation n := (Id 12).
Notation eq := (Id 13).
Notation m := (Id 14).
Notation evenodd := (Id 15).
Notation even := (Id 16).
Notation odd := (Id 17).
Notation eo := (Id 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 (*has_type* *G* (*tapp* *e1* *e1*) *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 x))).
```

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.
```

13.3.2 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 : ∀ t T,
     has_type empty t T →
     value t ∨ ∃ t', t ==> t'.
Proof with eauto.
  intros t T Ht.
  remember (@empty ty) as Gamma.
  generalize dependent HeqGamma.
  has_type_cases (induction Ht) Case; intros HeqGamma; subst.
  Case "T_Var".
    inversion H.
  Case "T_Abs".
```

```
    left...
  Case "T_App".
    right.
    destruct IHHt1; subst...
    SCase "t1 is a value".
      destruct IHHt2; subst...
      SSCase "t2 is a value".
        inversion H; subst; try (solve by inversion).
        ∃ (subst x t2 t12)...
      SSCase "t2 steps".
        inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
    SCase "t1 steps".
      inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...
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)

  .

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'.
  has_type_cases (induction H) Case;
    intros Gamma' Heqv...
  Case "T_Var".
    apply T_Var... rewrite ← Heqv...
  Case "T_Abs".
```

```
    apply T_Abs... apply IHhas_type. intros y Hafi.
    unfold extend. remember (beq_id x y) as e.
    destruct e...
Qed.

Lemma free_in_context : ∀ x t T Gamma,
   appears_free_in x t →
   has_type Gamma t T →
   ∃ T', Gamma x = Some T'.
Proof with eauto.
  intros x t T Gamma Hafi Htyp.
  has_type_cases (induction Htyp) Case; inversion Hafi; subst...
  Case "T_Abs".
    destruct IHHtyp as [T' Hctx]... ∃ T'.
    unfold extend in Hctx.
    apply not_eq_beq_id_false in H2. rewrite H2 in Hctx...
Qed.
```

Substitution

```
Lemma substitution_preserves_typing : ∀ Gamma x U v t S,
     has_type (extend Gamma x U) t S →
     has_type empty v U →
     has_type Gamma ([x:=v]t) S.
Proof with eauto.
  intros Gamma x U v t S Htypt Htypv.
  generalize dependent Gamma. generalize dependent S.
  t_cases (induction t) Case;
    intros S Gamma Htypt; simpl; inversion Htypt; subst...
  Case "tvar".
    simpl. rename i into y.
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      apply beq_id_eq in Heqe. subst.
      unfold extend in H1. rewrite ← beq_id_refl in H1.
      inversion H1; subst. clear H1.
      eapply context_invariance...
      intros x Hcontra.
      destruct (free_in_context _ _ S empty Hcontra) as [T' HT']...
      inversion HT'.
    SCase "x<>y".
      apply T_Var... unfold extend in H1. rewrite ← Heqe in H1...
  Case "tabs".
```

```
    rename i into y. rename t into T11.
    apply T_Abs...
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      eapply context_invariance...
      apply beq_id_eq in Heqe. subst.
      intros x Hafi. unfold extend.
      destruct (beq_id y x)...
    SCase "x<>y".
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold extend.
      remember (beq_id y z) as e0. destruct e0...
      apply beq_id_eq in Heqe0. subst.
      rewrite ← Heqe...
Qed.
```

Preservation

```
Theorem preservation : ∀ t t' T,
     has_type empty t T →
     t ==> t' →
     has_type empty t' T.
Proof with eauto.
  intros t t' T HT.
  remember (@empty ty) as Gamma. generalize dependent HeqGamma.
  generalize dependent t'.
  has_type_cases (induction HT) Case;
    intros t' HeqGamma HE; subst; inversion HE; subst...
  Case "T_App".
    inversion HE; subst...
    SCase "ST_AppAbs".
      apply substitution_preserves_typing with T1...
      inversion HT1...
Qed.
```

☐

End STLCExtended.

Chapter 14

Library Typechecking

14.1 MoreStlc: A Typechecker for STLC

```
Require Export 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.
```

14.1.1 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 : ∀ 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 : ∀ T1 T2,
  beq_ty T1 T2 = true → T1 = T2.
Proof with auto.
  intros T1. induction T1; intros T2 Hbeq; destruct T2; inversion Hbeq.
  Case "T1=TBool".
    reflexivity.
  Case "T1=TArrow T1_1 T1_2".
    apply andb_true in H0. inversion H0 as [Hbeq1 Hbeq2].
    apply IHT1_1 in Hbeq1. apply IHT1_2 in Hbeq2. subst... Qed.
```

14.1.2 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 *T1* and *T2*).

```
Fixpoint type_check (Gamma:context) (t:tm) : option ty :=
  match t with
  | tvar x ⇒ Gamma x
  | tabs x T11 t12 ⇒ match type_check (extend Gamma x T11) t12 with
                         | Some T12 ⇒ Some (TArrow T11 T12)
                         | _ ⇒ None
                       end
  | tapp t1 t2 ⇒ match type_check Gamma t1, type_check Gamma t2 with
                     | Some (TArrow T11 T12),Some T2 ⇒
                       if beq_ty T11 T2 then Some T12 else None
                     | _,_ ⇒ None
                   end
  | ttrue ⇒ Some TBool
  | tfalse ⇒ Some TBool
  | tif x t f ⇒ match type_check Gamma x with
                    | Some TBool ⇒
                      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.
```

14.1.3 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 : ∀ Gamma t T,
  type_check Gamma t = Some T → has_type Gamma t T.
Proof with eauto.
  intros Gamma t. generalize dependent Gamma.
  t_cases (induction t) Case; intros Gamma T Htc; inversion Htc.
  Case "tvar"...
  Case "tapp".
    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.
    remember (beq_ty T11 T2) as b.
    destruct b; try solve by inversion.
    symmetry in Heqb. apply beq_ty__eq in Heqb.
    inversion H0; subst...
  Case "tabs".
    rename i into y. rename t into T1.
    remember (extend Gamma y T1) as G'.
    remember (type_check G' t0) as TO2.
    destruct TO2; try solve by inversion.
    inversion H0; subst...
  Case "ttrue"...
  Case "tfalse"...
  Case "tif".
    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.
    remember (beq_ty T1 T2) as b.
    destruct b; try solve by inversion.
    symmetry in Heqb. apply beq_ty__eq in Heqb.
    inversion H0. subst. subst...
Qed.

Theorem type_checking_complete : ∀ Gamma t T,
  has_type Gamma t T → type_check Gamma t = Some T.
Proof with auto.
  intros Gamma t T Hty.
  has_type_cases (induction Hty) Case; simpl.
  Case "T_Var"...
  Case "T_Abs". rewrite IHHty...
  Case "T_App".
    rewrite IHHty1. rewrite IHHty2.
    rewrite (beq_ty_refl T11)...
  Case "T_True"...
  Case "T_False"...
  Case "T_If". rewrite IHHty1. rewrite IHHty2.
    rewrite IHHty3. rewrite (beq_ty_refl T)...
Qed.

End STLCChecker.
```

Chapter 15

Library Stlc

15.1 Stlc: The Simply Typed Lambda-Calculus

```
Require Export Types.
```

15.2 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 (which will take some work to deal with) all arise from the mechanisms of *variable binding* and *substitution*.

15.2.1 Overview

The STLC is built on some collection of *base types* – booleans, numbers, strings, etc. The exact choice of base types doesn't matter – the construction of the language and its theoretical properties work out pretty much the same – 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 the booleans, we add three things:

- variables
- function abstractions
- application

This gives us the following collection of abstract syntax constructors (written out here in informal BNF notation – we'll formalize it below).

Informal concrete syntax:

```
t ::= x                         variable
    | \x:T.t1                   abstraction
    | t1 t2                     application
    | true                      constant true
    | false                     constant false
    | if t1 then t2 else t3     conditional
```

The \ symbol in a function abstraction *x*:*T.t1* is often written as a greek "lambda" (hence the name of the calculus). The variable *x* is called the *parameter* to the function; the term *t1* is its *body*. The annotation :*T* specifies the type of arguments that the function can be applied to.

Some examples:

- *x*:*Bool*. *x*

  The identity function for booleans.

- (*x*:*Bool*. *x*) *true*

  The identity function for booleans, applied to the boolean *true*.

- *x*:*Bool*. `if` *x* `then` *false* `else` *true*

  The boolean "not" function.

- *x*:*Bool*. *true*

  The constant function that takes every (boolean) argument to *true*.

- *x*:*Bool*. *y*:*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.)

- (*x*:*Bool*. *y*:*Bool*. *x*) *false* *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 ((*x*:*Bool*. *y*:*Bool*. *x*) *false*) *true*.

- *f*:*Bool*→*Bool*. *f* (*f* *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.

- $(\backslash f{:}Bool{\rightarrow}Bool.\ f\ (f\ true))\ (\backslash x{:}Bool.\ false)$

  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 ::= Bool
    | T1 -> T2
```

For example:

- $\backslash x{:}Bool.\ false$ has type $Bool{\rightarrow}Bool$
- $\backslash x{:}Bool.\ x$ has type $Bool{\rightarrow}Bool$
- $(\backslash x{:}Bool.\ x)\ true$ has type $Bool$
- $\backslash x{:}Bool.\ \backslash y{:}Bool.\ x$ has type $Bool{\rightarrow}Bool{\rightarrow}Bool$ (i.e. $Bool \rightarrow (Bool{\rightarrow}Bool)$)
- $(\backslash x{:}Bool.\ \backslash y{:}Bool.\ x)\ false$ has type $Bool{\rightarrow}Bool$
- $(\backslash x{:}Bool.\ \backslash y{:}Bool.\ x)\ false\ true$ has type $Bool$
- $\backslash f{:}Bool{\rightarrow}Bool.\ f\ (f\ true)$ has type $(Bool{\rightarrow}Bool) \rightarrow Bool$
- $(\backslash f{:}Bool{\rightarrow}Bool.\ f\ (f\ true))\ (\backslash x{:}Bool.\ false)$ has type $Bool$

15.2.2 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.
Tactic Notation "t_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp"
  | Case_aux c "tabs" | Case_aux c "ttrue"
  | Case_aux c "tfalse" | Case_aux c "tif" ].
```

Note that an abstraction \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 := (Id 0).
Definition y := (Id 1).
Definition z := (Id 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))).
```

(We write these as `Notations` rather than `Definitions` to make things easier for `auto`.)

15.2.3 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 $\backslash x{:}T.t1$ 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 $\backslash x{:}T.t1$ 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 langauge) makes the first choice – for example, Eval simpl in (fun x:bool => 3 + 4) yields `fun` $x{:}bool \Rightarrow 7$. But 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.

Finally, 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).

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T t,
      value (tabs x T t)
  | t_true :
      value ttrue
  | t_false :
      value tfalse.

Hint Constructors value.
```

Free Variables and Substitution

Now we come to the heart of the matter: 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 (\x:Bool. if x then true else x) false to *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 \\$x$:*Bool*. x does *not* yield \\$x$:*Bool*. *true*! The reason for this is that the x in the body of \\$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(\x:T11.t12) = \x:T11. t12 x:=s(\y:T11.t12) = \y:T11. x:=st12 if x <> y x:=s(t1 t2) = (x:=st1) (x:=st2) x:=$s$true = true x:=$s$false = false x:=s(if t1 then t2 else t3) = if x:=st1 then x:=st2 else x:=st3]] ... and formally:

```
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 (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)
  end.
Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

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 y* that mention variables are not bound by some enclosing lambda), we can avoid this extra complexity here.

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 (\x:T.t12) v2 ==> *x*:=*v2*t12 is traditionally called "beta-reduction".

value v2

(ST_AppAbs) (\x:T.t12) v2 ==> *x*:=*v2*t12

t1 ==> t1'

(ST_App1) t1 t2 ==> t1' t2

value v1 t2 ==> t2'

(ST_App2) v1 t2 ==> v1 t2' ... plus the usual rules for booleans:

(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)

```
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).

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_AppAbs" | Case_aux c "ST_App1"
  | Case_aux c "ST_App2" | Case_aux c "ST_IfTrue"
  | Case_aux c "ST_IfFalse" | Case_aux c "ST_If" ].

Hint Constructors step.

Notation multistep := (multi step).
Notation "t1 '==>*' t2" := (multistep t1 t2) (at level 40).
```

Examples

Example: ((\x:Bool->Bool. x) (\x:Bool. x)) ==>* (\x:Bool. x) i.e. (idBB idB) ==>* idB

```
Lemma step_example1 :
  (tapp idBB idB) ==>* idB.
Proof.
  eapply multi_step.
    apply ST_AppAbs.
    apply v_abs.
  simpl.
  apply multi_refl. Qed.
```

A more automatic proof

```
Lemma step_example1' :
```

```
  (tapp idBB idB) ==>* idB.
Proof. normalize. Qed.

Lemma step_example2 :
  (tapp idBB (tapp idBB idB)) ==>* 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.
```

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.
```

Exercise: 2 stars (step_example3) Try to do this one both with and without *normalize*.

```
Lemma step_example3 :
       (tapp (tapp idBBBB idBB) idB)
  ==>* idB.
Proof.
   Admitted.
```

☐

15.2.4 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 $Gamma \vdash t : T$, where $Gamma$ is a "typing context" – a mapping from variables to their types.

We hide the definition of partial maps in a module since it is actually defined in *SfLib*.

```
Module PartialMap.

Definition partial_map (A:Type) := id → option A.
```

```
Definition empty {A:Type} : partial_map A := (fun _ ⇒ None).
```

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 extend {A:Type} (Gamma : partial_map A) (x:id) (T : A) :=
  fun x' ⇒ if beq_id x x' then Some T else Gamma x'.

Lemma extend_eq : ∀ A (ctxt: partial_map A) x T,
  (extend ctxt x T) x = Some T.
Proof.
  intros. unfold extend. rewrite ← beq_id_refl. auto.
Qed.

Lemma extend_neq : ∀ A (ctxt: partial_map A) x1 T x2,
  beq_id x2 x1 = false →
  (extend ctxt x2 T) x1 = ctxt x1.
Proof.
  intros. unfold extend. rewrite H. auto.
Qed.

End PartialMap.

Definition context := partial_map ty.
```

Typing Relation

Gamma x = T

(T_Var) Gamma |- x : T

Gamma , x:T11 |- t12 : T12

(T_Abs) Gamma |- \x:T11.t12 : T11->T12

Gamma |- t1 : T11->T12 Gamma |- t2 : T11

(T_App) Gamma |- t1 t2 : T12

(T_True) Gamma |- true : Bool

(T_False) Gamma |- false : Bool

Gamma |- t1 : Bool Gamma |- t2 : T Gamma |- t3 : T

(T_If) Gamma |- if t1 then t2 else t3 : T

```
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 (extend Gamma x T11) t12 T12 →
      has_type Gamma (tabs x T11 t12) (TArrow T11 T12)
  | T_App : ∀ T11 T12 Gamma t1 t2,
      has_type Gamma t1 (TArrow T11 T12) →
      has_type Gamma t2 T11 →
      has_type Gamma (tapp t1 t2) T12
  | T_True : ∀ Gamma,
       has_type Gamma ttrue TBool
  | T_False : ∀ Gamma,
       has_type Gamma tfalse TBool
  | T_If : ∀ t1 t2 t3 T Gamma,
       has_type Gamma t1 TBool →
       has_type Gamma t2 T →
       has_type Gamma t3 T →
       has_type Gamma (tif t1 t2 t3) T.

Tactic Notation "has_type_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "T_Var" | Case_aux c "T_Abs"
  | Case_aux c "T_App" | Case_aux c "T_True"
  | Case_aux c "T_False" | Case_aux c "T_If" ].

Hint Constructors has_type.
```

Examples

```
Example typing_example_1 :
  has_type empty (tabs x TBool (tvar x)) (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' :
  has_type empty (tabs x TBool (tvar x)) (TArrow TBool TBool).
Proof. auto. Qed.

Hint Unfold beq_id beq_nat extend.
```

Another example: empty |- \x:A. \y:A->A. y (y x)) : A -> (A->A) -> A.

```
Example typing_example_2 :
  has_type empty
    (tabs x TBool
       (tabs y (TArrow TBool TBool)
          (tapp (tvar y) (tapp (tvar y) (tvar x))))))
```

```
    (TArrow TBool (TArrow (TArrow TBool TBool) TBool)).
Proof with auto using extend_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 :
  has_type empty
    (tabs x TBool
       (tabs y (TArrow TBool TBool)
          (tapp (tvar y) (tapp (tvar y) (tvar x)))))
    (TArrow TBool (TArrow (TArrow TBool TBool) TBool)).
Proof.
   Admitted.
```

☐

Exercise: 2 stars (typing_example_3) Formally prove the following typing derivation holds:

empty |- (\x:Bool->B. \y:Bool->Bool. \z:Bool. y (x z)) : T.

```
Example typing_example_3 :
  ∃ T,
    has_type empty
      (tabs x (TArrow TBool TBool)
         (tabs y (TArrow TBool TBool)
            (tabs z TBool
               (tapp (tvar y) (tapp (tvar x) (tvar z))))))
      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 $\backslash x{:}Bool.\ \backslash y{:}Bool,\ x\ y$ – i.e., ~ exists T, empty |- (\x:Bool. \y:Bool, x y) : T.

```
Example typing_nonexample_1 :
  ¬ ∃ T,
      has_type empty
```

```
        (tabs x TBool
            (tabs y TBool
               (tapp (tvar x) (tvar y))))
        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 (typing_nonexample_3) Another nonexample: ~ (exists S, exists T, empty |- (\x:S. x x) : T).

```
Example typing_nonexample_3 :
  ¬ (∃ S, ∃ T,
        has_type empty
          (tabs x S
             (tapp (tvar x) (tvar x)))
          T).
Proof.
   Admitted.
```

□

Exercise: 1 star, optional (typing_statements) Which of the following propositions are provable?

- $y{:}Bool \vdash \backslash x{:}Bool.x : Bool{\to}Bool$
- $\exists\ T,\ empty \vdash (\backslash y{:}Bool{\to}Bool.\ \backslash x{:}Bool.\ y\ x) : T$
- $\exists\ T,\ empty \vdash (\backslash y{:}Bool{\to}Bool.\ \backslash x{:}Bool.\ x\ y) : T$
- $\exists\ S,\ x{:}S \vdash (\backslash y{:}Bool{\to}Bool.\ y)\ x : S$
- $\exists\ S,\ \exists\ T,\ x{:}S \vdash (x\ x\ x) : T$

□

Exercise: 1 star (more_typing_statements) Which of the following propositions are provable? For the ones that are, give witnesses for the existentially bound variables.

- $\exists\ T,\ empty \vdash (\backslash y{:}B{\to}B{\to}B.\ \backslash x{:}B,\ y\ x) : T$

- $\exists$ T, *empty* $\vdash$ (\x:A→B, \y:B→C, \z:A, y (x z)):T

- $\exists$ S, $\exists$ U, $\exists$ T, x:S, y:U $\vdash$ \z:A. x (y z) : T

- $\exists$ S, $\exists$ T, x:S $\vdash$ \y:A. x (x y) : T

- $\exists$ S, $\exists$ U, $\exists$ T, x:S $\vdash$ x (\z:U. z x) : T

☐

15.2.5 Properties

Progress

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.

```
Theorem progress : ∀ t T,
    has_type empty t T →
    value t ∨ ∃ t', t ==> t'.
```

Proof: by induction on the derivation of $\vdash$ *t* : *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* : *T2* → *T* and $\vdash$ *t2* : *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* = `if` *t1* `then` *t2* `else` *t3*, where *t1* has type *Bool*. By the IH, *t1* is either 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.
  has_type_cases (induction Ht) Case; subst Gamma...
  Case "T_Var".
    inversion H.
  Case "T_App".
    right. destruct IHHt1...
    SCase "t1 is a value".
      destruct IHHt2...
      SSCase "t2 is also a value".
        inversion H; subst. ∃ ([x0:=t2]t)...
        solve by inversion. solve by inversion.
      SSCase "t2 steps".
        inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...
    SCase "t1 steps".
      inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...
  Case "T_If".
    right. destruct IHHt1...
    SCase "t1 is a value".
      inversion H; subst. solve by inversion.
      SSCase "t1 = true". eauto.
      SSCase "t1 = false". eauto.
    SCase "t1 also steps".
      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' : ∀ t T,
     has_type empty t T →
     value t ∨ ∃ t', t ==> t'.
Proof.
  intros t.
  t_cases (induction t) Case; intros T Ht; auto.
   Admitted.
```

□

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 $\backslash x{:}T{\to}U.\ x\ y$
- both x and y appear free in $(\backslash x{:}T{\to}U.\ x\ y)\ x$
- no variables appear free in $\backslash x{:}T{\to}U.\ \backslash y{:}T.\ x\ y$

```
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).
```

```
Tactic Notation "afi_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "afi_var"
  | Case_aux c "afi_app1" | Case_aux c "afi_app2"
  | Case_aux c "afi_abs"
  | Case_aux c "afi_if1" | Case_aux c "afi_if2"
  | Case_aux c "afi_if3" ].
```

```
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.
```

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 →
   has_type Gamma t T →
   ∃ T', Gamma x = 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 *extend_neq*, noting that x and y are different variables.

```
Proof.
  intros x t T Gamma H H0. generalize dependent Gamma.
  generalize dependent T.
  afi_cases (induction H) Case;
         intros; try solve [inversion H0; eauto].
  Case "afi_abs".
    inversion H1; subst.
    apply IHappears_free_in in H7.
    apply not_eq_beq_id_false in H.
    rewrite extend_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 (typable_empty__closed) `Corollary` typable_empty__closed : ∀ t T,

  has_type empty t T →
  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 : ∀ $Gamma$ $Gamma'$ t T,
  has_type $Gamma$ t T →
  (∀ x, **appears_free_in** x t → $Gamma$ x = $Gamma'$ x) →
  has_type $Gamma'$ t T.

Proof: By induction on the derivation of $Gamma \vdash t : T$.

- 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 $Gamma' \vdash t : T$ by *T_Var*.

- If the last rule was *T_Abs*, then $t = \backslash y{:}T11.\ t12$, with $T = T11 \rightarrow T12$ and $Gamma, y{:}T11 \vdash t12 : T12$. 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 $Gamma' \vdash \backslash y{:}T11.\ t12 : T11 \rightarrow T12$.

  By *T_Abs*, it suffices to show that $Gamma', y{:}T11 \vdash t12 : T12$. 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 $t = \backslash y{:}T11.\ t12$, 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 $Gamma \vdash t1 : T2 \rightarrow T$ and $Gamma \vdash t2 : T2$. 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'.
  has_type_cases (induction H) Case; intros; auto.
  Case "T_Var".
    apply T_Var. rewrite <- H0...
  Case "T_Abs".
    apply T_Abs.
    apply IHhas_type. intros x1 Hafi.
    unfold extend. remember (beq_id x0 x1) as e. destruct e...
  Case "T_App".
    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 $Gamma,x{:}U \vdash t : T$ and $\vdash v : U$, then $Gamma \vdash [x{:=}v]t : T$.

```
Lemma substitution_preserves_typing : ∀ Gamma x U t t' T,
     has_type (extend Gamma x U) t T →
     has_type empty t' U →
     has_type Gamma ([x:=t']t) T.
```

One technical subtlety in the statement of the lemma is that we assign t' the type U in the *empty* context – in other words, we assume t' is closed. This assumption considerably simplifies the *T_Abs* case of the proof (compared to assuming $Gamma \vdash t' : U$, which would be the other reasonable assumption at this point) because the context invariance lemma then tells us that t' has type U in any context at all – we don't have to worry about free variables in t' 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 $Gamma,x{:}U \vdash t : T$ and $\vdash t' : U$, then $Gamma \vdash [x{:=}t']t : T$.

- 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 $Gamma,\ x{:}U \vdash x : T$ we conclude that $U = T$. We must show that $[x{:=}t']x = t'$ has type T under $Gamma$, given the

assumption that *t'* 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 \\$y{:}T11.\ t12$, then the IH tells us, for all *Gamma'* and *T'*, that if *Gamma'*,$x{:}U \vdash t12 : T'$ and $\vdash t' : U$, then *Gamma'* $\vdash [x{:=}t']t12 : T'$.

  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, $[x{:=}t']t = t$, so we just need to show *Gamma* $\vdash t : T$. But we know *Gamma*,$x{:}U \vdash t : T$, and since the variable y does not appear free in \\$y{:}T11.\ t12$, the context invariance lemma yields *Gamma* $\vdash t : T$.

  Second, suppose $x \neq y$. We know *Gamma*,$x{:}U$,$y{:}T11 \vdash t12 : T12$ by inversion of the typing relation, and *Gamma*,$y{:}T11$,$x{:}U \vdash t12 : T12$ follows from this by the context invariance lemma, so the IH applies, giving us *Gamma*,$y{:}T11 \vdash [x{:=}t']t12 : T12$. By *T_Abs*, *Gamma* $\vdash$ \\$y{:}T11.\ [x{:=}t']t12 : T11 \to T12$, and by the definition of substitution (noting that $x \neq y$), *Gamma* $\vdash$ \\$y{:}T11.\ [x{:=}t']t12 : T11 \to 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 *has_type* (*extend Gamma x U*) *t 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 t' T Ht Ht'.
  generalize dependent Gamma. generalize dependent T.
  t_cases (induction t) Case; intros T Gamma H;

    inversion H; subst; simpl...
  Case "tvar".
    rename i into y. remember (beq_id x y) as e. destruct e.
    SCase "x=y".
```

```
      apply beq_id_eq in Heqe. subst.
      rewrite extend_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'.
    SCase "x<>y".
      apply T_Var. rewrite extend_neq in H2...
  Case "tabs".
    rename i into y. apply T_Abs.
    remember (beq_id x y) as e. destruct e.
    SCase "x=y".
      eapply context_invariance...
      apply beq_id_eq in Heqe. subst.
      intros x Hafi. unfold extend.
      destruct (beq_id y x)...
    SCase "x<>y".
      apply IHt. eapply context_invariance...
      intros z Hafi. unfold extend.
      remember (beq_id y z) as e0. destruct e0...
      apply beq_id_eq in Heqe0. subst.
      rewrite ← Heqe...
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.

Preservation

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 : ∀ t t' T,
     has_type empty t T →
     t ==> t' →
     has_type empty t' T.
```

Proof: by induction on the derivation of $\vdash t : T$.

- 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* = \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* = `if` *t1* `then` *t2* `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'.
  has_type_cases (induction HT) Case;
       intros t' HE; subst Gamma; subst;
       try solve [inversion HE; subst; auto].
  Case "T_App".
    inversion HE; subst...
    SCase "ST_AppAbs".
      apply substitution_preserves_typing with T11...
      inversion HT1...
Qed.
```

Exercise: 2 stars, recommended (subject_expansion_stlc) An exercise in Types.v 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* ==> *t'* and *has_type t' T*, then *has_type t T*? If so, prove it. If not, give a counter-example.
☐

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:tm) : Prop :=
  (normal_form step) t ∧ ¬ value t.

Corollary soundness : ∀ t t' T,
  has_type empty t T →
  t ==>* t' →
  ~(stuck t').
Proof.
  intros t t' T Hhas_type Hmulti. unfold stuck.
  intros [Hnf Hnot_val]. unfold normal_form in Hnf.
  induction Hmulti.
   Admitted.
```

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.

☐

15.2.6 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 ==> zap and the following typing rule:

(T_Zap) Gamma |- 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) (\x:A. x) ==> foo

(ST_Foo2) foo ==> 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 (stlc_variation4) Suppose instead that we add the following new rule to the reduction relation:

(ST_FunnyIfTrue) (if true then t1 else t2) ==> 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_variation5) Suppose instead that we add the following new rule to the typing relation: Gamma |- t1 : Bool->Bool->Bool Gamma |- t2 : Bool

---

(T_FunnyApp) Gamma |- t1 t2 : 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 (stlc_variation6) Suppose instead that we add the following new rule to the typing relation: Gamma |- t1 : Bool Gamma |- t2 : Bool

---

(T_FunnyApp') Gamma |- t1 t2 : 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 (stlc_variation7) Suppose we add the following new rule to the typing relation of the STLC:

(T_FunnyAbs) |- \x:Bool.t : 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` STLC.

15.3 Optional 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.
```

15.3.1 Syntax and Operational Semantics

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.
```

```
Tactic Notation "t_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "tvar" | Case_aux c "tapp"
  | Case_aux c "tabs" | Case_aux c "tnat"
  | Case_aux c "tsucc" | Case_aux c "tpred"
  | Case_aux c "tmult" | Case_aux c "tif0" ].
```

Exercise: 4 stars, optional (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 of the original STLC to deal with the new syntactic forms. Make sure Coq accepts the whole file.

□

`End` STLCARITH.

Chapter 16

Library Types

16.1 Types: Type Systems

```
Require Export Smallstep.
```

Our next topic, a large one, 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*. Then we'll move on to the *simply typed lambda-calculus*, which lives at the core of every modern functional programming language (including Coq).

16.2 More Automation

Before we start, let's spend a little time learning to use some of Coq's more powerful automation features...

16.2.1 The `auto` and `eauto` Tactics

The `auto` tactic solves goals that are solvable by any combination of

- `intros`,
- `apply` (with a local hypothesis, by default), and
- `reflexivity`.

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.

Here is a contrived example:

```
Lemma auto_example_1 : ∀ P Q R S T U : Prop,
  (P → Q) →
  (P → R) →
  (T → R) →
  (S → T → U) →
  ((P→Q) → (P→S)) →
  T →
  P →
  U.
Proof. auto. 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., *conj*, *or_introl*, and *or_intror* – are installed in this hint database by default.

```
Lemma auto_example_2 : ∀ P Q R : Prop,
  Q →
  (Q → R) →
  P ∨ (Q ∧ R).
Proof.
  auto. Qed.
```

We can extend the hint database just for the purposes of one application of `auto` or `eauto` by writing `auto using` .... E.g., if *conj*, *or_introl*, and *or_intror* had *not* already been in the hint database, we could have done this instead:

```
Lemma auto_example_2a : ∀ P Q R : Prop,
  Q →
  (Q → R) →
  P ∨ (Q ∧ R).
Proof.
  auto using conj, or_introl, or_intror. Qed.
```

Of course, in any given development there will also be some of our own 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.

Here are some `Hints` we will find useful.

```
Hint Constructors multi.
Hint Resolve beq_id_eq beq_id_false_not_eq.
```

Warning: Just as with Coq's other automation facilities, it is easy to overuse `auto` and `eauto` and wind up with proofs that are impossible to understand later!

Also, overuse of `eauto` can make proof scripts very slow. Get in the habit of using `auto` most of the time and `eauto` only when necessary.

For much more detailed information about using `auto` and `eauto`, see the chapter *UseAuto*.

16.2.2 The `Proof with` Tactic

If you start a proof by saying `Proof with` (*tactic*) instead of just `Proof`, then writing ... instead of . after a tactic in the body of the proof will try to solve all generated subgoals with *tactic* (and fail if this doesn't work).

One common use of this facility is "`Proof with auto`" (or `eauto`). We'll see many examples of this later in the file.

16.2.3 The `solve by inversion` Tactic

Here's another nice automation feature: it often arises that the context contains a contradictory assumption and we want to use `inversion` on it to solve the goal. We'd like to be able to say to Coq, "find a contradictory assumption and invert it" without giving its name explicitly.

Doing `solve by inversion` will find a hypothesis that can be inverted to solve the goal, if there is one. The tactics `solve by inversion` 2 and `solve by inversion` 3 are slightly fancier versions which will perform two or three inversions in a row, if necessary, to solve the goal.

(These tactics are not actually built into Coq – their definitions are in *Sflib*.)

Caution: Overuse of `solve by inversion` can lead to slow proof scripts.

16.2.4 The `try solve` Tactic

If *t* is a tactic, then `try solve` [*t*] is a tactic that

- if *t* solves the goal, behaves just like *t*, or
- if *t* cannot completely solve the goal, does nothing.

More generally, `try solve` [*t1* | *t2* | ...] will try to solve the goal by using *t1*, *t2*, etc. If none of them succeeds in completely solving the goal, then `try solve` [*t1* | *t2* | ...] does nothing.

16.2.5 The `f_equal` Tactic

`f_equal` replaces a goal of the form *f x1 x2 ... xn = f y1 y2 ... yn*, where *f* is some function, with the subgoals *x1 = y1*, *x2 = y2*,...,*xn = yn*. It is useful for avoiding explicit rewriting steps, and often the generated subgoals can be quickly cleared by `auto`. This tactic is not fundamental, in the sense that it can always be replaced by a sequence of `asserts`. However in some cases it can be very handy.

16.2.6 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* ==>* *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* defined in the previous chapter:

```
Definition amultistep st := multi (astep st).
Notation " t '/' st '==>a*' t' " := (amultistep 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 applied *multi_step* until we got to a normal form. The proofs that the intermediate steps are possible 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 *multi_step* we print out the current goal, so that the user 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
  ==>a× (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''' : ∃ e',
  (APlus (ANum 3) (AMult (ANum 3) (ANum 4))) / empty_state
  ==>a× e'.
Proof.
  eapply ex_intro. normalize.
Qed.
```

Exercise: 1 star (normalize_ex) `Theorem normalize_ex : ∃ e',`

```
  (AMult (ANum 3) (AMult (ANum 2) (ANum 1))) / empty_state
  ==>a× e'.
Proof.
   Admitted.
```

☐

Exercise: 1 star, optional (normalize_ex') This time prove it by using `apply` instead of `eapply`.

```
Theorem normalize_ex' : ∃ e',
  (AMult (ANum 3) (AMult (ANum 2) (ANum 1))) / empty_state
  ==>a× e'.
Proof.
   Admitted.
```

☐

16.3 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 (just numbers) is too simple, but just two kinds (numbers and booleans) already gives us enough material to tell an interesting story.

The language definition is completely routine. The only thing to notice is that we are *not* using the *asnum*/*aslist* trick that we used in chapter *ImpList* to make all the operations total by forcibly coercing the arguments to + (for example) into numbers. Instead, we simply let terms get stuck if they try to use an operator with the wrong kind of operands: the *step* relation doesn't relate them to anything.

16.3.1 Syntax

Informally: t ::= true | false | if t then t else t | 0 | succ t | pred t | 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.
```

16.3.2 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).

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_IfTrue" | Case_aux c "ST_IfFalse" | Case_aux c "ST_If"
  | Case_aux c "ST_Succ" | Case_aux c "ST_PredZero"
  | Case_aux c "ST_PredSucc" | Case_aux c "ST_Pred"
  | Case_aux c "ST_IszeroZero" | Case_aux c "ST_IszeroSucc"
  | Case_aux c "ST_Iszero" ].

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 *one* step.

16.3.3 Normal Forms and Values

The first interesting thing 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 former set is included in the latter. This is important because it shows we did not accidentally define things so that some value could still take a step.

Exercise: 3 stars, optional (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 : ∀ t,
  value t → step_normal_form t.
Proof.
  Admitted.
```

□

Exercise: 3 stars, optional (step_deterministic) Using *value_is_nf*, we can show that the *step* relation is also deterministic...

```
Theorem step_deterministic:
  deterministic step.
Proof with eauto.
  Admitted.
```

□

16.3.4 Typing

The next critical observation about this language is that, although there are 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 $\vdash t : T$, 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) |- true : Bool

(T_False) |- false : Bool
|- t1 : Bool |- t2 : T |- t3 : T

(T_If) |- if t1 then t2 else t3 : T

(T_Zero) |- 0 : Nat
|- t1 : Nat

(T_Succ) |- succ t1 : Nat
|- t1 : Nat

(T_Pred) |- pred t1 : Nat
|- t1 : Nat

(T_IsZero) |- iszero t1 : Bool

```
Inductive has_type : tm → ty → Prop :=
  | T_True :
       has_type ttrue TBool
  | T_False :
       has_type tfalse TBool
  | T_If : ∀ t1 t2 t3 T,
       has_type t1 TBool →
       has_type t2 T →
       has_type t3 T →
       has_type (tif t1 t2 t3) T
  | T_Zero :
       has_type tzero TNat
  | T_Succ : ∀ t1,
       has_type t1 TNat →
       has_type (tsucc t1) TNat
  | T_Pred : ∀ t1,
       has_type t1 TNat →
       has_type (tpred t1) TNat
  | T_Iszero : ∀ t1,
       has_type t1 TNat →
       has_type (tiszero t1) TBool.

Tactic Notation "has_type_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "T_True" | Case_aux c "T_False" | Case_aux c "T_If"
  | Case_aux c "T_Zero" | Case_aux c "T_Succ" | Case_aux c "T_Pred"
  | Case_aux c "T_Iszero" ].

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 :
  has_type (tif tfalse tzero (tsucc tzero)) 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 :
  ¬ has_type (tif tfalse tzero ttrue) TBool.
Proof.
  intros Contra. solve by inversion 2. Qed.
```

Exercise: 1 star (succ_hastype_nat__hastype_nat) `Example succ_hastype_nat__hastype_nat : ∀ t,`

```
  has_type (tsucc t) TNat →
  has_type t TNat.
Proof.
  Admitted.
```

☐

16.3.5 Progress

The typing relation enjoys two critical properties. The first is that well-typed normal forms are values (i.e., not stuck).

Exercise: 3 stars, recommended (finish_progress_informal) Complete the following proof:

Theorem: If $\vdash t : T$, then either t is a value or else t ==> t' for some t'.

Proof: By induction on a derivation of $\vdash t : T$.

- If the last rule in the derivation is *T_If*, then t = `if` $t1$ `then` $t2$ `else` $t3$, with $\vdash t1 : Bool$, $\vdash t2 : T$ and $\vdash t3 : T$. By the IH, either $t1$ is a value or else $t1$ can step to some $t1'$.

- If *t1* is a value, then it is either an *nvalue* or a *bvalue*. But it cannot be an *nvalue*, because we know $\vdash$ *t1* : *Bool* and there are no rules assigning type *Bool* to any term that could be an *nvalue*. So *t1* is a *bvalue* – i.e., it is either *true* or *false*. If *t1* = *true*, then *t* steps to *t2* by *ST_IfTrue*, while if *t1* = *false*, then *t* steps to *t3* by *ST_IfFalse*. Either way, *t* can step, which is what we wanted to show.
- If *t1* itself can take a step, then, by *ST_If*, so can *t*.

☐

Exercise: 3 stars (finish_progress) Theorem progress : ∀ *t* *T*,

```
  has_type t T →
  value t ∨ ∃ t', t ==> t'.
Proof with auto.
  intros t T HT.
  has_type_cases (induction HT) Case...
  Case "T_If".
    right. inversion IHHT1; clear IHHT1.
    SCase "t1 is a value". inversion H; clear H.
      SSCase "t1 is a bvalue". inversion H0; clear H0.
        SSSCase "t1 is ttrue".
          ∃ t2...
        SSSCase "t1 is tfalse".
          ∃ t3...
      SSCase "t1 is an nvalue".
        solve by inversion 2.        SCase "t1 can take a step".
      inversion H as [t1' H1].
      ∃ (tif t1' t2 t3)...
  Admitted.
```

☐

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.

☐

16.3.6 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.

Exercise: 3 stars, recommended (finish_preservation_informal) Complete the following proof:

Theorem: If $\vdash t : T$ and $t \Longrightarrow t'$, then $\vdash t' : T$.

Proof: By induction on a derivation of $\vdash t : T$.

- If the last rule in the derivation is *T_If*, then *t* = `if` *t1* `then` *t2* `else` *t3*, with $\vdash t1 : Bool$, $\vdash t2 : T$ and $\vdash t3 : T$.

  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 $t \Longrightarrow t'$ are *ST_IfTrue*, *ST_IfFalse*, or *ST_If*.

  - If the last rule was *ST_IfTrue*, then $t' = t2$. But we know that $\vdash t2 : T$, so we are done.
  - If the last rule was *ST_IfFalse*, then $t' = t3$. But we know that $\vdash t3 : T$, so we are done.
  - If the last rule was *ST_If*, then *t'* = `if` *t1'* `then` *t2* `else` *t3*, where $t1 \Longrightarrow t1'$. We know $\vdash t1 : Bool$ so, by the IH, $\vdash t1' : Bool$. The *T_If* rule then gives us $\vdash$ `if` *t1'* `then` *t2* `else` *t3* : *T*, as required.

□

Exercise: 2 stars (finish_preservation) Theorem preservation : ∀ *t t' T*,

```
  has_type t T →
  t ==> t' →
  has_type t' T.
Proof with auto.
  intros t t' T HT HE.
  generalize dependent t'.
  has_type_cases (induction HT) Case;

        intros t' HE;

        try (solve by inversion).
```

```
  Case "T_If". inversion HE; subst.
    SCase "ST_IFTrue". assumption.
    SCase "ST_IfFalse". assumption.
    SCase "ST_If". apply T_If; try assumption.
      apply IHHT1; assumption.
Admitted.
```
□

Exercise: 3 stars (preservation_alternate_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' : ∀ t t' T,
  has_type t T →
  t ==> t' →
  has_type t' T.
Proof with eauto.
  Admitted.
```
□

16.3.7 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 : ∀ t t' T,
  has_type t T →
  t ==>* t' →
  ~(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.
```

16.3.8 Additional Exercises

Exercise: 2 stars, recommended (subject_expansion) Having seen the subject reduction property, it is reasonable to wonder whether the opposity property – subject *expansion* – also holds. That is, is it always the case that, if t ==> t' and *has_type* t' T, then *has_type t 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 we add the following two new rules to the reduction relation: | ST_PredTrue : (tpred ttrue) ==> (tpred tfalse) | ST_PredFalse : (tpred tfalse) ==> (tpred ttrue) Which of the following properties remain true in the presence of these rules? 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 typing relation: | T_IfFunny : forall t2 t3, has_type t2 TNat -> has_type (tif ttrue t2 t3) TNat Which of the following properties remain true in the presence of this rule? (Answer in the same style as above.)

- Determinism of *step*
- Progress
- Preservation

☐

Exercise: 2 stars (variation3) Suppose, instead, that we add this new rule to the typing relation: | T_SuccBool : forall t, has_type t TBool -> has_type (tsucc t) TBool Which of the following properties remain true in the presence of this rule? (Answer in the same style as above.)

- Determinism of *step*
- Progress
- Preservation

☐

Exercise: 2 stars (variation4) Suppose, instead, that we add this new rule to the *step* relation: | ST_Funny1 : forall t2 t3, (tif ttrue t2 t3) ==> 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 (variation5) Suppose instead that we add this rule: | ST_Funny2 : forall t1 t2 t2' t3, t2 ==> t2' -> (tif t1 t2 t3) ==> (tif 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 (variation6) Suppose instead that we add this rule: | ST_Funny3 : (tpred tfalse) ==> (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 (variation7) Suppose instead that we add this rule:

| T_Funny4 : has_type tzero 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 (variation8) Suppose instead that we add this rule:

| T_Funny5 : has_type (tpred tzero) 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, optional (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?

☐

Chapter 17

Library Smallstep

17.1 Smallstep: Small-step Operational Semantics

```
Require Export Imp.
```

The evaluators we have seen so far (e.g., the ones for *aexp*s, *bexp*s, 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 its use, 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 for some applications. Consider the variant of Imp with lists that we introduced in ImpList.v. We chose to define the meaning of programs like 0 + *nil* by specifying that a list should be interpreted as 0 when it occurs in a context expecting a number, but this was a bit of a hack. It would be better simply to say that the behavior of such a program is *undefined* – it doesn't evaluate to any result. We could easily do this: we'd just have to use the formulations of *aeval* and *beval* as inductive propositions (rather than Fixpoints), so that we can make them partial functions instead of total ones.

However, this way of defining Imp has 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 taking the successor of a boolean variable, and 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 different 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.

17.2 Relations

A *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.
```

Our main examples of such relations in this chapter will be the single-step and multi-step reduction relations on terms, ==> and ==>*, but there are many other examples – some that come to mind are 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.

The optional *Rel* chapter tells a more detailed story about how relations are treated in Coq.

17.3 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 the 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.

Tactic Notation "tm_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "C" | Case_aux c "P" ].

Module SimpleArith0.
```

Here is a standard evaluator for this language, written in the same (big-step) style as we've been using up to this point.

```
Fixpoint eval (t : tm) : nat :=
  match t with
  | C n ⇒ n
```

```
  | P a1 a2 ⇒ eval a1 + eval a2
  end.
End SimpleArith0.
```

Now, here is the same evaluator, written in exactly the same style, but formulated as an inductively defined relation. Again, we use the notation $t \parallel n$ for "t evaluates to n."

(E_Const) C n || n

t1 || n1 t2 || n2

(E_Plus) P t1 t2 || C (n1 + n2)

```
Reserved Notation " t '||' n " (at level 50, left associativity).

Inductive eval : tm → nat → Prop :=
  | E_Const : ∀ n,
      C n || n
  | E_Plus : ∀ t1 t2 n1 n2,
      t1 || n1 →
      t2 || n2 →
      P t1 t2 || (n1 + n2)

  where " t '||' n " := (eval t n).

Tactic Notation "eval_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E_Const" | Case_aux c "E_Plus" ].

Module SimpleArith1.
```

Now, here is a small-step version.

(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 → 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 : ∀ n1 t2 t2',
      t2 ==> t2' →
      P (C n1) t2 ==> P (C n1) t2'

  where " t '==>' t' " := (step t t').
Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_PlusConstConst"
  | Case_aux c "ST_Plus1" | Case_aux c "ST_Plus2" ].
```

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.

A couple of examples of reasoning with the *step* relation...

If *t1* can take a step to *t1'*, then *P t1 t2* steps to *plus 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: 2 stars (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.
```

□

One simple property of the ==> relation is that, like the evaluation relation for our language of Imp programs, 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 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 t1 t2* and the form *C n*. □

```
Definition deterministic {X: Type} (R: relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.

Theorem step_deterministic:
  deterministic step.
Proof.
  unfold deterministic. intros x y1 y2 Hy1 Hy2.
  generalize dependent y2.
  step_cases (induction Hy1) Case; intros y2 Hy2.
    Case "ST_PlusConstConst". step_cases (inversion Hy2) SCase.
      SCase "ST_PlusConstConst". reflexivity.
      SCase "ST_Plus1". inversion H2.
      SCase "ST_Plus2". inversion H2.
    Case "ST_Plus1". step_cases (inversion Hy2) SCase.
```

```
    SCase "ST_PlusConstConst". rewrite ← H0 in Hy1. inversion Hy1.
    SCase "ST_Plus1".
      rewrite ← (IHHy1 t1'0).
      reflexivity. assumption.
    SCase "ST_Plus2". rewrite ← H in Hy1. inversion Hy1.
  Case "ST_Plus2". step_cases (inversion Hy2) SCase.
    SCase "ST_PlusConstConst". rewrite ← H1 in Hy1. inversion Hy1.
    SCase "ST_Plus1". inversion H2.
    SCase "ST_Plus2".
      rewrite ← (IHHy1 t2'0).
      reflexivity. assumption. Qed.
End SimpleArith1.
```

17.3.1 Values

Let's take a moment to slightly generalize the way we state the definition of single-step reduction.

It is useful to think of the ==> 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 ==> 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 ==> relation to write *ST_Plus2* rule in a slightly more elegant way:

(ST_PlusConstConst) P (C n1) (C n2) ==> C (n1 + n2) t1 ==> t1'

(ST_Plus1) P t1 t2 ==> P t1' t2
value v1 t2 ==> t2'

(ST_Plus2) P v1 t2 ==> 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 the sake of brevity.)

```
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').

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_PlusConstConst"
  | Case_aux c "ST_Plus1" | Case_aux c "ST_Plus2" ].
```

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 t1 t2* 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. □

Most of this proof is the same as the one above. But to get maximum benefit from the exercise you should try to write it from scratch and just use the earlier one if you get stuck.

Exercise: 2 stars, optional (step_deterministic)
```
Theorem step_deterministic :
  deterministic step.
Proof.
  Admitted.
```
□

17.3.2 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 pretty easy to forget one of the rules and 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*): For all *t*:*tm*, either *t* is a value, or there exists a term *t'* such that *t* ==> *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* is either 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*. □

```
Theorem strong_progress : ∀ t,
  value t ∨ (∃ t', t ==> t').
Proof.
  tm_cases (induction t) Case.
    Case "C". left. apply v_const.
    Case "P". right. inversion IHt1.
      SCase "l". inversion IHt2.
        SSCase "l". inversion H. inversion H0.
          ∃ (C (n + n0)).
          apply ST_PlusConstConst.
```

```
      SSCase "r". inversion H0 as [t' H1].
        ∃ (P t1 t').
        apply ST_Plus2. apply H. apply H1.
    SCase "r". inversion H as [t' H0].
        ∃ (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 simply "progress.")

The idea of "making progress" can be extended to tell us something interesting about *value*s: in this language *value*s are exactly the terms that *cannot* make progress in this sense. To state this fact, let's begin by giving a name to terms that cannot make progress: We'll call them *normal forms*.

```
Definition normal_form {X:Type} (R:relation X) (t:X) : Prop :=
  ¬ ∃ t', R t t'.
```

This definition actually 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 : ∀ t,
  value t → normal_form step t.
Proof.
  unfold normal_form. intros t H. inversion H.
  intros contra. inversion contra. inversion H1.
Qed.
```

```
Lemma nf_is_value : ∀ t,
  normal_form step t → value t.
Proof.   unfold normal_form. intros t H.
  assert (G : value t ∨ ∃ t', t ==> t').
    SCase "Proof of assertion". apply strong_progress.
  inversion G.
    SCase "l". apply H0.
    SCase "r". apply ex_falso_quodlibet. apply H. assumption. Qed.
```

```
Corollary nf_same_as_value : ∀ t,
  normal_form step t ↔ value t.
Proof.
  split. apply nf_is_value. apply value_is_nf. Qed.
```

Why is this interesting? For two reasons:

- Because *value* is a syntactic concept – it is a 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, there are lots of languages in which the concepts of normal form and value do *not* coincide.

Let's examine how this can happen...

We might, for example, mistakenly define *value* so that it includes some terms that are not finished reducing.

```
Module Temp1.

Inductive value : tm → Prop :=
| v_const : ∀ n, value (C n)
| v_funny : ∀ 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').
```

Exercise: 3 stars, optional (value_not_same_as_normal_form) Lemma value_not_same_as_normal_

```
:
  ∃ t, value t ∧ ¬ normal_form step t.
Proof.
   Admitted.
   □

End Temp1.
```

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').
```

Exercise: 2 stars, optional (value_not_same_as_normal_form) Lemma value_not_same_as_normal_

```
:
  ∃ t, value t ∧ ¬ normal_form step t.
Proof.
   Admitted.
```

□

```
End Temp2.
```

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, evne 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.)

Exercise: 3 stars (value_not_same_as_normal_form') 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 plus 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, recommended (progress_bool) Just as we proved a progress theorem for plus expressions, we can do so for boolean expressions, as well.

```
Theorem strong_progress : ∀ t,
  value t ∨ (∃ t', t ==> 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: tif (tif ttrue ttrue ttrue) tfalse tfalse ==> tfalse.

Write an extra clause for the step relation that achieves this effect and prove *bool_step_prop4*.

```
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').
```

□

```
Definition bool_step_prop4 :=
         tif
            (tif ttrue ttrue ttrue)
            tfalse
            tfalse
     ==>
         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.
```

17.4 Multi-Step Reduction

Until now, we've been working with the *single-step reduction* relation ==>, which formalizes the individual steps of an *abstract machine* for executing programs.

We can also use this 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* ==>*, which relates terms *t* and *t'* if *t* can reach *t'* by any number of single reduction steps (including zero steps!).

- Then we define a "result" of a term *t* as a normal form that *t* can reach by multi-step reduction.

17.4.1 Definitions

Since we'll want to reuse the idea of multi-step reduction many times in this and future chapters, let's take a little extra trouble here and define it generically. Given a relation *R*, we define a relation *multi 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.
```

The effect of this definition is that *multi R* relates two elements *x* and *y* of *X* if either *x* = *y* or else there is some (possibly empty) sequence *z1*, *z2*, ..., *zn* such that: R x z1 R z1 z2 ... R zn y

```
Tactic Notation "multi_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "multi_refl" | Case_aux c "multi_step" ].

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.
```

The crucial properties of the *multi R* relation are

- *multi* R is reflexive
- *multi* R is transitive
- *multi* R relates everything related by R

```
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.
  multi_cases (induction G) Case.
    Case "multi_refl". assumption.
    Case "multi_step".
      apply multi_step with y. assumption.
      apply IHG. assumption. Qed.
```

We now write ==>* for the *multi step* relation – i.e., the relation that relates two terms *t* and *t'* if we can get from *t* to *t'* using the *step* relation zero or more times.

```
Definition multistep := multi step.
Notation " t '==>*' t' " := (multistep t t') (at level 40).
```

17.4.2 Examples

```
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 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.
  □
```

17.4.3 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' : tm) :=
  (t ==>* t' ∧ 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 *normal_form t 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 : ∀ t1 t1' t2,
     t1 ==>* t1' →
     P t1 t2 ==>* P t1' t2.
Proof.
  intros t1 t1' t2 H. multi_cases (induction H) Case.
    Case "multi_refl". apply multi_refl.
```

```
    Case "multi_step". 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 : ∀ t1 t2 t2',
     value t1 →
     t2 ==>* t2' →
     P t1 t2 ==>* P t1 t2'.
Proof.
  Admitted.
```

☐

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 $t2$ 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.
  tm_cases (induction t) Case.
    Case "C".
      ∃ (C n).
      split.
      SCase "l". apply multi_refl.
      SCase "r".
        rewrite nf_same_as_value. apply v_const.
    Case "P".
      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.
  SCase "l".
    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.
  SCase "r".
    rewrite nf_same_as_value. apply v_const. Qed.
```

17.4.4 Equivalence of Big-Step and Small-Step Reduction

Having defined the operational semantics of our tiny programming language in two different styles, 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 to you.

Exercise: 3 stars (eval__multistep) `Theorem eval__multistep : ∀ t n, t || n → t ==>* C n.`

The key idea behind the proof comes from 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 (eval__multistep_inf) Write a detailed informal version of the proof of *eval__multistep*.

☐

For the other direction of the correspondence, we need one lemma, which establishes a relation between single-step reduction and big-step evaluation.

Exercise: 3 stars (step__eval) Lemma step__eval : ∀ *t t' n*,

```
    t ==> t' →
    t' || n →
    t || n.
Proof.
  intros t t' n Hs. generalize dependent n.
   Admitted.
```

☐

The main theorem is now straightforward to prove, once it is stated correctly. The proof proceeds by induction on the multipstep reduction sequence that is buried in the hypothesis *normal_form_of t v*. Make sure you understand the statement before you start to work on the proof.

Exercise: 3 stars (multistep__eval) Theorem multistep__eval : ∀ *t v*,

```
  normal_form_of t v → ∃ n, v = C n ∧ t || n.
Proof.
   Admitted.
```

☐

17.4.5 Additional Exercises

Exercise: 4 stars (combined_properties) We've considered the 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.

Tactic Notation "tm_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "C" | Case_aux c "P"
  | Case_aux c "ttrue" | Case_aux c "tfalse" | Case_aux c "tif" ].

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').

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_PlusConstConst"
  | Case_aux c "ST_Plus1" | Case_aux c "ST_Plus2"
  | Case_aux c "ST_IfTrue" | Case_aux c "ST_IfFalse" | Case_aux c "ST_If" ].
```

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.
```

17.5 Small-Step Imp

For a more serious example, here is the 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 $==>a$ and $==>b$, respectively, 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 – they aren't needed in the definition of $==>b$ 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,
      AId 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 (ble_nat 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 '/' st '==>' t' '/' st' "
                  (at level 40, st at level 39, t' at level 39).

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 / (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 : ∀ st c1 c2,
      IFB BFalse THEN c1 ELSE c2 FI / st ==> c2 / st
  | CS_IfStep : ∀ st b b' c1 c2,
      b / st ==>b b' →
      IFB b THEN c1 ELSE c2 FI / st ==> (IFB b' THEN c1 ELSE c2 FI) / st
  | CS_While : ∀ st b c1,
          (WHILE b DO c1 END) / st
      ==> (IFB b THEN (c1; (WHILE b DO c1 END)) ELSE SKIP FI) / st

  where " t '/' st '==>' t' '/' st' " := (cstep (t,st) (t',st')).
```

17.6 Concurrent Imp (Optional)

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
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com

  | CPar : com → com → com.

Tactic Notation "com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "SKIP" | Case_aux c "::=" | Case_aux c ";"
  | Case_aux c "IFB" | Case_aux c "WHILE" | Case_aux c "PAR" ].

Notation "'SKIP'" :=
  CSkip.
Notation "l '::=' a" :=
  (CAss l 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'" :=
  (CIf e1 e2 e3) (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 / (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 : ∀ st c1 c2,
      (IFB BFalse THEN c1 ELSE c2 FI) / st ==> c2 / st
  | CS_IfStep : ∀ st b b' c1 c2,
      b /st ==>b b' →
      (IFB b THEN c1 ELSE c2 FI) / st ==> (IFB b' THEN c1 ELSE c2 FI) / st
  | CS_While : ∀ st b c1,
      (WHILE b DO c1 END) / st ==>
               (IFB b THEN (c1; (WHILE b DO c1 END)) ELSE SKIP FI) / st

  | CS_Par1 : ∀ st c1 c1' c2 st',
      c1 / st ==> c1' / st' →
      (PAR c1 WITH c2 END) / st ==> (PAR c1' WITH c2 END) / st'
  | CS_Par2 : ∀ st c1 c2 c2' st',
      c2 / st ==> c2' / st' →
      (PAR c1 WITH c2 END) / st ==> (PAR c1 WITH c2' END) / st'
  | CS_ParDone : ∀ st,
      (PAR SKIP WITH SKIP END) / st ==> SKIP / st
  where " t '/' st '==>' t' '/' st' " := (cstep (t,st) (t',st')).
Definition cmultistep := multi cstep.
Notation " t '/' st '==>*' 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 (AId Y) (ANum 0) DO
      X ::= APlus (AId X) (ANum 1)
    END
  END.
```

In particular, it can terminate with X set to 0:

```
Example par_loop_example_0:
  ∃ st',
       par_loop / empty_state ==>* SKIP / st'
```

```
    ∧ 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:
  ∃ st',
       par_loop / empty_state ==>* SKIP / st'
    ∧ 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 : ∀ n st,
  st X = n ∧ st Y = 0 →
  par_loop / st ==>* par_loop / (update st X (S n)).
Proof.
  Admitted.
```

Exercise: 3 stars, optional

```
Lemma par_body_n : ∀ n st,
  st X = 0 ∧ st Y = 0 →
  ∃ st',
    par_loop / st ==>* par_loop / st' ∧ st' X = n ∧ st' Y = 0.
Proof.
  Admitted.
```

... the above loop can exit with X having any value whatsoever.

```
Theorem par_loop_any_X:
  ∀ n, ∃ st',
    par_loop / empty_state ==>* SKIP / st'
    ∧ st' X = n.
Proof.
  intros n.
  destruct (par_body_n n empty_state).
    split; unfold update; reflexivity.
  rename x into st.
  inversion H as [H' [HX HY]]; clear H.
```

```
  ∃ (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 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.
  apply multi_refl.

  rewrite update_neq. assumption. reflexivity.
Qed.

End CImp.
```

Chapter 18

Library Rel

18.1 Rel: Properties of Relations

```
Require Export SfLib.
```

A (binary) *relation* is just a parameterized proposition. As you know from your undergraduate discrete math course, there are a lot of ways of discussing and describing relations *in general* – ways of classifying relations (are they reflexive, transitive, etc.), theorems that can be proved generically about classes of relations, constructions that build one relation from another, etc. Let us pause here to review a few that will be useful in what follows.

A (binary) relation *on* a set X is a proposition parameterized by two Xs – i.e., it is a logical assertion involving two values from the set X.

```
Definition relation (X: Type) := X→X→Prop.
```

Somewhat confusingly, the Coq standard library hijacks the generic term "relation" for this specific instance. 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, while 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-that-or-equal-to relation which we usually write like this $n1 \leq n2$.

```
Print le.
Check le : nat → nat → Prop.
Check le : relation nat.
```

18.2 Basic Properties of Relations

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., if $R\ x\ y1$ and $R\ x\ y2$ together imply $y1 = y2$.

```
Definition partial_function {X: Type} (R: relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.
```

For example, the *next_nat* relation defined in Logic.v is a partial function.

However, the $\leq$ relation on numbers is not a partial function.

This can be shown by contradiction. In short: 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 :
  ¬ (partial_function le).
Proof.
  unfold not. unfold partial_function. intros Hc.
  assert (0 = 1) as Nonsense.
   Case "Proof of assertion".
   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 Logic.v is not a partial function.

□

Exercise: 2 stars, optional Show that the *empty_relation* defined in Logic.v is a partial function.

□

A *reflexive* relation on a set X is one for which every element of X is related to itself.

```
Definition reflexive {X: Type} (R: relation X) :=
  ∀ a : X, R a a.

Theorem le_reflexive :
  reflexive le.
Proof.
  unfold reflexive. intros n. apply le_n. Qed.
```

A relation R is *transitive* if $R\ a\ c$ holds whenever $R\ a\ b$ and $R\ b\ c$ do.

```
Definition transitive {X: Type} (R: relation X) :=
  ∀ a b c : X, (R a b) → (R b c) → (R a c).

Theorem le_trans :
  transitive le.
Proof.
  intros n m o Hnm Hmo.
```

```
  induction Hmo.
  Case "le_n". apply Hnm.
  Case "le_S". 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 : ∀ n m, S n ≤ m → n ≤ 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,
  (S $n \leq$ 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, ~(S $n \leq n$)

A formal proof of this is an optional exercise below, but try the informal proof without doing the formal proof first.

Proof: ☐

Exercise: 1 star, optional Theorem le_Sn_n : $\forall$ n,
  $\neg$ (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, here are a few more common ones.

A relation R is *symmetric* if R a b implies R b a.

Definition symmetric {X: Type} (R: relation X) :=
  $\forall$ a b : X, (R a b) $\rightarrow$ (R b a).

Exercise: 2 stars, optional Theorem le_not_symmetric :
  $\neg$ (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: Type} (R: 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 : ∀ n m p,
  n < m →
  m ≤ S p →
  n ≤ p.
Proof.
  Admitted.
  □

A relation is an *equivalence* if it's reflexive, symmetric, and transitive.

```
Definition equivalence {X:Type} (R: relation X) :=
  (reflexive R) ∧ (symmetric R) ∧ (transitive R).
```

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:Type} (R: relation X) :=
  (reflexive R) ∧ (antisymmetric R) ∧ (transitive R).
```

A preorder is almost like a partial order, but doesn't have to be antisymmetric.

```
Definition preorder {X:Type} (R: relation X) :=
  (reflexive R) ∧ (transitive R).

Theorem le_order :
  order le.
Proof.
  unfold order. split.
    Case "refl". apply le_reflexive.
    split.
      Case "antisym". apply le_antisymmetric.
      Case "transitive.". apply le_trans. Qed.
```

18.3 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: Type} (R: relation A) : relation A :=
    | rt_step : ∀ x y, R x y → clos_refl_trans R x y
    | rt_refl : ∀ x, clos_refl_trans R x x
    | rt_trans : ∀ x y z,
          clos_refl_trans R x y →
          clos_refl_trans R y z →
          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 : ∀ n m,
  (n ≤ m) ↔ ((clos_refl_trans next_nat) n m).
Proof.
  intros n m. split.
    Case "->".
      intro H. induction H.
      SCase "le_n". apply rt_refl.
      SCase "le_S".
        apply rt_trans with m. apply IHle. apply rt_step. apply nn.
    Case "<-".
      intro H. induction H.
      SCase "rt_step". inversion H. apply le_S. apply le_n.
      SCase "rt_refl". apply le_n.
      SCase "rt_trans".
        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 – the "nondeterminism" of the *rt_trans* rule can sometimes lead to tricky inductions.

Here is a more useful definition...

```
Inductive refl_step_closure {X:Type} (R: relation X) : relation X :=
  | rsc_refl : ∀ (x : X), refl_step_closure R x x
  | rsc_step : ∀ (x y z : X),
                    R x y →
                    refl_step_closure R y z →
                    refl_step_closure R x z.
```

(Note that, aside from the naming of the constructors, this definition is the same as the *multi* step relation used in many other chapters.)

(The following `Tactic Notation` definitions are explained in Imp.v. You can ignore them if you haven't read that chapter yet.)

```
Tactic Notation "rt_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "rt_step" | Case_aux c "rt_refl"
  | Case_aux c "rt_trans" ].

Tactic Notation "rsc_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "rsc_refl" | Case_aux c "rsc_step" ].
```

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 *refl_step_closure* mimics the behavior of the two "missing" *clos_refl_trans* constructors.

```
Theorem rsc_R : ∀ (X:Type) (R:relation X) (x y : X),
       R x y → refl_step_closure R x y.
Proof.
  intros X R x y H.
  apply rsc_step with y. apply H. apply rsc_refl. Qed.
```

Exercise: 2 stars, optional (rsc_trans) Theorem rsc_trans :

```
  ∀ (X:Type) (R: relation X) (x y z : X),
      refl_step_closure R x y →
      refl_step_closure R y z →
      refl_step_closure 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 :

```
         ∀ (X:Type) (R: relation X) (x y : X),
  clos_refl_trans R x y ↔ refl_step_closure R x y.
Proof.
   Admitted.
```

□

Chapter 19

Library HoareAsLogic

19.1 HoareAsLogic: Hoare Logic as a Logic

```
Require Export 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.

```
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 ⇒ P st ∧ bassn b st) c P →
    hoare_proof P (WHILE b DO c END) (fun st ⇒ P st ∧ ¬ (bassn b st))
  | H_Consequence : ∀ (P Q P' Q' : Assertion) c,
    hoare_proof P' c Q' →
    (∀ st, P st → P' st) →
```

```
    (∀ st, Q' st → Q st) →
    hoare_proof P c Q
  | H_Consequence_pre : ∀ (P Q P' : Assertion) c,
    hoare_proof P' c Q →
    (∀ st, P st → P' st) →
    hoare_proof P c Q
  | H_Consequence_post : ∀ (P Q Q' : Assertion) c,
    hoare_proof P c Q' →
    (∀ st, Q' st → Q st) →
    hoare_proof P c Q.

Tactic Notation "hoare_proof_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "H_Skip" | Case_aux c "H_Asgn" | Case_aux c "H_Seq"
  | Case_aux c "H_If" | Case_aux c "H_While" | Case_aux c "H_Consequence"
  | Case_aux c "H_Consequence_pre" | Case_aux c "H_Consequence_post" ].
```

For example, let's construct a proof object representing a derivation for the hoare triple [1] X::=X+1; X::=X+2 [2]. We can use Coq's tactics to help us construct the proof object.

```
Example sample_proof
             : hoare_proof
                 (assn_sub X (APlus (AId X) (ANum 1))
                   (assn_sub X (APlus (AId X) (ANum 2))
                     (fun st ⇒ st X = 3) ))
                 (X ::= APlus (AId X) (ANum 1); (X ::= APlus (AId X) (ANum 2)))
                 (fun st ⇒ st X = 3).
Proof.
  apply H_Seq with (assn_sub X (APlus (AId X) (ANum 2))
                     (fun st ⇒ st X = 3)).
  apply H_Asgn. apply H_Asgn.
Qed.
```

Exercise: 2 stars Prove that such proof objects represent true claims.

```
Theorem hoare_proof_sound : ∀ P c Q,
  hoare_proof P c Q → {{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

[1] assn_subX(X+1)(assn_subX(X+2)(X=3))
[2] X=3

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 $\{\{P\}\}\ c\ \{\{True\}\}$ 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:
  ∀ c P, hoare_proof P c (fun _ ⇒ True).
Proof.
  intro c.
  com_cases (induction c) Case; intro P.
  Case "SKIP".
    eapply H_Consequence_pre.
    apply H_Skip.
    intros. apply I.
  Case "::=".
    eapply H_Consequence_pre.
    apply H_Asgn.
    intros. apply I.
  Case ";".
    eapply H_Consequence_pre.
    eapply H_Seq.
    apply (IHc1 (fun _ ⇒ True)).
    apply IHc2.
    intros. apply I.
  Case "IFB".
    apply H_Consequence_pre with (fun _ ⇒ True).
    apply H_If.
    apply IHc1.
    apply IHc2.
    intros. apply I.
  Case "WHILE".
    eapply H_Consequence.
    eapply H_While.
    eapply IHc.
    intros; apply I.
    intros; apply I.
Qed.
```

Similarly, we can show that $\{\{False\}\}\ c\ \{\{Q\}\}$ is provable for any c and Q.

```
Lemma False_and_P_imp: ∀ P Q,
  False ∧ P → 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:
  ∀ c Q, hoare_proof (fun _ ⇒ False) c Q.
Proof.
  intros c.
  com_cases (induction c) Case; intro Q.
  Case "SKIP". pre_false_helper H_Skip.
  Case "::=". pre_false_helper H_Asgn.
  Case ";". pre_false_helper H_Seq. apply IHc1. apply IHc2.
  Case "IFB".
    apply H_If; eapply H_Consequence_pre.
    apply IHc1. intro. eapply False_and_P_imp.
    apply IHc2. intro. eapply False_and_P_imp.
  Case "WHILE".
    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.
```

This 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 *Hoare* on formalizing decorated programs shows how we can do even better.

Chapter 20

Library HoareList

20.1 HoareList: Hoare Logic with Lists

```
Require Export SfLib.
```

20.2 Imp Programs with Lists

There are only so many numeric functions with interesting properties that have simple proofs. (Of course, there are lots of interesting functions on numbers and they have many interesting properties – this is the whole field of number theory! – but proving these properties often requires developing a lot of supporting lemmas.) In order to able to write a few more programs to reason about, we introduce here an extended version of Imp where variables can range over both numbers and lists of numbers. The basic operations are extended to also include taking the head and tail of lists, and testing lists for nonemptyness.

To do this, we only need to change the definitions of *state*, *aexp*, *aeval*, *bexp*, and *beval*. The definitions of *com* and *ceval* can be reused verbatim, although we need to copy-and-paste them in the context of the new definitions.

We start by repeating some material from chapter *Imp*.

20.2.1 Repeated Definitions

```
Inductive id : Type :=
  Id : nat → id.

Definition beq_id id1 id2 :=
  match (id1 , id2) with
    (Id n1 , Id n2) ⇒ beq_nat n1 n2
  end.

Theorem beq_id_refl : ∀ i,
  true = beq_id i i.
```

```
Proof.
  intros. destruct i.
  apply beq_nat_refl. Qed.

Theorem beq_id_eq : ∀ i1 i2,
  true = beq_id i1 i2 → i1 = i2.
Proof.
  intros i1 i2 H.
  destruct i1. destruct i2.
  apply beq_nat_eq in H. subst.
  reflexivity. Qed.

Theorem beq_id_false_not_eq : ∀ i1 i2,
  beq_id i1 i2 = false → i1 ≠ i2.
Proof.
  intros i1 i2 H.
  destruct i1. destruct i2.
  apply beq_nat_false in H.
  intros C. apply H. inversion C. reflexivity. Qed.

Theorem not_eq_beq_id_false : ∀ i1 i2,
  i1 ≠ i2 → beq_id i1 i2 = false.
Proof.
  intros i1 i2 H.
  destruct i1. destruct i2.
  assert (n ≠ n0).
    intros C. subst. apply H. reflexivity.
  apply not_eq_beq_false. assumption. Qed.

Definition X : id := Id 0.
Definition Y : id := Id 1.
Definition Z : id := Id 2.
```

20.2.2 Extensions

Now we come to the key changes.

Rather than evaluating to a *nat*, an *aexp* in our new language will evaluate to a *value* – an element of type *val* – which can be either a *nat* or a list of *nats*.

Similarly, *states* will now map identifiers to *vals* rather than *nats*, so that we can store lists in mutable variables.

```
Inductive val : Type :=
| VNat : nat → val
| VList : list nat → val.

Definition state := id → val.

Definition empty_state : state := fun _ ⇒ VNat 0.
```

```
Definition update (st : state) (X:id) (v : val) : state :=
  fun X' ⇒ if beq_id X X' then v else st X'.
```

Imp does not have a static type system, so nothing prevents the programmer from e.g. adding two lists or taking the head of a number. We have to decide what to do in such nonsensical situations.

We adopt a simple solution: if an arithmetic function is given a list as an argument we treat the list as if it was the number 0. Similarly, if a list function is given a number as an argument we treat the number as if it was *nil*. (Cf. Javascript, where adding 3 to the empty list evaluates to 3...)

The two functions *asnat* and *aslist* interpret *val*s in a numeric or a list context; *aeval* calls these whenever it evaluates an arithmetic or a list operation.

```
Definition asnat (v : val) : nat :=
  match v with
  | VNat n ⇒ n
  | VList _ ⇒ 0
  end.

Definition aslist (v : val) : list nat :=
  match v with
  | VNat n ⇒ []
  | VList xs ⇒ xs
  end.
```

Now we fill in the definitions of abstract syntax and evaluation functions for arithmetic and boolean expressions.

```
Inductive aexp : Type :=
  | ANum : nat → aexp
  | AId : id → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp

  | AHead : aexp → aexp
  | ATail : aexp → aexp
  | ACons : aexp → aexp → aexp
  | ANil : aexp.

Tactic Notation "aexp_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ANum" | Case_aux c "AId" | Case_aux c "APlus"
  | Case_aux c "AMinus" | Case_aux c "AMult"
  | Case_aux c "AHead" | Case_aux c "ATail"
  | Case_aux c "ACons" | Case_aux c "ANil" ].
```

```
Definition tail (l : list nat) :=
  match l with
  | x :: xs ⇒ xs
  | [] ⇒ []
  end.

Definition head (l : list nat) :=
  match l with
  | x :: xs ⇒ x
  | [] ⇒ 0
  end.

Fixpoint aeval (st : state) (e : aexp) : val :=
  match e with
  | ANum n ⇒ VNat n
  | AId i ⇒ st i
  | APlus a1 a2 ⇒ VNat (asnat (aeval st a1) + asnat (aeval st a2))
  | AMinus a1 a2 ⇒ VNat (asnat (aeval st a1) - asnat (aeval st a2))
  | AMult a1 a2 ⇒ VNat (asnat (aeval st a1) × asnat (aeval st a2))

  | ATail a ⇒ VList (tail (aslist (aeval st a)))
  | AHead a ⇒ VNat (head (aslist (aeval st a)))
  | ACons a1 a2 ⇒ VList (asnat (aeval st a1) :: aslist (aeval st a2))
  | ANil ⇒ VList []
  end.
```

We extend *bexps* with an operation to test if a list is nonempty and adapt *beval* acordingly.

```
Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp → aexp → bexp
  | BLe : aexp → aexp → bexp
  | BNot : bexp → bexp
  | BAnd : bexp → bexp → bexp

  | BIsCons : aexp → bexp.

Tactic Notation "bexp_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "BTrue" | Case_aux c "BFalse" | Case_aux c "BEq"
  | Case_aux c "BLe" | Case_aux c "BNot" | Case_aux c "BAnd"
  | Case_aux c "BIsCons" ].

Fixpoint beval (st : state) (e : bexp) : bool :=
  match e with
```

```
| BTrue ⇒ true
| BFalse ⇒ false
| BEq a1 a2 ⇒ beq_nat (asnat (aeval st a1)) (asnat (aeval st a2))
| BLe a1 a2 ⇒ ble_nat (asnat (aeval st a1)) (asnat (aeval st a2))
| BNot b1 ⇒ negb (beval st b1)
| BAnd b1 b2 ⇒ andb (beval st b1) (beval st b2)

| BIsCons a ⇒ match aslist (aeval st a) with
                 | _::_ ⇒ true
                 | [] ⇒ false
               end
end.
```

20.2.3 Repeated Definitions

Now we need to repeat a little bit of low-level work from Imp.v, plus the definitions of *com* and *ceval*. There are no interesting changes – it's just a matter of repeating the same definitions, lemmas, and proofs in the context of the new definitions of arithmetic and boolean expressions.

(Is all this cutting and pasting really necessary? No: Coq includes a powerful module system that we could use to abstract the repeated definitions with respect to the varying parts. But explaining how it works would distract us from the topic at hand.)

```
Theorem update_eq : ∀ n V st,
  (update st V n) V = n.
Proof.
  intros n V st.
  unfold update.
  rewrite ← beq_id_refl.
  reflexivity.
Qed.

Theorem update_neq : ∀ V2 V1 n st,
  beq_id V2 V1 = false →
  (update st V2 n) V1 = (st V1).
Proof.
  intros V2 V1 n st Hneq.
  unfold update.
  rewrite → Hneq.
  reflexivity. Qed.

Theorem update_shadow : ∀ x1 x2 k1 k2 (f : state),
   (update (update f k2 x1) k2 x2) k1 = (update f k2 x2) k1.
Proof.
  intros x1 x2 k1 k2 f.
```

```
  unfold update.
  destruct (beq_id k2 k1); reflexivity. Qed.

Theorem update_same : ∀ x1 k1 k2 (f : state),
  f k1 = x1 →
  (update f k1 x1) k2 = f k2.
Proof.
  intros x1 k1 k2 f Heq.
  unfold update. subst.
  remember (beq_id k1 k2) as b.
  destruct b.
  Case "true".
    apply beq_id_eq in Heqb. subst. reflexivity.
  Case "false".
    reflexivity. Qed.

Theorem update_permute : ∀ x1 x2 k1 k2 k3 f,
  beq_id k2 k1 = false →
  (update (update f k2 x1) k1 x2) k3 = (update (update f k1 x2) k2 x1) k3.
Proof.
  intros x1 x2 k1 k2 k3 f H.
  unfold update.
  remember (beq_id k1 k3) as b13.
  remember (beq_id k2 k3) as b23.
  apply beq_id_false_not_eq in H.
  destruct b13; try reflexivity.
  Case "true".
    destruct b23; try reflexivity.
    SCase "true".
      apply beq_id_eq in Heqb13.
      apply beq_id_eq in Heqb23.
      subst. apply ex_falso_quodlibet. apply H. reflexivity. Qed.
```

We can keep exactly the same old definitions of *com* and *ceval*.

```
Inductive com : Type :=
  | CSkip : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.

Tactic Notation "com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "SKIP" | Case_aux c "::=" | Case_aux c ";"
  | Case_aux c "IFB" | Case_aux c "WHILE" ].
```

```
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'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).

Reserved Notation "c1 '/' st '||' st'" (at level 40, st at level 39).

Inductive ceval : state → com → state → Prop :=
  | E_Skip : ∀ st,
      SKIP / st || st
  | E_Asgn : ∀ st a1 n X,
      aeval st a1 = n →
      (X ::= a1) / st || (update st X n)
  | E_Seq : ∀ c1 c2 st st' st'',
      c1 / st || st' →
      c2 / st' || st'' →
      (c1 ; c2) / st || st''
  | E_IfTrue : ∀ st st' b1 c1 c2,
      beval st b1 = true →
      c1 / st || st' →
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_IfFalse : ∀ st st' b1 c1 c2,
      beval st b1 = false →
      c2 / st || st' →
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_WhileEnd : ∀ b1 st c1,
      beval st b1 = false →
      (WHILE b1 DO c1 END) / st || st
  | E_WhileLoop : ∀ st st' st'' b1 c1,
      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 st c1 st').

Tactic Notation "ceval_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E_Skip" | Case_aux c "E_Asgn" | Case_aux c "E_Seq"
```

| *Case_aux c* "E_IfTrue" | *Case_aux c* "E_IfFalse"
| *Case_aux c* "E_WhileEnd" | *Case_aux c* "E_WhileLoop"].

20.3 Hoare Rules

We copy verbatim the Hoare rules from Hoare.v.

(hoare_asgn) [1] X::=a [2]

(hoare_skip) [3] SKIP [4]
[5] c1 [6] [7] c2 [8]

(hoare_seq) [9] c1;c2 [10]
[11] c1 [12] [13] c2 [14]

(hoare_if) [15] IFB b THEN c1 ELSE c2 FI [16]
[17] c [18]

(hoare_while) [19] WHILE b DO c END [20]
[21] c [22] P ~~> P' Q' ~~> Q

---

[1] `assn_subXaQ`
[2] `Q`
[3] `P`
[4] `P`
[5] `P`
[6] `Q`
[7] `Q`
[8] `R`
[9] `P`
[10] `R`
[11] `P/\b`
[12] `Q`
[13] `P/\~b`
[14] `Q`
[15] `P`
[16] `Q`
[17] `P/\b`
[18] `P`
[19] `P`
[20] `P/\~b`
[21] `P'`
[22] `Q'`

(hoare_consequence) [23] c [24]

```
Definition Assertion := state → Prop.

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.

Open Scope hoare_spec_scope.

Definition assn_sub X a Q : Assertion :=
  fun (st : state) ⇒
    Q (update st X (aeval st a)).

Theorem hoare_asgn : ∀ Q X a,
  {{assn_sub X a Q}} (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.

Theorem hoare_skip : ∀ P,
     {{P}} SKIP {{P}}.
Proof.
  intros P st st' H HP. inversion H. subst.
  assumption. Qed.

Theorem hoare_seq : ∀ P Q R c1 c2,
     {{Q}} c2 {{R}} →
     {{P}} c1 {{Q}} →
     {{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.

Definition bassn b : Assertion :=
  fun st ⇒ (beval st b = true).

Lemma bexp_eval_true : ∀ b st,
```

[23] P
[24] Q

```
  beval st b = true → (bassn b) st.
Proof.
  intros b st Hbe.
  unfold bassn. assumption. Qed.

Lemma bexp_eval_false : ∀ b st,
  beval st b = false → ¬ ((bassn b) st).
Proof.
  intros b st Hbe contra.
  unfold bassn in contra.
  rewrite → contra in Hbe. inversion Hbe. Qed.

Theorem hoare_if : ∀ P Q b c1 c2,
  {{fun st ⇒ P st ∧ bassn b st}} c1 {{Q}} →
  {{fun st ⇒ P st ∧ ~(bassn b st)}} c2 {{Q}} →
  {{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.
  Case "b is true".
    apply (HTrue st st').
      assumption.
      split. assumption.
             apply bexp_eval_true. assumption.
  Case "b is false".
    apply (HFalse st st').
      assumption.
      split. assumption.
             apply bexp_eval_false. assumption. Qed.

Lemma hoare_while : ∀ P b c,
  {{fun st ⇒ P st ∧ bassn b st}} c {{P}} →
  {{P}} WHILE b DO c END {{fun st ⇒ P st ∧ ¬ (bassn b st)}}.
Proof.
  intros P b c Hhoare st st' He HP.
  remember (WHILE b DO c END) as wcom.
  ceval_cases (induction He) Case; try (inversion Heqwcom); subst.

  Case "E_WhileEnd".
    split. assumption. apply bexp_eval_false. assumption.

  Case "E_WhileLoop".
    apply IHHe2. reflexivity.
    apply (Hhoare st st'); try assumption.
      split. assumption. apply bexp_eval_true. assumption. Qed.

Definition assert_implies (P Q : Assertion) : Prop :=
```

```
  ∀ st, P st → Q st.
Notation "P ~~> Q" := (assert_implies P Q) (at level 80).
Notation "P <~~> Q" := (P ~~> Q ∧ Q ~~> P) (at level 80).
Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P ~~> P' →
  {{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 : ∀ (P Q Q' : Assertion) c,
  {{P}} c {{Q'}} →
  Q' ~~> Q →
  {{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.
Theorem hoare_consequence : ∀ (P P' Q Q' : Assertion) c,
  {{P'}} c {{Q'}} →
  P ~~> P' →
  Q' ~~> Q →
  {{P}} c {{Q}}.
Proof.
  intros P P' Q Q' c Hht HPP' HQ'Q.
  intros st st' Hc HP.
  apply HQ'Q. apply (Hht st st'). assumption.
  apply HPP'. assumption. Qed.
```

20.3.1 Reasoning About Programs with Lists

Now let's look at a formal Hoare Logic proof for a program that works with lists. We will verify the following program, which checks if the number Y occurs in the list X, and if so sets Z to 1.

```
Definition list_member :=
  WHILE BIsCons (AId X) DO
    IFB (BEq (AId Y) (AHead (AId X))) THEN
      Z ::= (ANum 1)
```

```
    ELSE
      SKIP
    FI;
    X ::= ATail (AId X)
  END.
```

The informal proof looks like this: [25] => [26] WHILE (BIsCons X) DO [27] IFB (Y == head X) THEN [28] => [29] Z ::= 1 [30] ELSE [31] => [32] SKIP [33] FI; [34] X ::= ATail X [35] END [36] => [37]

The only interesting part of the proof is the choice of loop invariant: exists p, p ++ X = l /\ (Z = 1 <-> appears_in n p) This states that at each iteration of the loop, the original list l is equal to the append of the current value of X and some other list p which is not the value of any variable in the program, but keeps track of enough information from the original state to make the proof go through. (Such a p is sometimes called a "ghost variable").

In order to show that such a list p exists, in each iteration we add the head of X to the *end* of p. This needs the function *snoc*, from Poly.v.

```
Fixpoint snoc {X:Type} (l:list X) (v:X) : (list X) :=
  match l with
  | nil ⇒ [ v ]
  | cons h t ⇒ h :: (snoc t v)
  end.
```

The main proof uses several lemmas about *snoc* and ++.

```
Lemma snoc_equation : ∀ (A : Type) (h:A) (x y : list A),
  snoc x h ++ y = x ++ h :: y.
Proof.
  intros A h x y.
  induction x.
    Case "x = []". reflexivity.
    Case "x = cons". simpl. rewrite IHx. reflexivity.
Qed.
```

```
Lemma appears_in_snoc1 : ∀ a l,
```

[25] X=l/\Y=n/\Z=0
[26] Y=n/\existsp,p++X=l/\(Z=1<->appears_innp)
[27] Y=n/\(existsp,p++X=l/\(Z=1<->appears_innp))/\(BIsConsX)
[28] Y=n/\(existsp,p++X=l/\(Z=1<->appears_innp))/\(BIsConsX)/\Y==AHeadX
[29] Y=n/\(existsp,p++tailX=l/\(1=1<->appears_innp))
[30] Y=n/\(existsp,p++tailX=l/\(Z=1<->appears_innp))
[31] Y=n/\(existsp,p++X=l/\(Z=1<->appears_innp))/\(BIsConsX)/\~(Y==headX)
[32] Y=n/\(existsp,p++tailX=l/\(Z=1<->appears_innp))
[33] Y=n/\(existsp,p++tailX=l/\(Z=1<->appears_innp))
[34] Y=n/\(existsp,p++tailX=l/\(Z=1<->appears_innp))
[35] Y=n/\(existsp,p++X=l/\(Z=1<->appears_innp))
[36] Y=n/\(existsp,p++X=l/\(Z=1<->appears_innp))/\~(BIsConsX)
[37] Z=1<->appears_innl

```
  appears_in a (snoc l a).
Proof.
  induction l.
    Case "l = []". apply ai_here.
    Case "l = cons". simpl. apply ai_later. apply IHl.
Qed.

Lemma appears_in_snoc2 : ∀ a b l,
  appears_in a l →
  appears_in a (snoc l b).
Proof.
  induction l; intros H; inversion H; subst; simpl.
    Case "l = []". apply ai_here.
    Case "l = cons". apply ai_later. apply IHl. assumption.
Qed.

Lemma appears_in_snoc3 : ∀ a b l,
   appears_in a (snoc l b) →
   (appears_in a l ∨ a = b).
Proof.
   induction l; intros H.
   Case "l = []". inversion H.
     SCase "ai_here". right. reflexivity.
     SCase "ai_later". left. assumption.
   Case "l = cons". inversion H; subst.
     SCase "ai_here". left. apply ai_here.
     SCase "ai_later". destruct (IHl H1).
       left. apply ai_later. assumption.
       right. assumption.
Qed.

Lemma append_singleton_equation : ∀ (x : nat) l l',
  (l ++ [x]) ++ l' = l ++ x :: l'.
Proof.
  intros x l l'.
  induction l.
    reflexivity.
    simpl. rewrite IHl. reflexivity.
Qed.

Lemma append_nil : ∀ (A : Type) (l : list A),
  l ++ [] = l.
Proof.
  induction l.
    reflexivity.
```

```
    simpl. rewrite IHl. reflexivity.
Qed.

Theorem list_member_correct : ∀ l n,
  {{ fun st ⇒ aslist (st X) = l ∧ asnat (st Y) = n ∧ asnat (st Z) = 0 }}
  list_member
  {{ fun st ⇒ asnat (st Z) = 1 ↔ appears_in n l }}.
Proof.
  intros l n.
  eapply hoare_consequence.
  apply hoare_while with (P := fun st ⇒
     asnat (st Y) = n
     ∧ ∃ p, p ++ aslist (st X) = l
                  ∧ (asnat (st Z) = 1 ↔ appears_in n p)).
    eapply hoare_seq.
    apply hoare_asgn.
    apply hoare_if.
    Case "If taken".
      eapply hoare_consequence_pre.
      apply hoare_asgn.
      intros st [[[H1 [p [H2 H3]]] H9] H10].
      unfold assn_sub. split.
        rewrite update_neq; try reflexivity.
        rewrite update_neq; try reflexivity.
        assumption.
        remember (aslist (st X)) as x.
        destruct x as [|h x'].
          unfold bassn in H9. unfold beval in H9. unfold aeval in H9.
          rewrite ← Heqx in H9. inversion H9.

          ∃ (snoc p h).
          rewrite update_eq.
          unfold aeval. rewrite update_neq; try reflexivity.
          rewrite ← Heqx.
          split.
            rewrite snoc_equation. assumption.

            rewrite update_neq; try reflexivity.
            rewrite update_eq.
            split.
              simpl.
              unfold bassn in H10. unfold beval in H10.
              unfold aeval in H10. rewrite H1 in H10.
              rewrite ← Heqx in H10. simpl in H10.
              rewrite (beq_nat_true _ _ H10).
```

```
            intros. apply appears_in_snoc1.
            intros. reflexivity.
  Case "If not taken".
    eapply hoare_consequence_pre. apply hoare_skip.
    unfold assn_sub.
    intros st [[[H1 [p [H2 H3]]] H9] H10].
    split.
      rewrite update_neq; try reflexivity.
      assumption.
      remember (aslist (st X)) as x.
      destruct x as [|h x'].
        unfold bassn in H9. unfold beval in H9. unfold aeval in H9.
        rewrite ← Heqx in H9. inversion H9.
        ∃ (snoc p h).
        split.
          rewrite update_eq.
          unfold aeval. rewrite ← Heqx.
          rewrite snoc_equation. assumption.
          rewrite update_neq; try reflexivity.
          split.
            intros. apply appears_in_snoc2. apply H3. assumption.
            intros. destruct (appears_in_snoc3 _ _ _ H).
            SCase "later".
              inversion H3 as [_ H3'].
              apply H3'. assumption.
            SCase "here (absurd)".
              subst.
              unfold bassn, beval, aeval in H10.
              rewrite not_true_iff_false in H10.
              apply beq_nat_false in H10.
              rewrite ← Heqx in H10. simpl in H10.
              apply ex_falso_quodlibet. apply H10. assumption.
intros st [H1 [H2 H3]].
rewrite H1. rewrite H2. rewrite H3.
split.
  reflexivity.
  ∃ []. split.
    reflexivity.
    split; intros H; inversion H.
simpl. intros st [[H1 [p [H2 H3]]] H5].
unfold bassn in H5. unfold beval in H5. unfold aeval in H5.
```

```
  destruct (aslist (st X)) as [|h x'].
    rewrite append_nil in H2.
    rewrite ← H2.
    assumption.
    apply ex_falso_quodlibet. apply H5. reflexivity.
Qed.
```

Exercise: 3 stars (list_sum) Here is a direct definition of the sum of the elements of a list, and an Imp program that computes the sum.

```
Definition sum l := fold_right plus 0 l.

Definition sum_program :=
  Y ::= ANum 0;
  WHILE (BIsCons (AId X)) DO
    Y ::= APlus (AId Y) (AHead (AId X)) ;
    X ::= ATail (AId X)
  END.
```

Provide an *informal* proof of the following specification of *sum_program* in the form of a decorated version of the program.

```
Definition sum_program_spec := ∀ l,
  {{ fun st ⇒ aslist (st X) = l }}
  sum_program
  {{ fun st ⇒ asnat (st Y) = sum l }}.
```

☐

Exercise: 4 stars (list_reverse) Recall the function *rev* from Poly.v, for reversing lists.

```
Fixpoint rev {X:Type} (l:list X) : list X :=
  match l with
  | nil ⇒ []
  | cons h t ⇒ snoc (rev t) h
  end.
```

Write an Imp program *list_reverse_program* that reverses lists. Formally prove that it satisfies the following specification: forall l : list nat, [38] list_reverse_program [39]. You may find the lemmas *append_nil* and *rev_equation* useful.

```
Lemma rev_equation : ∀ (A : Type) (h : A) (x y : list A),
  rev (h :: x) ++ y = rev x ++ h :: y.
Proof.
  intros. simpl. apply snoc_equation.
```

[38] X=l/\Y=nil
[39] Y=revl

Qed.

□

Finally, for a bigger example, let's redo the proof of *list_member_correct* from above using our new tools.

Notice that the *verify* tactic leaves subgoals for each "interesting" use of *hoare_consequence* – that is, for each ⇒ that occurs in the decorated program, except for the ones that can be eliminated by repeated application of a few simple automated tactics. Each of these implications relies on a fact about lists, for example that l ++ [] = l. In other words, the Hoare logic infrastructure has taken care of the boilerplate reasoning about the execution of imperative programs, while the user has to prove lemmas that are specific to the problem domain (e.g. lists or numbers).

20.4 Formal Decorated Programs

Again, the definitions are copied verbatim from Hoare.v

```
Inductive dcom : Type :=
  | DCSkip : Assertion → dcom
  | DCSeq : dcom → dcom → dcom
  | DCAsgn : id → aexp → Assertion → dcom
  | DCIf : bexp → Assertion → dcom → Assertion → dcom
             → Assertion→ dcom
  | DCWhile : bexp → Assertion → dcom → Assertion → dcom
  | DCPre : Assertion → dcom → dcom
  | DCPost : dcom → Assertion → dcom.

Tactic Notation "dcom_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "Skip" | Case_aux c "Seq" | Case_aux c "Asgn"
  | Case_aux c "If" | Case_aux c "While"
  | Case_aux c "Pre" | Case_aux c "Post" ].

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 91, right associativity) : dcom_scope.
Notation " d ; d' "
      := (DCSeq d d')
      (at level 80, right associativity) : dcom_scope.

Delimit Scope dcom_scope with dcom.

Example dec_while : dcom := (
  {{ fun st ⇒ True }}
  WHILE (BNot (BEq (AId X) (ANum 0)))
  DO
    {{ fun st ⇒ True ∧ bassn (BNot (BEq (AId X) (ANum 0))) st}}
    X ::= (AMinus (AId X) (ANum 1))
    {{ fun _ ⇒ True }}
  END
  {{ fun st ⇒ True ∧ ¬bassn (BNot (BEq (AId X) (ANum 0))) st}} ⇒
  {{ fun st ⇒ asnat (st X) = 0 }}
) % dcom.

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.

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.

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.

Definition dec_correct (d:dcom) :=
  {{pre d}} (extract d) {{post d}}.

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 ~~> assn_sub X a Q)
  | 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.
```

```
Theorem verification_correct : ∀ d P,
  verification_conditions P d → {{P}} (extract d) {{post d}}.
Proof.
  dcom_cases (induction d) Case; intros P H; simpl in *.
  Case "Skip".
    eapply hoare_consequence_pre.
      apply hoare_skip.
      assumption.
  Case "Seq".
    inversion H as [H1 H2]. clear H.
    eapply hoare_seq.
      apply IHd2. apply H2.
      apply IHd1. apply H1.
  Case "Asgn".
    eapply hoare_consequence_pre.
      apply hoare_asgn.
      assumption.
  Case "If".
    inversion H as [HPre1 [HPre2 [Hd1 [Hd2 [HThen HElse]]]]]; clear H.
    subst.
    apply hoare_if.
      eapply hoare_consequence_pre. apply IHd1. eassumption. assumption.
      rewrite Hd2.
      eapply hoare_consequence_pre. apply IHd2. eassumption. assumption.
  Case "While".
    inversion H as [Hpre [Hbody [Hpost Hd]]]; subst; clear H.
    eapply hoare_consequence_pre.
    apply hoare_while with (P := post d).
      apply IHd. apply Hd.
      assumption.
  Case "Pre".
    inversion H as [HP Hd]; clear H.
    eapply hoare_consequence_pre. apply IHd. apply Hd. assumption.
  Case "Post".
    inversion H as [Hd HQ]; clear H.
    eapply hoare_consequence_post. apply IHd. apply Hd. assumption.
Qed.

Tactic Notation "verify" :=
  try apply verification_correct;
  repeat split;
  simpl; unfold assert_implies;
  unfold bassn in *; unfold beval in *; unfold aeval in *;
```

```
unfold assn_sub; intros;
repeat rewrite update_eq;
repeat (rewrite update_neq; [| reflexivity]);
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 *;
try eauto; try omega.
```

Finally, for a bigger example, let's redo the proof of *list_member_correct* from above using our new tools.

Notice that the *verify* tactic leaves subgoals for each "interesting" use of *hoare_consequence* – that is, for each ⇒ that occurs in the decorated program, except for the ones that can be eliminated by repeated application of a few simple automated tactics. Each of these implications relies on a fact about lists, for example that l ++ [] = l. In other words, the Hoare logic infrastructure has taken care of the boilerplate reasoning about the execution of imperative programs, while the user has to prove lemmas that are specific to the problem domain (e.g. lists or numbers).

```
Definition list_member_dec (n : nat) (l : list nat) : dcom := (
    {{ fun st ⇒ aslist (st X) = l ∧ asnat (st Y) = n ∧ asnat (st Z) = 0 }}
  WHILE BIsCons (AId X)
  DO {{ fun st ⇒ (asnat (st Y) = n
                ∧ (∃ p, p ++ aslist (st X) = l
                ∧ (asnat (st Z) = 1 ↔ appears_in n p)))
                ∧ bassn (BIsCons (AId X)) st }}
    IFB (BEq (AId Y) (AHead (AId X))) THEN
      {{ fun st ⇒
        ((asnat (st Y) = n
          ∧ (∃ p, p ++ aslist (st X) = l
              ∧ (asnat (st Z) = 1 ↔ appears_in n p)))
        ∧ bassn (BIsCons (AId X)) st)
        ∧ bassn (BEq (AId Y) (AHead (AId X))) st }}
      ⇒
      {{ fun st ⇒
          asnat (st Y) = n
          ∧ (∃ p, p ++ tail (aslist (st X)) = l
                ∧ (1 = 1 ↔ appears_in n p)) }}
     Z ::= ANum 1
      {{ fun st ⇒ asnat (st Y) = n
```

```
              ∧ (∃ p, p ++ tail (aslist (st X)) = l
                   ∧ (asnat (st Z) = 1 ↔ appears_in n p)) }}
  ELSE
      {{ fun st ⇒
         ((asnat (st Y) = n
           ∧ (∃ p, p ++ aslist (st X) = l
                   ∧ (asnat (st Z) = 1 ↔ appears_in n p)))
         ∧ bassn (BIsCons (AId X)) st)
         ∧ ¬bassn (BEq (AId Y) (AHead (AId X))) st }}
      ⇒
      {{ fun st ⇒
         asnat (st Y) = n
         ∧ (∃ p, p ++ tail (aslist (st X)) = l
              ∧ (asnat (st Z) = 1 ↔ appears_in n p)) }}
    SKIP
      {{ fun st ⇒ asnat (st Y) = n
           ∧ (∃ p, p ++ tail (aslist (st X)) = l
                 ∧ (asnat (st Z) = 1 ↔ appears_in n p)) }}
  FI
    {{ fun st ⇒ asnat (st Y) = n
       ∧ (∃ p, p ++ tail (aslist (st X)) = l
         ∧ (asnat (st Z) = 1 ↔ appears_in n p)) }}
  ;
  X ::= (ATail (AId X))
    {{ fun st ⇒
        asnat (st Y) = n ∧
        (∃ p : list nat, p ++ aslist (st X) = l
          ∧ (asnat (st Z) = 1 ↔ appears_in n p)) }}
 END
  {{ fun st ⇒
    (asnat (st Y) = n
      ∧ (∃ p, p ++ aslist (st X) = l
           ∧ (asnat (st Z) = 1 ↔ appears_in n p)))
    ∧ ¬bassn (BIsCons (AId X)) st }}
  ⇒
  {{ fun st ⇒ asnat (st Z) = 1 ↔ appears_in n l }}
) %dcom.

Theorem list_member_correct' : ∀ n l,
  dec_correct (list_member_dec n l).
Proof.
  intros n l. verify.
  Case "The loop precondition holds.".
```

```
    ∃ []. simpl. split.
      rewrite H. reflexivity.
      rewrite H1. split; intro Hc; inversion Hc.
  Case "IF taken".
    destruct H2 as [p [H3 H4]].
    remember (aslist (st X)) as x.
    destruct x as [|h x'].
      inversion H1.
      ∃ (snoc p h).
      simpl. split.
         rewrite snoc_equation. assumption.
         split.
           rewrite H in H0.
           simpl in H0. rewrite H0.
           intros _. apply appears_in_snoc1.
           intros _. reflexivity.
  Case "If not taken".
    destruct H2 as [p [H3 H4]].
    remember (aslist (st X)) as x.
    destruct x as [|h x'].
      inversion H1.
      ∃ (snoc p h).
      split. simpl.
        rewrite snoc_equation. assumption.
        split.
          intros. apply appears_in_snoc2. apply H4. assumption.
          intros Hai. destruct (appears_in_snoc3 _ _ _ Hai).
          SCase "later". apply H4. assumption.
          SCase "here (absurd)".
            subst. simpl in H0.
            apply ex_falso_quodlibet. apply H0. assumption.
  Case "Loop postcondition implies desired conclusion (->)".
    destruct H2 as [p [H3 H4]].
    destruct (aslist (st X)) as [|h x'].
       rewrite append_nil in H3. subst. apply H4. assumption.
       inversion H1.
  Case "loop postcondition implies desired conclusion (<-)".
    destruct H2 as [p [H3 H4]].
    destruct (aslist (st X)) as [|h x'].
       rewrite append_nil in H3. subst. apply H4. assumption.
       inversion H1.
Qed.
```

Chapter 21

Library Hoare

21.1 Hoare: Hoare Logic

```
Require Export Imp.
```

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 properties 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 optional chapter *Equiv*).

  If we stopped here, we would still 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 – or, in some cases, actually even wrong!

We'll return to this theme later in the course when we discuss *types* and *type soundness*.

- We saw a couple of examples of *program verification* – using the precise definition of Imp to prove formally that certain particular programs (e.g., factorial and slow subtraction) satisfied particular specifications of their behavior.

In this chapter, we'll take this last idea further. We'll develop a reasoning system called *Floyd-Hoare Logic* – commonly shortened to just *Hoare Logic* – in which each of the syntactic constructs of Imp is equipped with a single, generic "proof rule" that can be used to reason about programs involving this construct.

Hoare Logic originates 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 huge variety of tools that are now being used to specify and verify real software systems.

21.2 Hoare Logic

Hoare Logic combines two beautiful ideas: a natural way of writing down *specifications* of programs, and a *compositional proof technique* for proving that these specifications are met – where by "compositional" we mean that the structure of proofs directly mirrors the structure of the programs that they are about.

21.2.1 Assertions

If we're going to talk about specifications of programs, the first thing we'll want is a way of making *assertions* about properties that hold at particular points during a program's execution – i.e., properties that may or may not be true of a given state of the memory.

`Definition Assertion := state → Prop.`

Exercise: 1 star (assertions) Paraphrase the following assertions in English. 1) fun st => st X = 3 2) fun st => st X = x 3) fun st => st X <= st Y 4) fun st => st X = 3 \/ st X <= st Y 5) fun st => st Z * st Z <= x /\ ~ (((S (st Z)) * (S (st Z))) <= x) 6) fun st => True 7) fun st => False

☐

This way of writing assertions is formally correct – it precisely captures what we mean, and it is exactly what we will use in Coq proofs. We'll also want a lighter, less formal notation for discussing examples, since this one is a bit heavy: (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 (we will never need to talk about two different memory states at the same time). So, when writing down assertions informally, we'll make some simplifications: drop the initial `fun` *st* ⇒, and write just X instead of *st* X. Informally, instead of writing fun st => (st Z) * (st Z) <= x /\ ~ ((S (st Z)) * (S (st Z)) <= x) we'll write just Z * Z <= x /\ ~((S Z) * (S Z) <= x).

21.2.2 Hoare Triples

Next, we need a way of specifying – making claims about – the behavior of commands.

Since we've defined assertions as a way of making claims about the properties of states, and since the behavior of a command is to transform one state to another, it is natural to express claims about commands in the following way:

- "If command c is started in a state satisfying assertion P, and if c eventually terminates, then the final state is guaranteed to 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* of c.

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  ∀ st st',
       c / st || st' →
       P st →
       Q st'.
```

Since we'll be working a lot with Hoare triples, it's useful to have a compact notation: [1] c [2]. (Traditionally, Hoare triples are written $\{P\}\ c\ \{Q\}$, 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.
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. The first notation – with missing postcondition – will not actually be used for a while; it's just a placeholder for a notation that we'll want to define later, when we discuss decorated programs.)

Exercise: 1 star (triples) Paraphrase the following Hoare triples in English. 1) [3] c [4]

[1] P
[2] Q
[3] True
[4] X=5

2) [5] c [6]
3) [7] c [8]
4) [9] c [10]
5) [11] c [12].
6) [13] c [14] □

Exercise: 1 star (valid_triples) Which of the following Hoare triples are *valid* – i.e., the claimed relation between P, c, and Q is true? 1) [15] X ::= 5 [16]
2) [17] X ::= X + 1 [18]
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. We'll continue doing so throughout the chapter.)

To get us warmed up, here are two simple facts about Hoare triples.

---

[5] `X=x`
[6] `X=x+5)`
[7] `X<=Y`
[8] `Y<=X`
[9] `True`
[10] `False`
[11] `X=x`
[12] `Y=real_factx`
[13] `True`
[14] `(Z*Z)<=x/\~(((SZ)*(SZ))<=x)`
[15] `True`
[16] `X=5`
[17] `X=2`
[18] `X=3`
[19] `True`
[20] `X=5`
[21] `X=2/\X=3`
[22] `X=0`
[23] `True`
[24] `False`
[25] `False`
[26] `True`
[27] `True`
[28] `False`
[29] `X=0`
[30] `X=1`
[31] `X=1`
[32] `X=100`

```
Theorem hoare_post_true : ∀ (P Q : Assertion) c,
  (∀ st, Q st) →
  {{P}} c {{Q}}.
Proof.
  intros P Q c H. unfold hoare_triple.
  intros st st' Heval HP.
  apply H. Qed.
```

```
Theorem hoare_pre_false : ∀ (P Q : Assertion) c,
  (∀ st, ~(P st)) →
  {{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.
```

21.2.3 Weakest Preconditions

Some Hoare triples are more interesting than others. For example, [33] X ::= Y + 1 [34] is *not* very interesting: it is perfectly valid, but 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, [35] X ::= Y + 1 [36] 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 a given command and postcondition) is this one: [37] X ::= Y + 1 [38] 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
- whenever P' is an assertion such that $\{\{P'\}\}\ c\ \{\{Q\}\}$, we have P' *st* implies P *st* for all states *st*.

[33] `False`
[34] `X<=5`
[35] `Y<=4/\Z=0`
[36] `X<=5`
[37] `Y<=4`
[38] `X<=5`

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 Q after executing c.

The second of the conditions above is essentially a form of logical implication at the level of assertions. Because of the frequency of its occurrence, it is useful to define a little notation:

```
Definition assert_implies (P Q : Assertion) : Prop :=
  ∀ st, P st → Q st.
```

We will write P ~~> Q (in ASCII, P ¬~> Q) for *assert_implies* P Q.

```
Notation "P ~~> Q" := (assert_implies P Q) (at level 80).
Notation "P <~~> Q" := (P ~~> Q ∧ Q ~~> P) (at level 80).
```

Exercise: 1 star (wp) What are the weakest preconditions of the following commands for the following postconditions? 1) [39] SKIP [40]

2) [41] X ::= Y + Z [42]

3) [43] X ::= Y [44]

4) [45] IFB X == 0 THEN Y ::= Z + 1 ELSE Y ::= W + 2 FI [46]

5) [47] X ::= 5 [48]

6) [49] WHILE True DO X ::= 0 END [50] ☐

Exercise: 3 stars, optional (is_wp_formal) Weakest preconditions can be defined formally as follows:

```
Definition is_wp P c Q :=
  {{P}} c {{Q}} ∧
  ∀ P', {{P'}} c {{Q}} → (∀ st, P' st → P st).
```

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 ⇒ st Y ≤ 4)
    (X ::= APlus (AId Y) (ANum 1)) (fun st ⇒ st X ≤ 5).
Proof.
```

---

[39] ?
[40] X=5
[41] ?
[42] X=5
[43] ?
[44] X=Y
[45] ?
[46] Y=5
[47] ?
[48] X=0
[49] ?
[50] X=0

Admitted.

□

21.2.4 Proof Rules

The goal of Hoare logic is to provide a *compositional* method for proving the validity of Hoare triples. That is, the structure of a program's correctness proof should mirror the structure of the program itself. To this end, in the sections below, we'll introduce one 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 that are useful for gluing things together. We will prove programs correct using these proof rules, without ever unfolding the definition of *hoare_triple*.

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: [51] X ::= Y [52] 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 [53] X ::= Y + Z [54] 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 [55] X ::= a [56] is a valid Hoare triple.

Even more generally, a is *any* arithmetic expression and Q is *any* property of numbers, then [57] X ::= a [58] is a valid Hoare triple.

Rephrasing this a bit gives us the general Hoare rule for assignment: [59] X ::= a [60] For example, these are valid applications of the assignment rule: [61] X ::= X + 1 [62]

[63] X ::= 3 [64]

[65] X ::= 3 [66]

[51] `Y=1`
[52] `X=1`
[53] `Y+Z=1`
[54] `X=1`
[55] `a=1`
[56] `X=1`
[57] `Q(a)`
[58] `Q(X)`
[59] `QwhereaissubstitutedforX`
[60] `Q`
[61] `(X<=5)whereX+1issubstitutedforXi.e.,X+1<=5`
[62] `X<=5`
[63] `(X=3)where3issubstitutedforXi.e.,3=3`
[64] `X=3`
[65] `(0<=X/\X<=5)where3issubstitutedforXi.e.,(0<=3/\3<=5)`
[66] `0<=X/\X<=5`

To formalize the rule, we begin with the notion of "substitution in an assertion":

```
Definition assn_sub X a Q : Assertion :=
  fun (st : state) ⇒
    Q (update st X (aeval st a)).
```

We ask that Q holds for the state obtained by assigning a to X, i.e. the updated state in which X is bound to the result of evaluating a. Since we've chosen to represent assertions using Coq propositions, this is the only way we can "substitute" a variable inside an assertion.

Now the precise proof rule for assignment:

(hoare_asgn) [67] X::=a [68]

```
Theorem hoare_asgn : ∀ Q X a,
  {{assn_sub X a Q}} (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 :
  {{assn_sub X (ANum 3) (fun st ⇒ st X = 3)}}
  (X ::= (ANum 3))
  {{fun st ⇒ st X = 3}}.
Proof.
  apply hoare_asgn. Qed.
```

Exercise: 2 stars (hoare_asgn_examples) Translate these informal Hoare triples... [69] X ::= X + 1 [70] [71] X ::= 3 [72] ...into formal statements 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 backward to you, it may help to think a little about alternative "forward" rules. Here is a seemingly natural one:

(hoare_asgn_wrong) [73] X ::= a [74] Give a counterexample showing that this rule is incorrect

[67] assn_subXaQ
[68] Q
[69] assn_subX(X+1)(X<=5)
[70] X<=5
[71] assn_subX3(0<=X/\X<=5)
[72] 0<=X/\X<=5
[73] True
[74] X=a

(informally). 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, optional (hoare_asgn_fwd) However, using an auxiliary variable x 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) [75] X ::= a [76] (where st' = update st X x) 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 :
  (∀ {X Y: Type} {f g : X → Y}, (∀ (x: X), f x = g x) → f = g) →
  ∀ x a Q,
  {{fun st ⇒ Q st ∧ st X = x}}
    X ::= a
  {{fun st ⇒ Q (update st X x) ∧ st X = aeval (update st X x) a }}.
Proof.
  intros functional_extensionality v a Q.
   Admitted.
```

☐

Exercise: 2 stars (hoare_asgn_weakest) Show that the precondition in the rule *hoare_asgn* is in fact the weakest precondition.

```
Theorem hoare_asgn_weakest : ∀ P X a Q,
  {{P}} (X ::= a) {{Q}} →
  P ~~> assn_sub X a Q.
Proof.
   Admitted.
```

☐

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.

---

[75] funst=>Qst/\stX=x
[76] funst=>Qst'/\stX=aevalst'a

For instance, while [77] X ::= 3 [78], follows directly from the assignment rule, [79] X ::= 3 [80]. does not. This triple is also valid, but it is not an instance of *hoare_asgn* because *True* and *assn_sub X* 3 (X = 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 could capture this observation with the following rule: [81] c [82] P <~~> P'

(hoare_consequence_pre_equiv) [83] c [84] Generalizing this line of thought a bit further, if we can derive {{P}} c {{Q}}, it is valid to change P to P' as long as P' is strong enough to imply P, and change Q to Q' as long as Q implies Q'. This observation is captured by two *Rules of Consequence*. [85] c [86] P ~~> P'

(hoare_consequence_pre) [87] c [88]

[89] c [90] Q' ~~> Q

(hoare_consequence_post) [91] c [92]

Here are the formal versions:

```
Theorem hoare_consequence_pre : ∀ (P P' Q : Assertion) c,
  {{P'}} c {{Q}} →
  P ~~> P' →
  {{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 : ∀ (P Q Q' : Assertion) c,
  {{P}} c {{Q'}} →
  Q' ~~> Q →
  {{P}} c {{Q}}.
```

---

[77] assn_subX3(X=3)
[78] X=3
[79] True
[80] X=3
[81] P'
[82] Q
[83] P
[84] Q
[85] P'
[86] Q
[87] P
[88] Q
[89] P
[90] Q'
[91] P
[92] 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 might use the first consequence rule like this: [93] => [94] X ::= 1 [95] Or, formally...

```
Example hoare_asgn_example1 :
  {{fun st ⇒ True}} (X ::= (ANum 1)) {{fun st ⇒ st X = 1}}.
Proof.
  apply hoare_consequence_pre
    with (P' := assn_sub X (ANum 1) (fun st ⇒ st X = 1)).
  apply hoare_asgn.
  intros st H. 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. [96] c [97] P ~~> P' Q' ~~> Q

(hoare_consequence) [98] c [99]

```
Theorem hoare_consequence : ∀ (P P' Q Q' : Assertion) c,
  {{P'}} c {{Q'}} →
  P ~~> P' →
  Q' ~~> Q →
  {{P}} c {{Q}}.
Proof.
  intros P P' Q Q' c Hht HPP' HQ'Q.
  intros st st' Hc HP.
  apply HQ'Q. apply (Hht st st'). assumption.
  apply HPP'. assumption. Qed.
```

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* above, to make sure that

---

[93] True
[94] 1=1
[95] X=1
[96] P'
[97] Q'
[98] P
[99] Q

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 a little annoying, both because the assertion is a bit long and also because 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 soon."

```
Example hoare_asgn_example1' :
  {{fun st ⇒ True}}
  (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*) 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 (rightfully) 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. Admitted.
```

Coq gives a warning after `apply` *HP*: No more subgoals but non-instantiated existential variables: Existential 1 = ?171 : P : *nat* → *nat* → `Prop` Q : *nat* → `Prop` *HP* : ∀ x y : *nat*, P x y *HQ* : ∀ x y : *nat*, P x y → Q x ⊢ *nat* Trying to finish the proof with `Qed` instead of *Admitted* gives an error:

```
Error: Attempt to save a proof with existential variables still
non-instantiated
```

An additional constraint is that existential variables cannot be instantiated with terms containing (normal) variables that did not exist at the time the existential variable was created.

```
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']. Admitted.
```

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*.

```
Lemma silly2_fixed : ∀ (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. destruct HP as [y HP']. eapply HQ. apply HP'.
Qed.
```

In the last step we did `apply` *HP'* which unifies the existential variable in the goal with the variable *y*. 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'*.

```
Lemma silly2_eassumption : ∀ (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. destruct HP as [y HP']. eapply HQ. eassumption.
Qed.
```

Exercise: 2 stars (hoare_asgn_examples_2) Translate these informal Hoare triples... [100] X ::= X + 1 [101] [102] X ::= 3 [103] ...into formal statements and use *hoare_asgn* and *hoare_consequence_pre* to prove them.

□

Skip

Since *SKIP* doesn't change the state, it preserves any property P:

---

[100] `X+1<=5`
[101] `X<=5`
[102] `0<=3/\3<=5`
[103] `0<=X/\X<=5`

(hoare_skip) [104] SKIP [105]

```
Theorem hoare_skip : ∀ P,
     {{P}} SKIP {{P}}.
Proof.
  intros P st st' H HP. inversion H. subst.
  assumption. Qed.
```

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: [106] c1 [107] [108] c2 [109]

(hoare_seq) [110] c1;c2 [111]

```
Theorem hoare_seq : ∀ P Q R c1 c2,
     {{Q}} c2 {{R}} →
     {{P}} c1 {{Q}} →
     {{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: 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 recording a proof using the sequencing rule is as a "decorated program" where the intermediate assertion Q is written between *c1* and *c2*: [112] X ::= a; [113] <—- decoration for Q SKIP [114]

```
Example hoare_asgn_example3 : ∀ a n,
```

---

[104] P
[105] P
[106] P
[107] Q
[108] Q
[109] R
[110] P
[111] R
[112] a=n
[113] X=n
[114] X=n

```
  {{fun st ⇒ aeval st a = n}}
  (X ::= a; SKIP)
  {{fun st ⇒ st X = n}}.
Proof.
  intros a n. eapply hoare_seq.
  Case "right part of seq".
    apply hoare_skip.
  Case "left part of seq".
    eapply hoare_consequence_pre. apply hoare_asgn.
    intros st H. subst. reflexivity. Qed.
```

You will most often use *hoare_seq* and *hoare_consequence_pre* in conjunction with the `eapply` tactic, as done above.

Exercise: 2 stars (hoare_asgn_example4) Translate this decorated program into a formal proof: [115] => [116] X ::= 1; [117] => [118] Y ::= 2 [119]

```
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 (in Coq) that it satisfies the following specification: [120] c [121]

☐

Exercise: 3 stars, optional (hoarestate1) Explain why the following proposition can't be proven: forall (a : aexp) (n : nat), [122] (X ::= (ANum 3); Y ::= a) [123].

☐

Conditionals

What sort of rule do we want for reasoning about conditional commands? Certainly, if the same assertion *Q* holds after executing either branch, then it holds after the whole

[115] `True`
[116] `1=1`
[117] `X=1`
[118] `X=1/\2=2`
[119] `X=1/\Y=2`
[120] `X<=Y`
[121] `Y<=X`
[122] `funst=>aevalsta=n`
[123] `funst=>stY=n`

conditional. So we might be tempted to write: [124] c1 [125] [126] c2 [127]

---

[128] IFB b THEN c1 ELSE c2 [129] However, this is rather weak. For example, using this rule, we cannot show that: [130] IFB X == 0 THEN Y ::= 2 ELSE Y ::= X + 1 FI [131] since the rule tells us nothing about the state in which the assignments take place in the "then" and "else" branches.

But, actually, 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*). [132] c1 [133] [134] c2 [135]

(hoare_if) [136] IFB b THEN c1 ELSE c2 FI [137]

To interpret this rule formally, we need to do a little work.

Strictly speaking, the assertion we've written, $P \land b$, is the conjunction of an assertion and a boolean expression, which doesn't typecheck. To fix this, we need a way of formally "lifting" any bexp *b* to an assertion. We'll write *bassn b* for the assertion "the boolean expression *b* evaluates to *true* (in the given state)."

```
Definition bassn b : Assertion :=
  fun st ⇒ (beval st b = true).
```

A couple of useful facts about *bassn*:

```
Lemma bexp_eval_true : ∀ b st,
  beval st b = true → (bassn b) st.
Proof.
  intros b st Hbe.
  unfold bassn. assumption. Qed.
```

```
Lemma bexp_eval_false : ∀ b st,
  beval st b = false → ¬ ((bassn b) st).
Proof.
```

---

[124] P
[125] Q
[126] P
[127] Q
[128] P
[129] Q
[130] True
[131] X<=Y
[132] P/\b
[133] Q
[134] P/\~b
[135] Q
[136] P
[137] Q

```
  intros b st Hbe contra.
  unfold bassn in contra.
  rewrite → contra in Hbe. inversion Hbe. Qed.
```

Now we can formalize the Hoare proof rule for conditionals and prove it correct.

```
Theorem hoare_if : ∀ P Q b c1 c2,
  {{fun st ⇒ P st ∧ bassn b st}} c1 {{Q}} →
  {{fun st ⇒ P st ∧ ~(bassn b st)}} c2 {{Q}} →
  {{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.
  Case "b is true".
    apply (HTrue st st').
      assumption.
      split. assumption.
             apply bexp_eval_true. assumption.
  Case "b is false".
    apply (HFalse st st').
      assumption.
      split. assumption.
             apply bexp_eval_false. assumption. Qed.
```

Here is a formal proof that the program we used to motivate the rule satisfies the specification we gave.

```
Example if_example :
    {{fun st ⇒ True}}
  IFB (BEq (AId X) (ANum 0))
    THEN (Y ::= (ANum 2))
    ELSE (Y ::= APlus (AId X) (ANum 1))
  FI
    {{fun st ⇒ st X ≤ st Y}}.
Proof.
  apply hoare_if.
  Case "Then".
    eapply hoare_consequence_pre. apply hoare_asgn.
    unfold bassn, assn_sub, update, assert_implies. simpl. intros st [_ H].
    symmetry in H; apply beq_nat_eq in H.
    rewrite H. omega.
  Case "Else".
    eapply hoare_consequence_pre. apply hoare_asgn.
    unfold assn_sub, update, assert_implies; simpl; intros st _. omega.
Qed.
```

Exercise: One-sided conditionals

Exercise: 4 stars, recommended (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.

We first extend the syntax of commands, and introduce the usual notations. We use a separate module to prevent polluting the global name space.

```
Module If1.

Inductive com : Type :=
  | CSkip : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CIf1 : bexp → com → com.

Tactic Notation "com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "SKIP" | Case_aux c "::=" | Case_aux c ";"
  | Case_aux c "IFB" | Case_aux c "WHILE" | Case_aux c "CIF1" ].

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'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'IF1' b 'THEN' c 'FI'" :=
  (CIf1 b c) (at level 80, right associativity).
```

We now extend the evaluation relation to accommodate *IF1* branches. 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 || 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').
Tactic Notation "ceval_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E_Skip" | Case_aux c "E_Ass" | Case_aux c "E_Seq"
  | Case_aux c "E_IfTrue" | Case_aux c "E_IfFalse"
  | Case_aux c "E_WhileEnd" | Case_aux c "E_WhileLoop"

  ].
```

We repeat 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.
```

Now 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 that your rule is precise enough to show the following valid

Hoare triple: [138] IF1 Y <> 0 THEN X ::= X + Y FI [139]

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 (AId Y) (ANum 0)) THEN
    X ::= APlus (AId X) (AId Y)
  FI
  {{ fun st ⇒ st X = st Z }}.
Proof. Admitted.
```

```
End If1.
```

☐

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 [140] WHILE b DO c END [141] is a valid triple.

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 [142] WHILE b DO c END [143]. 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: [144] WHILE b DO c END [145] 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: [146] c [147]

---

[148] WHILE b DO c END [149] The proposition P is called an *invariant*.

---

[138] X+Y=Z
[139] X=Z
[140] P
[141] Q
[142] P
[143] P
[144] P
[145] P/\~b
[146] P
[147] P
[148] P
[149] P/\~b

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: [150] c [151]

(hoare_while) [152] WHILE b DO c END [153]

```
Lemma hoare_while : ∀ P b c,
  {{fun st ⇒ P st ∧ bassn b st}} c {{P}} →
  {{P}} WHILE b DO c END {{fun st ⇒ P st ∧ ¬ (bassn b st)}}.
Proof.
  intros P b c Hhoare st st' He HP.
  remember (WHILE b DO c END) as wcom.
  ceval_cases (induction He) Case; try (inversion Heqwcom); subst.
  Case "E_WhileEnd".
    split. assumption. apply bexp_eval_false. assumption.
  Case "E_WhileLoop".
    apply IHHe2. reflexivity.
    apply (Hhoare st st'). assumption.
      split. assumption. apply bexp_eval_true. assumption. Qed.
Example while_example :
    {{fun st ⇒ st X ≤ 3}}
  WHILE (BLe (AId X) (ANum 2))
  DO X ::= APlus (AId 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, update. simpl.
    intros st [H1 H2]. apply ble_nat_true in H2. omega.
  unfold bassn, assert_implies. intros st [Hle Hb].
    simpl in Hb. remember (ble_nat (st X) 2) as le. destruct le.
    apply ex_falso_quodlibet. apply Hb; reflexivity.
    symmetry in Heqle. apply ble_nat_false in Heqle. omega.
Qed.
```

---

[150] P/\b
[151] P
[152] P
[153] P/\~b

We can also use the while rule to prove the following Hoare triple, which may seem surprising at first...

```
Theorem always_loop_hoare : ∀ P Q,
  {{P}} 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.
  Case "Loop body preserves invariant".
    apply hoare_post_true. intros st. apply I.
  Case "Loop invariant and negated guard imply postcondition".
    simpl. intros st [Hinv Hguard].
    apply ex_falso_quodlibet. apply Hguard. reflexivity.
  Case "Precondition implies invariant".
    intros st H. constructor. Qed.
```

Actually, this result shouldn't be surprising. If we look back at the definition of *hoare_triple*, we can see that it asserts something meaningful *only* when the command terminates.

```
Print hoare_triple.
```

If the command doesn't terminate, we can prove anything we like about the post-condition. Here's a more direct proof of the same fact:

```
Theorem always_loop_hoare' : ∀ P Q,
  {{P}} WHILE BTrue DO SKIP END {{Q}}.
Proof.
  unfold hoare_triple. intros P Q st st' contra.
  apply loop_never_stops in contra. inversion contra.
Qed.
```

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 focus only on partial correctness.

Exercise: *REPEAT*

```
Module RepeatExercise.
```

Exercise: 4 stars (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
| CIf : 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.

```
Tactic Notation "com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "SKIP" | Case_aux c "::=" | Case_aux c ";"
  | Case_aux c "IFB" | Case_aux c "WHILE" | Case_aux c "CRepeat" ].
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) (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''

.
Tactic Notation "ceval_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E_Skip" | Case_aux c "E_Ass" | Case_aux c "E_Seq"
  | Case_aux c "E_IfTrue" | Case_aux c "E_IfFalse"
  | Case_aux c "E_WhileEnd" | Case_aux c "E_WhileLoop"

].
```

A couple of definitions from above, copied here so they use the new *ceval*.

```
Notation "c1 '/' st '||' 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 || 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 (AId Y) (ANum 1)
  UNTIL (BEq (AId X) (ANum 1)) END.

Theorem ex1_repeat_works :
  ex1_repeat / empty_state || update (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: [154] REPEAT Y ::= X; X ::= X - 1 UNTIL X = 0 END [155]

End REPEATEXERCISE.

☐

21.2.5 Exercise: *HAVOC*

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
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CHavoc : id → com.

Tactic Notation "com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "SKIP" | Case_aux c "::=" | Case_aux c ";"
  | Case_aux c "IFB" | Case_aux c "WHILE" | Case_aux c "HAVOC" ].

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'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'HAVOC' X" := (CHavoc X) (at level 60).
```

[154] X>0

[155] X=0/\Y>0

```
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 || 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 || update st X n

  where "c1 '/' st '||' st'" := (ceval c1 st st').

Tactic Notation "ceval_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E_Skip" | Case_aux c "E_Ass" | Case_aux c "E_Seq"
  | Case_aux c "E_IfTrue" | Case_aux c "E_IfFalse"
  | Case_aux c "E_WhileEnd" | Case_aux c "E_WhileLoop"
  | Case_aux c "E_Havoc" ].
```

The definition of Hoare triples is exactly as before. Unlike our notion of equivalence, which had subtle consequences with occassionally 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 :=
  ∀ 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.
```

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 : ∀ (Q : Assertion) (X : **id**),
  {{ havoc_pre X Q }} HAVOC X {{ Q }}.
Proof.
  Admitted.

Like in the *hoare_asgn_weakest* exercise above, show that your *havoc_pre* returns the weakest precondition.

Lemma hoare_havoc_weakest : ∀ (P Q : Assertion) (X : **id**),
  {{ P }} HAVOC X {{ Q }} →
  P ~~> havoc_pre X Q.
Proof.
  Admitted.

End HIMP.
☐

21.3 Review of Hoare Logic

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) [156] X::=a [157]

(hoare_skip) [158] SKIP [159]
[160] c1 [161] [162] c2 [163]

(hoare_seq) [164] c1;c2 [165]
[166] c1 [167] [168] c2 [169]

[156] assn_subXaQ
[157] Q
[158] P
[159] P
[160] P
[161] Q
[162] Q
[163] R
[164] P
[165] R
[166] P/\b
[167] Q
[168] P/\~b
[169] Q

(hoare_if) [170] IFB b THEN c1 ELSE c2 FI [171]
[172] c [173]

(hoare_while) [174] WHILE b DO c END [175]
[176] c [177] P ~~> P' Q' ~~> Q

(hoare_consequence) [178] c [179]

21.4 Decorated Programs

The beauty of Hoare Logic is that it is *compositional* – the structure of proofs exactly follows the structure of programs. This fact suggests that we could record the essential ideas of a proof informally (leaving out some low-level calculational details) by decorating programs with appropriate assertions around each statement. Such a *decorated program* carries with it an (informal) proof of its own correctness.

For example, here is a complete decorated program:

[180] => [181] X ::= x; [182] => [183] Z ::= z; [184] => [185] WHILE X <> 0 DO [186] => [187] Z ::= Z - 1; [188] X ::= X - 1 [189] END; [190] => [191] => [192] Z ::= Z + 1 [193]

Concretely, a decorated program consists of the program text interleaved with assertions. To check that a decorated program represents a valid proof, we check that each individual

[170] `P`
[171] `Q`
[172] `P/\b`
[173] `P`
[174] `P`
[175] `P/\~b`
[176] `P'`
[177] `Q'`
[178] `P`
[179] `Q`
[180] `True`
[181] `x=x`
[182] `X=x`
[183] `X=x/\z=z`
[184] `X=x/\Z=z`
[185] `Z-X=z-x`
[186] `Z-X=z-x/\X<>0`
[187] `(Z-1)-(X-1)=z-x`
[188] `Z-(X-1)=z-x`
[189] `Z-X=z-x`
[190] `Z-X=z-x/\~(X<>0)`
[191] `Z=z-x`
[192] `Z+1=z-x+1`
[193] `Z=z-x+1`

command is *locally* consistent with its accompanying assertions in the following sense:

- *SKIP* is locally consistent if its precondition and postcondition are the same: [194] SKIP [195]

- The sequential composition of commands *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): [196] c1; [197] c2 [198]

- An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition: [199] X ::= a [200]

- 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 $P \wedge b$ and $P/\backslash\tilde{\ } b$ and if its "then" branch is locally consistent (with respect to $P \wedge b$ and Q) and its "else" branch is locally consistent (with respect to $P/\backslash\tilde{\ } b$ and Q): [201] IFB b THEN [202] c1 [203] ELSE [204] c2 [205] FI [206]

- A while loop is locally consistent if its postcondition is $P/\backslash\tilde{\ } b$ (where P is its precondition) and if the pre- and postconditions of its body are exactly $P \wedge b$ and P: [207] WHILE b DO [208] c1 [209] END [210]

- A pair of assertions separated by ⇒ is locally consistent if the first implies the second (in all states): [211] => [212]

  This corresponds to the application of *hoare_consequence* and is the only place in a decorated program where checking if the decorations are correct is not fully mechanical and syntactic, but it involves logical and maybe arithmetic reasoning.

---

[194] `P`
[195] `P`
[196] `P`
[197] `Q`
[198] `R`
[199] `PwhereaissubstitutedforX`
[200] `P`
[201] `P`
[202] `P/\b`
[203] `Q`
[204] `P/\~b`
[205] `Q`
[206] `Q`
[207] `P`
[208] `P/\b`
[209] `P`
[210] `P/\~b`
[211] `P`
[212] `P,`

21.5 Sample Hoare Logic Proofs

21.5.1 Example: Slow Subtraction

We've seen an Imp program for subtracting one number from another by repeatedly subtracting one from each number until the one being subtracted reaches zero.

Here is a full proof – presented as a decorated program – that this program meets a natural specification: (1) [213] => (2) [214] WHILE X <> 0 DO (3) [215] => (4) [216] Z ::= Z - 1; (5) [217] X ::= X - 1 (6) [218] END (7) [219] => (8) [220] The decorations were constructed as follows:

- Begin with the undecorated program (the unnumbered lines).
- Add the specification – i.e., the outer precondition (1) and postcondition (8).
- Write down the invariant of the loop (6).
- Following the format dictated by the *hoare_while* rule, add the final use of the rule of consequence – the assertion in line (7) and the check that (7) implies (8).
- Check that the loop invariant *is* an invariant of the loop body by using the assignment rule twice to push the invariant backwards from the end of the loop body to the beginning (line (5) and then line (4)), and then filling in the rule of consequence asserting that the invariant plus the fact that the loop guard is true (line (3)) implies line (4).
- Check that the invariant is established at the beginning of the loop verifying that line (2) is implied by line (1).

As in most Hoare Logic proofs, the only challenging part of this process is finding the right loop invariant. There is no foolproof procedure for this, but a helpful heuristic is to begin by assumimng that the loop invariant is exactly the desired postcondition (i.e., that lines (6) and (8) are the same) and then calculating a bit to find out why this assertion is *not* an invariant of the loop body.

In this case, it quickly becomes clear that assertion (8) is not an invariant of the loop body because the loop body changes the variable Z, but (obviously) not the global constants x and z. So we need to generalize (8) to some statement that is equivalent to (8) when X

[213] `X=x/\Z=z`
[214] `Z-X=z-x`
[215] `Z-X=z-x/\X<>0`
[216] `(Z-1)-(X-1)=z-x`
[217] `Z-(X-1)=z-x`
[218] `Z-X=z-x`
[219] `Z-X=z-x/\~(X<>0)`
[220] `Z=z-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.

From this informal proof, it is now easy to read off a formal proof in terms of the Hoare rules:

```
Definition subtract_slowly : com :=
  WHILE BNot (BEq (AId X) (ANum 0)) DO
    Z ::= AMinus (AId Z) (ANum 1);
    X ::= AMinus (AId X) (ANum 1)
  END.

Definition subtract_slowly_invariant x z :=
  fun st ⇒ minus (st Z) (st X) = minus z x.

Theorem subtract_slowly_correct : ∀ x z,
  {{fun st ⇒ st X = x ∧ st Z = z}}
  subtract_slowly
  {{fun st ⇒ st Z = minus z x}}.
Proof.
  intros x z. unfold subtract_slowly.
  eapply hoare_consequence with (P' := subtract_slowly_invariant x z).
  apply hoare_while.
  Case "Loop body preserves invariant".
    eapply hoare_seq. apply hoare_asgn.
    eapply hoare_consequence_pre. apply hoare_asgn.
    unfold subtract_slowly_invariant, assn_sub, update, bassn. simpl.
    intros st [Inv GuardTrue].
    SearchAbout [negb true]. rewrite negb_true_iff in GuardTrue.
    SearchAbout [beq_nat false]. apply beq_nat_false in GuardTrue.
    omega.    Case "Initial state satisfies invariant".
    unfold subtract_slowly_invariant.
    intros st [HX HZ]. omega.
  Case "Invariant and negated guard imply postcondition".
    intros st [Inv GuardFalse].
    unfold subtract_slowly_invariant in Inv.
    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.
    omega. Qed.
```

21.5.2 Exercise: Reduce to Zero

Exercise: 2 stars (reduce_to_zero_correct) Here is a while loop that is so simple it needs no invariant: [221] WHILE X <> 0 DO [222] => [223] X ::= X - 1 [224] END [225] => [226] Your job is to translate this proof to Coq. Refer to the *slow_subtraction* example for ideas.

```
Definition reduce_to_zero : com :=
  WHILE BNot (BEq (AId X) (ANum 0)) DO
    X ::= AMinus (AId X) (ANum 1)
  END.

Theorem reduce_to_zero_correct :
  {{fun st ⇒ True}}
  reduce_to_zero
  {{fun st ⇒ st X = 0}}.
Proof.
  Admitted.
```

□

21.5.3 Exercise: Slow Addition

The following program adds the variable X into the variable Z by repeatedly decrementing X and incrementing Z.

```
Definition add_slowly : com :=
  WHILE BNot (BEq (AId X) (ANum 0)) DO
    Z ::= APlus (AId Z) (ANum 1);
    X ::= AMinus (AId X) (ANum 1)
  END.
```

Exercise: 3 stars (add_slowly_decoration) 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.

□

Exercise: 3 stars (add_slowly_formal) Now write down your specification of *add_slowly* formally, as a Coq *Hoare_triple*, and prove it valid.

□

[221] True
[222] True/\X<>0
[223] True
[224] True
[225] True/\X=0
[226] X=0

21.5.4 Example: Parity

Here's another, slightly trickier example. Make sure you understand the decorated program completely. You may find it instructive to start with the bare program and try to fill in the decorations yourself. Notice exactly where the condition $X \leq x$ comes up.

[227] => [228] Y ::= 0; [229] => [230] WHILE X <> 0 DO [231] => [232] Y ::= 1 - Y; [233] X ::= X - 1 [234] END [235] => [236]

And here is the formal version of this proof. Skim them, but you do not need to understand every detail (though the details are not actually very hard), since all the important ideas are already in the informal version.

```
Definition find_parity : com :=
  Y ::= (ANum 0);
  WHILE (BNot (BEq (AId X) (ANum 0))) DO
    Y ::= AMinus (ANum 1) (AId Y);
    X ::= AMinus (AId X) (ANum 1)
  END.

Definition find_parity_invariant x :=
  fun st ⇒
     st X ≤ x ∧ ((st Y = 0 ∧ ev (x - st X))
                       ∨ (st Y = 1 ∧ ¬ev (x - st X))).
```

We'll need the following lemma...

```
Lemma not_ev_ev_S_gen: ∀ n,
  (¬ ev n → ev (S n)) ∧
  (¬ ev (S n) → ev (S (S n))).
Proof.
  induction n as [| n'].
  Case "n = 0".
    split; intros H.
    SCase "->".
      apply ex_falso_quodlibet. apply H. apply ev_0.
    SCase "<-".
      apply ev_SS. apply ev_0.
  Case "n = S n'".
```

[227] X=x
[228] X=x/\0=0
[229] X=x/\Y=0
[230] (Y=0<->ev(x-X))/\X<=x
[231] (Y=0<->ev(x-X))/\X<=x/\X<>0
[232] (1-Y)=0<->ev(x-(X-1))/\X-1<=x
[233] Y=0<->ev(x-(X-1))/\X-1<=x
[234] Y=0<->ev(x-X)/\X<=x
[235] (Y=0<->ev(x-X))/\X<=x/\~(X<>0)
[236] Y=0<->evx

```
    inversion IHn' as [Hn HSn]. split; intros H.
    SCase "->".
      apply HSn. apply H.
    SCase "<-".
      apply ev_SS. apply Hn. intros contra.
      apply H. apply ev_SS. apply contra. Qed.

Lemma not_ev_ev_S : ∀ n,
  ¬ ev n → ev (S n).
Proof.
  intros n.
  destruct (not_ev_ev_S_gen n) as [H _].
  apply H.
Qed.

Theorem find_parity_correct : ∀ x,
  {{fun st ⇒ st X = x}}
  find_parity
  {{fun st ⇒ st Y = 0 ↔ ev x}}.
Proof.
  intros x. unfold find_parity.
  apply hoare_seq with (Q := find_parity_invariant x).
  eapply hoare_consequence.
  apply hoare_while with (P := find_parity_invariant x).
  Case "Loop body preserves invariant".
    eapply hoare_seq.
    apply hoare_asgn.
    eapply hoare_consequence_pre.
    apply hoare_asgn.
    intros st [[Inv1 Inv2] GuardTrue].
    unfold find_parity_invariant, bassn, assn_sub, aeval in *.
    rewrite update_eq.
    rewrite (update_neq Y X).
    rewrite (update_neq X Y).
    rewrite update_eq.
    simpl in GuardTrue. destruct (st X).
      inversion GuardTrue. simpl.
    split. omega.
    inversion Inv2 as [[H1 H2] | [H1 H2]]; rewrite H1;
                     [right|left]; (split; simpl; [omega |]).
    apply ev_not_ev_S in H2.
    replace (S (x - S n)) with (x-n) in H2 by omega.
    rewrite ← minus_n_O. assumption.
    apply not_ev_ev_S in H2.
```

```
    replace (S (x - S n)) with (x - n) in H2 by omega.
    rewrite ← minus_n_O. assumption.
    reflexivity. reflexivity.
  Case "Precondition implies invariant".
    intros st H. assumption.
  Case "Invariant implies postcondition".
    unfold bassn, find_parity_invariant. simpl.
    intros st [[Inv1 Inv2] GuardFalse].
    destruct (st X).
      split; intro.
        inversion Inv2.
           inversion H0 as [_ H1]. replace (x-0) with x in H1 by omega.
           assumption.
           inversion H0 as [H0' _]. rewrite H in H0'. inversion H0'.
        inversion Inv2.
           inversion H0. assumption.
           inversion H0 as [_ H1]. replace (x-0) with x in H1 by omega.
           apply ex_falso_quodlibet. apply H1. assumption.
      apply ex_falso_quodlibet. apply GuardFalse. reflexivity.
  Case "invariant established before loop".
    eapply hoare_consequence_pre.
    apply hoare_asgn.
    intros st H.
    unfold assn_sub, find_parity_invariant, update. simpl.
    subst.
    split.
    omega.
    replace (st X - st X) with 0 by omega.
    left. split. reflexivity.
    apply ev_0. Qed.
```

Exercise: 2 stars (wrong_find_parity_invariant) A plausible first attempt at stating an invariant for *find_parity* is the following.

```
Definition find_parity_invariant' x :=
  fun st ⇒
    (st Y) = 0 ↔ ev (x - st X).
```

Why doesn't this work? (Hint: Don't waste time trying to answer this exercise by attempting a formal proof and seeing where it goes wrong. Just think about whether the loop body actually preserves the property.)

☐

21.5.5 Example: Finding Square Roots

Here's another example, starting with the formal version this time.

```
Definition sqrt_loop : com :=
  WHILE BLe (AMult (APlus (ANum 1) (AId Z))
                   (APlus (ANum 1) (AId Z)))
            (AId X) DO
    Z ::= APlus (ANum 1) (AId Z)
  END.

Definition sqrt_com : com :=
  Z ::= ANum 0;
  sqrt_loop.

Definition sqrt_spec (x : nat) : Assertion :=
  fun st ⇒
       ((st Z) × (st Z)) ≤ x
    ∧ ¬ (((S (st Z)) × (S (st Z))) ≤ x).

Definition sqrt_inv (x : nat) : Assertion :=
  fun st ⇒
       st X = x
    ∧ ((st Z) × (st Z)) ≤ x.

Theorem random_fact_1 : ∀ st,
     (S (st Z)) × (S (st Z)) ≤ st X →
     bassn (BLe (AMult (APlus (ANum 1) (AId Z))
                       (APlus (ANum 1) (AId Z)))
                (AId X)) st.
Proof.
  intros st Hle. unfold bassn. simpl.
  destruct (st X) as [|x'].
  Case "st X = 0".
    inversion Hle.
  Case "st X = S x'".
    simpl in Hle. apply le_S_n in Hle.
    remember (ble_nat (plus (st Z)
                      ((st Z) × (S (st Z)))) x')
      as ble.
    destruct ble. reflexivity.
    symmetry in Heqble. apply ble_nat_false in Heqble.
    unfold not in Heqble. apply Heqble in Hle. inversion Hle.
Qed.

Theorem random_fact_2 : ∀ st,
     bassn (BLe (AMult (APlus (ANum 1) (AId Z))
```

```
                          (APlus (ANum 1) (AId Z)))
                  (AId X)) st →
        aeval st (APlus (ANum 1) (AId Z))
      × aeval st (APlus (ANum 1) (AId Z))
      ≤ st X.
Proof.
  intros st Hble. unfold bassn in Hble. simpl in *.
  destruct (st X) as [| x'].
  Case "st X = 0".
    inversion Hble.
  Case "st X = S x'".
    apply ble_nat_true in Hble. omega. Qed.

Theorem sqrt_com_correct : ∀ x,
  {{fun st ⇒ True}} (X ::= ANum x; sqrt_com) {{sqrt_spec x}}.
Proof.
  intros x.
  apply hoare_seq with (Q := fun st ⇒ st X = x).
  Case "sqrt_com".
    unfold sqrt_com.
    apply hoare_seq with (Q := fun st ⇒ st X = x
                                      ∧ st Z = 0).
    SCase "sqrt_loop".
      unfold sqrt_loop.
      eapply hoare_consequence.
      apply hoare_while with (P := sqrt_inv x).

      SSCase "loop preserves invariant".
        eapply hoare_consequence_pre.
        apply hoare_asgn. intros st H.
        unfold assn_sub. unfold sqrt_inv in *.
        inversion H as [[HX HZ] HP]. split.
        SSSCase "X is preserved".
          rewrite update_neq; try assumption; try reflexivity.
        SSSCase "Z is updated correctly".
          rewrite (update_eq (aeval st (APlus (ANum 1) (AId Z))) Z st).
          subst. apply random_fact_2. assumption.

      SSCase "invariant is true initially".
        intros st H. inversion H as [HX HZ].
        unfold sqrt_inv. split. assumption.
        rewrite HZ. simpl. omega.

      SSCase "after loop, spec is satisfied".
        intros st H. unfold sqrt_spec.
```

```
      inversion H as [HX HP].
      unfold sqrt_inv in HX. inversion HX as [HX' Harith].
      split. assumption.
      intros contra. apply HP. clear HP.
      simpl. simpl in contra.
      apply random_fact_1. subst x. assumption.
    SCase "Z set to 0".
     eapply hoare_consequence_pre. apply hoare_asgn.
     intros st HX.
     unfold assn_sub. split.
     rewrite update_neq. assumption. reflexivity.
     rewrite update_eq. reflexivity.
  Case "assignment of X".
    eapply hoare_consequence_pre. apply hoare_asgn.
    intros st H.
    unfold assn_sub. rewrite update_eq. reflexivity. Qed.
```

Exercise: 3 stars (sqrt_informal) Write an informal decorated program corresponding to this formal correctness proof.

□

21.5.6 Exercise: Factorial

Module Factorial.

Recall the mathematical factorial function...

```
Fixpoint real_fact (n:nat) : nat :=
  match n with
  | O ⇒ 1
  | S n' ⇒ n × (real_fact n')
  end.
```

... and the Imp program that we wrote to calculate factorials:

```
Definition fact_body : com :=
  Y ::= AMult (AId Y) (AId Z);
  Z ::= AMinus (AId Z) (ANum 1).

Definition fact_loop : com :=
  WHILE BNot (BEq (AId Z) (ANum 0)) DO
    fact_body
  END.

Definition fact_com : com :=
```

```
Z ::= (AId X);
Y ::= ANum 1;
fact_loop.
```

Exercise: 3 stars, optional (fact_informal) Decorate the *fact_com* program to show that it satisfies the specification given by the pre and postconditions below. As usual, feel free to elide the algebraic reasoning about arithmetic, the less-than relation, etc., that is formally required by the rule of consequence:

[237] Z ::= X; Y ::= 1; WHILE Z <> 0 DO Y ::= Y * Z; Z ::= Z - 1 END [238] □

Exercise: 4 stars, optional (fact_formal) Prove formally that fact_com satisfies this specification, using your informal proof as a guide. You may want to state the loop invariant separately (as we did in the examples).

```
Theorem fact_com_correct : ∀ x,
  {{fun st ⇒ st X = x}} fact_com
  {{fun st ⇒ st Y = real_fact x}}.
Proof.
   Admitted.
   □
End FACTORIAL.
```

21.6 Formalizing Decorated Programs (Optional)

The 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 the triple is reduced to simple logical and algebraic calculations showing that some assertions imply others.

In this optional section, we show that this informal presentation style can actually be made completely formal.

21.6.1 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
```

[237] X=x

[238] Y=real_factx

```
  | DCIf : bexp → Assertion → dcom → Assertion → dcom
             → Assertion→ dcom
  | DCWhile : bexp → Assertion → dcom → Assertion → dcom
  | DCPre : Assertion → dcom → dcom
  | DCPost : dcom → Assertion → dcom.

Tactic Notation "dcom_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "Skip" | Case_aux c "Seq" | Case_aux c "Asgn"
  | Case_aux c "If" | Case_aux c "While"
  | Case_aux c "Pre" | Case_aux c "Post" ].

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 91, 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 *com*mands, 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 (AId X) (ANum 0)))
  DO
    {{ fun st ⇒ True ∧ bassn (BNot (BEq (AId X) (ANum 0))) st}}
    X ::= (AMinus (AId X) (ANum 1))
    {{ fun _ ⇒ True }}
  END
  {{ fun st ⇒ True ∧ ¬bassn (BNot (BEq (AId X) (ANum 0))) st}} ⇒
  {{ 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 [239] ([240] SKIP [241]) ; ([242] SKIP [243]) [244], 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 $\{\{P\}\}$ $(extract\ d)$ $\{\{post\ d\}\}$, where *post* is defined as follows:

[239] P
[240] P
[241] P
[242] P
[243] P
[244] P

```
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.
```

We can define a similar function that extracts 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*).

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.

21.6.2 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 ~~> assn_sub X a Q)
  | 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, which captures the claim that the *verification_conditions* function does its job correctly. Not surprisingly, we need all of the Hoare Logic rules in the proof. We have used *in* variants of several tactics before to apply them to values in the context rather than the goal. An extension of this idea is the syntax *tactic* `in` *, which applies *tactic* in the goal and every hypothesis in the context. We most commonly use this facility in conjunction with the `simpl` tactic, as below.

```
Theorem verification_correct : ∀ d P,
  verification_conditions P d → {{P}} (extract d) {{post d}}.
Proof.
  dcom_cases (induction d) Case; intros P H; simpl in *.
  Case "Skip".
    eapply hoare_consequence_pre.
      apply hoare_skip.
```

```
      assumption.
  Case "Seq".
    inversion H as [H1 H2]. clear H.
    eapply hoare_seq.
      apply IHd2. apply H2.
      apply IHd1. apply H1.
  Case "Asgn".
    eapply hoare_consequence_pre.
      apply hoare_asgn.
      assumption.
  Case "If".
    inversion H as [HPre1 [HPre2 [Hd1 [Hd2 [HThen HElse]]]]]; clear H.
    subst.
    apply hoare_if.
      eapply hoare_consequence_pre. apply IHd1. eassumption. assumption.
      rewrite Hd2.
      eapply hoare_consequence_pre. apply IHd2. eassumption. assumption.
  Case "While".
    inversion H as [Hpre [Hbody [Hpost Hd]]]; subst; clear H.
    eapply hoare_consequence_pre.
    apply hoare_while with (P := post d).
      apply IHd. apply Hd.
      assumption.
  Case "Pre".
    inversion H as [HP Hd]; clear H.
    eapply hoare_consequence_pre. apply IHd. apply Hd. assumption.
  Case "Post".
    inversion H as [Hd HQ]; clear H.
    eapply hoare_consequence_post. apply IHd. apply Hd. assumption.
Qed.
```

21.6.3 Examples

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).
```

We can certainly work with them 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 applies splitting 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 update_eq;
  repeat (rewrite update_neq; [| reflexivity]);
  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 *;
  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 (x:nat) (z:nat) : dcom := (
    {{ fun st ⇒ st X = x ∧ st Z = z }}
  WHILE BNot (BEq (AId X) (ANum 0))
  DO {{ fun st ⇒ st Z - st X = z - x
                ∧ bassn (BNot (BEq (AId X) (ANum 0))) st }}
     Z ::= AMinus (AId Z) (ANum 1)
       {{ fun st ⇒ st Z - (st X - 1) = z - x }} ;
     X ::= AMinus (AId X) (ANum 1)
       {{ fun st ⇒ st Z - st X = z - x }}
  END
    {{ fun st ⇒ st Z
                - st X
                = z - x
              ∧ ¬ bassn (BNot (BEq (AId X) (ANum 0))) st }}
    ⇒
    {{ fun st ⇒ st Z = z - x }}
) % dcom.

Theorem subtract_slowly_dec_correct : ∀ x z,
```

dec_correct (subtract_slowly_dec x z).
Proof. intros x z. *verify*. Qed.

Exercise: 3 stars, optional (slow_assignment_dec) 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 an informal decorated program that implements this idea given a parameter x:

[245] X ::= x [246] ; Y ::= 0 [247] ; WHILE X <> 0 DO [248] X ::= X - 1 [249] ; Y ::= Y + 1 [250] END [251]

Write a corresponding formal decorated program (parametrized by x) and prove it correct.

```
Example slow_assignment_dec (x:nat) : dcom :=
admit.

Theorem slow_assignment_dec_correct : ∀ x,
  dec_correct (slow_assignment_dec x).
Proof. Admitted.
```

□

Exercise: 4 stars, optional (factorial_dec) Remember the factorial function we worked with before:

```
Fixpoint real_fact (n:nat) : nat :=
  match n with
  | O ⇒ 1
  | S n' ⇒ n × (real_fact n')
  end.
```

Following the pattern of *subtract_slowly_dec*, write a decorated Imp program that implements the factorial function, and prove it correct.

□

```
Definition div_mod_dec (a b : nat) : dcom := (
{{ fun st ⇒ True }} ⇒
  {{ fun st ⇒ b × 0 + a = a }}
  X ::= ANum a
  {{ fun st ⇒ b × 0 + st X = a }};
  Y ::= ANum 0
```

---

[245] True
[246] X=x
[247] X=x/\Y=0
[248] X+Y=x/\X>0
[249] Y+X+1=x
[250] Y+X=x
[251] Y=x/\X=0

```
  {{ fun st ⇒ b × st Y + st X = a }};
  WHILE (BLe (ANum b) (AId X)) DO
    {{ fun st ⇒ b × st Y + st X = a ∧ (bassn (BLe (ANum b) (AId X)) st) }} ⇒
    {{ fun st ⇒ b × (st Y + 1) + (st X - b) = a }}
    X ::= AMinus (AId X) (ANum b)
    {{ fun st ⇒ b × (st Y + 1) + st X = a }};
    Y ::= APlus (AId Y) (ANum 1)
    {{ fun st ⇒ b × st Y + st X = a }}
  END
  {{ fun st ⇒ b × st Y + st X = a ∧ ~(bassn (BLe (ANum b) (AId X)) st) }} ⇒
  {{ fun st ⇒ b × st Y + st X = a ∧ (st X < b) }}
)%dcom.

Theorem div_mod_dec_correct : ∀ a b,
  dec_correct (div_mod_dec a b).
Proof.
  intros a b. verify.
  Case "1". apply ble_nat_true in H0. rewrite mult_plus_distr_l. omega.
  Case "2". apply ble_nat_false in H0. omega.
Qed.
```

Chapter 22

Library Equiv

22.1 Equiv: Program Equivalence

```
Require Export Imp.
```

Some general advice for working on problems

- We've tried to make sure that most of the Coq proofs we ask you to do are similar to proofs that we've provided. Before starting to work on the homework 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 "following your nose" or random hacking. 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.
- 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.

22.2 Behavioral Equivalence

In the last 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.

22.2.1 Definitions

For *aexps* and *bexps* with variables, the definition we want is clear. We say that two *aexps* or *bexps* 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 (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, optional (pairs_equiv) Which of the following pairs of programs are equivalent? Write "yes" or "no" for each one.

(a) WHILE (BLe (ANum 1) (AId X)) DO X ::= APlus (AId X) (ANum 1) END and WHILE (BLe (ANum 2) (AId X)) DO X ::= APlus (AId X) (ANum 1) END

(b) WHILE BTrue DO WHILE BFalse DO X ::= APlus (AId X) (ANum 1) END END and WHILE BFalse DO WHILE BTrue DO X ::= APlus (AId X) (ANum 1) END END

☐

Exercise: 3 stars (equiv_classes) Given the following programs, group together those that are equivalent in *Imp*. 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: {a,b,c,d,e,f,g,h} {i}.

(a) WHILE X > 0 DO X ::= X + 1 END

(b) IFB X = 0 THEN X ::= X + 1; Y ::= 1 ELSE Y ::= 0 FI; X ::= X - Y; Y ::= 0

(c) SKIP

(d) WHILE X <> 0 DO X ::= X * Y + 1 END

(e) Y ::= 0

(f) Y ::= X + 1; WHILE X <> Y DO Y ::= X + 1 END
(g) WHILE BTrue DO SKIP END
(h) WHILE X <> X DO X ::= X + 1 END
(i) WHILE X <> Y DO X ::= Y + 1 END
□

22.2.2 Examples

Here are some simple examples of equivalences of arithmetic and boolean expressions.

```
Theorem aequiv_example:
  aequiv (AMinus (AId X) (AId X)) (ANum 0).
Proof.
  intros st. simpl. apply minus_diag.
Qed.
```

```
Theorem bequiv_example:
  bequiv (BEq (AMinus (AId X) (AId 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: ∀ c,
  cequiv
     (SKIP; c)
     c.
Proof.
  intros c st st'.
  split; intros H.
  Case "->".
    inversion H. subst.
    inversion H2. subst.
    assumption.
  Case "<-".
    apply E_Seq with st.
    apply E_Skip.
    assumption.
Qed.
```

Exercise: 2 stars (skip_right) Theorem skip_right: ∀ c,
  cequiv

```
    (c; SKIP)
    c.
Proof.
   Admitted.
   □
```

Similarly, here is a simple transformations that simplifies *IFB* commands:

```
Theorem IFB_true_simple: ∀ c1 c2,
  cequiv
    (IFB BTrue THEN c1 ELSE c2 FI)
    c1.
Proof.
  intros c1 c2.
  split; intros H.
  Case "->".
    inversion H; subst. assumption. inversion H5.
  Case "<-".
    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:

- (→) We must show, for all *st* and *st'*, that if *IFB b THEN c1 ELSE c2 FI / st || st'* then *c1 / st || st'*.

  Proceed by cases on the rules that could possibly have been used to show *IFB b THEN c1 ELSE c2 FI / st || st'*, namely *E_IfTrue* and *E_IfFalse*.

  - Suppose the final rule rule in the derivation of *IFB b THEN c1 ELSE c2 FI / st || st'* was *E_IfTrue*. We then have, by the premises of *E_IfTrue*, that *c1 / st || 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 || st'* was *E_IfFalse*. We then know that *beval st b = false* and *c2 / st || st'*.

    Recall that *b* is equivalent to *BTrue*, i.e. forall *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*.

- (←) We must show, for all *st* and *st'*, that if *c1 / st || st'* then *IFB b THEN c1 ELSE c2 FI / st || st'*.

Since *b* is equivalent to *BTrue*, we know that *beval st b* = *beval st BTrue* = *true*. Together with the assumption that *c1 / st* || *st'*, we can apply *E_IfTrue* to derive *IFB b THEN c1 ELSE c2 FI / st* || *st'*. □

Here is the formal version of this proof:

```
Theorem IFB_true: ∀ b c1 c2,
    bequiv b BTrue →
    cequiv
      (IFB b THEN c1 ELSE c2 FI)
      c1.
Proof.
  intros b c1 c2 Hb.
  split; intros H.
  Case "->".
    inversion H; subst.
    SCase "b evaluates to true".
      assumption.
    SCase "b evaluates to false (contradiction)".
      rewrite Hb in H5.
      inversion H5.
  Case "<-".
    apply E_IfTrue; try assumption.
    rewrite Hb. reflexivity. Qed.
```

Exercise: 2 stars, recommended (IFB_false)

```
Theorem IFB_false: ∀ b c1 c2,
  bequiv b BFalse →
  cequiv
    (IFB b THEN c1 ELSE c2 FI)
    c2.
Proof.
   Admitted.
```

□

Exercise: 3 stars (swap_if_branches)

```
Theorem swap_if_branches: ∀ b e1 e2,
  cequiv
    (IFB b THEN e1 ELSE e2 FI)
    (IFB BNot b THEN e2 ELSE e1 FI).
Proof.
   Admitted.
```

□

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 : ∀ b c,
     bequiv b BFalse →
     cequiv
       (WHILE b DO c END)
       SKIP.
Proof.
  intros b c Hb. split; intros H.
  Case "->".
    inversion H; subst.
    SCase "E_WhileEnd".
      apply E_Skip.
    SCase "E_WhileLoop".
      rewrite Hb in H2. inversion H2.
  Case "<-".
    inversion H; subst.
    apply E_WhileEnd.
    rewrite Hb.
    reflexivity. Qed.
```

Exercise: 2 stars (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.
  ceval_cases (induction H) Case;

    inversion Heqcw; subst; clear Heqcw.
  Case "E_WhileEnd". unfold bequiv in Hb.
    rewrite Hb in H. inversion H.
  Case "E_WhileLoop".     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) Hint: You'll want to use *WHILE_true_nonterm* here.

```
Theorem WHILE_true: ∀ b c,
     bequiv b BTrue →
     cequiv
       (WHILE b DO c END)
       (WHILE BTrue DO SKIP END).
Proof.
   Admitted.
```

☐

```
Theorem loop_unrolling: ∀ 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.
  Case "->".
    inversion Hce; subst.
    SCase "loop doesn't run".
      apply E_IfFalse. assumption. apply E_Skip.
    SCase "loop runs".
      apply E_IfTrue. assumption.
      apply E_Seq with (st' := st'0). assumption. assumption.
  Case "<-".
    inversion Hce; subst.
```

```
  SCase "loop runs".
    inversion H5; subst.
    apply E_WhileLoop with (st' := st'0).
    assumption. assumption. assumption.
  SCase "loop doesn't run".
    inversion H5; subst. apply E_WhileEnd. assumption. Qed.
```

Exercise: 2 stars, optional (seq_assoc)

```
Theorem seq_assoc : ∀ c1 c2 c3,
  cequiv ((c1;c2);c3) (c1;(c2;c3)).
Proof.
  Admitted.
```

☐

22.2.3 The Functional Equivalence Axiom

Finally, let's look at simple equivalences involving assignments. For example, we might expect to be able to show that *X ::= AId X* is equivalent to *SKIP*. However, when we try to show it, we get stuck in an interesting way.

```
Theorem identity_assignment_first_try : ∀ (X:id),
  cequiv (X ::= AId X) SKIP.
Proof.
  intros. split; intro H.
    Case "->".
      inversion H; subst. simpl.
      replace (update st X (st X)) with st.
      constructor.
Admitted.
```

What is going on here? Recall that our states are just functions from identifiers to values. For Coq, functions are only equal when their definitions are syntactically the same, modulo simplification. (This is the only way we can legally apply the *refl_equal* constructor of the inductively defined proposition *eq*!) In practice, for functions built up by repeated uses of the *update* operation, this means that two functions can be proven equal only if they were constructed using the *same update* operations, applied in the same order. In the theorem above, the sequence of updates on the first parameter *cequiv* is one longer than for the second parameter, so it is no wonder that the equality doesn't hold.

This problem is actually quite general. If we try to prove other simple facts, such as cequiv (X ::= APlus (AId X ANum 1) ; X ::= APlus (AId X ANum 1)) (X ::= APlus (AId X ANum 2)) or cequiv (X ::= ANum 1; Y ::= ANum 2) (y ::= ANum 2; X ::= ANum 1)

we'll get stuck in the same way: we'll have two functions that behave the same way on all inputs, but cannot be proven to be *eq* to each other.

The reasoning principle we would like to use in these situations is called *functional extensionality*: forall x, f x = g x f = g Although this principle is not derivable in Coq's built-in logic, it is safe to add it as an additional *axiom*.

```
Axiom functional_extensionality : ∀ {X Y: Type} {f g : X → Y},
  (∀ (x: X), f x = g x) → f = g.
```

It can be shown that adding this axiom doesn't introduce any inconsistencies into Coq. (In this way, it is similar to adding one of the classical logic axioms, such as *excluded_middle*.)

With the benefit of this axiom we can prove our theorem.

```
Theorem identity_assignment : ∀ (X:id),
  cequiv
    (X ::= AId X)
    SKIP.
Proof.
   intros. split; intro H.
     Case "->".
       inversion H; subst. simpl.
       replace (update st X (st X)) with st.
       constructor.
       apply functional_extensionality. intro.
       rewrite update_same; reflexivity.
     Case "<-".
       inversion H; subst.
       assert (st' = (update st' X (st' X))).
          apply functional_extensionality. intro.
          rewrite update_same; reflexivity.
       rewrite H0 at 2.
       constructor. reflexivity.
Qed.
```

Exercise: 2 stars, recommended (assign_aequiv)

```
Theorem assign_aequiv : ∀ X e,
  aequiv (AId X) e →
  cequiv SKIP (X ::= e).
Proof.
   Admitted.
```

□

Exercise: 2 stars, optional (functional_extensionality_failed_false) As we just mentioned, functional extensionality doesn't add any inconsistencies into Coq. However, consider the following theorem. Inspired by it, we present an attempt to prove *False*. Briefly explain why it will fail.

```
Theorem feff_1 : true = false → False.
```

```
Proof.
  intros. inversion H.
Qed.
```

```
Lemma feff_2 : empty_state = update empty_state X 0.
Proof.
  apply functional_extensionality. intros.
  destruct x as [n]. destruct n as [| n'].
  Case "x = AId 0". reflexivity.
  Case "x = AId (S n')". reflexivity.
Qed.
```

If we were able to prove *feff_3*, then together with *feff_2*, a proof of *False* is immediate. Explain why the `inversion` *H* fails, though it works in *feff_1*.

22.3 Properties of Behavioral Equivalence

We now turn to developing some of the properties of the program equivalences we have defined.

22.3.1 Behavioral Equivalence is an Equivalence

First, we verify that the equivalences on *aexps*, *bexps*, and *coms* really are *equivalences* – i.e., that they are reflexive, symmetric, and transitive:

```
Lemma refl_aequiv : ∀ (a : aexp), aequiv a a.
Proof.
  intros a st. reflexivity. Qed.

Lemma sym_aequiv : ∀ (a1 a2 : aexp),
  aequiv a1 a2 → aequiv a2 a1.
Proof.
  intros a1 a2 H. intros st. symmetry. apply H. Qed.

Lemma trans_aequiv : ∀ (a1 a2 a3 : aexp),
  aequiv a1 a2 → aequiv a2 a3 → aequiv a1 a3.
Proof.
  unfold aequiv. intros a1 a2 a3 H12 H23 st.
  rewrite (H12 st). rewrite (H23 st). reflexivity. Qed.

Lemma refl_bequiv : ∀ (b : bexp), bequiv b b.
Proof.
  unfold bequiv. intros b st. reflexivity. Qed.

Lemma sym_bequiv : ∀ (b1 b2 : bexp),
  bequiv b1 b2 → bequiv b2 b1.
```

```
Proof.
  unfold bequiv. intros b1 b2 H. intros st. symmetry. apply H. Qed.
Lemma trans_bequiv : ∀ (b1 b2 b3 : bexp),
  bequiv b1 b2 → bequiv b2 b3 → bequiv b1 b3.
Proof.
  unfold bequiv. intros b1 b2 b3 H12 H23 st.
  rewrite (H12 st). rewrite (H23 st). reflexivity. Qed.
Lemma refl_cequiv : ∀ (c : com), cequiv c c.
Proof.
  unfold cequiv. intros c st st'. apply iff_refl. Qed.
Lemma sym_cequiv : ∀ (c1 c2 : com),
  cequiv c1 c2 → cequiv c2 c1.
Proof.
  unfold cequiv. intros c1 c2 H st st'.
  assert (c1 / st || st' ↔ c2 / st || st') as H'.
    SCase "Proof of assertion". apply H.
  apply iff_sym. assumption.
Qed.
Lemma iff_trans : ∀ (P1 P2 P3 : Prop),
  (P1 ↔ P2) → (P2 ↔ P3) → (P1 ↔ 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 : ∀ (c1 c2 c3 : com),
  cequiv c1 c2 → cequiv c2 c3 → 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.
```

22.3.2 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: aequiv a1 a1'

---

cequiv (i ::= a1) (i ::= a1')
  cequiv c1 c1' cequiv c2 c2'

---

cequiv (c1;c2) (c1';c2') ...and so on. (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 : ∀ i a1 a1',
  aequiv a1 a1' →
  cequiv (CAss i a1) (CAss i a1').
Proof.
  intros i a1 a2 Heqv st st'.
  split; intros Hceval.
  Case "->".
    inversion Hceval. subst. apply E_Ass.
    rewrite Heqv. reflexivity.
  Case "<-".
    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 || st'* iff *WHILE b1' DO c1' END / st || st'*. We consider the two directions separately.

- (→) We show that *WHILE b1 DO c1 END / st || st'* implies *WHILE b1' DO c1' END / st || st'*, by induction on a derivation of *WHILE b1 DO c1 END / st || 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 *beval st b1 = false* and *st = st'*. But then, since *b1* and *b1'* are equivalent, we have *beval st b1' = false*, and *E-WhileEnd* applies, giving us *WHILE b1' DO c1' END / st || st'*, as required.
  - *E_WhileLoop*: The form of the rule now gives us *beval st b1 = true*, with *c1 / st || st'0* and *WHILE b1 DO c1 END / st'0 || st'* for some state *st'0*, with the induction hypothesis *WHILE b1' DO c1' END / st'0 || st'*.

    Since *c1* and *c1'* are equivalent, we know that *c1' / st || st'0*. And since *b1* and *b1'* are equivalent, we have *beval st b1' = true*. Now *E-WhileLoop* applies, giving us *WHILE b1' DO c1' END / st || st'*, as required.

- (←) Similar. □

```
Theorem CWhile_congruence : ∀ b1 b1' c1 c1',
  bequiv b1 b1' → cequiv c1 c1' →
  cequiv (WHILE b1 DO c1 END) (WHILE b1' DO c1' END).
Proof.
  unfold bequiv,cequiv.
  intros b1 b1' c1 c1' Hb1e Hc1e st st'.
  split; intros Hce.
  Case "->".
    remember (WHILE b1 DO c1 END) as cwhile.
    induction Hce; inversion Heqcwhile; subst.
    SCase "E_WhileEnd".
      apply E_WhileEnd. rewrite ← Hb1e. apply H.
    SCase "E_WhileLoop".
      apply E_WhileLoop with (st' := st').
      SSCase "show loop runs". rewrite ← Hb1e. apply H.
      SSCase "body execution".
        apply (Hc1e st st'). apply Hce1.
      SSCase "subsequent loop execution".
        apply IHHce2. reflexivity.
  Case "<-".
    remember (WHILE b1' DO c1' END) as c'while.
    induction Hce; inversion Heqc'while; subst.
    SCase "E_WhileEnd".
      apply E_WhileEnd. rewrite → Hb1e. apply H.
    SCase "E_WhileLoop".
      apply E_WhileLoop with (st' := st').
      SSCase "show loop runs". rewrite → Hb1e. apply H.
      SSCase "body execution".
        apply (Hc1e st st'). apply Hce1.
      SSCase "subsequent loop execution".
        apply IHHce2. reflexivity. Qed.
```

Exercise: 3 stars, optional (CSeq_congruence) Theorem CSeq_congruence : ∀ *c1 c1' c2 c2'*,

```
  cequiv c1 c1' → cequiv c2 c2' →
  cequiv (c1;c2) (c1';c2').
Proof.
   Admitted.
```

□

Exercise: 3 stars (CIf_congruence) Theorem CIf_congruence : ∀ *b b' c1 c1' c2 c2'*,
bequiv *b b'* → cequiv *c1 c1'* → cequiv *c2 c2'* →
cequiv (IFB *b* THEN *c1* ELSE *c2* FI) (IFB *b'* THEN *c1'* ELSE *c2'* 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 (AId X) (ANum 0))
     THEN
       Y ::= ANum 0
     ELSE
       Y ::= ANum 42
     FI)
    (X ::= ANum 0;
     IFB (BEq (AId X) (ANum 0))
     THEN
       Y ::= AMinus (AId X) (AId X)
     ELSE
       Y ::= ANum 42
     FI).
Proof.
  apply CSeq_congruence.
    apply refl_cequiv.
    apply CIf_congruence.
      apply refl_bequiv.
      apply CAss_congruence. unfold aequiv. simpl.
        symmetry. apply minus_diag.
      apply refl_cequiv.
Qed.
```

22.4 Case Study: Constant Folding

A *program transformation* is a function that takes a program as input and produces some variant of the program as its output. Compiler optimizations such as constant folding are a canonical example, but there are many others.

22.4.1 Soundness of Program Transformations

A program transformation is *sound* if it preserves the behavior of the original program.

We can define a notion of soundness for translations of *aexps*, *bexps*, and *coms*.

```
Definition atrans_sound (atrans : aexp → aexp) : Prop :=
  ∀ (a : aexp),
    aequiv a (atrans a).

Definition btrans_sound (btrans : bexp → bexp) : Prop :=
  ∀ (b : bexp),
    bequiv b (btrans b).

Definition ctrans_sound (ctrans : com → com) : Prop :=
  ∀ (c : com),
    cequiv c (ctrans c).
```

22.4.2 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
  | AId i ⇒ AId 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.

Example fold_aexp_ex1 :
    fold_constants_aexp
      (AMult (APlus (ANum 1) (ANum 2)) (AId X))
  = AMult (ANum 3) (AId 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 (AId X) (APlus (AMult (ANum 0) (ANum 6)) (AId Y)))
  = AMinus (AId X) (APlus (ANum 0) (AId Y)).
Proof. reflexivity. Qed.
```

Not only can we lift *fold_constants_aexp* to *bexp*s (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 ble_nat 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 (AId X) (AId Y))
            (BEq (ANum 0)
                 (AMinus (ANum 2) (APlus (ANum 1) (ANum 1)))))
  = BAnd (BEq (AId X) (AId 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 (AId X) (ANum 3);
     IFB BEq (AMinus (AId X) (AId 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 (AId Y) (ANum 0) DO
        X ::= APlus (AId X) (ANum 1)
      END) =
   (X ::= ANum 9;
    Y ::= AMinus (AId X) (ANum 3);
    IFB BEq (AMinus (AId X) (AId Y)) (ANum 6) THEN
      SKIP
    ELSE
      (Y ::= ANum 0)
    FI;
    Y ::= ANum 0;
    WHILE BEq (AId Y) (ANum 0) DO
      X ::= APlus (AId X) (ANum 1)
    END).
Proof. reflexivity. Qed.
```

22.4.3 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.
  aexp_cases (induction a) Case; 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_BEq_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*, 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 a1) (aeval a2) = beq_nat n1 n2. Also, it is easy to see (by considering the cases *n1* = *n2* and *n1* $\neq$ *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) aeval st a2 = aeval st (fold_constants_aexp a2), completing the case. □

```
Theorem fold_constants_bexp_sound:
  btrans_sound fold_constants_bexp.
Proof.
  unfold btrans_sound. intros b. unfold bequiv. intros st.
  bexp_cases (induction b) Case;

    try reflexivity.
  Case "BEq".
    rename a into a1. rename a0 into a2. simpl.
    remember (fold_constants_aexp a1) as a1'.
    remember (fold_constants_aexp a2) as a2'.
    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.
      simpl. destruct (beq_nat n n0); reflexivity.
  Case "BLe".
   admit.
  Case "BNot".
    simpl. remember (fold_constants_bexp b) as b'.
    rewrite IHb.
    destruct b'; reflexivity.
  Case "BAnd".
    simpl.
    remember (fold_constants_bexp b1) as b1'.
    remember (fold_constants_bexp b2) as b2'.
    rewrite IHb1. rewrite IHb2.
    destruct b1'; destruct b2'; reflexivity. Qed.
```
□

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.
  com_cases (induction c) Case; simpl.
  Case "SKIP". apply refl_cequiv.
  Case "::=". apply CAss_congruence. apply fold_constants_aexp_sound.
  Case ";". apply CSeq_congruence; assumption.
  Case "IFB".
    assert (bequiv b (fold_constants_bexp b)).
      SCase "Pf of assertion". apply fold_constants_bexp_sound.
    remember (fold_constants_bexp b) as b'.
    destruct b';

      try (apply CIf_congruence; assumption).
    SCase "b always true".
      apply trans_cequiv with c1; try assumption.
      apply IFB_true; assumption.
    SCase "b always false".
      apply trans_cequiv with c2; try assumption.
      apply IFB_false; assumption.
```

Case "WHILE".
Admitted.
☐

Soundness of (0 + n) Elimination, Redux

Exercise: 4 stars, optional (optimize_0plus) Recall the definition *optimize_0plus* from Imp.v: 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 *aexp*s, without states.

Write a new version of this function that accounts for variables, and analogous ones for *bexp*s 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.)

☐

22.5 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 (AId X) (ANum 1);
            Y ::= AId X)
      as c1.
  remember (X ::= APlus (AId X) (ANum 1);
            Y ::= APlus (AId X) (ANum 1))
      as c2.
  assert (cequiv c1 c2) by (subst; apply Contra).
  remember (update (update empty_state X 1) Y 1) as st1.
  remember (update (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' := (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)
  | VNUId: ∀ Y, X ≠ Y → var_not_used_in_aexp X (AId 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 →
      var_not_used_in_aexp X a2 →
      var_not_used_in_aexp X (AMult a1 a2).
Lemma aeval_weakening : ∀ i st a ni,
  var_not_used_in_aexp i a →
  aeval (update st i ni) a = aeval st a.
Proof.
   Admitted.
```

Using *var_not_used_in_aexp*, formalize and prove a correct verson of *subst_equiv_property*.

☐

Exercise: 3 stars, recommended (inequiv_exercise)
```
Theorem inequiv_exercise:
  ¬ cequiv (WHILE BTrue DO SKIP END) SKIP.
Proof.
   Admitted.
```

□

22.6 Extended exercise: Non-deterministic 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 non-deterministic 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*, non-deterministically. 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 non-deterministic code.

In a sense a variable on which we do *HAVOC* roughly corresponds to an unitialized variable in the C programming language. 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 the language, we first add a clause to the definition of commands.

```
Inductive com : Type :=
  | CSkip : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com
  | CHavoc : id → com.

Tactic Notation "com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "SKIP" | Case_aux c "::=" | Case_aux c ";"
  | Case_aux c "IFB" | Case_aux c "WHILE" | Case_aux c "HAVOC" ].

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'" :=
  (CIf 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 || 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').

Tactic Notation "ceval_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E_Skip" | Case_aux c "E_Ass" | Case_aux c "E_Seq"
  | Case_aux c "E_IfTrue" | Case_aux c "E_IfFalse"
  | Case_aux c "E_WhileEnd" | Case_aux c "E_WhileLoop"
```

].

As a sanity check, the following claims should be provable for your definition:

```
Example havoc_example1 : (HAVOC X) / empty_state || update empty_state X 0.
Proof.
  Admitted.

Example havoc_example2 :
  (SKIP; HAVOC Z) / empty_state || 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'.
```

This definition still makes perfect sense in the case of always terminating programs, so let's apply it to prove some non-deterministic programs equivalent or non-equivalent.

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 (havoc_copy) Are the following two programs equivalent?

```
Definition ptwice :=
  HAVOC X; HAVOC Y.

Definition pcopy :=
  HAVOC X; Y ::= AId 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 ∨ ¬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 non-determinism, like Himp, some programs always terminate, some programs always diverge, and some programs can non-deterministically terminate in some runs and diverge in others. The final part of the following optional exercise illustrates this phenomenon.

Exercise: 5 stars, optional (havoc_diverge) Prove the following program equivalences and non-equivalences, and try to understand why the *cequiv* definition has the behavior it has on these examples.

```
Definition p1 : com :=
  WHILE (BNot (BEq (AId X) (ANum 0))) DO
    HAVOC Y;
    X ::= APlus (AId X) (ANum 1)
  END.

Definition p2 : com :=
  WHILE (BNot (BEq (AId X) (ANum 0))) DO
    SKIP
  END.

Theorem p1_p2_equiv : cequiv p1 p2.
Proof. Admitted.

Definition p3 : com :=
  Z ::= ANum 1;
  WHILE (BNot (BEq (AId 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.

Definition p5 : com :=
  WHILE (BNot (BEq (AId X) (ANum 1))) DO
    HAVOC X
  END.

Definition p6 : com :=
  X ::= ANum 1.

Theorem p5_p6_equiv : cequiv p5 p6.
Proof. Admitted.
```

□

End HIMP.

22.7 Doing Without Extensionality (Optional)

Purists might object to using the *functional_extensionality* axiom. In general, it can be quite dangerous to add axioms, particularly several at once (as they may be mutually inconsistent). In fact, *functional_extensionality* and *excluded_middle* can both be assumed without any problems, but some Coq users prefer to avoid such "heavyweight" general techniques, and instead craft solutions for specific problems that stay within Coq's standard logic.

For our particular problem here, rather than extending the definition of equality to do what we want on functions representing states, we could instead give an explicit notion of *equivalence* on states. For example:

```
Definition stequiv (st1 st2 : state) : Prop :=
  ∀ (X:id), st1 X = st2 X.
```

```
Notation "st1 '¬' st2" := (stequiv st1 st2) (at level 30).
```

It is easy to prove that *stequiv* is an *equivalence* (i.e., it is reflexive, symmetric, and transitive), so it partitions the set of all states into equivalence classes.

Exercise: 1 star, optional (stequiv_refl) Lemma stequiv_refl : ∀ (*st* : state),

```
  st ¬ st.
Proof.
  Admitted.
```

□

Exercise: 1 star, optional (stequiv_sym) Lemma stequiv_sym : ∀ (*st1 st2* : state),

```
  st1 ¬ st2 →
  st2 ¬ st1.
Proof.
  Admitted.
```

□

Exercise: 1 star, optional (stequiv_trans) Lemma stequiv_trans : ∀ (*st1 st2 st3* : state),

```
  st1 ¬ st2 →
  st2 ¬ st3 →
  st1 ¬ st3.
Proof.
  Admitted.
```

□

Another useful fact...

Exercise: 1 star, optional (stequiv_update) Lemma stequiv_update : ∀ (*st1 st2* : state),
st1 ¬ *st2* →
∀ (*X*:**id**) (*n*:**nat**),
update *st1 X n* ¬ update *st2 X n*.
Proof.
Admitted.
□

It is then straightforward to show that *aeval* and *beval* behave uniformly on all members of an equivalence class:

Exercise: 2 stars, optional (stequiv_aeval) Lemma stequiv_aeval : ∀ (*st1 st2* : state),
st1 ¬ *st2* →
∀ (*a*:**aexp**), aeval *st1 a* = aeval *st2 a*.
Proof.
Admitted.
□

Exercise: 2 stars, optional (stequiv_beval) Lemma stequiv_beval : ∀ (*st1 st2* : state),
st1 ¬ *st2* →
∀ (*b*:**bexp**), beval *st1 b* = beval *st2 b*.
Proof.
Admitted.
□

We can also characterize the behavior of *ceval* on equivalent states (this result is a bit more complicated to write down because *ceval* is a relation).

```
Lemma stequiv_ceval: ∀ (st1 st2 : state),
  st1 ¬ st2 →
  ∀ (c: com) (st1': state),
    (c / st1 || st1') →
    ∃ st2' : state,
    ((c / st2 || st2') ∧ st1' ¬ st2').
Proof.
  intros st1 st2 STEQV c st1' CEV1. generalize dependent st2.
  induction CEV1; intros st2 STEQV.
  Case "SKIP".
    ∃ st2. split.
      constructor.
      assumption.
  Case ":=".
    ∃ (update st2 X n). split.
       constructor. rewrite ← H. symmetry. apply stequiv_aeval.
       assumption. apply stequiv_update. assumption.
```

```
  Case ";".
    destruct (IHCEV1_1 st2 STEQV) as [st2' [P1 EQV1]].
    destruct (IHCEV1_2 st2' EQV1) as [st2'' [P2 EQV2]].
    ∃ st2''. split.
      apply E_Seq with st2'; assumption.
      assumption.
  Case "IfTrue".
    destruct (IHCEV1 st2 STEQV) as [st2' [P EQV]].
    ∃ st2'. split.
      apply E_IfTrue. rewrite ← H. symmetry. apply stequiv_beval.
      assumption. assumption. assumption.
  Case "IfFalse".
    destruct (IHCEV1 st2 STEQV) as [st2' [P EQV]].
    ∃ st2'. split.
     apply E_IfFalse. rewrite ← H. symmetry. apply stequiv_beval.
     assumption. assumption. assumption.
  Case "WhileEnd".
    ∃ st2. split.
      apply E_WhileEnd. rewrite ← H. symmetry. apply stequiv_beval.
      assumption. assumption.
  Case "WhileLoop".
    destruct (IHCEV1_1 st2 STEQV) as [st2' [P1 EQV1]].
    destruct (IHCEV1_2 st2' EQV1) as [st2'' [P2 EQV2]].
    ∃ st2''. split.
      apply E_WhileLoop with st2'. rewrite ← H. symmetry.
      apply stequiv_beval. assumption. assumption. assumption.
      assumption.
Qed.
```

Now we need to redefine *cequiv* to use ¬ instead of =. It is not completely trivial to do this in a way that keeps the definition simple and symmetric, but here is one approach (thanks to Andrew McCreight). We first define a looser variant of || that "folds in" the notion of equivalence.

```
Reserved Notation "c1 '/' st '||'' st'" (at level 40, st at level 39).

Inductive ceval' : com → state → state → Prop :=
  | E_equiv : ∀ c st st' st'',
     c / st || st' →
     st' ¬ st'' →
     c / st ||' st''
  where "c1 '/' st '||'' st'" := (ceval' c1 st st').
```

Now the revised definition of *cequiv'* looks familiar:

```
Definition cequiv' (c1 c2 : com) : Prop :=
```

```
  ∀ (st st' : state),
    (c1 / st ||' st') ↔ (c2 / st ||' st').
```

A sanity check shows that the original notion of command equivalence is at least as strong as this new one. (The converse is not true, naturally.)

```
Lemma cequiv__cequiv' : ∀ (c1 c2: com),
  cequiv c1 c2 → cequiv' c1 c2.
Proof.
  unfold cequiv, cequiv'; split; intros.
    inversion H0 ; subst. apply E_equiv with st'0.
    apply (H st st'0); assumption. assumption.
    inversion H0 ; subst. apply E_equiv with st'0.
    apply (H st st'0). assumption. assumption.
Qed.
```

Exercise: 2 stars, optional (identity_assignment') Finally, here is our example once more... (You can complete the proof.)

```
Example identity_assignment' :
  cequiv' SKIP (X ::= AId X).
Proof.
    unfold cequiv'. intros. split; intros.
    Case "->".
      inversion H; subst; clear H. inversion H0; subst.
      apply E_equiv with (update st'0 X (st'0 X)).
      constructor. reflexivity. apply stequiv_trans with st'0.
      unfold stequiv. intros. apply update_same.
      reflexivity. assumption.
    Case "<-".
   Admitted.
```

☐

On the whole, this explicit equivalence approach is considerably harder to work with than relying on functional extensionality. (Coq does have an advanced mechanism called "setoids" that makes working with equivalences somewhat easier, by allowing them to be registered with the system so that standard rewriting tactics work for them almost as well as for equalities.) But it is worth knowing about, because it applies even in situations where the equivalence in question is *not* over functions. For example, if we chose to represent state mappings as binary search trees, we would need to use an explicit equivalence of this kind.

22.8 Additional Exercises

Exercise: 4 stars, optional (for_while_equiv) This exercise extends the optional *add_for_loop* exercise from Imp.v, 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) Theorem swap_noninterfering_assig

```
∀ l1 l2 a1 a2,
  l1 ≠ l2 →
  var_not_used_in_aexp l1 a2 →
  var_not_used_in_aexp l2 a1 →
  cequiv
    (l1 ::= a1; l2 ::= a2)
    (l2 ::= a2; l1 ::= a1).
Proof.
   Admitted.
   □
```

Chapter 23

Library Extraction

23.1 Extraction: Extracting ML from Coq

23.2 Basic Extraction

In its simplest form, program extraction from 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` 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` "imp1.ml" *ceval_step*.

When Coq processes this command, it generates a file *imp1.ml* containing an extracted version of *ceval_step*, together with everything that it recursively depends on. Have a look at this file now.

23.3 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*.

23.4 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 somehow 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 more 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.)

23.5 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 is not certified in any interesting sense, since we didn't prove anything about it.)

Chapter 24

Library ImpCEvalFun

24.1 ImpCEvalFun: Evaluation Function for Imp

24.1.1 Evaluation Function

```
Require Import Imp.
```

Here's a 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 ⇒
        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.
```

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...

Second try, using an extra numeric argument as a "step index" to ensure that evaluation always terminates.

```
Fixpoint ceval_step2 (st : state) (c : com) (i : nat) : state :=
  match i with
  | O ⇒ empty_state
  | S i' ⇒
    match c with
      | SKIP ⇒
          st
      | l ::= a1 ⇒
          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.

Third try, returning 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
  | O ⇒ None
  | S i' ⇒
    match c with
      | SKIP ⇒
          Some st
      | l ::= a1 ⇒
          Some (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 definition 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 (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 ⇒ 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: $1 + 2 + ... + X$) 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.

☐

24.1.2 Equivalence of Relational and Step-Indexed Evaluation

As with arithmetic and boolean expressions, we'd hope that the two alternative definitions of evaluation actually boil down to the same thing. This section shows that this is the case. Make sure you understand the statements of the theorems and can follow the structure of the proofs.

```
Theorem ceval_step__ceval: ∀ c st st',
      (∃ i, ceval_step st c i = Some st') →
      c / st || 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' ].
  Case "i = 0 – contradictory".
    intros c st st' H. inversion H.
  Case "i = S i'".
    intros c st st' H.
    com_cases (destruct c) SCase;
           simpl in H; inversion H; subst; clear H.
      SCase "SKIP". apply E_Skip.
      SCase "::=". apply E_Ass. reflexivity.
      SCase ";".
        remember (ceval_step st c1 i') as r1. destruct r1.
        SSCase "Evaluation of r1 terminates normally".
          apply E_Seq with s.
            apply IHi'. rewrite Heqr1. reflexivity.
            apply IHi'. simpl in H1. assumption.
        SSCase "Otherwise – contradiction".
          inversion H1.
      SCase "IFB".
        remember (beval st b) as r. destruct r.
        SSCase "r = true".
          apply E_IfTrue. rewrite Heqr. reflexivity.
          apply IHi'. assumption.
        SSCase "r = false".
          apply E_IfFalse. rewrite Heqr. reflexivity.
          apply IHi'. assumption.
```

```
SCase "WHILE". remember (beval st b) as r. destruct r.
  SSCase "r = true".
    remember (ceval_step st c i') as r1. destruct r1.
    SSSCase "r1 = Some s".
      apply E_WhileLoop with s. rewrite Heqr. reflexivity.
      apply IHi'. rewrite Heqr1. reflexivity.
      apply IHi'. simpl in H1. assumption.
    SSSCase "r1 = None".
      inversion H1.
  SSCase "r = false".
    inversion H1.
    apply E_WhileEnd.
    rewrite Heqr. 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: ∀ i1 i2 st st' c,
  i1 ≤ i2 →
  ceval_step st c i1 = Some st' →
  ceval_step st c i2 = Some st'.
Proof.
induction i1 as [|i1']; intros i2 st st' c Hle Hceval.
  Case "i1 = 0".
    simpl in Hceval. inversion Hceval.
  Case "i1 = S i1'".
    destruct i2 as [|i2']. inversion Hle.
    assert (Hle': i1' ≤ i2') by omega.
    com_cases (destruct c) SCase.
    SCase "SKIP".
      simpl in Hceval. inversion Hceval.
      reflexivity.
    SCase "::=".
      simpl in Hceval. inversion Hceval.
      reflexivity.
    SCase ";".
      simpl in Hceval. simpl.
      remember (ceval_step st c1 i1') as st1'o.
      destruct st1'o.
```

```
    SSCase "st1'o = Some".
      symmetry in Heqst1'o.
      apply (IHi1' i2') in Heqst1'o; try assumption.
      rewrite Heqst1'o. simpl. simpl in Hceval.
      apply (IHi1' i2') in Hceval; try assumption.
    SSCase "st1'o = None".
      inversion Hceval.

  SCase "IFB".
    simpl in Hceval. simpl.
    remember (beval st b) as bval.
    destruct bval; apply (IHi1' i2') in Hceval; assumption.

  SCase "WHILE".
    simpl in Hceval. simpl.
    destruct (beval st b); try assumption.
    remember (ceval_step st c i1') as st1'o.
    destruct st1'o.
    SSCase "st1'o = Some".
      symmetry in Heqst1'o.
      apply (IHi1' i2') in Heqst1'o; try assumption.
      rewrite → Heqst1'o. simpl. simpl in Hceval.
      apply (IHi1' i2') in Hceval; try assumption.
    SSCase "i1'o = None".
      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 ≤ and *plus*.

```
Theorem ceval__ceval_step: ∀ c st st',
      c / st || st' →
      ∃ i, ceval_step st c i = Some st'.
Proof.
  intros c st st' Hce.
  ceval_cases (induction Hce) Case.
   Admitted.
   □

Theorem ceval_and_ceval_step_coincide: ∀ c st st',
      c / st || st'
  ↔ ∃ i, ceval_step st c i = Some st'.
Proof.
  intros c st st'.
  split. apply ceval__ceval_step. apply ceval_step__ceval.
Qed.
```

24.1.3 Determinism of Evaluation (Simpler Proof)

Here's a slicker proof showing that the evaluation relation is deterministic, using the fact that the relational and step-indexed definition of evaluation are the same.

```
Theorem ceval_deterministic' : ∀ c st st1 st2,
     c / st || st1 →
     c / st || st2 →
     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.
```

Chapter 25

Library ImpParser

25.1 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 the abstract syntax trees defined by the datatypes *aexp*, *bexp*, and *com*. In this file 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.

This development is not intended to be understood in detail: 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 skip down to the Examples section at the very end to get the punchline.

25.2 Internals

```
Require Import SfLib.
Require Import Imp.

Require Import String.
Require Import Ascii.

Open Scope list_scope.
```

25.2.1 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" := (ble_nat 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.
```

25.2.2 Parsing

Options with Errors

```
Inductive optionE (X:Type) : Type :=
  | SomeE : X → optionE X
  | NoneE : string → optionE X.

Implicit Arguments SomeE [[X]].
Implicit Arguments NoneE [[X]].

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

```
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.

Fixpoint many {T} (p : parser T) (steps : nat) : parser (list T) :=
  many_helper p [] steps.

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.

Definition expect (t : token) : parser unit :=
  firstExpect t (fun xs ⇒ SomeE(tt, xs)).
```

A Recursive-Descent Parser for Imp

```
Definition parseIdentifier (symtable :string→nat) (xs : list token)
                         : optionE (id × 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.

Definition parseNumber (xs : list token) : optionE (nat × 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 × n + (nat_of_ascii d - nat_of_ascii "0"%char))
                (list_of_string x)
                0,
              xs')
    else
      NoneE "Expected number"
end.

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) <- parseIdentifier symtable xs ;;
          SomeE (AId 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.

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.

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) <==
         parseIdentifier 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.
```

25.3 Examples

Chapter 26

Library Imp

26.1 Imp: Simple Imperative Programs

In this chapter, we begin a new direction that will continue for the rest of the course. Up to now we've been mostly studying Coq itself, but from now on we'll mostly be using Coq to formalize other things.

Our first case study is a *simple imperative programming language* called Imp. 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*, the best-known logic for reasoning about imperative programs.

Sflib

A minor technical point: Instead of asking Coq to import our earlier definitions from chapter *Logic*, we import a small library called *Sflib.v*, containing just a few definitions and theorems from earlier chapters that we'll actually use in the rest of the course. This change should be nearly invisible, since most of what's missing from Sflib has identical definitions in the Coq standard library. The main reason for doing it is to tidy the global Coq environment so that, for example, it is easier to search for relevant theorems.

```
Require Export SfLib.
```

26.2 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.

26.2.1 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 file 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: aexp ::= nat | aexp '+' aexp | aexp '-' aexp | aexp '*' aexp

bexp ::= true | false | aexp '=' aexp | aexp '<=' aexp | bexp 'and' bexp | 'not' bexp

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 ×) unspecified. Some additional information – and human intelligence – would be required to turn this description into a formal definition (when implementing a compiler, for example).

  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 needed.

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.

26.2.2 Evaluation

Evaluating an arithmetic expression produces a number.

```
Fixpoint aeval (e : aexp) : nat :=
  match e 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 (e : bexp) : bool :=
  match e with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ beq_nat (aeval a1) (aeval a2)
  | BLe a1 a2 ⇒ ble_nat (aeval a1) (aeval a2)
  | BNot b1 ⇒ negb (beval b1)
  | BAnd b1 b2 ⇒ andb (beval b1) (beval b2)
  end.
```

26.2.3 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 (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.
```

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: ∀ e,
  aeval (optimize_0plus e) = aeval e.
Proof.
  intros e. induction e.
  Case "ANum". reflexivity.
  Case "APlus". destruct e1.
    SCase "e1 = ANum n". destruct n.
      SSCase "n = 0". simpl. apply IHe2.
      SSCase "n <> 0". simpl. rewrite IHe2. reflexivity.
    SCase "e1 = APlus e1_1 e1_2".
      simpl. simpl in IHe1. rewrite IHe1.
      rewrite IHe2. reflexivity.
    SCase "e1 = AMinus e1_1 e1_2".
      simpl. simpl in IHe1. rewrite IHe1.
      rewrite IHe2. reflexivity.
    SCase "e1 = AMult e1_1 e1_2".
      simpl. simpl in IHe1. rewrite IHe1.
      rewrite IHe2. reflexivity.
  Case "AMinus".
    simpl. rewrite IHe1. rewrite IHe2. reflexivity.
  Case "AMult".
    simpl. rewrite IHe1. rewrite IHe2. reflexivity. Qed.
```

26.3 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.

26.3.1 Tacticals

Tacticals is Coq's term for tactics that take other tactics as arguments – "higher-order tactics," if you will.

The `repeat` Tactical

The `repeat` tactical takes another tactic and keeps applying this tactic until the tactic fails. Here is an example showing that 100 is even using repeat.

```
Theorem ev100 : ev 100.
Proof.
  repeat (apply ev_SS).   apply ev_0.
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 (it repeats zero times).

```
Theorem ev100' : ev 100.
Proof.
  repeat (apply ev_0).   repeat (apply ev_SS). apply ev_0. Qed.
```

The `repeat` T tactic does not have any 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.

The `try` Tactical

A similar tactical is `try`: 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 : ∀ ae, aeval ae = aeval ae.
Proof. try reflexivity. Qed.
```

```
Theorem silly2 : ∀ (P : Prop), P → P.
```

```
Proof.
  intros P HP.
  try reflexivity.   apply HP. Qed.
```

Using `try` in a completely manual proof is a bit silly, but we'll see below that `try` is very useful for doing automated proofs in conjunction with the ; tactical.

The ; Tactical (Simple Form)

In its simplest and most commonly used form, the ; tactical takes 2 tactics as argument: *T*;*T'* first performs the tactic *T* and then performs the tactic *T'* on *each subgoal* generated by *T*.

For example, consider the following trivial lemma:

```
Lemma foo : ∀ n, ble_nat 0 n = true.
Proof.
  intros.
  destruct n.
    Case "n=0". simpl. reflexivity.
    Case "n=Sn'". simpl. reflexivity.
Qed.
```

We can simplify this proof using the ; tactical:

```
Lemma foo' : ∀ n, ble_nat 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': ∀ e,
  aeval (optimize_0plus e) = aeval e.
Proof.
  intros e.
  induction e;

    try (simpl; rewrite IHe1; rewrite IHe2; reflexivity).
  Case "ANum". reflexivity.
  Case "APlus".
    destruct e1;

      try (simpl; simpl in IHe1; rewrite IHe1;
           rewrite IHe2; reflexivity).
```

```
  SCase "e1 = ANum n". destruct n;
    simpl; rewrite IHe2; 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 e, aeval (optimize_0plus e) = aeval e. *Proof*: By induction on e. The *AMinus* and *AMult* cases follow directly from the IH. The remaining cases are as follows:

- Suppose e = *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 e = *APlus e1 e2* for some *e1* and *e2*. We must show aeval (optimize_0plus (APlus e1 e2)) = aeval (APlus e1 e2). Consider the possible forms of *e1*. For most of them, *optimize_0plus* simply calls itself recursively for the subexpressions and rebuilds a new expression of the same form as *e1*; in these cases, the result follows directly from the IH.

  The interesting case is when *e1* = *ANum n* for some n. If n = *ANum* 0, then optimize_0plus (APlus e1 e2) = optimize_0plus e2 and the IH for *e2* is exactly what we need. On the other hand, if n = *S n'* for some *n'*, then again *optimize_0plus* simply calls itself recursively, and the result follows from the IH. ☐

This proof can still be improved: the first case (for e = *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'': ∀ e,
  aeval (optimize_0plus e) = aeval e.
Proof.
  intros e.
  induction e;

    try (simpl; rewrite IHe1; rewrite IHe2; reflexivity);

    try reflexivity.
  Case "APlus".
    destruct e1; try (simpl; simpl in IHe1; rewrite IHe1;
                      rewrite IHe2; reflexivity).
    SCase "e1 = ANum n". destruct n;
      simpl; rewrite IHe2; reflexivity. Qed.
```

The ; Tactical (General Form)

The ; tactical has a more general than the simple *T*;*T'* we've seen above, and which is sometimes also useful. If *T*, *T1*, ..., *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 just a shorthand for: T; *T'* | *T'* | ... | *T'* The form *T*;*T'* is used most often in practice.

26.3.2 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 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* which 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`."

The next subsection gives a more sophisticated use of this feature...

Bulletproofing Case Analyses

Being able to deal with most of the cases of an `induction` or `destruct` all at the same time is very convenient, but it can also be a little confusing. One problem that often comes up is that *maintaining* proofs written in this style can be difficult. For example, suppose that, later, we extended the definition of *aexp* with another constructor that also required a special argument. The above proof might break because Coq generated the subgoals for this constructor before the one for *APlus*, so that, at the point when we start working on the *APlus* case, Coq is actually expecting the argument for a completely different constructor. What we'd like is to get a sensible error message saying "I was expecting the *AFoo* case at this point, but the proof script is talking about *APlus*." Here's a nice trick (due to Aaron Bohannon) that smoothly achieves this.

```
Tactic Notation "aexp_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ANum" | Case_aux c "APlus"
  | Case_aux c "AMinus" | Case_aux c "AMult" ].
```

(*Case_aux* implements the common functionality of *Case*, *SCase*, *SSCase*, etc. For example, *Case* "*foo*" is defined as *Case_aux Case* "*foo*".)

For example, if e is a variable of type *aexp*, then doing aexp_cases (induction e) Case will perform an induction on e (the same as if we had just typed `induction` e) and *also* add a *Case* tag to each subgoal generated by the `induction`, labeling which constructor it comes from. For example, here is yet another proof of *optimize_0plus_sound*, using *aexp_cases*:

```
Theorem optimize_0plus_sound''': ∀ e,
  aeval (optimize_0plus e) = aeval e.
Proof.
  intros e.
  aexp_cases (induction e) Case;
    try (simpl; rewrite IHe1; rewrite IHe2; reflexivity);
    try reflexivity.
  Case "APlus".
    aexp_cases (destruct e1) SCase;
      try (simpl; simpl in IHe1; rewrite IHe1; rewrite IHe2; reflexivity).
    SCase "ANum". destruct n;
      simpl; rewrite IHe2; reflexivity. Qed.
```

Exercise: 3 stars (optimize_0plus_b) Since the *optimize_0plus* tranformation doesn't change the value of *aexp*s, we should be able to apply it to all the *aexp*s that appear in a *bexp* without changing the *bexp*'s value. Write a function which performs that transformation on *bexp*s, and prove it is sound. Use the tacticals we've just seen to make the proof as elegant as possible.

□

Exercise: 4 stars, optional (optimizer) *Design exercise*: The optimization implemented by our *optimize_0plus* function is only one of many imaginable optimizations on arithmetic and boolean expressions. Write a more sophisticated optimizer and prove it correct.

□

26.3.3 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.

```
Example silly_presburger_example : ∀ m n o p,
  m + n ≤ n + o ∧ o + 3 = p + 3 →
  m ≤ p.
Proof.
  intros. omega.
Qed.
```

Andrew Appel calls this the "Santa Claus tactic." We'll see examples of its use below.

26.3.4 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.

26.4 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* || *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 vertical bar as the closest approximation in *.v* files.)

```
Notation "e '||' n" := (aevalR e n) : 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* || *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.

Tactic Notation "aevalR_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E_ANum" | Case_aux c "E_APlus"
  | Case_aux c "E_AMinus" | Case_aux c "E_AMult" ].
```

It is straightforward to prove that the relational and functional definitions of evaluation agree on all possible arithmetic expressions...

```
Theorem aeval_iff_aevalR : ∀ a n,
  (a || n) ↔ aeval a = n.
Proof.
 split.
 Case "->".
   intros H.
   aevalR_cases (induction H) SCase; simpl.
   SCase "E_ANum".
     reflexivity.
   SCase "E_APlus".
     rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
   SCase "E_AMinus".
     rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
   SCase "E_AMult".
     rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
 Case "<-".
   generalize dependent n.
   aexp_cases (induction a) SCase;
      simpl; intros; subst.
   SCase "ANum".
     apply E_ANum.
   SCase "APlus".
     apply E_APlus.
```

```
      apply IHa1. reflexivity.
      apply IHa2. reflexivity.
   SCase "AMinus".
     apply E_AMinus.
      apply IHa1. reflexivity.
      apply IHa2. reflexivity.
   SCase "AMult".
     apply E_AMult.
      apply IHa1. reflexivity.
      apply IHa2. reflexivity.
Qed.
```

We can make the proof quite a bit shorter by making more aggressive use of tacticals...

```
Theorem aeval_iff_aevalR' : ∀ a n,
  (a || n) ↔ aeval a = n.
Proof.
  split.
  Case "->".
    intros H; induction H; subst; reflexivity.
  Case "<-".
    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*.

☐

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 necessarily 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 computational mechanism.

However, there are circumstances where relational definitions of evaluation are greatly preferable to functional ones, as we'll see shortly.

26.4.1 Inference Rule Notation

In informal discussions, it is convenient 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. 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 || n1 e2 || n2

(E_APlus) APlus e1 e2 || n1+n2 Formally, there is nothing very 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 →. All the variables mentioned in the rule (*e1*, *n1*, etc.) are implicitly bound by universal quantifiers at the beginning. The whole collection of rules is understood as being wrapped in an `Inductive` declaration (informally, this is either elided or else indicated by saying something like "Let *aevalR* be the smallest relation closed under the following rules...").

For example, || is the smallest relation closed under these rules:

(E_ANum) ANum n || n

e1 || n1 e2 || n2

(E_APlus) APlus e1 e2 || n1+n2

e1 || n1 e2 || n2

(E_AMinus) AMinus e1 e2 || n1-n2

e1 || n1 e2 || n2

(E_AMult) AMult e1 e2 || n1*n2

`End` AExp.

26.5 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.

26.5.1 Identifiers

To begin, we'll need to formalize *identifiers* such as program variables. We could use strings for this – or, in a real compiler, fancier structures like pointers into a symbol table. But for simplicity let's just use natural numbers as identifiers.

(We hide this section in a module because these definitions are actually in *SfLib*, but we want to repeat them here so that we can explain them.)

`Module` Id.

We define a new inductive datatype *Id* so that we won't confuse identifiers and numbers.

```
Inductive id : Type :=
  Id : nat → id.
```

```
Definition beq_id X1 X2 :=
  match (X1, X2) with
    (Id n1, Id n2) ⇒ beq_nat n1 n2
  end.
```

After we "wrap" numbers as identifiers in this way, it is convenient to recapitulate a few properties of numbers as analogous properties of identifiers, so that we can work with identifiers in definitions and proofs abstractly, without unwrapping them to expose the underlying numbers. Since all we need to know about identifiers is whether they are the same or different, just a few basic facts are all we need.

```
Theorem beq_id_refl : ∀ X,
  true = beq_id X X.
Proof.
  intros. destruct X.
  apply beq_nat_refl. Qed.
```

Exercise: 1 star, optional (beq_id_eq) For this and the following exercises, do not use induction, but rather apply similar results already proved for natural numbers. Some of the tactics mentioned above may prove useful. Theorem beq_id_eq : ∀ i1 i2,

```
  true = beq_id i1 i2 → i1 = i2.
Proof.
   Admitted.
```

□

Exercise: 1 star, optional (beq_id_false_not_eq) Theorem beq_id_false_not_eq : ∀ i1 i2,

```
  beq_id i1 i2 = false → i1 ≠ i2.
Proof.
   Admitted.
```

□

Exercise: 1 star, optional (not_eq_beq_id_false) Theorem not_eq_beq_id_false : ∀ i1 i2,

```
  i1 ≠ i2 → beq_id i1 i2 = false.
Proof.
   Admitted.
```

□

Exercise: 1 star, optional (beq_id_sym) Theorem beq_id_sym: ∀ i1 i2,

```
  beq_id i1 i2 = beq_id i2 i1.
Proof.
   Admitted.
```

□

End Id.

26.5.2 States

A *state* represents the current values of all the variables at some point in the execution of a program. For simplicity (to avoid dealing with partial functions), we let the state be defined for *all* variables, even though any given program is only going to mention a finite number of them.

```
Definition state := id → nat.

Definition empty_state : state :=
  fun _ ⇒ 0.

Definition update (st : state) (X:id) (n : nat) : state :=
  fun X' ⇒ if beq_id X X' then n else st X'.
```

For proofs involving states, we'll need several simple properties of *update*.

Exercise: 1 star (update_eq) Theorem update_eq : ∀ n X st,

```
  (update st X n) X = n.
Proof.
  Admitted.
```

□

Exercise: 1 star (update_neq) Theorem update_neq : ∀ V2 V1 n st,

```
  beq_id V2 V1 = false →
  (update st V2 n) V1 = (st V1).
Proof.
  Admitted.
```

□

Exercise: 1 star (update_example) Before starting to play with tactics, make sure you understand exactly what the theorem is saying!

```
Theorem update_example : ∀ (n:nat),
  (update empty_state (Id 2) n) (Id 3) = 0.
Proof.
  Admitted.
```

□

Exercise: 1 star, recommended (update_shadow) Theorem update_shadow : ∀ x1 x2 k1 k2 (f : state),

```
   (update (update f k2 x1) k2 x2) k1 = (update f k2 x2) k1.
```

```
Proof.
  Admitted.
  □
```

Exercise: 2 stars (update_same)
```
Theorem update_same : ∀ x1 k1 k2 (f : state),
  f k1 = x1 →
  (update f k1 x1) k2 = f k2.
Proof.
  Admitted.
  □
```

Exercise: 3 stars (update_permute)
```
Theorem update_permute : ∀ x1 x2 k1 k2 k3 f,
  beq_id k2 k1 = false →
  (update (update f k2 x1) k1 x2) k3 = (update (update f k1 x2) k2 x1) k3.
Proof.
  Admitted.
  □
```

26.5.3 Syntax

We can add variables to the arithmetic expressions we had before by simply adding one more constructor:

```
Inductive aexp : Type :=
  | ANum : nat → aexp
  | AId : id → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp.
```

```
Tactic Notation "aexp_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ANum" | Case_aux c "AId" | Case_aux c "APlus"
  | Case_aux c "AMinus" | Case_aux c "AMult" ].
```

Defining a few variable names as notational shorthands will make examples easier to read:

```
Definition X : id := Id 0.
Definition Y : id := Id 1.
Definition Z : id := Id 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 the same as before (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.

Tactic Notation "bexp_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "BTrue" | Case_aux c "BFalse" | Case_aux c "BEq"
  | Case_aux c "BLe" | Case_aux c "BNot" | Case_aux c "BAnd" ].
```

26.5.4 Evaluation

The arith and boolean evaluators can be extended to handle variables in the obvious way:

```
Fixpoint aeval (st : state) (e : aexp) : nat :=
  match e with
  | ANum n ⇒ n
  | AId 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) (e : bexp) : bool :=
  match e with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ beq_nat (aeval st a1) (aeval st a2)
  | BLe a1 a2 ⇒ ble_nat (aeval st a1) (aeval st a2)
  | BNot b1 ⇒ negb (beval st b1)
  | BAnd b1 b2 ⇒ andb (beval st b1) (beval st b2)
  end.

Example aexp1 :
  aeval (update empty_state X 5)
        (APlus (ANum 3) (AMult (AId X) (ANum 2)))
  = 13.
Proof. reflexivity. Qed.

Example bexp1 :
  beval (update empty_state X 5)
        (BAnd BTrue (BNot (BLe (AId X) (ANum 4))))
  = true.
```

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

26.6 Commands

Now we are ready define the syntax and behavior of Imp *commands* (or *statements*).

26.6.1 Syntax

Informally, commands are described by the following BNF grammar: com ::= 'SKIP' | X '::=' aexp | com ';' com | 'WHILE' bexp 'DO' com 'END' | 'IFB' bexp 'THEN' com 'ELSE' com 'FI' For example, here's the factorial function 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 syntax of commands:

```
Inductive com : Type :=
  | CSkip : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.

Tactic Notation "com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "SKIP" | Case_aux c "::=" | Case_aux c ";"
  | Case_aux c "IFB" | Case_aux c "WHILE" ].
```

As usual, we can use a few `Notation` declarations to make things more readable. We need to be a bit careful to avoid conflicts with Coq's built-in notations, so we'll keep this light – in particular, we won't introduce any notations for *aexps* and *bexps* to avoid confusion with the numerical and boolean operators we've already defined. We use the keyword *IFB* for conditionals instead of *IF*, for similar reasons.

```
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'" :=
  (CIf e1 e2 e3) (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 ::= AId X;
  Y ::= ANum 1;
  WHILE BNot (BEq (AId Z) (ANum 0)) DO
    Y ::= AMult (AId Y) (AId Z);
    Z ::= AMinus (AId Z) (ANum 1)
  END.
```

26.6.2 Examples

Assignment:

```
Definition plus2 : com :=
  X ::= (APlus (AId X) (ANum 2)).

Definition XtimesYinZ : com :=
  Z ::= (AMult (AId X) (AId Y)).

Definition subtract_slowly_body : com :=
  Z ::= AMinus (AId Z) (ANum 1) ;
  X ::= AMinus (AId X) (ANum 1).
```

Loops:

```
Definition subtract_slowly : com :=
  WHILE BNot (BEq (AId 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.
```

Factorial again (broken up into smaller pieces this time, for convenience when we come back to proving things about it later).

```
Definition fact_body : com :=
  Y ::= AMult (AId Y) (AId Z) ;
  Z ::= AMinus (AId Z) (ANum 1).

Definition fact_loop : com :=
  WHILE BNot (BEq (AId Z) (ANum 0)) DO
```

```
    fact_body
  END.
```

```
Definition fact_com : com :=
  Z ::= AId X ;
  Y ::= ANum 1 ;
  fact_loop.
```

26.7 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 ...

26.7.1 Evaluation 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
    | l ::= a1 ⇒
        update st l (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 b1 DO c1 END ⇒
        st
  end.
```

In a traditional functional programming language like ML or Haskell we could write the WHILE case as follows:

```
Fixpoint ceval_fun (st : state) (c : com) : state :=
  match c with
    ...
    | WHILE b1 DO c1 END =>
        if (beval st b1)
          then ceval_fun st (c1; WHILE b1 DO c1 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, 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, which make everything much more complicated (see chapter *ImpCEvalFun* if curious).

26.7.2 Evaluation as a Relation

Here's a better way: we define *ceval* as a *relation* rather than a *function* – i.e., we define it in `Prop` instead of `Type`, as we did for *aevalR* and *bevalR* above.

This is an important change. Besides freeing us from the hacks that would be needed to define an evaluation function, it gives us a lot more flexibility in the definition. For example, if we added concurrency features to the language, we'd want the definition of evaluation to be non-deterministic – i.e., not only would it not be total, it would not even be a partial function!

We'll use the notation *c / st || st'* for our *ceval* relation, that is *c / st || 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'*".

(E_Skip) SKIP / st || st
aeval st a1 = n

(E_Ass) X := a1 / st || (update st X n)
c1 / st || st' c2 / st' || st"

(E_Seq) c1;c2 / st || st"
beval st b1 = true c1 / st || st'

(E_IfTrue) IF b1 THEN c1 ELSE c2 FI / st || st'
beval st b1 = false c2 / st || st'

(E_IfFalse) IF b1 THEN c1 ELSE c2 FI / st || st'
beval st b1 = false

(E_WhileEnd) WHILE b1 DO c1 END / st || st

beval st b1 = true c1 / st || st' WHILE b1 DO c1 END / st' || st"

(E_WhileLoop) WHILE b1 DO c1 END / st || st"

Here is the formal definition. (Make sure you understand how it corresponds to the inference rules.)

```
Reserved Notation "c1 '/' st '||' st'" (at level 40, st at level 39).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      SKIP / st || st
  | E_Ass : ∀ st a1 n X,
      aeval st a1 = n →
      (X ::= a1) / st || (update st X n)
  | E_Seq : ∀ c1 c2 st st' st'',
      c1 / st || st' →
      c2 / st' || st'' →
      (c1 ; c2) / st || st''
  | E_IfTrue : ∀ st st' b1 c1 c2,
      beval st b1 = true →
      c1 / st || st' →
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_IfFalse : ∀ st st' b1 c1 c2,
      beval st b1 = false →
      c2 / st || st' →
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_WhileEnd : ∀ b1 st c1,
      beval st b1 = false →
      (WHILE b1 DO c1 END) / st || st
  | E_WhileLoop : ∀ st st' st'' b1 c1,
      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').

Tactic Notation "ceval_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "E_Skip" | Case_aux c "E_Ass" | Case_aux c "E_Seq"
  | Case_aux c "E_IfTrue" | Case_aux c "E_IfFalse"
  | Case_aux c "E_WhileEnd" | Case_aux c "E_WhileLoop" ].
```

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 (AId X) (ANum 1)
       THEN Y ::= ANum 3
       ELSE Z ::= ANum 4
     FI)
   / empty_state
   || (update (update empty_state X 2) Z 4).
Proof.
  apply E_Seq with (update empty_state X 2).
  Case "assignment command".
    apply E_Ass. reflexivity.
  Case "if command".
    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 ||
    (update (update (update empty_state X 0) Y 1) Z 2).
Proof.
   Admitted.
```

☐

Exercise: 3 stars, optional (pup_to_n) Write an Imp program that sums the numbers from 1 to X (inclusive: $1 + 2 + ... + X$) in the variable Y. Prove that this program executes as intended for X = 2 (this latter part is trickier than you might expect).

```
Definition pup_to_n : com :=
  admit.

Theorem pup_to_2_ceval :
  pup_to_n / (update empty_state X 2) ||
    update (update (update (update (update (update empty_state
      X 2) Y 0) Y 2) X 1) Y 3) X 0.
Proof.
   Admitted.
```

☐

26.7.3 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 (imposed by Coq's restrictions on `Fixpoint` definitions) that evaluation should be a total function. But it also raises a question: Is the second definition of evaluation actually a partial function? That is, 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. Here's the proof:

```
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.
  ceval_cases (induction E1) Case;
           intros st2 E2; inversion E2; subst.
  Case "E_Skip". reflexivity.
  Case "E_Ass". reflexivity.
  Case "E_Seq".
    assert (st' = st'0) as EQ1.
      SCase "Proof of assertion". apply IHE1_1; assumption.
    subst st'0.
    apply IHE1_2. assumption.
  Case "E_IfTrue".
    SCase "b1 evaluates to true".
      apply IHE1. assumption.
    SCase "b1 evaluates to false (contradiction)".
      rewrite H in H5. inversion H5.
  Case "E_IfFalse".
    SCase "b1 evaluates to true (contradiction)".
      rewrite H in H5. inversion H5.
    SCase "b1 evaluates to false".
      apply IHE1. assumption.
  Case "E_WhileEnd".
    SCase "b1 evaluates to true".
      reflexivity.
    SCase "b1 evaluates to false (contradiction)".
      rewrite H in H2. inversion H2.
  Case "E_WhileLoop".
    SCase "b1 evaluates to true (contradiction)".
      rewrite H in H4. inversion H4.
```

```
  SCase "b1 evaluates to false".
    assert (st' = st'0) as EQ1.
      SSCase "Proof of assertion". apply IHE1_1; assumption.
    subst st'0.
    apply IHE1_2. assumption. Qed.
```

26.8 Reasoning About 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.

26.8.1 Basic Examples

```
Theorem plus2_spec : ∀ st n st',
  st X = n →
  plus2 / st || st' →
  st' X = n + 2.
Proof.
  intros st n st' HX Heval.
  inversion Heval. subst. clear Heval. simpl.
  apply 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 : ∀ st st',

```
  ~(loop / st || st').
Proof.
  intros st st' contra. unfold loop in contra.
  remember (WHILE BTrue DO SKIP END) as loopdef.
   Admitted.
```

□

```
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.
```

Exercise: 3 stars, optional (no_whilesR) The *no_whiles* property yields *true* on just those 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 → Prop :=

  .

Theorem no_whiles_eqv:
   ∀ c, no_whiles c = true ↔ no_whilesR c.
Proof.
   Admitted.
```

□

Exercise: 4 stars, optional (no_whiles_terminating) Imp programs that don't involve while loops always terminate. State and prove a theorem that says this. (Use either *no_whiles* or *no_whilesR*, as you prefer.)

□

26.8.2 Proving a Program Correct (Optional)

Recall the factorial program:

`Print` *fact_body*. `Print` *fact_loop*. `Print` *fact_com*.

Here is an alternative "mathematical" definition of the factorial function:

```
Fixpoint real_fact (n:nat) : nat :=
  match n with
  | O ⇒ 1
  | S n' ⇒ n × (real_fact n')
  end.
```

We would like to show that they agree – if we start *fact_com* in a state where variable X contains some number x, then it will terminate in a state where variable Y contains the factorial of x.

To show this, we rely on the critical idea of a *loop invariant*.

```
Definition fact_invariant (x:nat) (st:state) :=
  (st Y) × (real_fact (st Z)) = real_fact x.

Theorem fact_body_preserves_invariant: ∀ st st' x,
      fact_invariant x st →
      st Z ≠ 0 →
```

```
     fact_body / st || st' →
     fact_invariant x st'.
Proof.
  unfold fact_invariant, fact_body.
  intros st st' x Hm HZnz He.
  inversion He; subst; clear He.
  inversion H1; subst; clear H1.
  inversion H4; subst; clear H4.
  unfold update. simpl.
  destruct (st Z) as [| z'].
    apply ex_falso_quodlibet. apply HZnz. reflexivity.
  rewrite ← Hm. rewrite ← mult_assoc.
  replace (S z' - 1) with z' by omega.
  reflexivity. Qed.

Theorem fact_loop_preserves_invariant : ∀ st st' x,
     fact_invariant x st →
     fact_loop / st || st' →
     fact_invariant x st'.
Proof.
  intros st st' x H Hce.
  remember fact_loop as c.
  ceval_cases (induction Hce) Case;
              inversion Heqc; subst; clear Heqc.
  Case "E_WhileEnd".
    assumption.
  Case "E_WhileLoop".
    apply IHHce2.
      apply fact_body_preserves_invariant with st;
            try assumption.
      intros Contra. simpl in H0; subst.
      rewrite Contra in H0. inversion H0.
      reflexivity. Qed.

Theorem guard_false_after_loop: ∀ b c st st',
     (WHILE b DO c END) / st || st' →
     beval st' b = false.
Proof.
  intros b c st st' Hce.
  remember (WHILE b DO c END) as cloop.
  ceval_cases (induction Hce) Case;
     inversion Heqcloop; subst; clear Heqcloop.
  Case "E_WhileEnd".
    assumption.
```

```
  Case "E_WhileLoop".
    apply IHHce2. reflexivity. Qed.
```

Patching it all together...

```
Theorem fact_com_correct : ∀ st st' x,
      st X = x →
      fact_com / st || st' →
      st' Y = real_fact x.
Proof.
  intros st st' x HX Hce.
  inversion Hce; subst; clear Hce.
  inversion H1; subst; clear H1.
  inversion H4; subst; clear H4.
  inversion H1; subst; clear H1.
  rename st' into st''. simpl in H5.
  remember (update (update st Z (st X)) Y 1) as st'.
  assert (fact_invariant (st X) st').
    subst. unfold fact_invariant, update. simpl. omega.
  assert (fact_invariant (st X) st'').
    apply fact_loop_preserves_invariant with st'; assumption.
  unfold fact_invariant in H0.
  apply guard_false_after_loop in H5. simpl in H5.
  destruct (st'' Z).
  Case "st'' Z = 0". simpl in H0. omega.
  Case "st'' Z > 0 (impossible)". inversion H5.
Qed.
```

One might wonder whether all this work with poking at states and unfolding definitions could be ameliorated with some more powerful lemmas and/or more uniform reasoning principles... Indeed, this is exactly the topic of the upcoming *Hoare* chapter!

Exercise: 4 stars, optional (subtract_slowly_spec) Prove a specification for subtract_slowly, using the above specification of *fact_com* and the invariant below as guides.

```
Definition ss_invariant (x:nat) (z:nat) (st:state) :=
  minus (st Z) (st X) = minus z x.
```

☐

26.9 Additional Exercises

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.)

☐

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, recommended (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:

```
[]            |    2 3 * 3 4 2 - * +
[2]           |    3 * 3 4 2 - * +
[3, 2]        |    * 3 4 2 - * +
[6]           |    3 4 2 - * +
[3, 6]        |    4 2 - * +
[4, 3, 6]     |    2 - * +
[2, 4, 3, 6]  |    - * +
[2, 3, 6]     |    * +
[6, 6]        |    +
[12]          |
```

The task of this exercise is to write a small compiler that translates *aexp*s into stack machine instructions, and to prove its correctness.

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 takes 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 returns 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, since our compiler will never emit such a malformed program. However, when you do the correctness proof you may find some choices makes the proof easier than others.

```
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 (update empty_state X 3) [3,4]
       [SPush 4, SLoad X, SMult, SPlus]
   = [15, 4].
   Admitted.
```

Next, write a function which 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.
```

Finally, prove the following theorem, stating that the *compile* function behaves correctly. You will need to start by stating a more general lemma to get a usable induction hypothesis.

```
Theorem s_compile_correct : ∀ (st : state) (e : aexp),
  s_execute st [] (s_compile e) = [ aeval st e ].
Proof.
  Admitted.
```

□

Chapter 27

Library SfLib

27.1 SfLib: Software Foundations Library

Here we collect together several useful definitions and theorems from Basics.v, List.v, Poly.v, Ind.v, and Logic.v that are not already in the Coq standard library. From now on we can `Import` or `Export` this file, instead of cluttering our environment with all the examples and false starts in those files.

27.2 From the Coq Standard Library

```
Require Omega. Require Export Bool.
Require Export List.
Require Export Arith.
Require Export Arith.EqNat.
```

27.3 From Basics.v

```
Definition admit {T: Type} : T. Admitted.

Require String. Open Scope string_scope.

Ltac move_to_top x :=
  match reverse goal with
  | H : _ ⊢ _ ⇒ try move x after H
  end.

Tactic Notation "assert_eq" ident(x) constr(v) :=
  let H := fresh in
  assert (x = v) as H by reflexivity;
  clear H.
```

```
Tactic Notation "Case_aux" ident(x) constr(name) :=
  first [
    set (x := name); move_to_top x
  | assert_eq x name; move_to_top x
  | fail 1 "because we are working on a different case" ].
Tactic Notation "Case" constr(name) := Case_aux Case name.
Tactic Notation "SCase" constr(name) := Case_aux SCase name.
Tactic Notation "SSCase" constr(name) := Case_aux SSCase name.
Tactic Notation "SSSCase" constr(name) := Case_aux SSSCase name.
Tactic Notation "SSSSCase" constr(name) := Case_aux SSSSCase name.
Tactic Notation "SSSSSCase" constr(name) := Case_aux SSSSSCase name.
Tactic Notation "SSSSSSCase" constr(name) := Case_aux SSSSSSCase name.
Tactic Notation "SSSSSSSCase" constr(name) := Case_aux SSSSSSSCase name.
Fixpoint ble_nat (n m : nat) : bool :=
  match n with
  | O ⇒ true
  | S n' ⇒
      match m with
      | O ⇒ false
      | S m' ⇒ ble_nat n' m'
      end
  end.
Theorem andb_true_elim1 : ∀ b c,
  andb b c = true → b = true.
Proof.
  intros b c H.
  destruct b.
  Case "b = true".
    reflexivity.
  Case "b = false".
    rewrite ← H. reflexivity. Qed.
Theorem andb_true_elim2 : ∀ b c,
  andb b c = true → c = true.
Proof.
Admitted.
Theorem beq_nat_sym : ∀ (n m : nat),
  beq_nat n m = beq_nat m n.
Admitted.
Notation "[ ]" := nil.
Notation "[ x , .. , y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y" := (app x y)
```

(at level 60, right associativity).

27.4 From Props.v

```
Inductive ev : nat → Prop :=
  | ev_0 : ev O
  | ev_SS : ∀ n:nat, ev n → ev (S (S n)).
```

27.5 From Logic.v

```
Theorem andb_true : ∀ b c,
  andb b c = true → b = true ∧ c = true.
Proof.
  intros b c H.
  destruct b.
    destruct c.
      apply conj. reflexivity. reflexivity.
      inversion H.
    inversion H. Qed.

Theorem not_eq_beq_false : ∀ n n' : nat,
     n ≠ n' →
     beq_nat n n' = false.
Proof.
Admitted.

Theorem ex_falso_quodlibet : ∀ (P:Prop),
  False → P.
Proof.
  intros P contra.
  inversion contra. Qed.

Theorem ev_not_ev_S : ∀ n,
  ev n → ¬ ev (S n).
Proof.
Admitted.

Theorem ble_nat_true : ∀ n m,
  ble_nat n m = true → n ≤ m.
Admitted.

Theorem ble_nat_false : ∀ n m,
  ble_nat n m = false → ~(n ≤ m).
Admitted.
```

```
Inductive appears_in (n : nat) : list nat → Prop :=
| ai_here : ∀ l, appears_in n (n::l)
| ai_later : ∀ m l, appears_in n l → appears_in n (m::l).

Inductive next_nat (n:nat) : nat → Prop :=
  | nn : next_nat n (S n).

Inductive total_relation : nat → nat → Prop :=
  tot : ∀ n m : nat, total_relation n m.

Inductive empty_relation : nat → nat → Prop := .
```

27.6 From Later Files

```
Definition relation (X:Type) := X → X → Prop.

Definition deterministic {X: Type} (R: relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.

Inductive multi (X:Type) (R: relation X)
                            : X → X → Prop :=
  | multi_refl : ∀ (x : X),
                 multi X R x x
  | multi_step : ∀ (x y z : X),
                    R x y →
                    multi X R y z →
                    multi X R x z.
Implicit Arguments multi [[X]].

Tactic Notation "multi_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "multi_refl" | Case_aux c "multi_step" ].

Theorem multi_R : ∀ (X:Type) (R:relation X) (x y : X),
       R x y → multi R x y.
Proof.
  intros X R x y r.
  apply multi_step with y. apply r. apply multi_refl. Qed.

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.
   Admitted.

Inductive id : Type :=
```

```
  Id : nat → id.
Definition beq_id id1 id2 :=
  match (id1 , id2) with
    (Id n1 , Id n2) ⇒ beq_nat n1 n2
  end.
Theorem beq_id_refl : ∀ i,
  true = beq_id i i.
Proof.
  intros. destruct i.
  apply beq_nat_refl. Qed.
Theorem beq_id_eq : ∀ i1 i2,
  true = beq_id i1 i2 → i1 = i2.
Proof.
  intros i1 i2 H.
  destruct i1. destruct i2.
  apply beq_nat_eq in H. subst.
  reflexivity. Qed.
Theorem beq_id_false_not_eq : ∀ i1 i2,
  beq_id i1 i2 = false → i1 ≠ i2.
Proof.
  intros i1 i2 H.
  destruct i1. destruct i2.
  apply beq_nat_false in H.
  intros C. apply H. inversion C. reflexivity. Qed.
Theorem not_eq_beq_id_false : ∀ i1 i2,
  i1 ≠ i2 → beq_id i1 i2 = false.
Proof.
  intros i1 i2 H.
  destruct i1. destruct i2.
  assert (n ≠ n0).
    intros C. subst. apply H. reflexivity.
  apply not_eq_beq_false. assumption. Qed.
Theorem beq_id_sym: ∀ i1 i2,
  beq_id i1 i2 = beq_id i2 i1.
Proof.
  intros i1 i2. destruct i1. destruct i2. apply beq_nat_sym. Qed.
Definition partial_map (A:Type) := id → option A.
Definition empty {A:Type} : partial_map A := (fun _ ⇒ None).
Definition extend {A:Type} (Gamma : partial_map A) (x:id) (T : A) :=
  fun x' ⇒ if beq_id x x' then Some T else Gamma x'.
```

```
Lemma extend_eq : ∀ A (ctxt: partial_map A) x T,
  (extend ctxt x T) x = Some T.
Proof.
  intros. unfold extend. rewrite ← beq_id_refl. auto.
Qed.

Lemma extend_neq : ∀ A (ctxt: partial_map A) x1 T x2,
  beq_id x2 x1 = false →
  (extend ctxt x2 T) x1 = ctxt x1.
Proof.
  intros. unfold extend. rewrite H. auto.
Qed.

Lemma extend_shadow : ∀ A (ctxt: partial_map A) t1 t2 x1 x2,
  extend (extend ctxt x2 t1) x2 t2 x1 = extend ctxt x2 t2 x1.
Proof with auto.
  intros. unfold extend. destruct (beq_id x2 x1)...
Qed.
```

27.7 Some useful tactics

```
Tactic Notation "solve_by_inversion_step" tactic(t) :=
  match goal with
  | H : _ ⊢ _ ⇒ 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.
```

Chapter 28

Library Prop

28.1 Prop: Propositions and Evidence

```
Require Export Poly.
```

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 ($P \to Q$), and with quantified propositions ($\forall\ x,\ P$).

In this chapter we take a deeper look at the way propositions are expressed in Coq and at the structure of the logical evidence that we construct when we carry out proofs.

Some of the concepts in this chapter may seem a bit abstract on a first encounter. We've included a *lot* of exercises, most of which should be quite approachable even if you're still working on understanding the details of the text. Try to work as many of them as you can, especially the one-starred exercises.

28.2 Inductively Defined Propositions

As a running example for the first part of the chapter, let's consider a simple property of natural numbers, and let's say that the numbers possessing this property are "beautiful."

Informally, a number is *beautiful* if it is 0, 3, 5, or the sum of two beautiful numbers. More pedantically, we can define beautiful numbers by giving four rules:

- Rule *b_0*: The number 0 is beautiful.
- Rule *b_3*: The number 3 is beautiful.
- Rule *b_5*: The number 5 is beautiful.
- Rule *b_sum*: If n and m are both beautiful, then so is their sum.

We will see many definitions like this one during the rest of the course, and 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:

(b_0) beautiful 0

(b_3) beautiful 3

(b_5) beautiful 5
beautiful n beautiful m

(b_sum) beautiful (n+m)
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 *b_sum* says that, if n and m are both beautiful numbers, then it follows that $n+m$ is beautiful too. The rules with no premises above the line are called *axioms*.

These rules *define* the property *beautiful*. That is, if we want to convince someone that some particular number is beautiful, our argument must be based on these rules. For a simple example, suppose we claim that the number 5 is beautiful. To support this claim, we just need to point out that rule *b_5* says it is. Or, if we want to claim that 8 is beautiful, we can support our claim by first observing that 3 and 5 are both beautiful (by rules *b_3* and *b_5*) and then pointing out that their sum, 8, is therefore beautiful by rule *b_sum*. This argument can be expressed graphically with the following *proof tree*:

(b_3) ——— (b_5) beautiful 3 beautiful 5

(b_sum) beautiful 8 Of course, there are other ways of using these rules to argue that 8 is beautiful – for instance:

(b_5) ——— (b_3) beautiful 5 beautiful 3

(b_sum) beautiful 8

Exercise: 1 star (varieties_of_beauty) How many different ways are there to show that 8 is beautiful?

☐

In Coq, we can express the definition of *beautiful* as follows:

```
Inductive beautiful : nat → Prop :=
  b_0 : beautiful 0
| b_3 : beautiful 3
| b_5 : beautiful 5
```

| b_sum : ∀ n m, **beautiful** $n \to$ **beautiful** $m \to$ **beautiful** $(n+m)$.

The first line declares that *beautiful* is a proposition – or, more formally, a family of propositions "indexed by" natural numbers. (For each number n, the claim that "n is *beautiful*" is a proposition.) Such a family of propositions is often called a *property* of numbers.

Each of the remaining lines embodies one of the rules for beautiful numbers.

We can use Coq's tactic scripting facility to assemble proofs that particular numbers are beautiful.

```
Theorem three_is_beautiful: beautiful 3.
Proof.
  apply b_3.
Qed.
```

```
Theorem eight_is_beautiful: beautiful 8.
Proof.
  apply b_sum with (n:=3) (m:=5).
  apply b_3.
  apply b_5.
Qed.
```

28.3 Proof Objects

Look again at the formal definition of the *beautiful* property. The opening keyword, `Inductive`, has been used up to this point to declare new types of *data*, such as numbers and lists. Does this interpretation also make sense for the Inductive definition of *beautiful*? That is, can we view evidence of beauty as some kind of data structure? Yes, we can!

The trick is to introduce an alternative pronunciation of ":". Instead of "has type," we can also say "is a proof of." For example, the second line in the definition of *beautiful* declares that *b_0* : *beautiful* 0. Instead of "*b_0* has type *beautiful* 0," we can say that "*b_0* is a proof of *beautiful* 0." Similarly for *b_3* and *b_5*.

This pun between ":" as "has type" and : as "is a proof of" is called the *Curry-Howard correspondence* (or sometimes *Curry-Howard isomorphism*). It proposes a deep connection between the world of logic and the world of computation.

```
propositions  ~  types
evidence      ~  data
```

Many useful things follow from this connection. To begin with, it gives us a natural interpretation of the *b_sum* constructor:

b_sum : forall n m, beautiful n -> beautiful m -> beautiful (n+m). If we read : as "has type," this says that *b_sum* is a data constructor that takes four arguments: two numbers, n and m, and two values of type *beautiful* n and *beautiful* m. That is, *b_sum* can be viewed as a *function* that, given evidence for the propositions *beautiful* n and *beautiful* m, gives us evidence for the proposition that *beautiful* $(n+m)$.

In view of this, we might wonder whether we can write an expression of type *beautiful* 8 by applying *b_sum* to appropriate arguments. Indeed, we can:

```
Check (b_sum 3 5 b_3 b_5).
```

The expression *b_sum* 3 5 *b_3 b_5* can be thought of as instantiating the parameterized constructor *b_sum* with the specific arguments 3 5 and the corresponding proof objects for its premises *beautiful* 3 and *beautiful* 5 (Coq is smart enough to figure out that 3+5=8). Alternatively, we can think of *b_sum* as a primitive "evidence constructor" that, when applied to two particular numbers, wants to be further applied to evidence that those two numbers are beautiful; its type, [$\forall$ *n m*, *beautiful n* $\rightarrow$ *beautiful m* $\rightarrow$ *beautiful* (*n*+*m*), *expresses this functionality*, `in` *the same way that the polymorphic type* [$\forall$ *X*, *list X*] `in` *the previous chapter expressed the fact that the* `constructor` [*nil*] *can be thought of* `as` *a function from types to empty lists* `with` *elements of that type.*

This gives us an alternative way to write the proof that 8 is beautiful:

```
Theorem eight_is_beautiful': beautiful 8.
Proof.
  apply (b_sum 3 5 b_3 b_5).
Qed.
```

Notice that we're using `apply` here in a new way: instead of just supplying the *name* of a hypothesis or previously proved theorem whose type matches the current goal, we are supplying an *expression* that directly builds evidence with the required type.

28.3.1 Proof Scripts and Proof Objects

These proof objects 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 the `Proof` command and the `Qed` instruct Coq 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 eight_is_beautiful'': beautiful 8.
Proof.
  apply b_sum with (n:=3) (m:=5).
  Show Proof.
  apply b_3.
  Show Proof.
  apply b_5.
  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. In the `Show Proof` output, lines of the form ?1 $\rightarrow$ *beautiful n* record these requirements. (The $\rightarrow$ here has nothing to do with either implication or function types – it is just an unfortunate choice of concrete syntax for the output!)

Each of the holes corresponds to a subgoal, and the proof is finished when there are no more subgoals. At this point, the `Theorem` command gives a name to the evidence we've built and stores it in the global context.

Tactic proofs are useful and convenient because they avoid building proof trees by hand, but they are not essential: in principle, we can always construct the required evidence by hand. Indeed, we don't even need the `Theorem` command: we can use `Definition` instead, to directly give a global name to a piece of evidence.

```
Definition eight_is_beautiful''' : beautiful 8 :=
  b_sum 3 5 b_3 b_5.
```

All these different ways of building the proof lead to exactly the same evidence being saved in the global environment.

```
Print eight_is_beautiful.
Print eight_is_beautiful'.
Print eight_is_beautiful''.
Print eight_is_beautiful'''.
```

Exercise: 1 star (six_is_beautiful) Give a tactic proof and a proof object showing that 6 is *beautiful*.

```
Theorem six_is_beautiful :
  beautiful 6.
Proof.
   Admitted.
Definition six_is_beautiful' : beautiful 6 :=
  admit.
```

☐

Exercise: 1 star (nine_is_beautiful) Give a tactic proof and a proof object showing that 9 is *beautiful*.

```
Theorem nine_is_beautiful :
  beautiful 9.
Proof.
   Admitted.
Definition nine_is_beautiful' : beautiful 9 :=
  admit.
```

☐

28.3.2 Implications and Functions

If we want to substantiate the claim that $P \to Q$, what sort of proof object should count as evidence?

We've seen one case above: the *b_sum* constructor, which is *primitive* evidence for an implication proposition – it is part of the very meaning of the word "beautiful" in this context. But what about other implications that we might want to prove?

For example, consider this statement:

```
Theorem b_plus3: ∀ n, beautiful n → beautiful (3+n).
Proof.
  intros n H.
  apply b_sum.
  apply b_3.
  apply H.
Qed.
```

What is the proof object corresponding to *b_plus3*?

We've made a notational pun between $\to$ as implication and $\to$ as the type of functions. If we take this pun seriously, then what we're looking for is an expression whose *type* is $\forall$ n, *beautiful* n $\to$ *beautiful* $(3+n)$ – 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 b_plus3' : ∀ n, beautiful n → beautiful (3+n) :=
  fun n ⇒ fun H : beautiful n ⇒
    b_sum 3 n b_3 H.
Check b_plus3'.
```

Recall that `fun n ⇒ blah` means "the function that, given n, yields *blah*." Another equivalent way to write this definition is:

```
Definition b_plus3'' (n : nat) (H : beautiful n) : beautiful (3+n) :=
  b_sum 3 n b_3 H.
Check b_plus3''.
```

Exercise: 2 stars (b_times2) `Theorem b_times2: ∀ n, beautiful n → beautiful (2×n).`

```
Proof.
  Admitted.
```

☐

Exercise: 3 stars, optional (b_times2') Write a proof object corresponding to *b_times2* above

```
Definition b_times2': ∀ n, beautiful n → beautiful (2×n) :=
  admit.
```

Exercise: 2 stars (b_timesm) `Theorem` b_timesm: ∀ n m, **beautiful** n → **beautiful** (m×n).
`Proof`.
  Admitted.
  □

28.3.3 Induction Over Proof Objects

Since we use the keyword `Induction` to define primitive propositions together with their evidence, we might wonder whether there are some sort of induction principles associated with these definitions. Indeed there are, and in this section we'll take a look at how they can be used.

Besides *constructing* evidence that numbers are beautiful, we can also *reason about* such evidence. The fact that we introduced *beautiful* with an `Inductive` declaration tells us not only that the constructors *b_0*, *b_3*, *b_5* and *b_sum* are ways to build evidence, but also that these two constructors are the *only* ways to build evidence that numbers are beautiful.

In other words, if someone gives us evidence E justifying the assertion *beautiful* n, then we know that E can only have one of four forms: either E is *b_0* (and n is O) or E is *b_3* (and n is 3), or E is *b_5* (and n is 5), or E is *b_sum n1 n2 E1 E2* (and n is (*n1*+*n2*), and *E1* is evidence that *n1* is beauiful and *E2* is evidence that *n2* is beautiful).

Thus, it makes sense to use the tactics that we have already seen for inductively defined *data* to reason instead about inductively defined *evidence*.

Let's introduce a new property of numbers to help illustrate the role of induction.

```
Inductive gorgeous : nat → Prop :=
  g_0 : gorgeous 0
| g_plus3 : ∀ n, gorgeous n → gorgeous (3+n)
| g_plus5 : ∀ n, gorgeous n → gorgeous (5+n).
```

Exercise: 1 star (gorgeous_tree) Write out the definition of gorgeous numbers using the *inference rule* notation.
  □

It seems intuitively obvious that, although *gorgeous* and *beautiful* are presented using slightly different rules, they are actually the same property in the sense that they are true of the same numbers. Indeed, we can prove this.

```
Theorem gorgeous__beautiful : ∀ n,
  gorgeous n → beautiful n.
Proof.
  intros.
  induction H as [|n'|n'].
  Case "g_0".
    apply b_0.
  Case "g_plus3".
```

```
        apply b_sum. apply b_3.
        apply IHgorgeous.
    Case "g_plus5".
        apply b_sum. apply b_5. apply IHgorgeous.
Qed.
```

Let's see what happens if we try to prove this by induction on n instead of induction on the evidence H.

```
Theorem gorgeous__beautiful_FAILED : ∀ n,
  gorgeous n → beautiful n.
Proof.
    intros. induction n as [| n'].
    Case "n = 0". apply b_0.
    Case "n = S n'". Admitted.
```

Exercise: 1 star (gorgeous_plus13) Theorem gorgeous_plus13: ∀ n,

```
  gorgeous n → gorgeous (13+n).
Proof.
    Admitted.
```

☐

Exercise: 2 stars (gorgeous_plus13_po): Give the proof object for theorem *gorgeous_plus13* above.

```
Definition gorgeous_plus13_po: ∀ n, gorgeous n → gorgeous (13+n):=
    admit.
```

☐

Exercise: 2 stars (gorgeous_sum) Theorem gorgeous_sum : ∀ n m,

```
  gorgeous n → gorgeous m → gorgeous (n + m).
Proof.
    Admitted.
```

☐

Exercise: 3 stars (beautiful__gorgeous) Theorem beautiful__gorgeous : ∀ n, **beautiful** n → **gorgeous** n.

```
Proof.
    Admitted.
```

☐

Exercise: 3 stars, optional (b_times2) Prove the *g_times2* theorem below without using *gorgeous__beautiful.* You might find the following helper lemma useful.

```
Lemma helper_g_times2 : ∀ x y z, x + (z + y)= z + x + y.
```

```
Proof.
    Admitted.
```

```
Theorem g_times2: ∀ n, gorgeous n → gorgeous (2×n).
Proof.
    intros n H. simpl.
    induction H.
    Admitted.
```

□

28.3.4 Evenness

In chapter *Basics* we defined a *function evenb* that tests a number for evenness, yielding *true* if so. This gives us an obvious way of defining the *concept* of evenness:

```
Definition even (n:nat) : Prop :=
  evenb n = true.
```

That is, we can define "n is even" to mean "the function *evenb* returns *true* when applied to n."

Another alternative is to define the concept of evenness directly. Instead of going via the *evenb* function ("a number is even if a certain computation yields *true*"), we can say what the concept of evenness means by giving two different ways of presenting *evidence* that a number is even.

```
Inductive ev : nat → Prop :=
  | ev_0 : ev O
  | ev_SS : ∀ n:nat, ev n → ev (S (S n)).
```

This definition says that there are two ways to give evidence that a number m is even. First, 0 is even, and *ev_0* is evidence for this. Second, if $m = S\ (S\ n)$ for some n and we can give evidence e that n is even, then m is also even, and *ev_SS* n e is the evidence.

Exercise: 1 star (double_even) Construct a tactic proof of the following proposition.

```
Theorem double_even : ∀ n,
  ev (double n).
Proof.
    Admitted.
```

□

Exercise: 4 stars, optional (double_even_pfobj) Try to predict what proof object is constructed by the above tactic proof. (Before checking your answer, you'll want to strip out any uses of *Case*, as these will make the proof object look a bit cluttered.) □

Discussion: Computational vs. Inductive Definitions

We have seen that the proposition "some number is even" can be phrased in two different ways – indirectly, via a boolean testing function *evenb*, or directly, by inductively describing what constitutes evidence for evenness. These two ways of defining evenness are about equally easy to state and work with. Which we choose is basically a question of taste.

However, for many other properties of interest, the direct inductive definition is preferable, since writing a testing function may be awkward or even impossible.

One such property is *beautiful*. This is a perfectly sensible definition of a set of numbers, but we cannot translate its definition directly as a Coq Fixpoint (or translate it directly into a recursive function in any other programming language). We might be able to find a clever way of testing this property using a `Fixpoint` (indeed, it is not too hard to find one in this case), but in general this could require arbitrarily deep thinking. In fact, if the property we are interested in is uncomputable, then we cannot define it as a `Fixpoint` no matter how hard we try, because Coq requires that all `Fixpoint`s correspond to terminating computations.

On the other hand, writing an inductive definition of what it means to give evidence for the property *beautiful* is straightforward.

28.3.5 Inverting Evidence

Besides induction, we can use the other tactics in our toolkit with evidence. For example, this proof uses `destruct` on evidence.

```
Theorem ev_minus2: ∀ n,
  ev n → ev (pred (pred n)).
Proof.
  intros n E.
  destruct E as [| n' E'].
  Case "E = ev_0". simpl. apply ev_0.
  Case "E = ev_SS n' E'". simpl. apply E'. Qed.
```

Exercise: 1 star, optional (ev_minus2_n) What happens if we try to `destruct` on n instead of E? □

Exercise: 1 star, recommended (ev__even) Here is a proof that the inductive definition of evenness implies the computational one.

```
Theorem ev__even : ∀ n,
  ev n → even n.
Proof.
  intros n E. induction E as [| n' E'].
  Case "E = ev_0".
    unfold even. reflexivity.
```

Case "E = ev_SS n' E'".
  unfold even. apply *IHE'*.
Qed.

Could this proof also be carried out by induction on n instead of E? If not, why not?

☐

The induction principle for inductively defined propositions does not follow quite the same form as that of inductively defined sets. For now, you can take the intuitive view that induction on evidence *ev n* is similar to induction on n, but restricts our attention to only those numbers for which evidence *ev n* could be generated. We'll look at the induction principle of *ev* in more depth below, to explain what's really going on.

Exercise: 1 star (l_fails) The following proof attempt will not succeed. Theorem l : forall n, ev n. Proof. intros n. induction n. Case "O". simpl. apply ev_0. Case "S". ... Briefly explain why.

☐

Exercise: 2 stars (ev_sum) Here's another exercise requiring induction.

```
Theorem ev_sum : ∀ n m,
   ev n → ev m → ev (n+m).
Proof.
   Admitted.
```

☐

Here's another situation where we want to analyze evidence for evenness: proving that if $n+2$ is even, then n is. Our first idea might be to use `destruct` for this kind of case analysis:

```
Theorem SSev_ev_firsttry : ∀ n,
  ev (S (S n)) → ev n.
Proof.
  intros n E.
  destruct E as [| n' E'].
Admitted.
```

In the first sub-goal, we've lost the information that n is 0. We could have used *remember*, but then we still need `inversion` on both cases.

```
Theorem SSev_ev_secondtry : ∀ n,
  ev (S (S n)) → ev n.
Proof.
  intros n E. remember (S (S n)) as n2.
  destruct E as [| n' E'].
  Case "n = 0". inversion Heqn2.
  Case "n = S n'". inversion Heqn2. rewrite ← H0. apply E'.
Qed.
```

There is a much simpler way to this: we can use `inversion` directly on the inductively defined proposition *ev* (*S* (*S* *n*)).

```
Theorem SSev__even : ∀ n,
  ev (S (S n)) → ev n.
Proof.
  intros n E. inversion E as [| n' E']. apply E'. Qed.
```

This use of `inversion` 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 *I* 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` *I* generates a subgoal in which *I* 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.

In this particular case, the `inversion` analyzed the construction *ev* (*S* (*S* *n*)), determined that this could only have been constructed using *ev_SS*, and generated a new subgoal with the arguments of that constructor as new hypotheses. (It also produced an auxiliary equality, which happens to be useless here.) We'll begin exploring this more general behavior of inversion in what follows.

Exercise: 1 star (inversion_practice)

```
Theorem SSSSev__even : ∀ n,
  ev (S (S (S (S n)))) → ev n.
Proof.
  Admitted.
```

The `inversion` tactic can also be used to derive goals by showing the absurdity of a hypothesis.

```
Theorem even5_nonsense :
  ev 5 → 2 + 2 = 9.
Proof.
  Admitted.
```

☐

We can generally use `inversion` on inductive propositions. This illustrates that in general, we get one case for each possible constructor. Again, we also get some auxiliary equalities that are rewritten in the goal but not in the other hypotheses.

```
Theorem ev_minus2': ∀ n,
  ev n → ev (pred (pred n)).
Proof.
  intros n E. inversion E as [| n' E'].
  Case "E = ev_0". simpl. apply ev_0.
```

Case "E = ev_SS n' E'". simpl. apply *E'*. Qed.

Exercise: 3 stars, recommended (ev_ev__ev) Finding the appropriate thing to do induction on is a bit tricky here:

```
Theorem ev_ev__ev : ∀ n m,
  ev (n+m) → ev n → ev m.
Proof.
  Admitted.
```

☐

Exercise: 3 stars, optional (ev_plus_plus) Here's an exercise that just requires applying existing lemmas. No induction or even case analysis is needed, but some of the rewriting may be tedious. You'll want the `replace` tactic used for *plus_swap'* in Basics.v

```
Theorem ev_plus_plus : ∀ n m p,
  ev (n+m) → ev (n+p) → ev (m+p).
Proof.
  Admitted.
```

☐

28.4 Programming with Propositions

A *proposition* is a statement expressing a factual claim, like "two plus two equals four." In Coq, propositions are written as expressions of type `Prop`. Although we haven't mentioned it explicitly, we have already seen numerous examples.

```
Check (2 + 2 = 4).

Check (ble_nat 3 2 = false).

Check (beautiful 8).
```

Both provable and unprovable claims are perfectly good propositions. Simply *being* a proposition is one thing; being *provable* is something else!

```
Check (2 + 2 = 5).

Check (beautiful 4).
```

Both $2 + 2 = 4$ and $2 + 2 = 5$ are legal expressions of type `Prop`.

We've seen one way that propositions can be used in Coq: in `Theorem` (and `Lemma` and `Example`) declarations.

```
Theorem plus_2_2_is_4 :
  2 + 2 = 4.
Proof. reflexivity. Qed.
```

But they 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 (numbers, functions, types, type functions, ...).

```
Definition plus_fact : Prop := 2 + 2 = 4.
Check plus_fact.
```

Now we can 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.
```

There are many ways of constructing propositions. We can define a new proposition primitively using `Inductive`, we can form an equality proposition from two computational expressions, and we can build up a new proposition from existing ones using implication and quantification.

```
Definition strange_prop1 : Prop :=
  (2 + 2 = 5) → (99 + 26 = 42).
```

Also, given a proposition P with a free variable n, we can form the proposition ∀ n, P.

```
Definition strange_prop2 :=
  ∀ n, (ble_nat n 17 = true) → (ble_nat n 99 = true).
```

We can also define *parameterized propositions*, such as the property of being even.

```
Check even.
Check (even 4).
Check (even 3).
```

The type of *even*, *nat*→`Prop`, can be pronounced in three equivalent ways: (1) "*even* is a *function* from numbers to propositions," (2) "*even* is a *family* of propositions, indexed by a number n," or (3) "*even* is a *property* of numbers."

Propositions – including parameterized propositions – are first-class citizens in Coq. For example, we can define them to take multiple arguments...

```
Definition between (n m o: nat) : Prop :=
  andb (ble_nat n o) (ble_nat o m) = true.
```

... and then partially apply them:

```
Definition teen : nat→Prop := between 13 19.
```

We can even pass propositions – including parameterized propositions – as arguments to functions:

```
Definition true_for_zero (P:nat→Prop) : Prop :=
  P 0.
```

Here are two more examples of passing parameterized propositions as arguments to a function. The first takes a proposition P as argument and builds the proposition that P is true for all natural numbers. The second takes P and builds the proposition that, if P is true for some natural number n', then it is also true by the successor of n' – i.e. that P is *preserved by successor*:

```
Definition true_for_all_numbers (P:nat→Prop) : Prop :=
  ∀ n, P n.
Definition preserved_by_S (P:nat→Prop) : Prop :=
  ∀ n', P n' → P (S n').
```

28.5 Induction Principles

This is a good point to pause and take a deeper look at induction principles in general.

28.5.1 Induction Principles for Inductively Defined Types

Every time we declare a new `Inductive` datatype, Coq automatically generates an axiom that embodies an *induction principle* for this type.

The induction principle for a type t 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 a little with using `apply` *nat_ind* directly, 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' : ∀ n:nat,
  n × 0 = 0.
Proof.
  apply nat_ind.
  Case "O". reflexivity.
  Case "S". 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. The `induction` tactic works either with a variable in the context or a quantified variable in the goal.

Third, the `apply` tactic automatically chooses variable names for us (in the second subgoal, here), whereas `induction` lets us specify (with the `as`... clause) what names should be used. The automatic choice is actually a little unfortunate, since it re-uses the name n for a variable that is different from the n in the original theorem. This is why the *Case* annotation is just S – if we tried to write it out in the more explicit form that we've been using for most proofs, we'd have to write $n = S\ n$, which doesn't make a lot of sense! All of 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 this little bit of bookkeeping, applying *nat_ind* is what we are really doing.

Exercise: 2 stars, optional (plus_one_r') Complete this proof as we did *mult_0_r'* above, without using the `induction` tactic.

```
Theorem plus_one_r' : ∀ n:nat,
  n + 1 = S n.
Proof.
  Admitted.
```

☐

The induction principles that Coq generates for other datatypes defined with `Inductive` 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 (rgb) Write out the induction principle that Coq will generate for the following datatype. Write down your answer on paper, 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 (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...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 (in English):
  - "for all values *x1...xn* of types *a1...an*, if *P* holds for each of the *x*s of type *t*, then *P* holds for *c x1 ... xn*".

Exercise: 1 star (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 :=
```

.

□

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 □ -> (forall (x : X) (l : list X), P l -> P (x :: l)) -> forall l : list X, P l Note the wording here (and, accordingly, the form of *list_ind*): 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 (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 (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, _____________________

☐

28.5.2 Induction Hypotheses

Where does the phrase "induction hypothesis" fit into this story?

The induction principle for numbers forall P : nat -> Prop, P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n is a generic statement that holds for all propositions P (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 nat→Prop.

We can make the proof 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 when we do the proof it is easier to see where *P_m0r* appears.

```
Theorem mult_0_r'' : ∀ n:nat,
  P_m0r n.
Proof.
  apply nat_ind.
  Case "n = O". reflexivity.
  Case "n = S n'".
    unfold P_m0r. simpl. intros n' IHn'.
    apply IHn'. Qed.
```

This extra naming step isn't something that we'll 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 ∀ *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 ∀ *n'*, *P_m0r n'* → *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'*.

28.6 Optional Material

This section offers some additional details on how induction works in Coq and the process of building proof trees. It can safely be skimmed on a first reading. (We recommend skimming rather than skipping over it outright: it answers some questions that occur to many Coq users at some point, so it is useful to have a rough idea of what's here.)

28.6.1 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 *gorgeous* are a tiny bit more complicated. As with all induction principles, we want to use the induction principle on *gorgeous* to prove things by inductively considering the possible shapes that something in *gorgeous* can have – either it is evidence that 0 is gorgeous, or it is evidence that, for some n, $3+n$ is gorgeous, or it is evidence that, for some n, $5+n$ is gorgeous and it includes evidence that n itself is. Intuitively speaking, however, what we want to prove are not statements about *evidence* but statements about *numbers*. So 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 *gorgeous*... Inductive gorgeous : nat -> Prop := g_0 : gorgeous 0 | g_plus3 : forall n, gorgeous n -> gorgeous (3+m) | g_plus5 : forall n, gorgeous n -> gorgeous (5+m). ...to give rise to an induction principle that looks like this... gorgeous_ind_max : forall P : (forall n : nat, gorgeous n -> Prop), P O g_0 -> (forall (m : nat) (e : gorgeous m), P m e -> P (3+m) (g_plus3 m e) -> (forall (m : nat) (e : gorgeous m), P m e -> P (5+m) (g_plus5 m e) -> forall (n : nat) (e : gorgeous n), P n e ... because:

- Since *gorgeous* is indexed by a number n (every *gorgeous* object e is a piece of evidence that some particular number n is gorgeous), the proposition P is parameterized by both n and e – that is, the induction principle can be used to prove assertions involving both a gorgeous number and the evidence that it is gorgeous.
- Since there are three ways of giving evidence of gorgeousness (*gorgeous* has three constructors), applying the induction principle generates three subgoals:
  - We must prove that P holds for O and *b_0*.
  - We must prove that, whenever n is a gorgeous number and e is an evidence of its gorgeousness, if P holds of n and e, then it also holds of $3+m$ and *g_plus3 n e*.
  - We must prove that, whenever n is a gorgeous number and e is an evidence of its gorgeousness, if P holds of n and e, then it also holds of $5+m$ and *g_plus5 n e*.
- If these subgoals can be proved, then the induction principle tells us that P is true for *all* gorgeous numbers n and evidence e of their gorgeousness.

But this is a little more flexibility than we actually need or want: it is giving us a way to prove logical assertions where the assertion involves properties of some piece of *evidence* of gorgeousness, while all we really care about is proving properties of *numbers* that are gorgeous – 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 gorgeous numbers n: forall P : nat -> Prop, ... -> forall n : nat, gorgeous n -> P n For this reason, Coq actually generates the following simplified induction principle for *gorgeous*:

```
Check gorgeous_ind.
```

In particular, Coq has dropped the evidence term e as a parameter of the the proposition P, and consequently has rewritten the assumption $\forall$ (n : *nat*) (e: *gorgeous* n), ... to be $\forall$ (n : *nat*), *gorgeous* $n \rightarrow$ ...; i.e., we no longer require explicit evidence of the provability of *gorgeous* n.

In English, *gorgeous_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 gorgeous, it suffices to show:
  - P holds for 0,
  - for any n, if n is gorgeous and P holds for n, then P holds for $3+n$,

- for any n, if n is gorgeous and P holds for n, then P holds for $5+n$.

We can apply *gorgeous_ind* directly instead of using `induction`.

```
Theorem gorgeous__beautiful' : ∀ n, gorgeous n → beautiful n.
Proof.
  intros.
  apply gorgeous_ind.
  Case "g_0".
    apply b_0.
  Case "g_plus3".
    intros.
    apply b_sum. apply b_3.
    apply H1.
  Case "g_plus5".
    intros.
    apply b_sum. apply b_5.
    apply H1.
  apply H.
Qed.
```

```
Module P.
```

Exercise: 3 stars, optional (p_provability) Consider the following inductively defined proposition:

```
Inductive p : (tree nat) → nat → Prop :=
  | c1 : ∀ n, p (leaf _ n) 1
  | c2 : ∀ t1 t2 n1 n2,
          p t1 n1 → p t2 n2 → p (node _ t1 t2) (n1 + n2)
  | c3 : ∀ t n, p t n → p t (S n).
```

Describe, in English, the conditions under which the proposition $p\ t\ n$ is provable.
☐

```
End P.
```

28.6.2 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 $P\ O$ 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 the proof above that *plus* is associative...

```
Theorem plus_assoc' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p.
  induction n as [| n'].
  Case "n = O". reflexivity.
  Case "n = S n'".
    simpl. rewrite → IHn'. reflexivity. Qed.
```

It also works to apply `induction` to a variable that is quantified in the goal.

```
Theorem plus_comm' : ∀ n m : nat,
  n + m = m + n.
Proof.
  induction n as [| n'].
  Case "n = O". intros m. rewrite → plus_0_r. reflexivity.
  Case "n = S n'". 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 ∀ *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'' : ∀ n m : nat,
  n + m = m + n.
Proof.
  induction m as [| m'].
  Case "m = O". simpl. rewrite → plus_0_r. reflexivity.
  Case "m = S m'". 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.

☐

One more quick digression, for adventurous souls: if we can define parameterized propositions using `Definition`, then can we also define them using `Fixpoint`? Of course we can! However, this kind of "recursive parameterization" doesn't correspond to anything very familiar from everyday mathematics. The following exercise gives a slightly contrived example.

Exercise: 4 stars, optional (true_upto_n__true_everywhere) Define a recursive function *true_upto_n__true_everywhere* that makes *true_upto_n_example* work.

☐

28.6.3 Building Proof Objects Incrementally

As you probably noticed while solving the exercises earlier in the chapter, constructing proof objects is more involved than constructing the corresponding tactic proofs. Fortunately, there is a bit of syntactic sugar that we've already introduced to help in the construction: the *admit* term, which we've sometimes used to force Coq into accepting incomplete exercies. As an example, let's walk through the process of constructing a proof object demonstrating the beauty of 16.

```
Definition b_16_atmpt_1 : beautiful 16 := admit.
```

Maybe we can use *b_sum* to construct a term of type *beautiful* 16? Recall that *b_sum* is of type

forall n m : nat, beautiful n -> beautiful m -> beautiful (n + m)

If we can demonstrate the beauty of 5 and 11, we should be done.

```
Definition b_16_atmpt_2 : beautiful 16 := b_sum 5 11 admit admit.
```

In the attempt above, we've omitted the proofs of the propositions that 5 and 11 are beautiful. But the first of these is already axiomatized in *b_5*:

```
Definition b_16_atmpt_3 : beautiful 16 := b_sum 5 11 b_5 admit.
```

What remains is to show that 11 is beautiful. We repeat the procedure:

```
Definition b_16_atmpt_4 : beautiful 16 :=
  b_sum 5 11 b_5 (b_sum 5 6 admit admit).

Definition b_16_atmpt_5 : beautiful 16 :=
  b_sum 5 11 b_5 (b_sum 5 6 b_5 admit).

Definition b_16_atmpt_6 : beautiful 16 :=
  b_sum 5 11 b_5 (b_sum 5 6 b_5 (b_sum 3 3 admit admit)).
```

And finally, we can complete the proof object:

```
Definition b_16 : beautiful 16 :=
  b_sum 5 11 b_5 (b_sum 5 6 b_5 (b_sum 3 3 b_3 b_3)).
```

To recap, we've been guided by an informal proof that we have in our minds, and we check the high level details before completing the intricacies of the proof. The *admit* term allows us to do this.

28.7 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 : forall l, l = rev l -> pal l may seem obvious, but will not work very well.)

- Prove that forall l, pal (l ++ rev l).
- Prove that forall l, pal l -> l = rev l.

□

Exercise: 5 stars, optional (palindrome_converse) Using your definition of *pal* from the previous exercise, prove that forall l, l = rev l -> pal l.

□

Exercise: 4 stars (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 that subsequence is reflexive, that is, any list is a subsequence of itself.
- Prove 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 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 (foo_ind_principle) Suppose we make the following inductive definition: Inductive foo (X : Set) (Y : Set) : Set := | foo1 : X -> foo X Y | foo2 : Y -> foo X Y | foo3 : foo X Y -> foo X Y. Fill in the blanks to complete the induction principle that will be generated by Coq. foo_ind : forall (X Y : Set) (P : foo X Y -> Prop), (forall x : X, ________________________________) -> (forall y : Y, ________________________________) -> (____) -> ____

□

Exercise: 2 stars, optional (bar_ind_principle) Consider the following induction principle: bar_ind : forall P : bar -> Prop, (forall n : nat, P (bar1 n)) -> (forall b : bar, P b -> P (bar2 b)) -> (forall (b : bool) (b0 : bar), P b0 -> P (bar3 b b0)) -> forall b : bar, P b Write out the corresponding inductive set definition. Inductive bar : Set := | bar1 : ______________________________________ | bar2 : ______________________________________ | bar3 : ______________________________________.

□

Exercise: 2 stars, optional (no_longer_than_ind) Given the following inductively defined proposition: Inductive no_longer_than (X : Set) : (list X) -> nat -> Prop := | nlt_nil : forall n, no_longer_than X □ n | nlt_cons : forall x l n, no_longer_than X l n -> no_longer_than X (x::l) (S n) | nlt_succ : forall l n, no_longer_than X l n -> no_longer_than X l (S n). write the induction principle generated by Coq. no_longer_than_ind : forall (X : Set) (P : list X -> nat -> Prop), (forall n : nat, __________________) -> (forall (x : X) (l : list X) (n : nat), no_longer_than X l n -> __________________ -> ____________________________ -> (forall (l : list X) (n : nat), no_longer_than X l n -> __________________ -> ______________________ -> forall (l : list X) (n : nat), no_longer_than X l n -> __________________

□

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]

□

Chapter 29

Library Logic

29.1 Logic: Logic in Coq

```
Require Export "Prop".
```

Coq's built-in logic is extremely small: only `Inductive` definitions, universal quantification ($\forall$), and implication ($\rightarrow$) are primitive, while all the other familiar logical connectives – conjunction, disjunction, negation, existential quantification, even equality – can be defined using just these.

29.2 Quantification and Implication

In fact, $\rightarrow$ and $\forall$ are the *same* primitive! Coq's $\rightarrow$ notation is actually just a shorthand for $\forall$. The $\forall$ notation is more general, because it allows us to *name* the hypothesis.

For example, consider this proposition:

```
Definition funny_prop1 :=
  ∀ n, ∀ (E : beautiful n), beautiful (n+3).
```

If we had a proof term inhabiting this proposition, it would be a function with two arguments: a number n and some evidence that n is beautiful. But the name E for this evidence is not used in the rest of the statement of *funny_prop1*, so it's a bit silly to bother making up a name. We could write it like this instead, using the dummy identifier _ in place of a real name:

```
Definition funny_prop1' :=
  ∀ n, ∀ (_ : beautiful n), beautiful (n+3).
```

Or, equivalently, we can write it in more familiar notation:

```
Definition funny_prop1'' :=
  ∀ n, beautiful n → beautiful (n+3).
```

This illustrates that "$P \rightarrow Q$" is just syntactic sugar for "$\forall$ (_:P), Q".

29.3 Conjunction

The logical conjunction of propositions P and Q can be represented using an `Inductive` definition with one constructor.

```
Inductive and (P Q : Prop) : Prop :=
  conj : P → Q → (and P Q).
```

Note that, like the definition of *ev* in the previous chapter, this definition is parameterized; however, in this case, the parameters are themselves propositions, rather than numbers.

The intuition behind this definition is simple: to construct evidence for *and P Q*, we must provide evidence for P and evidence for Q. More precisely:

- *conj p q* can be taken as evidence for *and P Q* if p is evidence for P and q is evidence for Q; and

- this is the *only* way to give evidence for *and P Q* – that is, if someone gives us evidence for *and P Q*, we know it must have the form *conj p q*, where p is evidence for P and q is evidence for Q.

Since we'll be using conjunction a lot, let's introduce a more familiar-looking infix notation for it.

```
Notation "P /\ Q" := (and P Q) : type_scope.
```

(The *type_scope* annotation tells Coq that this notation will be appearing in propositions, not values.)

Consider the "type" of the constructor *conj*:

```
Check conj.
```

Notice that it takes 4 inputs – namely the propositions P and Q and evidence for P and Q – and returns as output the evidence of $P \wedge Q$.

Besides the elegance of building everything up from a tiny foundation, what's nice about defining conjunction this way is that we can prove statements involving conjunction using the tactics that we already know. For example, if the goal statement is a conjuction, we can prove it by applying the single constructor *conj*, which (as can be seen from the type of *conj*) solves the current goal and leaves the two parts of the conjunction as subgoals to be proved separately.

```
Theorem and_example :
  (beautiful 0) ∧ (beautiful 3).
Proof.
  apply conj.
   apply b_0.
   apply b_3. Qed.
```

Let's take a look at the proof object for the above theorem.

```
Print and_example.
```

Note that the proof is of the form conj (beautiful 0) (beautiful 3) (...pf of beautiful 3...) (...pf of beautiful 3...) as you'd expect, given the type of *conj*.

Just for convenience, we can use the tactic `split` as a shorthand for `apply` *conj*.

```
Theorem and_example' :
  (ev 0) ∧ (ev 4).
Proof.
  split.
    Case "left". apply ev_0.
    Case "right". apply ev_SS. apply ev_SS. apply ev_0. Qed.
```

Conversely, the `inversion` tactic can be used to take a conjunction hypothesis in the context, calculate what evidence must have been used to build it, and add variables representing this evidence to the proof context.

```
Theorem proj1 : ∀ P Q : Prop,
  P ∧ Q → P.
Proof.
  intros P Q H.
  inversion H as [HP HQ].
  apply HP. Qed.
```

Exercise: 1 star, optional (proj2) `Theorem proj2 : ∀ P Q : Prop,`

```
  P ∧ Q → Q.
Proof.
   Admitted.
```

☐

```
Theorem and_commut : ∀ P Q : Prop,
  P ∧ Q → Q ∧ P.
Proof.
  intros P Q H.
  inversion H as [HP HQ].
  split.
   apply HQ.
   apply HP. Qed.
```

Once again, we have commented out the *Case* tactics to make the proof object for this theorem easy to understand. Examining it shows that all that is really happening is taking apart a record containing evidence for P and Q and rebuilding it in the opposite order:

```
Print and_commut.
```

Exercise: 2 stars (and_assoc) In the following proof, notice how the *nested pattern* in the `inversion` 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 : ∀ P Q R : Prop,
  $P \wedge (Q \wedge R) \rightarrow (P \wedge Q) \wedge R$.
Proof.
  intros P Q R H.
  inversion H as [HP [HQ HR]].
   Admitted.
   □

Exercise: 2 stars, recommended (even__ev) Now we can prove the other direction of the equivalence of *even* and *ev*, which we left hanging in chapter Prop. Notice that the left-hand conjunct here is the statement we are actually interested in; the right-hand conjunct is needed in order to make the induction hypothesis strong enough that we can carry out the reasoning in the inductive step. (To see why this is needed, try proving the left conjunct by itself and observe where things get stuck.)

Theorem even__ev : ∀ n : **nat**,
  (even n → **ev** n) ∧ (even (S n) → **ev** (S n)).
Proof.
   Admitted.
   □

Exercise: 2 stars, optional (conj_fact) Construct a proof object demonstrating the following proposition.

Definition conj_fact : ∀ P Q R, $P \wedge Q \rightarrow Q \wedge R \rightarrow P \wedge R$:=
  admit.
   □

29.3.1 Iff

The familiar logical "if and only if" is just the conjunction of two implications.

Definition iff (P Q : Prop) := $(P \rightarrow Q) \wedge (Q \rightarrow P)$.

Notation "P <-> Q" := (iff P Q)
                      (at level 95, no associativity) : *type_scope*.

Theorem iff_implies : ∀ P Q : Prop,
  $(P \leftrightarrow Q) \rightarrow P \rightarrow Q$.
Proof.
  intros P Q H.
  inversion H as [HAB HBA]. apply HAB. Qed.

Theorem iff_sym : ∀ P Q : Prop,
  $(P \leftrightarrow Q) \rightarrow (Q \leftrightarrow P)$.
Proof.
  intros P Q H.

```
  inversion H as [HAB HBA].
  split.
    Case "->". apply HBA.
    Case "<-". apply HAB. 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 : ∀ P : Prop,
  P ↔ P.
Proof.
   Admitted.

Theorem iff_trans : ∀ P Q R : Prop,
  (P ↔ Q) → (Q ↔ R) → (P ↔ R).
Proof.
   Admitted.
```

Hint: If you have an iff hypothesis in the context, you can use `inversion` to break it into two separate implications. (Think about why this works.) ☐

Exercise: 2 stars, optional (beautiful_iff_gorgeous) We have seen that the families of propositions *beautiful* and *gorgeous* actually characterize the same set of numbers. Prove that *beautiful* $n \leftrightarrow$ *gorgeous* n for all n. Just for fun, write your proof as an explicit proof object, rather than using tactics. (*Hint*: if you make use of previously defined theorems, you should only need a single line!)

```
Definition beautiful_iff_gorgeous :
  ∀ n, beautiful n ↔ gorgeous n :=
  admit.
```

☐

Some of Coq's tactics treat *iff* statements specially, thus avoiding the need for some low-level manipulation when reasoning with them. In particular, `rewrite` can be used with *iff* statements, not just equalities.

29.4 Disjunction

Disjunction ("logical or") can also be defined as an inductive proposition.

```
Inductive or (P Q : Prop) : Prop :=
  | or_introl : P → or P Q
  | or_intror : Q → or P Q.

Notation "P \/ Q" := (or P Q) : type_scope.
```

Consider the "type" of the constructor *or_introl*:

Check or_introl.

It takes 3 inputs, namely the propositions P, Q and evidence of P, and returns, as output, the evidence of $P \lor Q$. Next, look at the type of *or_intror*:

Check or_intror.

It is like *or_introl* but it requires evidence of Q instead of evidence of P.

Intuitively, there are two ways of giving evidence for $P \lor Q$:

- give evidence for P (and say that it is P you are giving evidence for – this is the function of the *or_introl* constructor), or
- give evidence for Q, tagged with the *or_intror* constructor.

Since $P \lor Q$ has two constructors, doing `inversion` on a hypothesis of type $P \lor Q$ yields two subgoals.

```
Theorem or_commut : ∀ P Q : Prop,
  P ∨ Q → Q ∨ P.
Proof.
  intros P Q H.
  inversion H as [HP | HQ].
    Case "left". apply or_intror. apply HP.
    Case "right". apply or_introl. apply HQ. Qed.
```

From here on, we'll use the shorthand tactics `left` and `right` in place of `apply` *or_introl* and `apply` *or_intror*.

```
Theorem or_commut' : ∀ P Q : Prop,
  P ∨ Q → Q ∨ P.
Proof.
  intros P Q H.
  inversion H as [HP | HQ].
    Case "left". right. apply HP.
    Case "right". left. apply HQ. Qed.
```

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!).

☐

```
Theorem or_distributes_over_and_1 : ∀ P Q R : Prop,
  P ∨ (Q ∧ R) → (P ∨ Q) ∧ (P ∨ R).
Proof.
  intros P Q R. intros H. inversion H as [HP | [HQ HR]].
    Case "left". split.
      SCase "left". left. apply HP.
      SCase "right". left. apply HP.
```

```
Case "right". split.
  SCase "left". right. apply HQ.
  SCase "right". right. apply HR. Qed.
```

Exercise: 2 stars, recommended (or_distributes_over_and_2) Theorem or_distributes_over_and_2

```
: ∀ P Q R : Prop,
  (P ∨ Q) ∧ (P ∨ R) → P ∨ (Q ∧ R).
Proof.
   Admitted.
```

☐

Exercise: 1 star, optional (or_distributes_over_and) Theorem or_distributes_over_and

```
: ∀ P Q R : Prop,
  P ∨ (Q ∧ R) ↔ (P ∨ Q) ∧ (P ∨ R).
Proof.
   Admitted.
```

☐

29.4.1 Relating ∧ and ∨ with *andb* and *orb*

We've already seen several places where analogous structures can be found in Coq's computational (Type) and logical (Prop) worlds. Here is one more: the boolean operators *andb* and *orb* are clearly analogs of the logical connectives ∧ and ∨. This analogy can be made more precise by the following theorems, which show how to translate knowledge about *andb* and *orb*'s behaviors on certain inputs into propositional facts about those inputs.

```
Theorem andb_true__and : ∀ b c,
  andb b c = true → b = true ∧ c = true.
Proof.
  intros b c H.
  destruct b.
    Case "b = true". destruct c.
      SCase "c = true". apply conj. reflexivity. reflexivity.
      SCase "c = false". inversion H.
    Case "b = false". inversion H. Qed.
```

```
Theorem and__andb_true : ∀ b c,
  b = true ∧ c = true → andb b c = true.
Proof.
  intros b c H.
  inversion H.
  rewrite H0. rewrite H1. reflexivity. Qed.
```

Exercise: 2 stars (bool_prop) Theorem andb_false : ∀ b c,

```
  andb b c = false → b = false ∨ c = false.
Proof.
    Admitted.
Theorem orb_true : ∀ b c,
  orb b c = true → b = true ∨ c = true.
Proof.
    Admitted.
Theorem orb_false : ∀ b c,
  orb b c = false → b = false ∧ c = false.
Proof.
    Admitted.
```

☐

29.5 Falsehood

Logical falsehood can be represented in Coq as an inductively defined proposition with no constructors.

```
Inductive False : Prop := .
```

Intuition: *False* is a proposition for which there is no way to give evidence.

Exercise: 1 star (False_ind_principle) Can you predict the induction principle for falsehood?

☐

Since *False* has no constructors, inverting an assumption of type *False* always yields zero subgoals, allowing us to immediately prove any goal.

```
Theorem False_implies_nonsense :
  False → 2 + 2 = 5.
Proof.
  intros contra.
  inversion contra. Qed.
```

How does this work? The `inversion` tactic breaks *contra* into each of its possible cases, and yields a subgoal for each case. As *contra* is evidence for *False*, it has *no* possible cases, hence, there are no possible subgoals and the proof is done.

Conversely, the only way to prove *False* is if there is already something nonsensical or contradictory in the context:

```
Theorem nonsense_implies_False :
  2 + 2 = 5 → False.
Proof.
```

```
  intros contra.
  inversion contra. Qed.
```

Actually, since the proof of *False_implies_nonsense* doesn't actually have anything to do with the specific nonsensical thing being proved; it can easily be generalized to work for an arbitrary P:

```
Theorem ex_falso_quodlibet : ∀ (P:Prop),
  False → P.
Proof.
  intros P contra.
  inversion contra. Qed.
```

The Latin *ex falso quodlibet* means, literally, "from falsehood follows whatever you please." This theorem is also known as the *principle of explosion*.

29.5.1 Truth

Since we have defined falsehood in Coq, one might wonder whether it is possible to define truth in the same way. We can.

Exercise: 2 stars, optional (True_induction) Define *True* as another inductively defined proposition. What induction principle will Coq generate for your definition? (The intution is that *True* should be a proposition for which it is trivial to give evidence. Alternatively, you may find it easiest to start with the induction principle and work backwards to the inductive definition.)

□

However, unlike *False*, which we'll use extensively, *True* is just a theoretical curiosity: it is trivial (and therefore uninteresting) to prove as a goal, and it carries no useful information as a hypothesis.

29.6 Negation

The logical complement of a proposition P is written *not* P or, for shorthand, $\neg P$:

```
Definition not (P:Prop) := P → False.
```

The intuition is that, if P is not true, then anything at all (even *False*) follows from assuming P.

```
Notation "~ x" := (not x) : type_scope.
```

```
Check not.
```

It takes a little practice to get used to working with negation in Coq. Even though you can see perfectly well why something is true, it can be a little hard at first to get things into the right configuration so that Coq can see it! Here are proofs of a few familiar facts about negation to get you warmed up.

```
Theorem not_False :
  ¬ False.
Proof.
  unfold not. intros H. inversion H. Qed.
```

```
Theorem contradiction_implies_anything : ∀ P Q : Prop,
  (P ∧ ¬P) → Q.
Proof.
  intros P Q H. inversion H as [HP HNA]. unfold not in HNA.
  apply HNA in HP. inversion HP. Qed.
```

```
Theorem double_neg : ∀ P : Prop,
  P → ~~P.
Proof.
  intros P H. unfold not. intros G. apply G. apply H. Qed.
```

Exercise: 2 stars, recommended (double_neg_inf) Write an informal proof of *double_neg*:

Theorem: P implies ~~P, for any proposition P.

Proof: □

Exercise: 2 stars, recommended (contrapositive)

```
Theorem contrapositive : ∀ P Q : Prop,
  (P → Q) → (¬Q → ¬P).
Proof.
  Admitted.
```

□

Exercise: 1 star (not_both_true_and_false)

```
Theorem not_both_true_and_false : ∀ P : Prop,
  ¬ (P ∧ ¬P).
Proof.
  Admitted.
```

□

Exercise: 1 star (informal_not_PNP) Write an informal proof (in English) of the proposition $\forall\ P$: Prop, ~($P \wedge \neg P$).

□

```
Theorem five_not_even :
  ¬ ev 5.
```

```
Proof.
  unfold not. intros Hev5. inversion Hev5 as [|n Hev3 Heqn].
  inversion Hev3 as [|n' Hev1 Heqn']. inversion Hev1. Qed.
```

Exercise: 1 star (ev_not_ev_S) Theorem *five_not_even* confirms the unsurprising fact that five is not an even number. Prove this more interesting fact:

```
Theorem ev_not_ev_S : ∀ n,
  ev n → ¬ ev (S n).
Proof.
  unfold not. intros n H. induction H. Admitted.
```

☐

Note that some theorems that are true in classical logic are *not* provable in Coq's (constructive) logic. E.g., let's look at how this proof gets stuck...

```
Theorem classic_double_neg : ∀ P : Prop,
  ~~P → P.
Proof.
  intros P H. unfold not in H.
  Admitted.
```

Exercise: 5 stars, optional (classical_axioms) For those who like a challenge, here is an exercise taken from the Coq'Art book (p. 123). The following five statements are often considered as characterizations of classical logic (as opposed to constructive logic, which is what is "built in" to Coq). We can't prove them in Coq, but we can consistently add any one of them as an unproven axiom if we wish to work in classical logic. Prove that these five propositions are equivalent.

```
Definition peirce := ∀ P Q: Prop,
  ((P→Q)->P)->P.
Definition classic := ∀ P:Prop,
  ~~P → P.
Definition excluded_middle := ∀ P:Prop,
  P ∨ ¬P.
Definition de_morgan_not_and_not := ∀ P Q:Prop,
  ~(~P∧¬Q) → P∨Q.
Definition implies_to_or := ∀ P Q:Prop,
  (P→Q) → (¬P∨Q).
```

☐

29.6.1 Inequality

Saying $x \neq y$ is just the same as saying ~$(x = y)$.

```
Notation "x <> y" := (¬ (x = y)) : type_scope.
```

Since inequality involves a negation, it again requires a little practice to be able to work with it fluently. Here is one very useful trick. If you are trying to prove a goal that is nonsensical (e.g., the goal state is *false* = *true*), apply the lemma *ex_falso_quodlibet* to change the goal to *False*. This makes it easier to use assumptions of the form ¬*P* that are available in the context – in particular, assumptions of the form *x*≠*y*.

```
Theorem not_false_then_true : ∀ b : bool,
  b ≠ false → b = true.
Proof.
  intros b H. destruct b.
  Case "b = true". reflexivity.
  Case "b = false".
    unfold not in H.
    apply ex_falso_quodlibet.
    apply H. reflexivity. Qed.
```

Exercise: 2 stars, recommended (not_eq_beq_false)

```
Theorem not_eq_beq_false : ∀ n n' : nat,
      n ≠ n' →
      beq_nat n n' = false.
Proof.
   Admitted.
```

☐

Exercise: 2 stars, optional (beq_false_not_eq)

```
Theorem beq_false_not_eq : ∀ n m,
  false = beq_nat n m → n ≠ m.
Proof.
   Admitted.
```

☐

29.7 Existential Quantification

Another critical logical connective is *existential quantification*. We can capture what this means with the following definition:

```
Inductive ex (X:Type) (P : X→Prop) : Prop :=
  ex_intro : ∀ (witness:X), P witness → ex X P.
```

That is, *ex* is a family of propositions indexed by a type X and a property P over X. In order to give evidence for the assertion "there exists an x for which the property P holds" we must actually name a *witness* – a specific value x – and then give evidence for P x, i.e., evidence that x has the property P.

For example, consider this existentially quantified proposition:

```
Definition some_nat_is_even : Prop :=
```

ex nat ev.

To prove this proposition, we need to choose a particular number as witness – say, 4 – and give some evidence that that number is even.

```
Definition snie : some_nat_is_even :=
  ex_intro _ ev 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

Coq's notation definition facility can be used to introduce more familiar notation for writing existentially quantified propositions, exactly parallel to the built-in syntax for universally quantified propositions. Instead of writing *ex nat ev* to express the proposition that there exists some number that is even, for example, we can write ∃ *x*:*nat*, *ev x*. (It is not necessary to understand exactly how the `Notation` definition works.)

```
Notation "'exists' x , p" := (ex _ (fun x ⇒ p))
  (at level 200, x ident, right associativity) : type_scope.
Notation "'exists' x : X , p" := (ex _ (fun x:X ⇒ p))
  (at level 200, x ident, right associativity) : type_scope.
```

We can use the same set of tactics as always for manipulating existentials. For example, if to prove an existential, we `apply` the constructor *ex_intro*. Since the premise of *ex_intro* involves a variable (*witness*) that does not appear in its conclusion, we need to explicitly give its value when we use `apply`.

```
Example exists_example_1 : ∃ n, n + (n × n) = 6.
Proof.
  apply ex_intro with (witness:=2).
  reflexivity. Qed.
```

Note, again, that we have to explicitly give the witness.

Or, instead of writing `apply` *ex_intro* `with` (*witness*:=*e*) all the time, we can use the convenient shorthand ∃ *e*, which means the same thing.

```
Example exists_example_1' : ∃ n,
  n + (n × n) = 6.
Proof.
  ∃ 2.
  reflexivity. Qed.
```

Conversely, if we have an existential hypothesis in the context, we can eliminate it with `inversion`. Note the use of the `as...` pattern to name the variable that Coq introduces to name the witness value and get evidence that the hypothesis holds for the witness. (If we don't explicitly choose one, Coq will just call it *witness*, which makes proofs confusing.)

```
Theorem exists_example_2 : ∀ n,
  (∃ m, n = 4 + m) →
  (∃ o, n = 2 + o).
Proof.
  intros n H.
  inversion H as [m Hm].
```

```
  ∃ (2 + m).
  apply Hm. Qed.
```

Exercise: 1 star, optional (english_exists) In English, what does the proposition ex nat (fun n => beautiful (S n))]] mean?

Complete the definition of the following proof object:

```
Definition p : ex nat (fun n ⇒ beautiful (S n)) :=
admit.
```

☐

Exercise: 1 star (dist_not_exists) Prove that "P holds for all x" and "there is no x for which P does not hold" are equivalent assertions.

```
Theorem dist_not_exists : ∀ (X:Type) (P : X → Prop),
  (∀ x, P x) → ¬ (∃ x, ¬ P x).
Proof.
   Admitted.
```

☐

Exercise: 3 stars, optional (not_exists_dist) The other direction requires the classical "law of the excluded middle":

```
Theorem not_exists_dist :
  excluded_middle →
  ∀ (X:Type) (P : X → Prop),
    ¬ (∃ x, ¬ P x) → (∀ x, P x).
Proof.
   Admitted.
```

☐

Exercise: 2 stars (dist_exists_or) Prove that existential quantification distributes over disjunction.

```
Theorem dist_exists_or : ∀ (X:Type) (P Q : X → Prop),
  (∃ x, P x ∨ Q x) ↔ (∃ x, P x) ∨ (∃ x, Q x).
Proof.
   Admitted.
```

☐

29.8 Equality

Even Coq's equality relation is not built in. It has roughly the following inductive definition. (We enclose the definition in a module to avoid confusion with the standard library equality, which we have used extensively already.)

```
Module MyEquality.

Inductive eq (X:Type) : X -> X -> Prop :=
  refl_equal : ∀ x, eq X x x.
```

Standard infix notation (using Coq's type argument synthesis):

```
Notation "x = y" := (eq _ x y)
                    (at level 70, no associativity) : type_scope.
```

This is a bit subtle. The way to think about it 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 members of this family: applying the constructor *refl_equal* to a type X and a value $x : X$ yields evidence that x is equal to x.

Here is a slightly different definition – the one that actually appears in the Coq standard library.

```
Inductive eq' (X:Type) (x:X) : X -> Prop :=
    refl_equal' : eq' X x x.

Notation "x =' y" := (eq' _ x y)
                     (at level 70, no associativity) : type_scope.
```

Exercise: 3 stars, optional (two_defs_of_eq_coincide) Verify that the two definitions of equality are equivalent.

```
Theorem two_defs_of_eq_coincide : ∀ (X:Type) (x y : X),
  x = y <-> x =' y.
Proof.
  Admitted.
```

☐

The advantage of the second definition is that the induction principle that Coq derives for it is precisely the familiar principle of *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.

```
Check eq'_ind.
```

One important consideration remains. Clearly, we can use *refl_equal* to construct evidence that, for example, $2 = 2$. Can we also use it to construct evidence that $1 + 1 = 2$? Yes: 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 `Eval simpl`, include evaluation of function application, inlining of definitions, and simplification of `match`es.

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 :=
```

```
  refl_equal nat 4.
Definition singleton : ∀ (X:Set) (x:X), []++[x] = x::[] :=
  fun (X:Set) (x:X) ⇒ refl_equal (list X) [x].
End MyEquality.
```

29.8.1 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.

29.9 Relations as Propositions

A proposition parameterized by a number (such as *ev* or *beautiful*) 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` LeFirstTry.

We've already seen an inductive definition of one fundamental relation: equality. Another useful one 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 → nat → Prop :=
  | le_n : ∀ n, le n n
  | le_S : ∀ n m, (le n m) → (le n (S m)).
```

`End` LeFirstTry.

This is a reasonable definition of the $\leq$ relation, but we can streamline it 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. This is similar to what we did in our second definition of the *eq* relation, above.

```
Inductive le (n:nat) : nat → Prop :=
  | le_n : le n n
  | le_S : ∀ m, (le n m) → (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. (The same was true of our second version of *eq*.)

`Check le_ind.`

By contrast, the induction principle that Coq calculates for the first definition has a lot of extra quantifiers, which makes it messier to work with when proving things by induction. Here is the induction principle for the first *le*:

Proofs of facts about $\leq$ using the constructors *le_n* and *le_S* follow the same patterns as proofs about properties, like *ev* in chapter `Prop`. We can `apply` the constructors to prove $\leq$ goals (e.g., to show that 3<=3 or 3<=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).)

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 ≤ 3.
Proof.
  apply le_n. Qed.

Theorem test_le2 :
  3 ≤ 6.
Proof.
  apply le_S. apply le_S. apply le_S. apply le_n. Qed.

Theorem test_le3 :
  ¬ (2 ≤ 1).
Proof.
  intros H. inversion H. inversion H1. Qed.
```

The "strictly less than" relation $n < m$ can now be defined in terms of *le*.

```
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 (n:nat) : nat → Prop :=
  | nn : next_nat n (S n).

Inductive next_even (n:nat) : nat → Prop :=
  | ne_1 : ev (S n) → next_even n (S n)
  | ne_2 : 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, recommended (R_provability) `Module R.`

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) State and prove an equivalent characterization of the relation R. That is, if $R\ m\ n\ o$ is true, what can we say about m, n, and o, and vice versa?

☐

`End` R.

Exercise: 3 stars, recommended (all_forallb) Inductively define a property *all* of lists, parameterized by a type X and a property $P : X \to$ `Prop`, such that *all* X P l asserts that P is true for every element of the list l.

```
Inductive all (X : Type) (P : X → Prop) : list X → Prop :=

.
```

Recall the function *forallb*, from the exercise *forall_exists_challenge* in chapter *Poly*:

```
Fixpoint forallb {X : Type} (test : X → bool) (l : list X) : bool :=
  match l with
    | [] ⇒ true
    | x :: l' ⇒ andb (test x) (forallb test l')
  end.
```

Using the property *all*, write down a specification for *forallb*, and prove that it satisfies the specification. Try to make your specification as precise as possible.

Are there any important properties of the function *forallb* which are not captured by your specification?

☐

Exercise: 4 stars, optional (filter_challenge) One of the main purposes of Coq is to prove that programs match their specifications. To this end, let's prove that our definition of *filter* matches a specification. Here is the specification, written out informally in English.

Suppose we have a set X, a function *test*: $X \rightarrow bool$, 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*.

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. Your job is to translate this specification into a Coq theorem and prove it. (Hint: 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, optional (filter_challenge_2) A different way to formally 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. Express this claim formally and prove it.

□

Exercise: 4 stars, optional (no_repeats) The following inductively defined proposition...

```
Inductive appears_in {X:Type} (a:X) : list X → Prop :=
  | ai_here : ∀ l, appears_in a (a::l)
  | ai_later : ∀ b l, appears_in a l → appears_in a (b::l).
```

...gives us a precise way of saying that a value a appears at least once as a member of a list l.

Here's a pair of warm-ups about *appears_in*.

```
Lemma appears_in_app : ∀ {X:Type} (xs ys : list X) (x:X),
     appears_in x (xs ++ ys) → appears_in x xs ∨ appears_in x ys.
Proof.
  Admitted.

Lemma app_appears_in : ∀ {X:Type} (xs ys : list X) (x:X),
     appears_in x xs ∨ appears_in x ys → appears_in x (xs ++ ys).
Proof.
  Admitted.
```

Now use *appears_in* 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 *appears_in* to define an inductive proposition *no_repeats 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, *no_repeats nat* [1,2,3,4] and *no_repeats bool* [] should be provable, while *no_repeats nat* [1,2,1] and *no_repeats bool* [*true*,*true*] should not be.

Finally, state and prove one or more interesting theorems relating *disjoint*, *no_repeats* and ++ (list append).

☐

29.9.1 Digression: More Facts about $\leq$ and $<$

Let's pause briefly to record several facts about the $\leq$ and $<$ relations that we are going to need later in the course. The proofs make good practice exercises.

Exercise: 2 stars, optional (le_exercises) Theorem O_le_n : ∀ *n*,
  0 ≤ *n*.
Proof.
  Admitted.

Theorem n_le_m__Sn_le_Sm : ∀ *n m*,
  n ≤ *m* → S *n* ≤ S *m*.
Proof.
  Admitted.

Theorem Sn_le_Sm__n_le_m : ∀ *n m*,
  S *n* ≤ S *m* → *n* ≤ *m*.
Proof.
  intros *n m*. generalize dependent *n*. induction *m*.
  Admitted.

Theorem le_plus_l : ∀ *a b*,
  a ≤ *a* + *b*.
Proof.
  Admitted.

Theorem plus_lt : ∀ *n1 n2 m*,
  n1 + *n2* < *m* →
  n1 < *m* ∧ *n2* < *m*.
Proof.
  Admitted.

Theorem lt_S : ∀ *n m*,
  n < *m* →
  n < S *m*.
Proof.
  Admitted.

Theorem ble_nat_true : ∀ *n m*,
  ble_nat *n m* = true → *n* ≤ *m*.

```
Proof.
  Admitted.

Theorem ble_nat_n_Sn_false : ∀ n m,
  ble_nat n (S m) = false →
  ble_nat n m = false.
Proof.
  Admitted.

Theorem ble_nat_false : ∀ n m,
  ble_nat n m = false → ~(n ≤ m).
Proof.
  Admitted.
```
☐

Exercise: 3 stars, recommended (nostutter) Formulating inductive definitions of predicates is an important skill you'll need in this course. Try to solve this exercise without any help at all (except from your study group partner, if you have one).

We say that a list of numbers "stutters" if it repeats the same number consecutively. The predicate "*nostutter mylist*" means that *mylist* does not stutter. Formulate an inductive definition for *nostutter*. (This is different from the *no_repeats* predicate in the exercise above; the sequence 1,4,1 repeats but does not stutter.)

```
Inductive nostutter: list nat → Prop :=

.
```

Make sure each of these tests succeeds, but you are free to change the proof if the given one doesn't work for you. Your definition might be different from mine and still correct, in which case the examples might need a different proof.

The suggested proofs for the examples (in comments) use a number of tactics we haven't talked about, to try to make them robust with respect to different possible ways of defining *nostutter*. You should be able to just uncomment and use them as-is, but if you prefer 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 [].
  Admitted.

Example test_nostutter_3: nostutter [5].
  Admitted.

Example test_nostutter_4: not (nostutter [3,1,1,4]).
  Admitted.
```
☐

Exercise: 4 stars, optional (pigeonhole principle) The "pigeonhole principle" states a basic fact about counting: if you distribute more than n items into n pigeonholes, some pigeonhole must contain at least two items. As is often the case, this apparently trivial fact about numbers requires non-trivial machinery to prove, but we now have enough...

First a pair of useful lemmas... (we already proved this for lists of naturals, but not for arbitrary lists.)

```
Lemma app_length : ∀ {X:Type} (l1 l2 : list X),
  length (l1 ++ l2) = length l1 + length l2.
Proof.
  Admitted.
```

```
Lemma appears_in_app_split : ∀ {X:Type} (x:X) (l:list X),
  appears_in x l →
  ∃ l1, ∃ l2, l = l1 ++ (x::l2).
Proof.
  Admitted.
```

Now define a predicate *repeats* (analogous to *no_repeats* in the exercise above), 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. List *l2* represents a list of pigeonhole labels, and list *l1* represents an assignment of items to labels: if there are more items than labels, at least two items must have the same label. You will almost certainly need to use the *excluded_middle* hypothesis.

```
Theorem pigeonhole_principle: ∀ {X:Type} (l1 l2:list X),
  excluded_middle →
  (∀ x, appears_in x l1 → appears_in x l2) →
  length l2 < length l1 →
  repeats l1.
Proof. intros X l1. induction l1.
  Admitted.
  □
```

29.10 Informal Proofs

Q: What is the relation between a formal proof of a proposition P and an informal proof of the same proposition P?

A: The latter should *teach* the reader how to produce the former.

Q: How much detail is needed?

A: 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 amounts to just transcribing the formal one into words). This gives the reader the *ability* to reproduce the formal one for themselves, but it doesn't *teach* them anything.

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 usually writing the proof requires some deep 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 part of work that we went through to find the proof in the first place) and clear high-level suggestions for the more routine parts to save the reader from spending too much time reconstructing these parts (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.

Another key point: if we're talking about a formal proof of a proposition P and an informal proof of P, the proposition P doesn't change. That is, formal and informal proofs are *talking about the same world* and they *must play by the same rules.*

29.10.1 Informal Proofs by Induction

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. The ideal is somewhere in between, of course, but while you are 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 below – 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`). In the rest of this course, please follow one of the two for *all* of your inductive proofs.

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 $n = c\ a1\ ...\ ak$, 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 $P(c\ a1\ ...\ ak)$>.

    <go on and prove $P(n)$ to finish the case...>

– <other cases similarly...> □

Example:

- *Theorem*: For all sets X, lists l : *list* X, and numbers n, if *length* $l = n$ then *index* (S n) l = *None*.

  Proof: By induction on l.

  – Suppose l = []. We must show, for all numbers n, that, if length [] = n, then *index* (S n) [] = *None*.

    This follows immediately from the definition of index.

  – Suppose $l = x$ :: l' for some x and l', where *length* l' = n' implies *index* (S n') l' = *None*, for any number n'. We must show, for all n, that, if *length* (x::l') = n then *index* (S n) (x::l') = *None*.

    Let n be a number with *length* $l = n$. Since length l = length (x::l') = S (length l'), it suffices to show that index (S (length l')) l' = None.

]] But this follows directly from the induction hypothesis, picking n' to be length l'. □

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...> □

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 $m \leq o$ is *le_n*. Then $m = o$ and we must show that $n \leq m$, which is immediate by hypothesis.
  - Suppose the final rule used to show $m \leq o$ is *le_S*. Then $o = S\ o'$ for some o' with $m \leq o'$. We must show that $n \leq S\ o'$. By induction hypothesis, $n \leq o'$. But then, by *le_S*, $n \leq S\ o'$. □

29.11 Optional Material

29.11.1 Induction Principles for ∧ and ∨

The induction principles for conjunction and disjunction are a good illustration of Coq's way of generating simplified induction principles for `Inductive`ly defined propositions, which we discussed in the last chapter. You try first:

Exercise: 1 star, optional (and_ind_principle) See if you can predict the induction principle for conjunction.

□

Exercise: 1 star, optional (or_ind_principle) See if you can predict the induction principle for disjunction.

□

```
Check and_ind.
```

From the inductive definition of the proposition *and P Q* Inductive and (P Q : Prop) : Prop := conj : P -> Q -> (and P Q). we might expect Coq to generate this induction principle and_ind_max : forall (P Q : Prop) (P0 : P /\ Q -> Prop), (forall (a : P) (b : Q), P0 (conj P Q a b)) -> forall a : P /\ Q, P0 a but actually it generates this simpler and more useful one: and_ind : forall P Q P0 : Prop, (P -> Q -> P0) -> P /\ Q -> P0 In the same way, when given the inductive definition of *or P Q* Inductive or (P Q : Prop) : Prop := | or_introl : P -> or P Q | or_intror : Q -> or P Q. instead of the "maximal induction principle" or_ind_max : forall (P Q : Prop) (P0 : P \/ Q -> Prop), (forall a : P, P0 (or_introl P Q a)) -> (forall b : Q, P0 (or_intror P Q b)) -> forall o : P \/ Q, P0 o what Coq actually generates is this: or_ind : forall P Q P0 : Prop, (P -> P0) -> (Q -> P0) -> P \/ Q -> P0]]

29.11.2 Explicit Proof Objects for Induction

Although tactic-based proofs are normally much easier to work with, the ability to write a proof term directly is sometimes very handy, particularly when we want Coq to do something slightly non-standard.

Recall the induction principle on naturals that Coq generates for us automatically from the Inductive declation for *nat*.

There's nothing magic about this induction lemma: it's just another Coq lemma that requires a proof. Coq generates the proof automatically too...

`Print` *nat_ind*.

We can read this as follows: Suppose we have evidence f that P holds on 0, and evidence *f0* that $\forall$ *n*:*nat*, $P\ n \rightarrow P\ (S\ n)$. Then we can prove that P holds of an arbitrary nat n via a recursive function F (here defined using the expression form `Fix` rather than by a top-level `Fixpoint` declaration). F pattern matches on n:

- If it finds 0, F uses f to show that $P\ n$ holds.
- If it finds S *n0*, F applies itself recursively on *n0* to obtain evidence that P *n0* holds; then it applies *f0* on that evidence to show that $P\ (S\ n)$ holds.

F is just an ordinary recursive function that happens to operate on evidence in `Prop` rather than on terms in `Set`.

Aside to those interested in functional programming: You may notice that the `match` in F requires an annotation `as` *n0* `return` (P *n0*) to help Coq's typechecker realize that the two arms of the `match` actually return the same type (namely $P\ n$). This is essentially like matching over a GADT (generalized algebraic datatype) in Haskell. In fact, F has a *dependent* type: its result type depends on its argument; GADT's can be used to describe simple dependent types like this.

We can adapt this approach to proving *nat_ind* to help prove *non-standard* induction principles too. Recall our desire to prove that

$\forall$ n : *nat*, *even* $n \rightarrow$ *ev* n.

Attempts to do this by standard induction on n fail, because the induction principle only lets us proceed when we can prove that *even* $n \rightarrow$ *even* $(S\ n)$ – which is of course never provable. What we did earlier in this chapter was a bit of a hack:

`Theorem` *even__ev* : $\forall$ n : *nat*, (*even* $n \rightarrow$ *ev* n) $\land$ (*even* $(S\ n) \rightarrow$ *ev* $(S\ n)$).

We can make a much better proof by defining and proving a non-standard induction principle that goes "by twos":

```
Definition nat_ind2 :
  ∀ (P : nat → Prop),
  P 0 →
  P 1 →
  (∀ n : nat, P n → P (S(S n))) →
```

```
∀ n : nat , P n :=
    fun P ⇒ fun P0 ⇒ fun P1 ⇒ fun PSS ⇒
        fix f (n:nat) := match n return P n with
                             0 ⇒ P0
                           | 1 ⇒ P1
                           | S (S n') ⇒ PSS n' (f n')
                          end.
```

Once you get the hang of it, it is entirely straightforward to give an explicit proof term for induction principles like this. Proving this as a lemma using tactics is much less intuitive (try it!).

The `induction` ... `using` tactic gives a convenient way to specify a non-standard induction principle like this.

```
Lemma even__ev' : ∀ n, even n → ev n.
Proof.
 intros.
 induction n as [ | |n'] using nat_ind2.
  Case "even 0".
    apply ev_0.
  Case "even 1".
    inversion H.
  Case "even (S(S n'))".
    apply ev_SS.
    apply IHn'. unfold even. unfold even in H. simpl in H. apply H.
Qed.
```

29.11.3 The Coq Trusted Computing Base

One issue that arises with any automated proof assistant is "why trust it?": what if there is a bug in the implementation that renders all its reasoning suspect?

While it is impossible to allay such concerns completely, the fact that Coq is based on the Curry-Howard Correspondence gives it a strong foundation. Because propositions are just types and proofs are just terms, checking that an alleged proof of a proposition is valid just amounts to *type-checking* the term. Type checkers are relatively small and straightforward programs, so the "trusted computing base" for Coq – the part of the code that we have to believe is operating correctly – is small too.

What must a typechecker do? Its primary job is to make sure that in each function application the expected and actual argument types match, that the arms of a `match` expression are constructor patterns belonging to the inductive type being matched over and all arms of the `match` return the same type, and so on.

There are a few additional wrinkles:

- Since Coq types can themselves be expressions, the checker must normalize these (by using the conversion rules) before comparing them.

- The checker must make sure that `match` expressions are *exhaustive*. That is, there must be an arm for every possible constructor. To see why, consider the following alleged proof object: Definition or_bogus : forall P Q, P \/ Q -> P := fun (P Q : Prop) (A : P \/ Q) => match A with | or_introl H => H end. All the types here match correctly, but the `match` only considers one of the possible constructors for *or*. Coq's exhaustiveness check will reject this definition.

- The checker must make sure that each `fix` expression terminates. It does this using a syntactic check to make sure that each recursive call is on a subexpression of the original argument. To see why this is essential, consider this alleged proof: Definition nat_false : forall (n:nat), False := fix f (n:nat) : False := f n. Again, this is perfectly well-typed, but (fortunately) Coq will reject it.

Note that the soundness of Coq depends only on the correctness of this typechecking engine, not on the tactic machinery. If there is a bug in a tactic implementation (and this certainly does happen!), that tactic might construct an invalid proof term. But when you type `Qed`, Coq checks the term for validity from scratch. Only lemmas whose proofs pass the type-checker can be used in further proof developments.

Chapter 30

Library Gen

30.1 Gen: Generalizing Induction Hypotheses

```
Require Export Poly.
```

In the previous chapter, we noticed the importance of controlling 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. In this short chapter, we consider this point in a little more depth and introduce one new tactic, called `generalize dependent`, that is sometimes useful in helping massage the induction hypothesis into the required form.

First, let's review the basic issue. Suppose we want to show that the *double* function is injective – i.e., that it always maps different arguments to different results. The way we *start* this proof is a little bit delicate: if we begin it 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 : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction n as [| n'].
  Case "n = O". simpl. intros eq. destruct m as [| m'].
    SCase "m = O". reflexivity.
    SCase "m = S m'". inversion eq.
  Case "n = S n'". intros eq. destruct m as [| m'].
    SCase "m = O". inversion eq.
    SCase "m = S m'".
      assert (n' = m') as H.
      SSCase "Proof of assertion".
      Admitted.
```

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 *this 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 that the proposition

- $P\ n =$ "if $double\ n = double\ m$, then $n = m$"

holds for all n by showing

- $P\ O$

  (i.e., "if $double\ O = double\ m$ then $O = m$")

- $P\ n \to P\ (S\ n)$

  (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 $double\ (S\ n) = double\ m$ 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 can prove

- $Q =$ "if $double\ n = 10$ then $n = 5$"

then we can prove

- $R =$ "if $double\ (S\ n) = 10$ then $S\ n = 5$".

But knowing Q doesn't give us any help with proving R! (If we tried to prove R from Q, we would say something like "Suppose $double\ (S\ n) = 10$..." but then we'd be stuck: knowing that $double\ (S\ n)$ is 10 tells us nothing about whether $double\ n$ is 10, so Q is useless at this point.)

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 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'].
  Case "n = O". simpl. intros m eq. destruct m as [| m'].
    SCase "m = O". reflexivity.
    SCase "m = S m'". inversion eq.
  Case "n = S n'".
    intros m eq.
    destruct m as [| m'].
    SCase "m = O". inversion eq.        SCase "m = S m'".
      assert (n' = m') as H.
      SSCase "Proof of assertion". apply IHn'.
        inversion eq. reflexivity.
      rewrite → H. reflexivity. Qed.
```

What this teaches us 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.

However, this strategy doesn't always apply directly; sometimes a little rearrangement is needed. Suppose, for example, that we had decided we wanted to prove *double_injective* by induction on m instead of n.

```
Theorem double_injective_take2_FAILED : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction m as [| m'].
  Case "m = O". simpl. intros eq. destruct n as [| n'].
    SCase "n = O". reflexivity.
    SCase "n = S n'". inversion eq.
  Case "m = S m'". intros eq. destruct n as [| n'].
    SCase "n = O". inversion eq.
    SCase "n = S n'".
      assert (n' = m') as H.
      SSCase "Proof of assertion".
Admitted.
```

The problem 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 mangle 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, taking them out of the context and putting them back at the beginning of the goal. The `generalize dependent` tactic does this.

```
Theorem double_injective_take2 : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m.
  generalize dependent n.
  induction m as [| m'].
  Case "m = O". simpl. intros n eq. destruct n as [| n'].
    SCase "n = O". reflexivity.
    SCase "n = S n'". inversion eq.
  Case "m = S m'". intros n eq. destruct n as [| n'].
    SCase "n = O". inversion eq.
    SCase "n = S n'".
      assert (n' = m') as H.
      SSCase "Proof of assertion".
        apply IHm'. inversion eq. reflexivity.
      rewrite → H. 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 this is what we wanted to show. Otherwise, if $n = S\ n'$ for some n', we derive a contradiction: by the definition of *double* we would have $n = S\ (S\ (double\ n'))$, but this contradicts the assumption that *double* $n = 0$.

- Otherwise, 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 $S\ (S\ (double\ n'))$ $= S\ (S\ (double\ m'))$, which implies by inversion that $double\ n' = double\ m'$.

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. □

Exercise: 3 stars, recommended (gen_dep_practice) Carry out this proof by induction on m.

```
Theorem plus_n_n_injective_take2 : ∀ n m,
    n + n = m + m →
    n = m.
Proof.
  Admitted.
```

Now prove this by induction on l.

```
Theorem index_after_last: ∀ (n : nat) (X : Type) (l : list X),
    length l = n →
    index (S n) l = None.
Proof.
  Admitted.
```

□

Exercise: 3 stars, optional (index_after_last_informal) Write an informal proof corresponding to your Coq proof of *index_after_last*:

Theorem: For all sets X, lists l : *list* X, and numbers n, if *length* $l = n$ then *index* $(S\ n)$ $l = None$.

Proof: □

Exercise: 3 stars (gen_dep_practice_opt) Prove this by induction on l.

```
Theorem length_snoc''' : ∀ (n : nat) (X : Type)
                              (v : X) (l : list X),
    length l = n →
    length (snoc l v) = S n.
Proof.
  Admitted.
```

□

Exercise: 3 stars (app_length_cons) Prove this by induction on $l1$, without using *app_length*.

```
Theorem app_length_cons : ∀ (X : Type) (l1 l2 : list X)
```

$(x : X)$ $(n :$ **nat**$)$,
    length $(l1$ ++ $(x :: l2)) = n \rightarrow$
    S (length $(l1$ ++ $l2)) = n$.

`Proof.`
  Admitted.
  □

Exercise: 4 stars, optional (app_length_twice) Prove this by induction on l, without using app_length.

`Theorem` app_length_twice : $\forall$ $(X$:`Type`$)$ $(n$:**nat**$)$ $(l$:**list** $X)$,
    length $l = n \rightarrow$
    length $(l$ ++ $l) = n + n$.

`Proof.`
  Admitted.
  □

Chapter 31

Library Poly

31.1 Poly: Polymorphism and Higher-Order Functions

```
Require Export Lists.
```

31.2 Polymorphism

31.2.1 Polymorphic Lists

Up to this point, we've been working with lists of numbers. Of course, interesting programs also need to be able to manipulate lists whose elements are drawn 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 `Inductive`ly 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, ∀ is usually typeset as the usual mathematical "upside down A," but you'll see the spelled-out "forall" in a few places. This is just a quirk of typesetting: there is no difference in meaning.

The "∀ X" in these types should 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 gone back to writing *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 (or "generic") versions of all the list-processing functions that we wrote before. Here is *length*, for example:

```
Fixpoint length (X:Type) (l:list X) : nat :=
  match l with
  | nil ⇒ 0
  | cons h t ⇒ S (length X t)
  end.
```

Note that the uses of *nil* and *cons* in `match` patterns do not require any type annotations: we already know that the list l contains elements of type X, so there's no reason to include X in the pattern. (More formally, the type X is a parameter of the whole definition of *list*, not of the individual constructors. We'll come back to this point later.)

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

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

To use our length with other kinds of lists, we simply instantiate it with an appropriate type parameter:

```
Example test_length2 :
    length bool (cons bool true (nil bool)) = 1.
Proof. reflexivity. Qed.
```

Let's close this subsection 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 X h (app X t l2)
  end.

Fixpoint snoc (X:Type) (l:list X) (v:X) : (list X) :=
  match l with
  | nil ⇒ cons X v (nil X)
  | cons h t ⇒ cons X h (snoc X t v)
  end.

Fixpoint rev (X:Type) (l:list X) : list X :=
  match l with
  | nil ⇒ nil X
  | cons h t ⇒ snoc X (rev X t) h
  end.

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

Example test_rev2:
  rev bool (nil bool) = nil bool.
Proof. reflexivity. Qed.
```

Type Annotation Inference

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

```
Fixpoint app' X l1 l2 : list X :=
  match l1 with
  | nil ⇒ l2
  | cons h t ⇒ cons X h (app' X t l2)
  end.
```

Indeed it will. Let's see what type Coq has assigned to *app'*:

```
Check app'.
```

```
Check app.
```

It has exactly the same type type as *app*. Coq was able to use a process called *type inference* to deduce what the types of X, *l1*, and *l2* 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 *l1* with *nil* and *cons* means it must be a *list*; 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. You should try to find a balance in your own code between too many type annotations (so many that they 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

Whenever 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 *length* function above must pass along the type X. But just like providing explicit type annotations everywhere, this is heavy and verbose. Since the second argument to *length* is a list of Xs, 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 figure out for yourself what type 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 – and, indeed, the two procedures rely on the same underlying mechanisms. Instead of simply omitting the types of some arguments to a function, like app' X l1 l2 : list X := we can also replace the types with _, like app' (X : _) (l1 l2 : _) : list X := which tells Coq to attempt to infer the missing information, just as with argument synthesis.

Using implicit arguments, the *length* function can be written like this:

```
Fixpoint length' (X:Type) (l:list X) : nat :=
  match l with
  | nil ⇒ 0
  | cons h t ⇒ S (length' _ t)
  end.
```

In this instance, we don't save much by writing _ instead of X. But in many cases the difference can be significant. 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

If fact, we can go further. To avoid having to sprinkle _'s throughout our programs, we can tell Coq *always* to infer the type argument(s) of a given function.

```
Implicit Arguments nil [[X]].
Implicit Arguments cons [[X]].
Implicit Arguments length [[X]].
Implicit Arguments app [[X]].
Implicit Arguments rev [[X]].
Implicit Arguments snoc [[X]].
```

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

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

```
Fixpoint length'' {X:Type} (l:list X) : nat :=
  match l with
  | nil ⇒ 0
  | cons h t ⇒ S (length'' t)
  end.
```

(Note that we didn't even have to provide a type argument to the recursive call to *length''*.) We will use this style whenever possible, although we will continue to use use explicit `Implicit` *Argument* declarations for `Inductive` constructors.

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 this time, even though we've globally declared it to be `Implicit`. For example, if we write:

If we uncomment this definition, Coq will give 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: Polymorphic Lists

Exercise: 2 stars, optional (poly_exercises) Here are a few simple exercises, just like ones in the *Lists* chapter, for practice with polymorphism. Fill in the definitions and complete the proofs below.

```
Fixpoint repeat (X : Type) (n : X) (count : nat) : list X :=
  admit.

Example test_repeat1:
  repeat bool true 2 = cons true (cons true nil).
   Admitted.

Theorem nil_app : ∀ X:Type, ∀ l:list X,
  app [] l = l.
Proof.
   Admitted.

Theorem rev_snoc : ∀ X : Type,
                     ∀ v : X,
                     ∀ s : list X,
  rev (snoc s v) = v :: (rev s).
Proof.
   Admitted.

Theorem rev_involutive : ∀ X : Type, ∀ l : list X,
  rev (rev l) = l.
Proof.
   Admitted.

Theorem snoc_with_append : ∀ X : Type,
                             ∀ l1 l2 : list X,
                             ∀ v : X,
  snoc (l1 ++ l2) v = l1 ++ (snoc l2 v).
Proof.
   Admitted.
```

□

31.2.2 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* (or *products*):

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

```
Implicit 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 pair *types*:

```
Notation "X * Y" := (prod X Y) : type_scope.
```

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

A note of caution: it is easy at first to get (x,y) and $X \times Y$ confused. Remember that (x,y) is a *value* built from two other values; $X \times Y$ is a *type* built from two other types. If x has type X and y has type Y, then (x,y) has type $X \times Y$.

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 many functional programming languages, it is called *zip*. We call it *combine* for consistency with Coq's standard library. Note that the pair notation can be used both in expressions and in patterns...

```
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.
```

Indeed, when no ambiguity results, we can even drop the enclosing parens:

```
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 Eval simpl in (combine 1,2 *false*,*false*,*true*,*true*). print? □

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 programing languages, this function is called *unzip*.

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

(If you're reading the HTML version of this file, note that there's an unresolved typesetting problem in the example: several square brackets are missing. Refer to the .v file for the correct version. □

31.2.3 Polymorphic Options

One last polymorphic type for now: *polymorphic options*. The type declaration generalizes the one for *natoption* in the previous chapter:

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

Implicit Arguments Some [[X]].
Implicit Arguments None [[X]].
```

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

```
Fixpoint index {X : Type} (n : nat)
               (l : list X) : option X :=
  match l with
  | [] ⇒ None
  | a :: l' ⇒ if beq_nat n O then Some a else index (pred n) l'
  end.

Example test_index1 : index 0 [4,5,6,7] = Some 4.
Proof. reflexivity. Qed.
Example test_index2 : index 1 [[1],[2]] = Some [2].
Proof. reflexivity. Qed.
Example test_index3 : index 2 [true] = None.
Proof. reflexivity. Qed.
```

Exercise: 1 star, optional (hd_opt_poly) Complete the definition of a polymorphic version of the *hd_opt* function from the last chapter. Be sure that it passes the unit tests below.

```
Definition hd_opt {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_opt.

Example test_hd_opt1 : hd_opt [1,2] = Some 1.
  Admitted.
Example test_hd_opt2 : hd_opt [[1],[2]] = Some [1].
  Admitted.
```

☐

31.3 Functions as Data

31.3.1 Higher-Order Functions

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

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.
```

31.3.2 Partial Application

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 → in this expression is actually a *binary* operator on types. (This is the same as saying that Coq primitively supports only one-argument functions – do you see why?) This operator is *right-associative*, so the type of *plus* is really a shorthand for *nat* → (*nat* → *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.
```

31.3.3 Digression: Currying

Exercise: 2 stars, optional (currying) In Coq, a function $f : A \to B \to C$ really has the type $A \to (B \to C)$. That is, if you give f a value of type A, it will give you function *f'* : $B \to C$. 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 $A \to B \to C$ as $(A \times B) \to C$. 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 : Type}
  (f : X × Y → 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 : Type}
  (f : X → Y → Z) (p : X × Y) : Z :=
  admit.
```

(Thought exercise: before running these commands, can you calculate the types of *prod_curry* and *prod_uncurry*?)

```
Check @prod_curry.
Check @prod_uncurry.
```

```
Theorem uncurry_curry : ∀ (X Y Z : Type) (f : X → Y → Z) x y,
  prod_curry (prod_uncurry f) x y = f x y.
```

```
Proof.
  Admitted.

Theorem curry_uncurry : ∀ (X Y Z : Type)
                               (f : (X × Y) → Z) (p : X × Y),
  prod_uncurry (prod_curry f) p = f p.
Proof.
  Admitted.
  □
```

31.3.4 Filter

Here is a useful higher-order function, which takes a list of Xs and a *predicate* on X (a function from X to *bool*) and "filters" 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.
```

31.3.5 Anonymous Functions

It is a little annoying 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. It is also possible to construct a function "on the fly" without declaring it at the top level or giving it a name; this is analogous to the notation we've been using for writing down constant lists, natural numbers, and so on.

```
Example test_anon_fun':
  doit3times (fun n ⇒ n × n) 2 = 256.
Proof. reflexivity. Qed.
```

Here is the motivating example from before, 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* which takes a list of natural numbers as input and keeps only those numbers which 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 $X \rightarrow$ *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 ⇒ false) [5,9,0] = ([], [5,9,0]).
  Admitted.
```

☐

31.3.6 Map

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

```
Fixpoint map {X Y:Type} (f:X→Y) (l:list X)
             : (list Y) :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ (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 (plus 3) [2,0,2] = [5,3,5].
Proof. reflexivity. Qed.
```

The element types of the input and output lists need not be the same (*map* takes *two* type arguments, X and Y). This version of *map* 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 ⇒ [evenb n,oddb n]) [2,1,2,5]
  = [[true,false],[false,true],[true,false],[false,true]].
Proof. reflexivity. Qed.
```

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

```
Theorem map_rev : ∀ (X Y : Type) (f : X → 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 $X \to Y$. We can define a similar function, *flat_map*, which maps a *list X* to a *list Y* using a function *f* of type $X \to$ *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. □

31.3.7 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 $1 + (2 + (3 + (4 + 0)))$. Here are some more examples:

```
Check (fold plus).
Eval simpl in (fold plus [1,2,3,4] 0).

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, optional (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?

31.3.8 Functions For Constructing Functions

Most of the higher-order functions we have talked about so far take functions as *arguments*. Now let's look at some examples involving *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 *nat* to X that yields x whenever it is called, ignoring its *nat* argument.

```
Definition constfun {X: Type} (x: X) : nat→X :=
  fun (k:nat) ⇒ x.

Definition ftrue := constfun true.

Example constfun_example1 : ftrue 0 = true.
Proof. reflexivity. Qed.

Example constfun_example2 : (constfun 5) 99 = 5.
Proof. reflexivity. Qed.
```

Similarly, but a bit more interestingly, here is a function that takes a function f from numbers to some type X, a number k, and a value x, and constructs a function that behaves exactly like f except that, when called with the argument k, it returns x.

```
Definition override {X: Type} (f: nat→X) (k:nat) (x:X) : nat→X:=
```

fun (k':**nat**) ⇒ if beq_nat k k' then x else f k'.

For example, we can apply *override* twice to obtain a function from numbers to booleans that returns *false* on 1 and 3 and returns *true* on all other arguments.

```
Definition fmostlytrue := override (override ftrue 1 false) 3 false.

Example override_example1 : fmostlytrue 0 = true.
Proof. reflexivity. Qed.

Example override_example2 : fmostlytrue 1 = false.
Proof. reflexivity. Qed.

Example override_example3 : fmostlytrue 2 = true.
Proof. reflexivity. Qed.

Example override_example4 : fmostlytrue 3 = false.
Proof. reflexivity. Qed.
```

Exercise: 1 star (override_example) Before starting to work on the following proof, make sure you understand exactly what the theorem is saying and can paraphrase it in your own words. The proof itself is straightforward.

```
Theorem override_example : ∀ (b:bool),
  (override (constfun b) 3 true) 2 = b.
Proof.
  Admitted.
```

□

We'll use function overriding heavily in parts of the rest of the course, and we will end up needing to know quite a bit about its properties. To prove these properties, though, we need to know about a few more of Coq's tactics; developing these is the main topic of the rest of the chapter.

31.4 Optional Material

31.4.1 Non-Uniform Inductive Families (GADTs)

This section needs more text!

Recall the definition of lists of booleans: Inductive boollist : Type := boolnil : boollist | boolcons : bool -> boollist -> boollist.

We saw how it could be generalized to "polymorphic lists" with elements of an arbitrary type X. Here's another way of generalizing it: an inductive family of "length-indexed" lists of booleans:

```
Inductive boolllist : nat → Type :=
  boollnil : boolllist O
| boollcons : ∀ n, bool → boolllist n → boolllist (S n).
```

```
Implicit Arguments boollcons [[n]].

Check (boollcons true (boollcons false (boollcons true boollnil))).

Fixpoint blapp {n1} (l1: boolllist n1)
               {n2} (l2: boolllist n2)
             : boolllist (n1 + n2) :=
  match l1 with
  | boollnil ⇒ l2
  | boollcons _ h t ⇒ boollcons h (blapp t l2)
  end.
```

Of course, these generalizions can be combined. Here's the length-indexed polymorphic version:

```
Inductive llist (X:Type) : nat → Type :=
  lnil : llist X O
| lcons : ∀ n, X → llist X n → llist X (S n).

Implicit Arguments lnil [[X]].
Implicit Arguments lcons [[X] [n]].

Check (lcons true (lcons false (lcons true lnil))).

Fixpoint lapp (X:Type)
              {n1} (l1: llist X n1)
              {n2} (l2: llist X n2)
            : llist X (n1 + n2) :=
  match l1 with
  | lnil ⇒ l2
  | lcons _ h t ⇒ lcons h (lapp X t l2)
  end.
```

31.5 More About Coq

31.5.1 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 : ∀ (n m o p : nat),
     n = m →
     [n,o] = [n,p] →
     [n,o] = [m,p].
Proof.
  intros n m o p eq1 eq2.
  rewrite ← eq1.
```

```
  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 ∀ binding 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 : ∀ (n : nat),
     true = beq_nat n 5 →
     beq_nat (S (S n)) 7 = true.
```

```
Proof.
  intros n H.
  simpl.
Admitted.
```

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

```
Theorem silly3 : ∀ (n : nat),
     true = beq_nat n 5 →
     beq_nat (S (S n)) 7 = true.
Proof.
  intros n H.
  symmetry.
  simpl.   apply H. Qed.
```

Exercise: 3 stars, recommended (apply_exercise1) `Theorem rev_exercise1 : ∀ (l l' : list nat),`

```
     l = rev l' →
     l' = rev l.
Proof.
   Admitted.
```

□

Exercise: 1 star (apply_rewrite) Briefly explain the difference between the tactics `apply` and `rewrite`. Are there situations where both can usefully be applied?

□

31.5.2 The `unfold` Tactic

Sometimes, a proof will get stuck because Coq doesn't automatically expand a function call into its definition. (This is a feature, not a bug: if Coq automatically expanded everything possible, our proof goals would quickly become enormous – hard to read and slow for Coq to manipulate!)

```
Theorem unfold_example_bad : ∀ m n,
  3 + n = m →
  plus3 n + 1 = m + 1.
Proof.
  intros m n H.
  Admitted.
```

The `unfold` tactic can be used to explicitly replace a defined name by the right-hand side of its definition.

```
Theorem unfold_example : ∀ m n,
```

```
  3 + n = m →
  plus3 n + 1 = m + 1.
Proof.
  intros m n H.
  unfold plus3.
  rewrite → H.
  reflexivity. Qed.
```

Now we can prove a first property of *override*: If we override a function at some argument k and then look up k, we get back the overridden value.

```
Theorem override_eq : ∀ {X:Type} x k (f:nat→X),
  (override f k x) k = x.
Proof.
  intros X x k f.
  unfold override.
  rewrite ← beq_nat_refl.
  reflexivity. Qed.
```

This proof was straightforward, but note that it requires `unfold` to expand the definition of *override*.

Exercise: 2 stars (override_neq) Theorem override_neq : ∀ {X:Type} x1 x2 k1 k2 (f : nat→X),

```
  f k1 = x1 →
  beq_nat k2 k1 = false →
  (override f k2 x2) k1 = x1.
Proof.
  Admitted.
```

☐

As the inverse of `unfold`, Coq also provides a tactic `fold`, which can be used to "unexpand" a definition. It is used much less often.

31.5.3 Inversion

Recall the definition of natural numbers: Inductive nat : Type := | O : nat | S : nat -> nat. It is clear 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 other facts:

- The constructor S is *injective*. That is, the only way we can have $S\ n = S\ m$ is if $n = m$.
- 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 unequal. (Since neither *true* nor *false* take any arguments, their injectivity is not an issue.)

Coq provides a tactic, called `inversion`, that allows us to exploit these principles in making proofs.

The `inversion` tactic is used like this. 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 ... an* and *b1 ... bm*. Then `inversion` H instructs Coq to "invert" this equality to extract the information it contains about these terms:

- If c and d are the same constructor, then we know, by the injectivity of this constructor, 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. That is, a false assumption has crept into the context, and this means that any goal whatsoever is provable! In this case, `inversion` H marks the current goal as completed and pops it off the goal stack.

The `inversion` tactic is probably easier to understand by seeing it in action than from general descriptions like the above. Below you will find example theorems that demonstrate the use of `inversion` and exercises to test your understanding.

```
Theorem eq_add_S : ∀ (n m : nat),
     S n = S m →
     n = m.
Proof.
  intros n m eq. inversion eq. reflexivity. Qed.

Theorem silly4 : ∀ (n m : nat),
     [n] = [m] →
     n = m.
Proof.
  intros n o eq. inversion eq. reflexivity. Qed.
```

As a convenience, the `inversion` tactic can also destruct equalities between complex values, binding multiple variables as it goes.

```
Theorem silly5 : ∀ (n m o : nat),
     [n,m] = [o,o] →
     [n] = [m].
Proof.
  intros n m o eq. inversion eq. reflexivity. Qed.
```

Exercise: 1 star (sillyex1) Example sillyex1 : $\forall$ (X : Type) (x y z : X) (l j : **list** X),

x :: y :: l = z :: j $\rightarrow$
y :: l = x :: j $\rightarrow$
x = y.

Proof.
Admitted.
☐

Theorem silly6 : $\forall$ (n : **nat**),
S n = O $\rightarrow$
2 + 2 = 5.
Proof.
intros n *contra*. inversion *contra*. Qed.

Theorem silly7 : $\forall$ (n m : **nat**),
false = true $\rightarrow$
[n] = [m].
Proof.
intros n m *contra*. inversion *contra*. Qed.

Exercise: 1 star (sillyex2) Example sillyex2 : $\forall$ (X : Type) (x y z : X) (l j : **list** X),

x :: y :: l = [] $\rightarrow$
y :: l = z :: j $\rightarrow$
x = z.

Proof.
Admitted.
☐

While the injectivity of constructors allows us to reason $\forall$ (n m : nat), S n = S m $\rightarrow$ n = m, the reverse direction of the implication, provable by standard equational reasoning, is a useful fact to record for cases we will see several times.

Lemma eq_remove_S : $\forall$ n m,
n = m $\rightarrow$ S n = S m.
Proof. intros n m *eq*. rewrite $\rightarrow$ *eq*. reflexivity. Qed.

Here's another illustration of `inversion`. This is a slightly roundabout way of stating a fact that we have already proved above. The extra equalities force us to do a little more equational reasoning and exercise some of the tactics we've seen recently.

Theorem length_snoc' : $\forall$ (X : Type) (v : X)
(l : **list** X) (n : **nat**),
length l = n $\rightarrow$
length (snoc l v) = S n.
Proof.
intros X v l. induction l as [| v' l'].
Case "l = []". intros n *eq*. rewrite $\leftarrow$ *eq*. reflexivity.

```
Case "l = v' :: l'". intros n eq. simpl. destruct n as [| n'].
  SCase "n = 0". inversion eq.
  SCase "n = S n'".
    apply eq_remove_S. apply IHl'. inversion eq. reflexivity. Qed.
```

31.5.4 Varying the Induction Hypothesis

Here is a more realistic use of inversion to prove a property that is useful in many places later on...

```
Theorem beq_nat_eq_FAILED : ∀ n m,
  true = beq_nat n m → n = m.
Proof.
  intros n m H. induction n as [| n'].
  Case "n = 0".
    destruct m as [| m'].
    SCase "m = 0". reflexivity.
    SCase "m = S m'". simpl in H. inversion H.
  Case "n = S n'".
    destruct m as [| m'].
    SCase "m = 0". simpl in H. inversion H.
    SCase "m = S m'".
      apply eq_remove_S.
      Admitted.
```

The inductive proof above fails because we've set up things so that the induction hypothesis (in the second subgoal generated by the `induction` tactic) is

true = beq_nat n' m → n' = m .

This hypothesis makes a statement about *n'* together with the *particular* natural number *m* – that is, the number *m*, which was introduced into the context by the `intros` at the top of the proof, is "held constant" in the induction hypothesis. This induction hypothesis is not strong enough to make the induction step of the proof go through.

If we set up the proof slightly differently by introducing just *n* into the context at the top, then we get an induction hypothesis that makes a stronger claim:

∀ *m* : *nat*, *true = beq_nat n' m → n' = m*

Setting up the induction hypothesis this way makes the proof of *beq_nat_eq* go through:

```
Theorem beq_nat_eq : ∀ n m,
  true = beq_nat n m → n = m.
Proof.
  intros n. induction n as [| n'].
  Case "n = 0".
    intros m. destruct m as [| m'].
    SCase "m = 0". reflexivity.
    SCase "m = S m'". simpl. intros contra. inversion contra.
```

Case "n = S n'".
  `intros` *m*. `destruct` *m* `as` [| *m'*].
  SCase "m = 0". `simpl`. `intros` *contra*. `inversion` *contra*.
  SCase "m = S m'". `simpl`. `intros` *H*.
    `apply` eq_remove_S. `apply` *IHn'*. `apply` *H*. `Qed`.

Similar issues will come up in *many* of the proofs below. If you ever find yourself in a situation where the induction hypothesis is insufficient to establish the goal, consider going back and doing fewer `intros` to make the IH stronger.

Exercise: 2 stars (beq_nat_eq_informal) Give an informal proof of *beq_nat_eq*.

□

Exercise: 3 stars (beq_nat_eq') We can also prove beq_nat_eq by induction on *m*, though we have to be a little careful about which order we introduce the variables, so that we get a general enough induction hypothesis – this is done for you below. Finish the following proof. To get maximum benefit from the exercise, try first to do it without looking back at the one above.

```
Theorem beq_nat_eq' : ∀ m n,
  beq_nat n m = true → n = m.
Proof.
  intros m. induction m as [| m'].
   Admitted.
```

□

Practice Session

Exercise: 2 stars, optional (practice) Some nontrivial but not-too-complicated proofs to work together in class, and some for you to work as exercises. Some of the exercises may involve applying lemmas from earlier lectures or homeworks.

```
Theorem beq_nat_0_l : ∀ n,
  true = beq_nat 0 n → 0 = n.
Proof.
   Admitted.
```

```
Theorem beq_nat_0_r : ∀ n,
  true = beq_nat 0 n → 0 = n.
Proof.
   Admitted.
```

□

Exercise: 3 stars (apply_exercise2) In the following proof opening, notice that we don't introduce m before performing induction. This leaves it general, so that the IH doesn't specify a particular m, but lets us pick. Finish the proof.

```
Theorem beq_nat_sym : ∀ (n m : nat),
  beq_nat n m = beq_nat m n.
Proof.
  intros n. induction n as [| n'].
   Admitted.
   □
```

Exercise: 3 stars (beq_nat_sym_informal) Provide 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: □

31.5.5 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 : ∀ (n m : nat) (b : bool),
     beq_nat (S n) (S m) = b →
     beq_nat n m = b.
Proof.
  intros n m b H. simpl in H. apply H. Qed.
```

Similarly, the tactic `apply` L `in` H matches some conditional statement L (of the form *L1* → *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* → *L2* and a hypothesis matching *L1*, it gives us a hypothesis matching *L2*. By contrast, `apply` L is "backward reasoning" – it says that if we know *L1*→*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, Coq tends to favor backward reasoning, but in some situations the forward style can be easier to use or to think about.

Exercise: 3 stars, recommended (plus_n_n_injective) You can practice using the "in" variants in this exercise.

```
Theorem plus_n_n_injective : ∀ n m,
     n + n = m + m →
     n = m.
Proof.
  intros n. induction n as [| n'].
   Admitted.
   □
```

31.5.6 Using destruct on Compound Expressions

We have seen many examples where the `destruct` tactic 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 : nat) : bool :=
  if beq_nat n 3 then false
  else if beq_nat n 5 then false
  else false.

Theorem sillyfun_false : ∀ (n : nat),
  sillyfun n = false.
Proof.
  intros n. unfold sillyfun.
  destruct (beq_nat n 3).
    Case "beq_nat n 3 = true". reflexivity.
    Case "beq_nat n 3 = false". destruct (beq_nat n 5).
      SCase "beq_nat n 5 = true". reflexivity.
      SCase "beq_nat n 5 = false". reflexivity. Qed.
```

After unfolding *sillyfun* in the above proof, we find that we are stuck on `if` (*beq_nat* n 3) `then` ... `else` .... Well, either n is equal to 3 or it isn't, so we use `destruct` (*beq_nat* n 3) to let us reason about the two cases.

Exercise: 1 star (override_shadow) `Theorem` override_shadow : ∀ {X:Type} *x1 x2 k1 k2* (f : **nat**→X),
  (override (override f *k1 x2*) *k1 x1*) *k2* = (override f *k1 x1*) *k2*.

`Proof.`

Admitted.

☐

Exercise: 3 stars, recommended (combine_split) ☐

Exercise: 3 stars, optional (split_combine) Thought exercise: We have just proven that for all lists of pairs, *combine* is the inverse of `split`. How would you state the theorem showing that `split` is the inverse of *combine*?

Hint: what property do you need of *l1* and *l2* for `split` *combine l1 l2* = (*l1*,*l2*) to be true?

State this theorem in Coq, and prove it. (Be sure to leave your induction hypothesis general by not doing `intros` on more things than necessary.)

☐

31.5.7 The *remember* Tactic

(Note: the *remember* tactic is not strictly needed until a bit later, so if necessary this section can be skipped and returned to when needed.)

We have seen how 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.

Sometimes, however, this substitution process loses information that we need in order to complete the 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.
```

And suppose that we want to convince Coq of the rather obvious observation 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).
Admitted.
```

We get stuck at this point because the context does not contain enough information to prove the goal! The problem is that the substitution peformed by `destruct` is too brutal – it threw away every occurrence of *beq_nat n* 3, but we need to keep at least one of these because we need to be able to reason that since, in this branch of the case analysis, *beq_nat n* 3 = *true*, it must be that $n = 3$, from which it follows that n is odd.

What we would really like is not to use `destruct` directly on *beq_nat n* 3 and substitute away all occurrences of this expression, but rather to use `destruct` on something else that is *equal* to *beq_nat n* 3. For example, if we had a variable that we knew was equal to *beq_nat n* 3, we could `destruct` this variable instead.

The *remember* tactic allows us to introduce such a variable.

```
Theorem sillyfun1_odd : ∀ (n : nat),
      sillyfun1 n = true →
      oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
   remember (beq_nat n 3) as e3.
   destruct e3.
     Case "e3 = true". apply beq_nat_eq in Heqe3.
       rewrite → Heqe3. reflexivity.
     Case "e3 = false".
       remember (beq_nat n 5) as e5. destruct e5.
         SCase "e5 = true".
           apply beq_nat_eq in Heqe5.
           rewrite → Heqe5. reflexivity.
         SCase "e5 = false". inversion eq. Qed.
```

Exercise: 2 stars (override_same) `Theorem override_same : ∀ {X:Type} x1 k1 k2 (f : nat→X),`

```
  f k1 = x1 →
  (override f k1 x1) k2 = f k2.
Proof.
   Admitted.
```

□

Exercise: 3 stars, optional (filter_exercise) This one is a bit challenging. Be sure your initial `intros` go only up through the parameter on which you want to do induction!

```
Theorem filter_exercise : ∀ (X : Type) (test : X → bool)
                             (x : X) (l lf : list X),
     filter test l = x :: lf →
     test x = true.
Proof.
   Admitted.
   □
```

31.5.8 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 : ∀ (a b c d e f : nat),
     [a,b] = [c,d] →
     [c,d] = [e,f] →
     [a,b] = [e,f].
Proof.
  intros a b c d e f eq1 eq2.
  rewrite → eq1. rewrite → eq2. reflexivity. Qed.
```

Since this is a common pattern, we might abstract it out as a lemma recording once and for all the fact that equality is transitive.

```
Theorem trans_eq : ∀ {X:Type} (n m o : X),
  n = m → m = o → 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' : ∀ (a b c d e f : nat),
     [a,b] = [c,d] →
     [c,d] = [e,f] →
     [a,b] = [e,f].
Proof.
  intros a b c d e f eq1 eq2.
  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, recommended (apply_exercises) `Example trans_eq_exercise : ∀ (n m o p : nat),`

```
     m = (minustwo o) →
```

```
    (n + p) = m →
    (n + p) = (minustwo o).
Proof.
  Admitted.

Theorem beq_nat_trans : ∀ n m p,
  true = beq_nat n m →
  true = beq_nat m p →
  true = beq_nat n p.
Proof.
  Admitted.

Theorem override_permute : ∀ {X:Type} x1 x2 k1 k2 k3 (f : nat→X),
  false = beq_nat k2 k1 →
  (override (override f k2 x2) k1 x1) k3 = (override (override f k1 x1) k2 x2) k3.
Proof.
  Admitted.
```

□

31.6 Review

We've now seen a bunch of Coq's fundamental tactics – enough to do pretty much everything we'll want for a while. We'll introduce one or two more as we go along through the next few lectures, and later in the course we'll introduce some more powerful *automation* tactics that make Coq do more of the low-level work in many cases. 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
- `induction... as...`: induction on values of inductively defined types
- `inversion`: reason by injectivity and distinctness of constructors
- *remember* (e) `as` x: give a name (x) to an expression (e) so that we can destruct x without "losing" e
- `assert` (e) `as` H: introduce a "local lemma" e and call it H

31.7 Additional Exercises

Exercise: 2 stars, optional (fold_length) Many common functions on lists can be implemented in terms of `fold`. For example, here is an alternate 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 : ∀ X (l : list X),
  fold_length l = length l.
  Admitted.
```

□

Exercise: 3 stars, recommended (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 in Coq stating that *fold_map* is correct, and prove it.

□

```
Module MumbleBaz.
```

Exercise: 2 stars, optional (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*

□

Exercise: 2 stars, optional (baz_num_elts) Consider the following inductive definition:

```
Inductive baz : Type :=
  | x : baz → baz
  | y : baz → bool → baz.
```

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

End MumbleBaz.

Exercise: 4 stars, recommended (forall_exists_challenge) Challenge problem: 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 function *existsb* 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, create a *nonrecursive* `Definition`, *existsb'*, using *forallb* and *negb*.

Prove that *existsb'* and *existsb* have the same behavior.

□

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

Chapter 32

Library Lists

32.1 Lists: Working with Structured Data

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:

- CoqIDE:

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

- Command line:

  Run *coqc Basics.v*

In this file, we again use the `Module` feature to wrap all of the definitions for pairs and lists of numbers in a module so that, later, we can reuse the same names for improved (generic) versions of the same operations.

```
Module NatList.
```

32.2 Pairs of Numbers

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

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

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

We can construct an element of *natprod* like this:

```
Eval simpl in (pair 3 5).
```

Here are two simple function definitions 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.

Eval simpl in (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 notation is provided as part of the standard library):

```
Eval simpl in (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 and prove a few simple facts about pairs. If we state the lemmas in a particular (and slightly peculiar) way, we can prove them 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. Admitted.
```

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`.

Notice that, unlike for *nat*s, `destruct` doesn't generate an extra subgoal here. That's because *natprod*s can only be constructed in one way.

```
Theorem surjective_pairing : ∀ (p : natprod),
  p = (fst p, snd p).
Proof.
  intros p. destruct p as (n,m). simpl. reflexivity. Qed.
```

Notice that Coq allows us to use the notation we introduced for pairs in the "`as`..." pattern that tells it what variables to bind.

Exercise: 1 star (snd_fst_is_swap)

```
Theorem snd_fst_is_swap : ∀ (p : natprod),
  (snd p, fst p) = swap_pair p.
Proof.
  Admitted.
```

□

Exercise: 1 star, optional (fst_swap_is_snd)

```
Theorem fst_swap_is_snd : ∀ (p : natprod),
  fst (swap_pair p) = snd p.
Proof.
  Admitted.
```

□

32.3 Lists of Numbers

Generalizing the definition of pairs a little, 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 two 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 fully understand 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 are there to instruct the "coqdoc" tool that the bracketed part should be displayed as Coq code rather than running text. These brackets don't appear in the generated HTML.)

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.

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
  | O ⇒ nil
  | S count' ⇒ n :: (repeat n count')
  end.
```

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

```
Fixpoint length (l:natlist) : nat :=
  match l with
  | nil ⇒ O
  | h :: t ⇒ S (length t)
```

```
  end.
```

The *app* ("append") function concatenates 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.
```

Here are two smaller examples of programming with lists. The *hd* function returns the first element (the "head") of the list, while *tail* returns everything but the first element. 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 => default
  | h :: t => h
  end.
Definition tail (l:natlist) : natlist :=
  match l with
  | nil => nil
  | h :: t => 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_tail: tail [1,2,3] = [2,3].
Proof. reflexivity. Qed.
```

Exercise: 2 stars, recommended (list_funs) Complete the definitions of *nonzeros*, *oddmembers* and *countoddmembers* below.

```
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, recommended (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.
```

□

32.3.1 Bags via Lists

A *bag* (or *multiset*) is like a set, but each element can appear multiple times instead of just once. One reasonable implementation of bags 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:nat) (s: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:nat) (s: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:bag) (s2: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 about bags involving the functions *count* and *add*, and prove it. 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!

☐

32.4 Reasoning About Lists

Just 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 : ∀ l:natlist,
  [] ++ l = l.
Proof.
   reflexivity. Qed.
```

... because the [] is substituted into the match position 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 : ∀ l:natlist,
  pred (length l) = length (tail l).
Proof.
  intros l. destruct l as [| n l'].
  Case "l = nil".
    reflexivity.
  Case "l = cons n l'".
    reflexivity. Qed.
```

Here, the *nil* case works because we've chosen to define *tail 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.

32.4.1 Micro-Sermon

Simply reading example proofs will not get you very far! It is very important to work through the details of each one, using Coq and thinking about what each step of the proof achieves. Otherwise it is more or less guaranteed that the exercises will make no sense.

32.4.2 Induction on Lists

Proofs by induction over datatypes like *natlist* are perhaps a little less familiar than standard natural number induction, but the basic idea is equally simple. Each `Inductive` declaration defines a set of data values that can be built up from the declared constructors: a boolean can be either *true* or *false*; a number can be either *O* or *S* applied to a 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 things together establish the truth of P for all lists l. Here's a concrete example:

```
Theorem app_ass : ∀ l1 l2 l3 : natlist,
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
Proof.
  intros l1 l2 l3. induction l1 as [| n l1'].
  Case "l1 = nil".
    reflexivity.
  Case "l1 = cons n l1'".
    simpl. rewrite → IHl1'. reflexivity. Qed.
```

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.

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. □

Here is an exercise to be worked together in class:

```
Theorem app_length : ∀ l1 l2 : natlist,
  length (l1 ++ l2) = (length l1) + (length l2).
Proof.
  intros l1 l2. induction l1 as [| n l1'].
  Case "l1 = nil".
```

```
    reflexivity.
  Case "l1 = cons".
    simpl. rewrite → IHl1'. reflexivity. Qed.
```

For a slightly more involved example of an inductive proof over lists, suppose we define a "cons on the right" function *snoc* like this...

```
Fixpoint snoc (l:natlist) (v:nat) : natlist :=
  match l with
  | nil ⇒ [v]
  | h :: t ⇒ h :: (snoc t v)
  end.
```

... and use it to define a list-reversing function *rev* like this:

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ snoc (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.
```

Now let's prove some more list theorems using our newly defined *snoc* and *rev*. For something a little more challenging than the inductive proofs we've seen so far, let's prove that reversing a list does not change its length. Our first attempt at this proof gets stuck in the successor case...

```
Theorem rev_length_firsttry : ∀ l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l'].
  Case "l = []".
    reflexivity.
  Case "l = n :: l'".
    simpl.
    rewrite ← IHl'.
Admitted.
```

So let's take the equation about *snoc* that would have enabled us to make progress and prove it as a separate lemma.

```
Theorem length_snoc : ∀ n : nat, ∀ l : natlist,
  length (snoc l n) = S (length l).
Proof.
  intros n l. induction l as [| n' l'].
```

```
  Case "l = nil".
    reflexivity.
  Case "l = cons n' l'".
    simpl. rewrite → IHl'. reflexivity. Qed.
```

Now we can complete the original proof.

```
Theorem rev_length : ∀ l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l'].
  Case "l = nil".
    reflexivity.
  Case "l = cons".
    simpl. rewrite → length_snoc.
    rewrite → IHl'. reflexivity. Qed.
```

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

Theorem: For all numbers n and lists l, $length\ (snoc\ l\ n) = S\ (length\ l)$.

Proof: By induction on l.

- First, suppose $l =$ []. We must show length (snoc ☐ n) = S (length ☐), which follows directly from the definitions of *length* and *snoc*.

- Next, suppose $l = n'::l'$, with length (snoc l' n) = S (length l'). We must show length (snoc (n' :: l') n) = S (length (n' :: l')). By the definitions of *length* and *snoc*, this follows from S (length (snoc l' n)) = S (S (length l')),

]] which is immediate from the induction hypothesis. ☐

Theorem: For all lists l, $length\ (rev\ l) = length\ l$.

Proof: By induction on l.

- First, suppose $l =$ []. We must show length (rev ☐) = length ☐, which follows directly from the definitions of *length* and *rev*.

- Next, suppose $l = n::l'$, with length (rev l') = length l'. We must show length (rev (n :: l')) = length (n :: l'). By the definition of *rev*, this follows from length (snoc (rev l') n) = S (length l') which, by the previous lemma, is the same as S (length (rev l')) = S (length l'). This is immediate from the induction hypothesis. ☐

Obviously, the style of these proofs is rather longwinded and pedantic. After the first few, we might find it easier to follow proofs that give a little less detail overall (since we can easily work them 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 more like this:

Theorem: For all lists l, $length\ (rev\ l) = length\ l$.

Proof: First, observe that length (snoc l n) = S (length l) for any *l*. This follows by a straightforward induction on *l*. The main property now follows by another straightforward induction on *l*, using the observation together with the induction hypothesis in the case where *l* = *n'*::*l'*. □

Which style is preferable in a given situation depends on the sophistication of the expected audience and on 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 present purposes.

32.4.3 `SearchAbout`

We've seen that proofs can make use of other theorems we've already proved, using `rewrite`, and later we will see other ways of reusing previous theorems. But in order to refer to a theorem, we need to know its name, and remembering the names of all the theorems we might ever want to use can become quite difficult! It is often hard even to remember what theorems have been proven, much less what they are named.

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 course; it can save you a lot of time!

Also, 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-*;.

32.4.4 List Exercises, Part 1

Exercise: 3 stars, recommended (list_exercises) More practice with lists.

```
Theorem app_nil_end : ∀ l : natlist,
  l ++ [] = l.
Proof.
  Admitted.

Theorem rev_involutive : ∀ l : natlist,
  rev (rev l) = l.
Proof.
  Admitted.
```

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

```
Theorem app_ass4 : ∀ l1 l2 l3 l4 : natlist,
  l1 ++ (l2 ++ (l3 ++ l4)) = ((l1 ++ l2) ++ l3) ++ l4.
Proof.
  Admitted.

Theorem snoc_append : ∀ (l:natlist) (n:nat),
```

```
  snoc l n = l ++ [n].
Proof.
  Admitted.

Theorem distr_rev : ∀ l1 l2 : natlist,
  rev (l1 ++ l2) = (rev l2) ++ (rev l1).
Proof.
  Admitted.
```

An exercise about your implementation of *nonzeros*:

```
Lemma nonzeros_length : ∀ l1 l2 : natlist,
  nonzeros (l1 ++ l2) = (nonzeros l1) ++ (nonzeros l2).
Proof.
  Admitted.
```

☐

32.4.5 List Exercises, Part 2

Exercise: 2 stars, recommended (list_design) Design exercise:

- Write down a non-trivial theorem involving *cons* (::), *snoc*, and *append* (++).
- Prove it.

☐

Exercise: 2 stars, optional (bag_proofs) If you did the optional exercise about bags above, here are a couple of little theorems to prove about your definitions.

```
Theorem count_member_nonzero : ∀ (s : bag),
  ble_nat 1 (count 1 (1 :: s)) = true.
Proof.
  Admitted.
```

The following lemma about *ble_nat* might help you in the next proof.

```
Theorem ble_n_Sn : ∀ n,
  ble_nat n (S n) = true.
Proof.
  intros n. induction n as [| n'].
  Case "0".
    simpl. reflexivity.
  Case "S n'".
    simpl. rewrite IHn'. reflexivity. Qed.

Theorem remove_decreases_count: ∀ (s : bag),
  ble_nat (count 0 (remove_one 0 s)) (count 0 s) = true.
```

```
Proof.
  Admitted.
  □
```

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

□

Exercise: 4 stars, optional (rev_injective) Prove that the *rev* function is injective, that is,

forall (l1 l2 : natlist), rev l1 = rev l2 -> l1 = l2.

There is a hard way and an easy way to solve this exercise.

□

32.5 Options

Here is another type definition that is often useful in day-to-day programming:

```
Inductive natoption : Type :=
  | Some : nat → natoption
  | None : natoption.
```

One use of *natoption* is as a way of returning "error codes" from functions. For example, suppose we want to write a function that returns the nth element of some list. If we give it type *nat* → *natlist* → *nat*, then we'll have to return some number when the list is too short!

```
Fixpoint index_bad (n:nat) (l:natlist) : nat :=
  match l with
  | nil ⇒ 42
  | a :: l' ⇒ match beq_nat n O with
               | true ⇒ a
               | false ⇒ index_bad (pred n) l'
               end
  end.
```

On the other hand, if we give it type *nat* → *natlist* → *natoption*, then we can return *None* when the list is too short and *Some a* when the list has enough members and a appears at position n.

```
Fixpoint index (n:nat) (l:natlist) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ match beq_nat n O with
               | true ⇒ Some a
               | false ⇒ index (pred n) l'
```

```
        end
  end.
Example test_index1 : index 0 [4,5,6,7] = Some 4.
Proof. reflexivity. Qed.
Example test_index2 : index 3 [4,5,6,7] = Some 7.
Proof. reflexivity. Qed.
Example test_index3 : index 10 [4,5,6,7] = None.
Proof. reflexivity. Qed.
```

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

```
Fixpoint index' (n:nat) (l:natlist) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ if beq_nat n O then Some a else index' (pred n) l'
  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_opt) 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_opt (l : natlist) : natoption :=
  admit.
Example test_hd_opt1 : hd_opt [] = None.
  Admitted.
Example test_hd_opt2 : hd_opt [1] = Some 1.
  Admitted.
Example test_hd_opt3 : hd_opt [5,6] = Some 5.
  Admitted.
```

☐

Exercise: 1 star, optional (option_elim_hd) This exercise relates your new *hd_opt* to the old *hd*.

```
Theorem option_elim_hd : ∀ (l:natlist) (default:nat),
  hd default l = option_elim default (hd_opt l).
Proof.
   Admitted.
   □
```

Exercise: 2 stars, recommended (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 : ∀ l:natlist,
  true = beq_natlist l l.
Proof.
   Admitted.
   □
```

32.6 Extended Exercise: Dictionaries

As a final illustration of how fundamental data structures can be defined in Coq, here is the declaration of a simple *dictionary* data type, using numbers for both the keys and the values stored under these keys. (That is, a dictionary represents a finite map from numbers to numbers.)

```
Module DICTIONARY.
Inductive dictionary : Type :=
  | empty : dictionary
  | record : nat → nat → dictionary → dictionary.
```

This declaration can be read: "There are two ways to construct a *dictionary*: either using the constructor *empty* to represent an empty dictionary, or by applying the constructor *record* to a key, a value, and an existing *dictionary* to construct a *dictionary* with an additional key to value mapping."

```
Definition insert (key value : nat) (d : dictionary) : dictionary :=
```

(record *key value d*).

Here is a function *find* that searches a *dictionary* for a given key. It evaluates evaluates to *None* if the key was not found and *Some val* if the key was mapped to *val* in the dictionary. If the same key is mapped to multiple values, *find* will return the first one it finds.

```
Fixpoint find (key : nat) (d : dictionary) : natoption :=
  match d with
  | empty ⇒ None
  | record k v d' ⇒ if (beq_nat key k) then (Some v) else (find key d')
  end.
```

Exercise: 1 star (dictionary_invariant1) Complete the following proof.

```
Theorem dictionary_invariant1 : ∀ (d : dictionary) (k v: nat),
  (find k (insert k v d)) = Some v.
Proof.
  Admitted.
```

□

Exercise: 1 star (dictionary_invariant2) Complete the following proof.

```
Theorem dictionary_invariant2 : ∀ (d : dictionary) (m n o: nat),
  (beq_nat m n) = false → (find m d) = (find m (insert n o d)).
Proof.
  Admitted.
```

□

End DICTIONARY.

End NATLIST.

Chapter 33

Library Basics

33.1 Basics: Functional Programming

33.2 Enumerated Types

In Coq's programming language, almost nothing is built in – not even booleans or numbers! Instead, it provides powerful tools for defining new types of data and functions that process and transform them.

33.2.1 Days of the Week

Let's start with a very simple example. The following declaration tells Coq that we are defining a new set of data values – a "type." The type is called *day*, and its members are *monday*, *tuesday*, etc. The lines of the definition can be read "*monday* is a *day*, *tuesday* is a *day*, etc."

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

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 work out these types even if they are not given explicitly – i.e., it performs some *type inference* – but we'll always 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 `Eval simpl` to evaluate a compound expression involving *next_weekday*.

```
Eval simpl in (next_weekday friday).
Eval simpl in (next_weekday (next_weekday saturday)).
```

If you have a computer handy, now 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.

The keyword `simpl` ("simplify") tells Coq precisely how to evaluate the expression we give it. For the moment, `simpl` is the only one we'll need; later on we'll see some alternatives that are sometimes useful.

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 are the same after simplification."

Third, we can ask Coq to "extract," from a `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 won't have space to dig further into this topic, but more information can be found in the Coq'Art book by Bertot and CastÃĺran, as well as the Coq reference manual.

33.2.2 Booleans

In a similar way, we can define the 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 the syntax for multi-argument function definitions.

The following four "unit tests" 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.
```

A note on notation: We use square brackets to delimit fragments of Coq code in comments in .v files; 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 following bit of Coq hackery defines a magic value called *admit* that can fill a hole in an incomplete definition or proof. We'll use it in the definition of *nandb* in the following exercise. In general, your job in the exercises is to replace *admit* or *Admitted* with real definitions or proofs.

```
Definition admit {T: Type} : T. Admitted.
```

Exercise: 1 star (nandb) Complete the definition of the following function, then make sure that the `Example` assertions below each can be verified by Coq.

This function should return *true* if either or both of its inputs are *false*.

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

Remove "*Admitted.*" and fill in each proof with "`Proof. simpl. reflexivity. Qed.`"

```
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)

```
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.
```

☐

33.2.3 Function Types

The `Check` command causes 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*."

33.2.4 Numbers

Technical digression: Coq provides a fairly fancy module system, to aid in organizing large developments. In this course, we won't need most of its features, but one of them 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`, all these definitions will be referred to by names like *X.foo* instead of just *foo*. This means that the new definition will not clash with the unqualified name *foo* later, which would otherwise be an error (a name can only be defined once in a given scope). Here, we use this feature to introduce the definition of the type *nat* in an inner module so that it does not shadow the one from the standard library.

```
Module Playground1.
```

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 (*weekday*, *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*.

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 $S\ (S\ \mathit{false})$ do not.

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

```
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)))).
Eval simpl in (minustwo 4).
```

The constructor S has the type $nat \to nat$, 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: functions like *pred* and *minustwo* come with *computation rules* – e.g., the definition of *pred* says that *pred n* can be simplified to `match` n `with` $\mid O \Rightarrow O \mid S\ m' \Rightarrow m'$ `end` – while the definition of S has no such behavior attached. Although it is 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, pure 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
  | O ⇒ true
```

```
  | S O ⇒ false
  | S (S n') ⇒ evenb n'
  end.
```

When Coq checks this definition, it notes that *evenb* 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 *evenb* will eventually terminate. Coq demands that some argument of *every* `Fixpoint` definition is decreasing.

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

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

```
Example test_oddb1: (oddb (S O)) = true.
Proof. simpl. reflexivity. Qed.
Example test_oddb2: (oddb (S (S (S (S O))))) = false.
Proof. simpl. reflexivity. Qed.
```

Naturally, we can also define multi-argument functions by recursion. (Once again, we use a module to avoid polluting the namespace.)

```
Module Playground2.
```

```
Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

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

```
Eval simpl in (plus (S (S (S O))) (S (S O))).
```

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
    | O ⇒ O
    | S n' ⇒ plus m (mult n' m)
  end.
```

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.

Example test_mult1: (mult 3 3) = 9.
Proof. simpl. reflexivity. Qed.
```

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

```
    factorial(0)  =  1
    factorial(n)  =  n * factorial(n-1)     (if n>0)
```

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).
```

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 $x + y$.

Each notation-symbol in Coq is active in a *notation scope*. Coq tries to guess what scope you mean, so when you write $S(O \times O)$ it guesses *nat_scope*, but when you write the cartesian product (tuple) type *bool*×*bool* it guesses *type_scope*. Occasionally you have to help it out with percent-notation by writing $(x \times y)\%nat$, 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 O, or $0\%Z$ which means the Integer zero.

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 *nat*ural numbers for *eq*uality, yielding a *b*oolean. Note the use of nested `match`es (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.
```

Similarly, the *ble_nat* function tests *nat*ural numbers for *l*ess-or-*e*qual, yielding a *b*oolean.

```
Fixpoint ble_nat (n m : nat) : bool :=
  match n with
  | O ⇒ true
  | S n' ⇒
      match m with
      | O ⇒ false
      | S m' ⇒ ble_nat n' m'
      end
  end.
Example test_ble_nat1: (ble_nat 2 2) = true.
Proof. simpl. reflexivity. Qed.
Example test_ble_nat2: (ble_nat 2 4) = true.
Proof. simpl. reflexivity. Qed.
Example test_ble_nat3: (ble_nat 4 2) = false.
```

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

Exercise: 2 stars (blt_nat) The *blt_nat* function tests *nat*ural numbers for *l*ess-*t*han, yielding a *b*oolean. Instead of making up a new `Fixpoint` for this one, define it in terms of a previously defined function.

Note: If you have trouble with the `simpl` tactic, try using `compute`, which is like `simpl` on steroids. However, there is a simple, elegant solution for which `simpl` suffices.

```
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.
```

☐

33.3 Proof By Simplification

Now that we've defined a few datatypes and functions, let's turn to the question of how to state and prove properties of their behavior. Actually, in a sense, 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 the function's definition to simplify the expressions on both sides of the $=$ and notice that they become identical.

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, since the definition of $+$ is recursive in its first argument.

```
Theorem plus_O_n : ∀ n:nat, 0 + n = n.
Proof.
  simpl. reflexivity. Qed.
```

The `reflexivity` command implicitly simplifies both sides of the equality before testing to see if they are the same, so we can shorten the proof a little. (It will be useful later to know that `reflexivity` actually does somwhat more than `simpl` – for example, it tries "unfolding" defined terms, replacing them with their right-hand sides. The reason for this difference is that, when reflexivity succeeds, the whole goal is finished and we don't need to look at whatever expanded expressions `reflexivity` has found; by contrast, `simpl` is used in situations where we may have to read and understand the new goal, so we would not want it blindly expanding definitions.)

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

The form of this theorem and proof are almost exactly the same as the examples above: the only differences are that we've added the quantifier ∀ *n*:*nat* and that we've used the keyword `Theorem` instead of `Example`. Indeed, the latter 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.

The keywords `simpl` and `reflexivity` are examples of *tactics*. A tactic is a command that is used between `Proof` and `Qed` to tell Coq how it should check the correctness of some claim we are making. We will see several more tactics in the rest of this lecture, and yet more in future lectures.

Exercise: 1 star, optional (simpl_plus) What will Coq print in response to this query?

What about this one?

Explain the difference. □

33.4 The `intros` Tactic

Aside from unit tests, which apply functions to particular arguments, most of the properties we will be interested in proving about programs will begin with some quantifiers (e.g., "for all numbers n, ...") and/or hypothesis ("assuming m=n, ..."). In such situations, we will need to be able to reason by *assuming the hypothesis* – i.e., we start by saying "OK, suppose n is some arbitrary number," or "OK, suppose m=n."

The `intros` tactic permits us to do this by moving one or more quantifiers or hypotheses from the goal to a "context" of current assumptions.

For example, here is a slightly different proof of the same theorem.

```
Theorem plus_O_n'' : ∀ n:nat, 0 + n = n.
Proof.
  intros n. reflexivity. Qed.
```

Step through this proof in Coq and notice how the goal and context change.

```
Theorem plus_1_l : ∀ n:nat, 1 + n = S n.
Proof.
  intros n. reflexivity. Qed.
Theorem mult_0_l : ∀ n:nat, 0 × n = 0.
Proof.
  intros n. reflexivity. Qed.
```

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

33.5 Proof by Rewriting

Here is a slightly more interesting theorem:

Theorem plus_id_example : $\forall\ n\ m$:**nat**,
  $n = m \rightarrow$
  $n + n = m + m$.

Instead of making a completely universal claim about all numbers n and m, this theorem talks about a more specialized property that only holds when $n = m$. The arrow symbol is pronounced "implies."

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 in Coq's behavior.)

Exercise: 1 star (plus_id_exercise) Remove "*Admitted.*" and fill in the proof.

Theorem plus_id_exercise : $\forall\ n\ m\ o$: **nat**,
  $n = m \rightarrow m = o \rightarrow n + m = m + o$.
Proof.
  Admitted.
  □

The *Admitted* command tells Coq that we want to give up 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 facts that we believe will be useful for making some larger argument, use *Admitted* to accept them on faith for the moment, and continue thinking about the larger 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 *admit* or *Admitted* 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 : $\forall\ n\ m$: **nat**,
  $(0 + n) \times m = n \times m$.

```
Proof.
  intros n m.
  rewrite → plus_O_n.
  reflexivity. Qed.
```

Exercise: 2 stars, recommended (mult_1_plus) Theorem mult_1_plus : ∀ n m : **nat**,

```
  (1 + n) × m = m + (n × m).
Proof.
```

Admitted.

☐

33.6 Case Analysis

Of course, not everything can be proved by simple calculation: In general, unknown, hypothetical values (arbitrary numbers, booleans, lists, etc.) can show up in the "head position" of functions that we want to reason about, blocking 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. Admitted.
```

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.

What we need is to be able 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 : ∀ n : 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 as proved. (No special command is needed for moving from one subgoal to the other. When the first subgoal has been proved, it just disappears and we are left with the other "in focus.") In this proof, each of the subgoals is easily proved by a single use of `reflexivity`.

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 |. Here, the first component is empty, since the O constructor is nullary (it doesn't carry any data). The second component gives a single name, *n'*, since S is a unary constructor.

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

```
Theorem negb_involutive : ∀ b : 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. Although this is convenient, it is arguably bad style, since Coq often makes confusing choices of names when left to its own devices.

Exercise: 1 star (zero_nbeq_plus_1)

```
Theorem zero_nbeq_plus_1 : ∀ n : nat,
  beq_nat 0 (n + 1) = false.
Proof.
  Admitted.
```

□

33.7 Naming Cases

The fact that there is no explicit command for moving from one branch of a case analysis to the next can make proof scripts rather hard to read. In larger proofs, with nested case analyses, it can even become hard to stay oriented when you're sitting with Coq and stepping through the proof. (Imagine trying to remember that the first five subgoals belong to the inner case analysis and the remaining seven cases are what remains of the outer one...) Disciplined use of indentation and comments can help, but a better way is to use the *Case* tactic.

Case is not built into Coq: we need to define it ourselves. There is no need to understand how it works – just skip over the definition to the example that follows. It uses some facilities of Coq that we have not discussed – the string library (just for the concrete syntax of quoted strings) and the Ltac command, which allows us to declare custom tactics. Kudos to Aaron Bohannon for this nice hack!

```
Require String. Open Scope string_scope.

Ltac move_to_top x :=
  match reverse goal with
  | H : _ ⊢ _ ⇒ try move x after H
  end.

Tactic Notation "assert_eq" ident(x) constr(v) :=
  let H := fresh in
  assert (x = v) as H by reflexivity;
  clear H.

Tactic Notation "Case_aux" ident(x) constr(name) :=
  first [
    set (x := name); move_to_top x
  | assert_eq x name; move_to_top x
  | fail 1 "because we are working on a different case" ].

Tactic Notation "Case" constr(name) := Case_aux Case name.
Tactic Notation "SCase" constr(name) := Case_aux SCase name.
Tactic Notation "SSCase" constr(name) := Case_aux SSCase name.
Tactic Notation "SSSCase" constr(name) := Case_aux SSSCase name.
Tactic Notation "SSSSCase" constr(name) := Case_aux SSSSCase name.
Tactic Notation "SSSSSCase" constr(name) := Case_aux SSSSSCase name.
Tactic Notation "SSSSSSCase" constr(name) := Case_aux SSSSSSCase name.
Tactic Notation "SSSSSSSCase" constr(name) := Case_aux SSSSSSSCase name.
```

Here's an example of how *Case* is used. Step through the following proof and observe how the context changes.

```
Theorem andb_true_elim1 : ∀ b c : bool,
  andb b c = true → b = true.
Proof.
  intros b c H.
  destruct b.
  Case "b = true".
    reflexivity.
  Case "b = false".
    rewrite ← H. reflexivity. Qed.
```

Case does something very trivial: It simply adds a string that we choose (tagged with the identifier "Case") to the context for the current goal. When subgoals are generated, this string is carried over into their contexts. When the last of these subgoals is finally proved and the next top-level goal (a sibling of the current one) becomes active, this string will no longer appear in the context and we will be able to see that the case where we introduced it is complete. Also, as a sanity check, if we try to execute a new *Case* tactic while the string left by the previous one is still in the context, we get a nice clear error message.

For nested case analyses (i.e., when we want to use a `destruct` to solve a goal that has itself been generated by a `destruct`), there is an *SCase* ("subcase") tactic.

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

```
Theorem andb_true_elim2 : ∀ b c : bool,
  andb b c = true → c = true.
Proof.
  Admitted.
```

☐

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 *Case* tactics placed 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 a good place to mention one other piece of (possibly obvious) advice about line lengths. Beginning Coq users sometimes tend to the extremes, either writing each tactic on its own line or entire proofs on one line. Good style lies somewhere in the middle. In particular, one reasonable convention is to limit yourself to 80-character lines. Lines longer than this are hard to read and can be inconvenient to display and print. Many editors have features that help enforce this.

33.8 Induction

We proved above that 0 is a neutral element for + on the left using a simple partial evaluation argument. The fact that it is also a neutral element on the *right*...

```
Theorem plus_0_r_firsttry : ∀ n:nat,
  n + 0 = n.
```

... cannot be proved in the same simple way. Just applying `reflexivity` doesn't work: the n in $n + 0$ is an arbitrary unknown number, so the `match` in the definition of + can't be simplified. 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, 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 going this way we'll never be done.

```
Proof.
  intros n.
  simpl. Admitted.
```

Case analysis gets us a little further, but not all the way:

```
Theorem plus_0_r_secondtry : ∀ n:nat,
  n + 0 = n.
Proof.
  intros n. destruct n as [| n'].
  Case "n = 0".
    reflexivity.    Case "n = S n'".
    simpl. Admitted.
```

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

Recall (from high school) 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 $P(O)$ 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 $P(O)$ and then showing $P(n') \to P(S\ n')$. Here's how this works for the theorem we are trying to prove at the moment:

```
Theorem plus_0_r : ∀ n:nat, n + 0 = n.
Proof.
  intros n. induction n as [| n'].
  Case "n = 0". reflexivity.
  Case "n = S n'". 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'). The goal in this case becomes $(S\ n') + 0 = S\ n'$, which simplifies to $S\ (n' + 0) = S\ n'$, which in turn follows from the induction hypothesis.

```
Theorem minus_diag : ∀ n,
  minus n n = 0.
Proof.
  intros n. induction n as [| n'].
  Case "n = 0".
    simpl. reflexivity.
  Case "n = S n'".
    simpl. rewrite → IHn'. reflexivity. Qed.
```

Exercise: 2 stars, recommended (basic_induction) Theorem mult_0_r : ∀ n:**nat**,
  $n \times 0 = 0$.
Proof.
  Admitted.

Theorem plus_n_Sm : ∀ n m : **nat**,
  S (n + m) = n + (S m).
Proof.
  Admitted.

Theorem plus_comm : ∀ n m : **nat**,
  n + m = m + n.
Proof.
  Admitted.
  □

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

Exercise: 2 stars (double_plus) Lemma double_plus : ∀ n, double n = n + n .
Proof.
  Admitted.
  □

Exercise: 1 star (destruct_induction) Briefly explain the difference between the tactics `destruct` and `induction`.
  □

33.9 Formal vs. Informal Proof

"Informal proofs are algorithms; formal proofs are code."

The question of what, exactly, constitutes a "proof" of a mathematical claim has challenged philosophers for millenia. A rough and ready definition, though, could be this: a proof of a mathematical proposition P is a written (or spoken) text that instills in the reader or hearer the certainty that P is true. That is, a proof is an act of communication.

Now, acts of communication may involve different sorts of readers. On one hand, the "reader" can be a program like Coq, in which case the "belief" that is instilled is a simple mechanical check that P can be derived from a certain set of formal logical rules, and the proof is a recipe that guides the program in performing this check. Such recipes are *formal* proofs.

Alternatively, the reader can be a human being, in which case the proof will be written in English or some other natural language, thus necessarily *informal*. Here, the criteria for success are less clearly specified. A "good" proof is one that makes the reader believe P. But the same proof may be read by many different readers, some of whom may be convinced by a particular way of phrasing the argument, while others may not be. One reader may be particularly pedantic, inexperienced, or just plain thick-headed; the only way to convince them will be to make the argument in painstaking detail. But another reader, more familiar in the area, may find all this detail so overwhelming that they lose the overall thread. All they want is to be told the main ideas, because it is easier to fill in the details for themselves. Ultimately, there is no universal standard, because there is no single way of writing an informal proof that is guaranteed to convince every conceivable reader. In practice, however, mathematicians have developed a rich set of conventions and idioms for writing about complex mathematical objects that, within a certain community, make communication fairly reliable. The conventions of this stylized form of communication give a fairly clear standard for judging proofs good or bad.

Because we are using Coq in this course, we will be working heavily with formal proofs. But this doesn't mean we can ignore the informal ones! Formal proofs are useful in many ways, but they are *not* very efficient ways of communicating ideas between human beings.

For example, here is a proof that addition is associative:

```
Theorem plus_assoc' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof. intros n m p. induction n as [| n']. reflexivity.
  simpl. rewrite → IHn'. reflexivity. Qed.
```

Coq is perfectly happy with this as a proof. For a human, however, it is difficult to make much sense of it. If you're used to Coq you can probably step through the tactics one after the other in your mind and imagine the state of the context and goal stack at each point, but if the proof were even a little bit more complicated this would be next to impossible. Instead, a mathematician mighty write it like this:

- *Theorem*: For any n, m and p, n + (m + p) = (n + m) + p. *Proof*: By induction on n.
  - First, suppose $n = 0$. We must show 0 + (m + p) = (0 + m) + p. This follows directly from the definition of +.
  - Next, suppose $n = S\ n'$, where n' + (m + p) = (n' + m) + p. We must show (S n') + (m + p) = ((S n') + m) + p. By the definition of +, this follows from S (n' + (m + p)) = S ((n' + m) + p), which is immediate from the induction hypothesis. ☐

The overall form of the proof is basically similar. This is no accident, of course: Coq has been designed so that its `induction` tactic generates the same sub-goals, in the same order, as the bullet points that a mathematician would write. But there are significant differences of detail: the formal proof is much more explicit in some ways (e.g., the use of `reflexivity`) but much less explicit in others; in particular, the "proof state" at any given point in the Coq proof is completely implicit, whereas the informal proof reminds the reader several times where things stand.

Here is a formal proof that shows the structure more clearly:

```
Theorem plus_assoc : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n'].
  Case "n = 0".
    reflexivity.
  Case "n = S n'".
    simpl. rewrite → IHn'. reflexivity. Qed.
```

Exercise: 2 stars (plus_comm_informal) Translate your solution for *plus_comm* into an informal proof.

Theorem: Addition is commutative.

Proof: □

Exercise: 2 stars, optional (beq_nat_refl_informal) Write an informal proof of the following theorem, using the informal proof of *plus_assoc* as a model. Don't just paraphrase the Coq tactics into English!

Theorem: *true* = *beq_nat* *n* *n* for any *n*.

Proof: □

Exercise: 1 star, optional (beq_nat_refl) `Theorem beq_nat_refl : ∀ n : nat,`

```
  true = beq_nat n n.
Proof.
   Admitted.
```

□

33.10 Proofs Within Proofs

In Coq, as in informal mathematics, large proofs are very often broken into a sequence of theorems, with later proofs referring to earlier theorems. Occasionally, however, a proof will need 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_O_n*. We can also use `assert` to state and prove *plus_O_n* in-line:

```
Theorem mult_0_plus' : ∀ n m : nat,
  (0 + n) × m = n × m.
Proof.
  intros n m.
  assert (H: 0 + n = n).
    Case "Proof of assertion". 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. (Note that we could also name the assertion with as just as we did above with destruct and induction, i.e., assert $(0 + n = n)$ as H. Also note that we mark the proof of this assertion with a *Case*, both for readability and so that, when using Coq interactively, we can see when we're finished proving the assertion by observing when the "Proof *of assertion*" string disappears from the context.) The second goal is the same as the one at the point where we invoke assert, except that, in the context, we 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.

Actually, assert will turn out to be 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 $\rightarrow$ *plus_comm* will affect only the *outer* one.

```
Theorem plus_rearrange_firsttry : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  rewrite → plus_comm.
Admitted.
```

To get *plus_comm* to apply at the point where we want it, 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 this lemma to do the desired rewrite.

```
Theorem plus_rearrange : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  assert (H: n + m = m + n).
    Case "Proof of assertion".
    rewrite → plus_comm. reflexivity.
```

rewrite → H. reflexivity. Qed.

Exercise: 4 stars, recommended (mult_comm) Use assert to help prove this theorem. You shouldn't need to use induction.

Theorem plus_swap : ∀ n m p : **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 : ∀ m n : **nat**,
 $m \times n$ = $n \times m$.
Proof.
 Admitted.
 □

Exercise: 2 stars, optional (evenb_n__oddb_Sn) Theorem evenb_n__oddb_Sn : ∀ n : **nat**,
 evenb n = negb (evenb (S n)).
Proof.
 Admitted.
 □

33.11 More Exercises

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 hacking!)

Theorem ble_nat_refl : ∀ n:**nat**,
 true = ble_nat n n.
Proof.
 Admitted.

Theorem zero_nbeq_S : ∀ n:**nat**,
 beq_nat 0 (S n) = false.
Proof.
 Admitted.

Theorem andb_false_r : ∀ b : **bool**,

```
  andb b false = false.
Proof.
   Admitted.

Theorem plus_ble_compat_l : ∀ n m p : nat,
  ble_nat n m = true → ble_nat (p + n) (p + m) = true.
Proof.
   Admitted.

Theorem S_nbeq_0 : ∀ n:nat,
  beq_nat (S n) 0 = false.
Proof.
   Admitted.

Theorem mult_1_l : ∀ n:nat, 1 × n = n.
Proof.
   Admitted.

Theorem all3_spec : ∀ b c : bool,
    orb
      (andb b c)
      (orb (negb b)
               (negb c))
  = true.
Proof.
   Admitted.

Theorem mult_plus_distr_r : ∀ n m p : nat,
  (n + m) × p = (n × p) + (m × p).
Proof.
   Admitted.

Theorem mult_assoc : ∀ n m p : nat,
  n × (m × p) = (n × m) × p.
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. More precisely, `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' : ∀ n m p : nat,
```

$n + (m + p) = m + (n + p)$.
Proof.
Admitted.
□

Exercise: 3 stars, optional Theorem bool_fn_applied_thrice :
∀ (f : **bool** → **bool**) (b : **bool**),
f (f (f b)) = f b.

```
Proof.
  intros f b.
  destruct b.
  Case "b = true".
  remember (f true) as ftrue.
    destruct ftrue.
    SCase "f true = true".
      rewrite ← Heqftrue.
      symmetry.
      apply Heqftrue.
    SCase "f true = false".
      remember (f false) as ffalse.
      destruct ffalse.
      SSCase "f false = true".
        symmetry.
        apply Heqftrue.
      SSCase "f false = false".
        symmetry.
        apply Heqffalse.
  remember (f false) as ffalse.
    destruct ffalse.
    SCase "f false = true".
      remember (f true) as ftrue.
      destruct ftrue.
      SSCase "f true = true".
        symmetry.
        apply Heqftrue.
      SSCase "f true = false".
        symmetry.
        apply Heqffalse.
    SCase "f false = false".
      rewrite ← Heqffalse.
      symmetry.
      apply Heqffalse.
```

`Qed.`

☐

Exercise: 4 stars, recommended (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 a nat (whatever that is), and if n is a nat 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 for binary numbers, and a function to convert binary numbers to unary numbers.

(c) Finally, prove that your increment and binary-to-unary functions commute: 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.

☐

Exercise: 5 stars (binary_inverse) This exercise is a continuation of the previous exercise about binary numbers. You will need your definitions and theorems from the previous exercise 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, it is not true! Explain what the problem is.

(c) Define 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.

Exercise: 2 stars, optional (decreasing) The requirement that some argument to each function be "decreasing" 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.

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 *not* accept because of this restriction.

□

Chapter 34

Library Preface

34.1 Preface

This electronic book is a one-semester course on *Software Foundations* – the mathematical theory of programming and programming languages – suitable for graduate or upper-level undergraduate students. It develops basic concepts of functional programming, logic, operational semantics, lambda-calculus, and static type systems, using the Coq proof assistant.

The main novelty of the course is that the development is formalized and machine-checked: the text is literally a script for the Coq proof assistant. It is intended to be read hand-in-hand with the accompanying Coq source file in an interactive session with Coq. All the details of the lectures are fully developed 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 several "appendices" covering additional topics.

34.2 Overview

The course weaves together several fundamental themes:

- *Logic* as the mathematical basis for software engineering:

```
        logic                          calculus
--------------------   =   ----------------------------
software engineering       mechanical/civil engineering
```

  In particular, inductively defined sets and relations and inductive proofs about them are ubiquitous in all of computer science. The course examines them from many angles.

- *Functional programming*, an increasingly important part of the software developer's bag of tricks:

- Advanced programming idioms in mainstream software development methodologies are increasingly incorporating ideas from functional programming.
- In particular, using persistent data structures and avoiding mutable state enormously simplifies many concurrent programming tasks.

- *Foundations of programming languages* (the second part of the course):
  - Notations and techniques for rigorously describing and stress-testing new programming languages and language features. (Designing new programming languages is a surprisingly common activity! Most large software systems include subsystems that are basically programming languages – think of regular expressions, command-line formats, preference and configuration files, SQL, Flash, PDF, etc., etc.)
  - A more sophisticated understanding of the everyday tools used to build software... what's going on under the hood of your favorite programming language.

- *Coq*, an industrial-strength proof assistant:
  - Proof assistants are becoming more and more popular in both software and hardware industries. Coq is not the only one out there, but learning one thoroughly will give you a big advantage in coming to grips with another.

34.3 Practicalities

34.3.1 Chapter Dependencies

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

34.3.2 Required Background

These notes are intended to be accessible to a broad range of readers, from advanced undergraduates to PhD students and researchers. They assume little specific background in programming languages or logic. However, a degree of mathematical maturity will be helpful.

34.4 Coq

Our laboratory for this course is the Coq proof assistant. Coq can be seen as a combination of two things:

- a simple and slightly idiosyncratic (but, in its way, extremely expressive) *programming language*, together with
- a set of tools for stating *logical assertions* (including assertions about the behavior of programs) and marshalling evidence of their truth.

We will be investigating both aspects in tandem.

34.4.1 System Requirements

Coq runs on Windows and pretty much all flavors of Unix (including Linux and OS X). You will need:

- A current installation of Coq, available from the Coq home page. (Everything is known to compile with 8.3.)
- 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, but on some platforms compiling it involves installing additional packages for GUI libraries and such.

34.4.2 Access to 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.

34.5 Exercises

Each chapter of the notes includes numerous exercises. Some are marked "optional", some "recommended." Doing just the non-optional exercises should provide good coverage of the material in approximately six or eight hours of study time (for the longer chapters).

The "star ratings" for the exercises can be interpreted as follows:

- One star: easy exercises that for most readers should take only a minute or two. None of these are explicitly marked "Recommended"; rather, *all* of them should be considered as recommended: readers should be in the habit of working them as they reach them.
- Two stars: straightforward exercises (five or ten minutes).
- Three stars: exercises requiring a bit of thought (fifteen minutes to half an hour).
- Four stars: more difficult exercises (an hour or two).

34.6 Recommended Reading

Some suggestions for supplemental texts can be found in the *Postscript* chapter.

34.7 Translations

Thanks to the efforts of a team of volunteer translators, *Software Foundations* can now be enjoyed in Japanese:

- http://proofcafe.org/sf

34.8 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, and 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.

Chapter 35

Library Symbols

35.1 Symbols: Special symbols