

# Contents

<b>1</b>	<b>Library LF.Preface</b>	<b>2</b>
1.1	Preface . . . . .	2
1.2	Welcome . . . . .	2
1.3	Overview . . . . .	2
1.3.1	Logic . . . . .	3
1.3.2	Proof Assistants . . . . .	3
1.3.3	Functional Programming . . . . .	5
1.3.4	Further Reading . . . . .	6
1.4	Practicalities . . . . .	6
1.4.1	System Requirements . . . . .	6
1.4.2	Exercises . . . . .	7
1.4.3	Downloading the Coq Files . . . . .	7
1.4.4	Chapter Dependencies . . . . .	7
1.4.5	Recommended Citation Format . . . . .	8
1.5	Resources . . . . .	8
1.5.1	Sample Exams . . . . .	8
1.5.2	Lecture Videos . . . . .	8
1.6	Note for Instructors . . . . .	8
1.7	Translations . . . . .	9
1.8	Thanks . . . . .	9
<b>2</b>	<b>Library LF.Basics</b>	<b>10</b>
2.1	Basics: Functional Programming in Coq . . . . .	10
2.2	Introduction . . . . .	10
2.3	Data and Functions . . . . .	11
2.3.1	Enumerated Types . . . . .	11
2.3.2	Days of the Week . . . . .	11
2.3.3	Homework Submission Guidelines . . . . .	12
2.3.4	Booleans . . . . .	13
2.3.5	Types . . . . .	15
2.3.6	New Types from Old . . . . .	16
2.3.7	Modules . . . . .	18
2.3.8	Tuples . . . . .	18

2.3.9	Numbers . . . . .	19
2.4	Proof by Simplification . . . . .	25
2.5	Proof by Rewriting . . . . .	26
2.6	Proof by Case Analysis . . . . .	28
2.6.1	More on Notation (Optional) . . . . .	32
2.6.2	Fixpoints and Structural Recursion (Optional) . . . . .	33
2.7	More Exercises . . . . .	33
2.8	Testing Your Solutions . . . . .	35
<b>3</b>	<b>Library LF.Induction</b>	<b>37</b>
3.1	Induction: Proof by Induction . . . . .	37
3.2	Separate Compilation . . . . .	37
3.3	Proof by Induction . . . . .	38
3.4	Proofs Within Proofs . . . . .	41
3.5	Formal vs. Informal Proof . . . . .	42
3.6	More Exercises . . . . .	45
<b>4</b>	<b>Library LF.Lists</b>	<b>49</b>
4.1	Lists: Working with Structured Data . . . . .	49
4.2	Pairs of Numbers . . . . .	49
4.3	Lists of Numbers . . . . .	51
4.4	Reasoning About Lists . . . . .	57
4.4.1	Induction on Lists . . . . .	58
4.4.2	<b>Search</b> . . . . .	62
4.4.3	List Exercises, Part 1 . . . . .	62
4.4.4	List Exercises, Part 2 . . . . .	63
4.5	Options . . . . .	64
4.6	Partial Maps . . . . .	66
<b>5</b>	<b>Library LF.Poly</b>	<b>69</b>
5.1	Poly: Polymorphism and Higher-Order Functions . . . . .	69
5.2	Polymorphism . . . . .	69
5.2.1	Polymorphic Lists . . . . .	69
5.2.2	Polymorphic Pairs . . . . .	76
5.2.3	Polymorphic Options . . . . .	78
5.3	Functions as Data . . . . .	79
5.3.1	Higher-Order Functions . . . . .	79
5.3.2	Filter . . . . .	79
5.3.3	Anonymous Functions . . . . .	80
5.3.4	Map . . . . .	81
5.3.5	Fold . . . . .	83
5.3.6	Functions That Construct Functions . . . . .	84
5.4	Additional Exercises . . . . .	85

<b>6</b>	<b>Library LF.Tactics</b>	<b>89</b>
6.1	Tactics: More Basic Tactics . . . . .	89
6.2	The <code>apply</code> Tactic . . . . .	89
6.3	The <code>apply with</code> Tactic . . . . .	91
6.4	The <code>injection</code> and <code>discriminate</code> Tactics . . . . .	93
6.5	Using Tactics on Hypotheses . . . . .	96
6.6	Varying the Induction Hypothesis . . . . .	97
6.7	Unfolding Definitions . . . . .	102
6.8	Using <code>destruct</code> on Compound Expressions . . . . .	104
6.9	Review . . . . .	106
6.10	Additional Exercises . . . . .	107
<b>7</b>	<b>Library LF.Logic</b>	<b>111</b>
7.1	Logic: Logic in Coq . . . . .	111
7.2	Logical Connectives . . . . .	113
7.2.1	Conjunction . . . . .	113
7.2.2	Disjunction . . . . .	116
7.2.3	Falsehood and Negation . . . . .	117
7.2.4	Truth . . . . .	120
7.2.5	Logical Equivalence . . . . .	120
7.2.6	Setoids and Logical Equivalence . . . . .	121
7.2.7	Existential Quantification . . . . .	122
7.3	Programming with Propositions . . . . .	123
7.4	Applying Theorems to Arguments . . . . .	126
7.5	Coq vs. Set Theory . . . . .	129
7.5.1	Functional Extensionality . . . . .	130
7.5.2	Propositions vs. Booleans . . . . .	131
7.5.3	Classical vs. Constructive Logic . . . . .	136
<b>8</b>	<b>Library LF.IndProp</b>	<b>140</b>
8.1	IndProp: Inductively Defined Propositions . . . . .	140
8.2	Inductively Defined Propositions . . . . .	140
8.2.1	Inductive Definition of Evenness . . . . .	141
8.3	Using Evidence in Proofs . . . . .	142
8.3.1	Inversion on Evidence . . . . .	143
8.3.2	Induction on Evidence . . . . .	147
8.4	Inductive Relations . . . . .	149
8.5	Case Study: Regular Expressions . . . . .	154
8.5.1	The <i>remember</i> Tactic . . . . .	161
8.6	Case Study: Improving Reflection . . . . .	166
8.7	Additional Exercises . . . . .	169
8.7.1	Extended Exercise: A Verified Regular-Expression Matcher . . . . .	172

<b>9</b>	<b>Library LF.Maps</b>	<b>179</b>
9.1	Maps: Total and Partial Maps . . . . .	179
9.2	The Coq Standard Library . . . . .	179
9.3	Identifiers . . . . .	180
9.4	Total Maps . . . . .	181
9.5	Partial maps . . . . .	184
<b>10</b>	<b>Library LF.ProofObjects</b>	<b>187</b>
10.1	ProofObjects: The Curry-Howard Correspondence . . . . .	187
10.2	Proof Scripts . . . . .	189
10.3	Quantifiers, Implications, Functions . . . . .	190
10.4	Programming with Tactics . . . . .	191
10.5	Logical Connectives as Inductive Types . . . . .	192
10.5.1	Conjunction . . . . .	192
10.5.2	Disjunction . . . . .	193
10.5.3	Existential Quantification . . . . .	194
10.5.4	<b>True</b> and <b>False</b> . . . . .	195
10.6	Equality . . . . .	196
10.6.1	Inversion, Again . . . . .	197
10.7	The Coq Trusted Computing Base . . . . .	198
<b>11</b>	<b>Library LF.IndPrinciples</b>	<b>200</b>
11.1	IndPrinciples: Induction Principles . . . . .	200
11.2	Basics . . . . .	200
11.3	Polymorphism . . . . .	203
11.4	Induction Hypotheses . . . . .	205
11.5	More on the <b>induction</b> Tactic . . . . .	205
11.6	Induction Principles for Propositions . . . . .	207
11.7	Another Form of Induction Principles on Propositions (Optional) . . . . .	208
11.8	Formal vs. Informal Proofs by Induction . . . . .	209
11.8.1	Induction Over an Inductively Defined Set . . . . .	210
11.8.2	Induction Over an Inductively Defined Proposition . . . . .	211
11.9	Explicit Proof Objects for Induction (Optional) . . . . .	211
<b>12</b>	<b>Library LF.Rel</b>	<b>214</b>
12.1	Rel: Properties of Relations . . . . .	214
12.2	Relations . . . . .	214
12.3	Basic Properties . . . . .	215
12.4	Reflexive, Transitive Closure . . . . .	219
<b>13</b>	<b>Library LF.Imp</b>	<b>222</b>
13.1	Imp: Simple Imperative Programs . . . . .	222
13.2	Arithmetic and Boolean Expressions . . . . .	222

13.2.1	Syntax . . . . .	223
13.2.2	Evaluation . . . . .	224
13.2.3	Optimization . . . . .	224
13.3	Coq Automation . . . . .	225
13.3.1	Tacticals . . . . .	226
13.3.2	Defining New Tactic Notations . . . . .	230
13.3.3	The <b>omega</b> Tactic . . . . .	230
13.3.4	A Few More Handy Tactics . . . . .	231
13.4	Evaluation as a Relation . . . . .	232
13.4.1	Inference Rule Notation . . . . .	233
13.4.2	Equivalence of the Definitions . . . . .	234
13.4.3	Computational vs. Relational Definitions . . . . .	236
13.5	Expressions With Variables . . . . .	238
13.5.1	States . . . . .	238
13.5.2	Syntax . . . . .	238
13.5.3	Notations . . . . .	239
13.5.4	Evaluation . . . . .	240
13.6	Commands . . . . .	241
13.6.1	Syntax . . . . .	241
13.6.2	Desugaring notations . . . . .	242
13.6.3	The <b>Locate</b> command . . . . .	243
13.6.4	More Examples . . . . .	243
13.7	Evaluating Commands . . . . .	244
13.7.1	Evaluation as a Function (Failed Attempt) . . . . .	244
13.7.2	Evaluation as a Relation . . . . .	245
13.7.3	Determinism of Evaluation . . . . .	248
13.8	Reasoning About Imp Programs . . . . .	249
13.9	Additional Exercises . . . . .	250
<b>14</b>	<b>Library LF.ImpParser</b>	<b>257</b>
14.1	ImpParser: Lexing and Parsing in Coq . . . . .	257
14.2	Internals . . . . .	257
14.2.1	Lexical Analysis . . . . .	257
14.2.2	Parsing . . . . .	259
14.3	Examples . . . . .	266
<b>15</b>	<b>Library LF.ImpCEvalFun</b>	<b>267</b>
15.1	ImpCEvalFun: An Evaluation Function for Imp . . . . .	267
15.2	A Broken Evaluator . . . . .	267
15.3	A Step-Indexed Evaluator . . . . .	268
15.4	Relational vs. Step-Indexed Evaluation . . . . .	271
15.5	Determinism of Evaluation Again . . . . .	274

<b>16 Library LF.Extraction</b>	<b>275</b>
16.1 Extraction: Extracting ML from Coq . . . . .	275
16.2 Basic Extraction . . . . .	275
16.3 Controlling Extraction of Specific Types . . . . .	275
16.4 A Complete Example . . . . .	276
16.5 Discussion . . . . .	277
16.6 Going Further . . . . .	277
<b>17 Library LF.Auto</b>	<b>278</b>
17.1 Auto: More Automation . . . . .	278
17.2 The <code>auto</code> Tactic . . . . .	279
17.3 Searching For Hypotheses . . . . .	283
17.4 Tactics <code>eapply</code> and <code>eauto</code> . . . . .	288
17.5 Constraints on Existential Variables . . . . .	290
<b>18 Library LF.AltAuto</b>	<b>292</b>
18.1 AltAuto: More Automation . . . . .	292
18.2 Coq Automation . . . . .	293
18.3 Tacticals . . . . .	293
18.3.1 A Few More Handy Tactics . . . . .	301
18.3.2 Defining New Tactics . . . . .	301
18.4 Decision Procedures . . . . .	302
18.4.1 The Omega Tactic . . . . .	302
18.5 Search Tactics . . . . .	302
18.5.1 The <code>constructor</code> tactic. . . . .	303
18.5.2 The <code>auto</code> Tactic . . . . .	303
18.5.3 The <code>eapply</code> and <code>eauto</code> variants . . . . .	306
<b>19 Library LF.Postscript</b>	<b>309</b>
19.1 Postscript . . . . .	309
19.2 Looking Back . . . . .	309
19.3 Looking Forward . . . . .	310
19.4 Resources . . . . .	310
<b>20 Library LF.Bib</b>	<b>312</b>
20.1 Bib: Bibliography . . . . .	312
20.2 Resources cited in this volume . . . . .	312
<b>21 Library LF.PrefaceTest</b>	<b>313</b>
<b>22 Library LF.BasicsTest</b>	<b>315</b>
<b>23 Library LF.InductionTest</b>	<b>321</b>

24 Library LF.ListsTest	326
25 Library LF.PolyTest	338
26 Library LF.TacticsTest	348
27 Library LF.LogicTest	354
28 Library LF.IndPropTest	363
29 Library LF.MapsTest	370
30 Library LF.ProofObjectsTest	373
31 Library LF.IndPrinciplesTest	377
32 Library LF.RelTest	380
33 Library LF.ImpTest	382
34 Library LF.ImpParserTest	389
35 Library LF.ImpCEvalFunTest	391
36 Library LF.ExtractionTest	394
37 Library LF.AutoTest	396
38 Library LF.AltAutoTest	398
39 Library LF.PostscriptTest	400
40 Library LF.BibTest	402

# Chapter 1

## Library LF.Preface

### 1.1 Preface

### 1.2 Welcome

This is the entry point to a series of electronic textbooks on various aspects of *Software Foundations*, the mathematical underpinnings of reliable software. Topics in the series include basic concepts of logic, computer-assisted theorem proving, the Coq proof assistant, functional programming, operational semantics, logics and techniques for reasoning about programs, static type systems, property-based random testing, and verification of practical C code. The exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers. No specific background in logic or programming languages is assumed, though a degree of mathematical maturity will be helpful.

The principal novelty of the series is that it is one hundred percent formalized and machine-checked: each text is literally a script for Coq. The books are intended to be read alongside (or inside) an interactive session with Coq. All the details in the text are fully formalized in Coq, and most of the exercises are designed to be worked using Coq.

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

This book, *Logical Foundations*, lays groundwork for the others, introducing the reader to the basic ideas of functional programming, constructive logic, and the Coq proof assistant.

### 1.3 Overview

Building reliable software is really hard – really hard. The scale and complexity of modern systems, the number of people involved, and the range of demands placed on them make it challenging to build software that is even more-or-less correct, much less 100% correct. At



the same time, the increasing degree to which information processing is woven into every aspect of society greatly amplifies the cost of bugs and insecurities.

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

This volume weaves together three conceptual threads:

- (1) basic tools from *logic* for making and justifying precise claims about programs;
- (2) the use of *proof assistants* to construct rigorous logical arguments;
- (3) *functional programming*, both as a method of programming that simplifies reasoning about programs and as a bridge between programming and logic.

### 1.3.1 Logic

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

In particular, the fundamental tools of *inductive proof* are ubiquitous in all of computer science. You have surely seen them before, perhaps in a course on discrete math or analysis of algorithms, but in this course we will examine them more deeply than you have probably done so far.

### 1.3.2 Proof Assistants

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

- *Automated theorem provers* provide “push-button” operation: you give them a proposition and they return either *true* or *false* (or, sometimes, *don’t know: ran out of time*). Although their capabilities are still limited to specific domains, they have matured

tremendously in recent years and are used now in a multitude of settings. Examples of such tools include SAT solvers, SMT solvers, and model checkers.

- *Proof assistants* are hybrid tools that automate the more routine aspects of building proofs while depending on human guidance for more difficult aspects. Widely used proof assistants include Isabelle, Agda, Twelf, ACL2, PVS, and Coq, among many others.

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

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

- As a *platform for modeling programming languages*, it has become a standard tool for researchers who need to describe and reason about complex language definitions. It has been used, for example, to check the security of the JavaCard platform, obtaining the highest level of common criteria certification, and for formal specifications of the x86 and LLVM instruction sets and programming languages such as C.
- As an *environment for developing formally certified software and hardware*, Coq has been used, for example, to build CompCert, a fully-verified optimizing compiler for C, and CertiKOS, a fully verified hypervisor, for proving the correctness of subtle algorithms involving floating point numbers, and as the basis for CertiCrypt, an environment for reasoning about the security of cryptographic algorithms. It is also being used to build verified implementations of the open-source RISC-V processor architecture.
- As a *realistic environment for functional programming with dependent types*, it has inspired numerous innovations. For example, the Ynot system embeds “relational Hoare reasoning” (an extension of the *Hoare Logic* we will see later in this course) in Coq.
- As a *proof assistant for higher-order logic*, it has been used to validate a number of important results in mathematics. For example, its ability to include complex computations inside proofs made it possible to develop the first formally verified proof of the 4-color theorem. This proof had previously been controversial among mathematicians because it required checking a large number of configurations using a program. In the Coq formalization, everything is checked, including the correctness of the computational part. More recently, an even more massive effort led to a Coq formalization of

the Feit-Thompson Theorem, the first major step in the classification of finite simple groups.

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

### 1.3.3 Functional Programming

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

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

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

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

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

Hadoop and is used by Google to index the entire web is a classic example of functional programming.

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

### 1.3.4 Further Reading

This text is intended to be self contained, but readers looking for a deeper treatment of particular topics will find some suggestions for further reading in the Postscript chapter. Bibliographic information for all cited works can be found in the file Bib.

## 1.4 Practicalities

### 1.4.1 System Requirements

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

- A current installation of Coq, available from the Coq home page. These files have been tested with Coq 8.12.
- 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”).

Adventurous users of Coq within Emacs may want to check out extensions such as *company-coq* and *control-lock*.

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

Users who like CoqIDE should consider running it with the “asynchronous” and “error resilience” modes disabled:

```
coqide -async-proofs off \
```

- `async-proofs-command-error-resilience off Foo.v &`

### 1.4.2 Exercises

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

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

Those using SF in a classroom setting should note that the autograder assigns extra points to harder exercises:

1 star = 1 point 2 stars = 2 points 3 stars = 3 points 4 stars = 6 points 5 stars = 10 points

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

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

### 1.4.3 Downloading the Coq Files

A tar file containing the full sources for the “release version” of this book (as a collection of Coq scripts and HTML files) is available at <https://softwarefoundations.cis.upenn.edu>.

If you are using the book as part of a class, your professor may give you access to a locally modified version of the files; you should use that one instead of the public release version, so that you get any local updates during the semester.

### 1.4.4 Chapter Dependencies

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

### 1.4.5 Recommended Citation Format

If you want to refer to this volume in your own writing, please do so as follows:

```
@book {Pierce:SF1, author = {Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris
Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, Brent
Yorgey}, title = “Logical Foundations”, series = “Software Foundations”, volume = “1”, year =
“2020”, publisher = “Electronic textbook”, note = {Version 5.8, \URLhttp://softwarefoundations.cis.upenn.
}, }
```

## 1.5 Resources

### 1.5.1 Sample Exams

A large compendium of exams from many offerings of CIS500 (“Software Foundations”) at the University of Pennsylvania can be found at <https://www.seas.upenn.edu/~cis500/current/exams/index.htm>. There has been some drift of notations over the years, but most of the problems are still relevant to the current text.

### 1.5.2 Lecture Videos

Lectures for two intensive summer courses based on *Logical Foundations* (part of the DeepSpec summer school series) can be found at <https://deepspec.org/event/dsss17> and <https://deepspec.org/ev>. The video quality in the 2017 lectures is poor at the beginning but gets better in the later lectures.

## 1.6 Note for Instructors

If you plan 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!

In order to keep the legalities simple and to have a single point of responsibility in case the need should ever arise to adjust the license terms, sublicense, etc., we ask all contributors (i.e., everyone with access to the developers’ repository) to assign copyright in their contributions to the appropriate “author of record,” as follows:

- I hereby assign copyright in my past and future contributions to the Software Foundations project to the Author of Record of each volume or component, to be licensed under the same terms as the rest of Software Foundations. I understand that, at present, the Authors of Record are as follows: For Volumes 1 and 2, known until 2016 as “Software Foundations” and from 2016 as (respectively) “Logical Foundations” and “Programming Foundations,” and for Volume 4, “QuickChick: Property-Based Testing in Coq,” the Author of Record is Benjamin C. Pierce. For Volume 3, “Verified Functional Algorithms”, the Author of Record is Andrew W. Appel. For components

outside of designated volumes (e.g., typesetting and grading tools and other software infrastructure), the Author of Record is Benjamin Pierce.

To get started, please send an email to Benjamin Pierce, describing yourself and how you plan to use the materials and including (1) the above copyright transfer text and (2) your github username.

We'll set you up with access to the git repository and developers' mailing lists. In the repository you'll find a file *INSTRUCTORS* with further instructions.

## 1.7 Translations

Thanks to the efforts of a team of volunteer translators, *Software Foundations* can be enjoyed in Japanese at <http://proofcafe.org/sf>. A Chinese translation is also underway; you can preview it at <https://coq-zh.github.io/SF-zh/>.

## 1.8 Thanks

Development of the *Software Foundations* series has been supported, in part, by the National Science Foundation under the NSF Expeditions grant 1521523, *The Science of Deep Specification*.

# Chapter 2

## Library LF.Basics

### 2.1 Basics: Functional Programming in Coq

### 2.2 Introduction

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

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

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

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



## 2.3 Data and Functions

### 2.3.1 Enumerated Types

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

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

### 2.3.2 Days of the Week

To see how this definition mechanism works, let's start with a very simple example. The following declaration tells Coq that we are defining a set of data values – a *type*.

**Inductive** `day` : `Type` :=

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

The new type is called `day`, and its members are `monday`, `tuesday`, etc.

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

**Definition** `next_weekday` (`d:day`) : `day` :=

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

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

Having defined a function, we should next check that it works on some examples. There are actually three different ways to do the examples in Coq. First, we can use the command `Compute` to evaluate a compound expression involving `next_weekday`.

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

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

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

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

```
Example test_next_weekday:
```

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

This declaration does two things: it makes an assertion (that the second weekday after `saturday` is `tuesday`), and it gives the assertion a name that can be used to refer to it later. Having made the assertion, we can also ask Coq to verify it like this:

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

The details are not important just now, but essentially this can be read as “The assertion we’ve just made can be proved by observing that both sides of the equality evaluate to the same thing.”

Third, we can ask Coq to *extract*, from our **Definition**, a program in another, more conventional, programming language (OCaml, Scheme, or Haskell) with a high-performance compiler. This facility is very interesting, since it gives us a path from proved-correct algorithms written in Gallina to efficient machine code. (Of course, we are trusting the correctness of the OCaml/Haskell/Scheme compiler, and of Coq’s extraction facility itself, but this is still a big step forward from the way most software is developed today.) Indeed, this is one of the main uses for which Coq was developed. We’ll come back to this topic in later chapters.

### 2.3.3 Homework Submission Guidelines

If you are using *Software Foundations* in a course, your instructor may use automatic scripts to help grade your homework assignments. In order for these scripts to work correctly (and give you that you get full credit for your work!), please be careful to follow these rules:

- The grading scripts work by extracting marked regions of the `.v` files that you submit. It is therefore important that you do not alter the “markup” that delimits exercises: the Exercise header, the name of the exercise, the “empty square bracket” marker at the end, etc. Please leave this markup exactly as you find it.
- Do not delete exercises. If you skip an exercise (e.g., because it is marked “optional,” or because you can’t solve it), it is OK to leave a partial proof in your `.v` file; in this case, please make sure it ends with *Admitted* (not, for example **Abort**).

- It is fine to use additional definitions (of helper functions, useful lemmas, etc.) in your solutions. You can put these between the exercise header and the theorem you are asked to prove.
- If you introduce a helper lemma that you end up being unable to prove, hence end it with *Admitted*, then make sure to also end the main theorem in which you use it with *Admitted*, not *Qed*. That will help you get partial credit, in case you use that main theorem to solve a later exercise.

You will also notice that each chapter (like *Basics.v*) is accompanied by a *test script* (*BasicsTest.v*) that automatically calculates points for the finished homework problems in the chapter. These scripts are mostly for the auto-grading tools, but you may also want to use them to double-check that your file is well formatted before handing it in. In a terminal window, either type “*make BasicsTest.vo*” or do the following:

```
coqc -Q . LF Basics.v coqc -Q . LF BasicsTest.v
```

See the end of this chapter for more information about how to interpret the output of test scripts.

There is no need to hand in *BasicsTest.v* itself (or *Preface.v*).

If your class is using the Canvas system to hand in assignments...

- If you submit multiple versions of the assignment, you may notice that they are given different names. This is fine: The most recent submission is the one that will be graded.
- To hand in multiple files at the same time (if more than one chapter is assigned in the same week), you need to make a single submission with all the files at once using the button “Add another file” just above the comment box.

The **Require Export** statement on the next line tells Coq to use the **String** module from the standard library. We’ll use strings ourselves in later chapters, but we need to **Require** it here so that the grading scripts can use it for internal purposes. **From Coq Require Export String**.

### 2.3.4 Booleans

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

```
Inductive bool : Type :=
```

```
| true
| false.
```

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.
```

(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, together with a multitude of useful functions and lemmas. Whenever possible, we'll name our own definitions and theorems so that they exactly coincide with the ones in the standard library.)

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

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

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

```
Notation "x && y" := (andb x y).  
Notation "x || y" := (orb x y).
```

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

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

**Exercise: 1 star, standard (`nandb`)** The command *Admitted* can be used as a placeholder for an incomplete proof. We use it in exercises to indicate the parts that we're leaving

for you – i.e., your job is to replace *Admitteds* with real proofs.

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

**Definition** `nandb (b1:bool) (b2:bool) : bool`

*. Admitted.*

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

**Definition** `andb3 (b1:bool) (b2:bool) (b3:bool) : bool`

*. Admitted.*

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

*Admitted.*

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

*Admitted.*

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

*Admitted.*

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

*Admitted.*

□

## 2.3.5 Types

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

**Check** `true.`

If the expression after **Check** is followed by a colon and a type, Coq will verify that the type of the expression matches the given type and halt with an error if not.

**Check** `true`  
: `bool.`

Check (negb true)  
: **bool**.

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  
: **bool** → **bool**.

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, each of type **bool**, this function produces an output of type **bool**.”

### 2.3.6 New Types from Old

The types we have defined so far are examples of “enumerated types”: their definitions explicitly enumerate a finite set of elements, called *constructors*. Here is a more interesting type definition, where one of the constructors takes an argument:

```
Inductive rgb : Type :=  
  | red  
  | green  
  | blue.
```

```
Inductive color : Type :=  
  | black  
  | white  
  | primary (p : rgb).
```

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

Every inductively defined type (**day**, **bool**, **rgb**, **color**, etc.) describes a set of *constructor expressions* built from *constructors*.

- We start with an infinite set of *constructors*. E.g., **red**, **primary**, **true**, **false**, **monday**, etc. are constructors.
- *Constructor expressions* are formed by applying a constructor to zero or more other constructors or constructor expressions. E.g.,
  - **red**
  - **true**
  - **primary**
  - **primary red**
  - **red primary**
  - **red true**

- `primary` (`primary primary`)
- etc.
- An **Inductive** definition carves out a subset of the whole space of constructor expressions and gives it a name, like **`bool`**, **`rgb`**, or **`color`**.

In particular, the definitions of **`rgb`** and **`color`** say which constructor expressions belong to the sets **`rgb`** and **`color`**:

- `red`, `green`, and `blue` belong to the set **`rgb`**;
- `black` and `white` belong to the set **`color`**;
- if  $p$  is a constructor expression belonging to the set **`rgb`**, then `primary  $p$`  (pronounced “the constructor `primary` applied to the argument  $p$ ”) is a constructor expression belonging to the set **`color`**; and
- constructor expressions formed in these ways are the *only* ones belonging to the sets **`rgb`** and **`color`**.

We can define functions on colors using pattern matching just as we did for **`day`** and **`bool`**.

```
Definition monochrome (c : color) : bool :=
  match c with
  | black => true
  | white => true
  | primary p => false
  end.
```

Since the `primary` constructor takes an argument, a pattern matching `primary` should include either a variable (as above – note that we can choose its name freely) or a constant of appropriate type (as below).

```
Definition isred (c : color) : bool :=
  match c with
  | black => false
  | white => false
  | primary red => true
  | primary _ => false
  end.
```

The pattern “`primary _`” here is shorthand for “the constructor `primary` applied to any **`rgb`** constructor except `red`.” (The wildcard pattern `_` has the same effect as the dummy pattern variable  $p$  in the definition of `monochrome`.)

### 2.3.7 Modules

Coq provides a *module system* to aid in organizing large developments. We won't need most of its features, but one is useful: If we enclose a collection of declarations between **Module** *X* and **End** *X* markers, then, in the remainder of the file after the **End**, these definitions are referred to by names like *X.foo* instead of just *foo*. We will use this feature to limit the scope of definitions, so that we are free to reuse names.

```
Module PLAYGROUND.  
  Definition b : rgb := blue.  
End PLAYGROUND.  
  
Definition b : bool := true.  
Check Playground.b : rgb.  
Check b : bool.
```

### 2.3.8 Tuples

```
Module TUPLEPLAYGROUND.
```

A single constructor with multiple parameters can be used to create a tuple type. As an example, consider representing the four bits in a nybble (half a byte). We first define a datatype **bit** that resembles **bool** (using the constructors **B0** and **B1** for the two possible bit values) and then define the datatype **nybble**, which is essentially a tuple of four bits.

```
Inductive bit : Type :=  
  | B0  
  | B1.  
  
Inductive nybble : Type :=  
  | bits (b0 b1 b2 b3 : bit).  
Check (bits B1 B0 B1 B0)  
  : nybble.
```

The **bits** constructor acts as a wrapper for its contents. Unwrapping can be done by pattern-matching, as in the **all\_zero** function which tests a nybble to see if all its bits are **B0**. We use underscore (**\_**) as a *wildcard pattern* to avoid inventing variable names that will not be used.

```
Definition all_zero (nb : nybble) : bool :=  
  match nb with  
  | (bits B0 B0 B0 B0) => true  
  | (bits _ _ _ _) => false  
  end.  
  
Compute (all_zero (bits B1 B0 B1 B0)).  
Compute (all_zero (bits B0 B0 B0 B0)).  
End TUPLEPLAYGROUND.
```



### 2.3.9 Numbers

We put this section in a module so that our own definition of natural numbers does not interfere with the one from the standard library. In the rest of the book, we'll want to use the standard library's.

**Module** NATPLAYGROUND.

All the types we have defined so far – both “enumerated types” such as **day**, **bool**, and **bit** and tuple types such as **nybble** built from them – are finite. The natural numbers, on the other hand, are an infinite set, so we'll need to use a slightly richer form of type declaration to represent them.

There are many representations of numbers to choose from. We are most familiar with decimal notation (base 10), using the digits 0 through 9, for example, to form the number 123. You may have encountered hexadecimal notation (base 16), in which the same number is represented as 7B, or octal (base 8), where it is 173, or binary (base 2), where it is 1111011. Using an enumerated type to represent digits, we could use any of these as our representation natural numbers. Indeed, there are circumstances where each of these choices would be useful.

The binary representation is valuable in computer hardware because the digits can be represented with just two distinct voltage levels, resulting in simple circuitry. Analogously, we wish here to choose a representation that makes *proofs* simpler.

In fact, there is a representation of numbers that is even simpler than binary, namely unary (base 1), in which only a single digit is used (as one might do to count days in prison by scratching on the walls). To represent unary numbers with a Coq datatype, we use two constructors. The capital-letter **O** constructor represents zero. When the **S** constructor is applied to the representation of the natural number  $n$ , the result is the representation of  $n+1$ , where **S** stands for “successor” (or “scratch” if one is in prison). Here is the complete datatype definition.

```
Inductive nat : Type :=  
  | O  
  | S ( $n$  : nat).
```

With this definition, 0 is represented by **O**, 1 by **S O**, 2 by **S (S O)**, and so on. Informally, the clauses of the definition can be read:

- **O** is a natural number (remember this is the letter “O,” not the numeral “0”).
- **S** can be put in front of a natural number to yield another one – if  $n$  is a natural number, then **S**  $n$  is too.

Again, let's look at this in a little more detail. The definition of **nat** says how expressions in the set **nat** can be built:

- the constructor expression **O** belongs to the set **nat**;

- if  $n$  is a constructor expression belonging to the set **nat**, then  $S\ n$  is also a constructor expression belonging to the set **nat**; and
- constructor expressions formed in these two ways are the only ones belonging to the set **nat**.

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

A critical point here is that what we’ve done so far is just to define a *representation* of numbers: a way of writing them down. The names  $O$  and  $S$  are arbitrary, and at this point they have no special meaning – they are just two different marks that we can use to write down numbers (together with a rule that says any **nat** will be written as some string of  $S$  marks followed by an  $O$ ). If we like, we can write essentially the same definition this way:

```
Inductive nat' : Type :=
| stop
| tick (foo : nat').
```

The *interpretation* of these marks comes from how we use them to compute.

We can do this by writing functions that pattern match on representations of natural numbers just as we did above with booleans and days – for example, here is the predecessor function:

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

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

The following **End** command closes the current module, so **nat** will refer back to the type from the standard library. As mentioned earlier, it comes with special notation (as decimal numbers) unlike the above redefinition of **nat**.

```
End NATPLAYGROUND.
```

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 decimal numerals can be used as an alternative to the “unary” notation defined by the constructors  $S$  and  $O$ . Coq prints numbers in decimal form by default:

```
Check (S (S (S (S O)))).
```

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

```

| S (S n') ⇒ n'
end.

```

Compute (minustwo 4).

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

Check **S** : **nat**→**nat**.

Check **pred** : **nat**→**nat**.

Check **minustwo** : **nat**→**nat**.

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

(Think about standard decimal numerals: the numeral 1 is not a computation; it's a piece of data. When we write 111 to mean the number one hundred and eleven, we are using 1, three times, to write down a concrete representation of a number.)

Now let's go on and define some more functions over numbers.

For most interesting computations involving numbers, simple 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. Such functions are introduced with the keyword **Fixpoint** instead of **Definition**.

```

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

```

We could define **oddb** by a similar **Fixpoint** declaration, but here is a simpler way:

```

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

```

Example test\_oddb1: oddb 1 = true.

Proof. simpl. reflexivity. Qed.

Example test\_oddb2: oddb 4 = false.

Proof. simpl. reflexivity. Qed.

(You may notice if you step through these proofs that **simpl** actually has no effect on the goal – all of the work is done by **reflexivity**. We'll discuss why that is shortly.)

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

```

Module NATPLAYGROUND2.

```

```

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

```

```

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

```

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

Compute (plus 3 2).

The steps of simplification that Coq performs 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,  $(n\ m : \mathbf{nat})$  means just the same as if we had written  $(n : \mathbf{nat})\ (m : \mathbf{nat})$ .

```

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

```

Example test\_mult1: (mult 3 3) = 9.

Proof. simpl. reflexivity. Qed.

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

```

Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | 0, _ => 0
  | S -, 0 => n
  | S n', S m' => minus n' m'
  end.

```

End NATPLAYGROUND2.

```

Fixpoint exp (base power : nat) : nat :=
  match power with
  | 0 => S 0
  | S p => mult base (exp base p)
  end.

```

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

$\text{factorial}(0) = 1$   $\text{factorial}(n) = n * \text{factorial}(n-1)$  (if  $n > 0$ )

Translate this into Coq.

```

Fixpoint factorial (n:nat) : nat
. Admitted.

```

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

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

9

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*.

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

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

When we say that Coq comes with almost nothing built-in, we really mean it: even equality testing is a user-defined operation! Here is a function `eqb`, which tests natural numbers for **e**quality, yielding a **b**oolean. Note the use of nested `matches` (we could also have used a simultaneous match, as we did in `minus`.)

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

29

```

| O ⇒ true
| S n' ⇒
  match m with
  | O ⇒ false
  | S m' ⇒ leb n' m'
  end
end.

```

Example test\_leb1: leb 2 2 = true.

Proof. simpl. reflexivity. Qed.

Example test\_leb2: leb 2 4 = true.

Proof. simpl. reflexivity. Qed.

Example test\_leb3: leb 4 2 = false.

Proof. simpl. reflexivity. Qed.

We'll be using these (especially `eqb`) a lot, so let's give them infix notations.

Notation "x =? y" := (`eqb x y`) (at level 70) : *nat\_scope*.

Notation "x <=? y" := (`leb x y`) (at level 70) : *nat\_scope*.

Example test\_leb3': (4 <=? 2) = false.

Proof. simpl. reflexivity. Qed.

We now have two symbols that look like equality: `=` and `=?`. We'll have much more to say about the differences and similarities between them later. For now, the main thing to notice is that  $x = y$  is a logical *claim* – a “proposition” – that we can try to prove, while  $x =? y$  is an *expression* whose value (either `true` or `false`) we can compute.

**Exercise: 1 star, standard (ltb)** The `ltb` function tests natural numbers for *less-than*, yielding a boolean. Instead of making up a new `Fixpoint` for this one, define it in terms of a previously defined function. (It can be done with just one previously defined function, but you can use two if you want.)

**Definition** `ltb (n m : nat) : bool`  
*. Admitted.*

Notation "x <? y" := (`ltb x y`) (at level 70) : *nat\_scope*.

Example test\_ltb1: (ltb 2 2) = false.

*Admitted.*

Example test\_ltb2: (ltb 2 4) = true.

*Admitted.*

Example test\_ltb3: (ltb 4 2) = false.

*Admitted.*

□

## 2.4 Proof by Simplification

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

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

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

**Proof.**

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

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

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

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

**Proof.**

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

Moreover, it will be useful to know that **reflexivity** does somewhat *more* simplification than **simpl** does – for example, it tries “unfolding” defined terms, replacing them with their right-hand sides. The reason for this difference is that, if **reflexivity** succeeds, the whole goal is finished and we don’t need to look at whatever expanded expressions **reflexivity** has created by all this simplification and unfolding; by contrast, **simpl** is used in situations where we may have to read and understand the new goal that it creates, so we would not want it blindly expanding definitions and leaving the goal in a messy state.

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

First, we’ve used the keyword **Theorem** instead of **Example**. This difference is mostly a matter of style; the keywords **Example** and **Theorem** (and a few others, including **Lemma**, **Fact**, and **Remark**) mean pretty much the same thing to Coq.

Second, we’ve added the quantifier  $\forall n : \text{nat}$ , so that our theorem talks about *all* natural numbers  $n$ . Informally, to prove theorems of this form, we generally start by saying “Suppose  $n$  is some number...” Formally, this is achieved in the proof by **intros**  $n$ , which moves  $n$

from the quantifier in the goal to a *context* of current assumptions. Note that we could have used another identifier instead of  $n$  in the `intros` clause, (though of course this might be confusing to human readers of the proof):

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

Proof.

`intros m. reflexivity. Qed.`

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

Other similar theorems can be proved with the same pattern.

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

Proof.

`intros n. reflexivity. Qed.`

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

Proof.

`intros n. reflexivity. Qed.`

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

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

## 2.5 Proof by Rewriting

The following theorem is a bit more interesting than the ones we’ve seen:

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

$n = m \rightarrow$

$n + n = m + m.$

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

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

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

Proof.

`intros n m.`

`intros H.`



```
rewrite → 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 ←`. Try making this change in the above proof and see what difference it makes.)

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

```
Theorem plus_id_exercise : ∀ n m o : nat,
  n = m → m = o → n + m = m + o.
```

Proof.

*Admitted.*

□

The *Admitted* command tells Coq that we want to skip trying to prove this theorem and just accept it as a given. This can be useful for developing longer proofs, since we can state subsidiary lemmas that we believe will be useful for making some larger argument, use *Admitted* to accept them on faith for the moment, and continue working on the main argument until we are sure it makes sense; then we can go back and fill in the proofs we skipped. Be careful, though: every time you say *Admitted* you are leaving a door open for total nonsense to enter Coq’s nice, rigorous, formally checked world!

The **Check** command can also be used to examine the statements of previously declared lemmas and theorems. The two examples below are lemmas about multiplication that are proved in the standard library. (We will see how to prove them ourselves in the next chapter.)

```
Check mult_n_0.
```

```
Check mult_n_Sm.
```

We can use the `rewrite` tactic with a previously proved theorem instead of a hypothesis from the context. If the statement of the previously proved theorem involves quantified variables, as in the example below, Coq tries to instantiate them by matching with the current goal.

```
Theorem mult_n_0_m_0 : ∀ p q : nat,
  (p × 0) + (q × 0) = 0.
```

Proof.

```
intros p q.
rewrite ← mult_n_0.
rewrite ← mult_n_0.
reflexivity. Qed.
```

**Exercise: 1 star, standard (mult\_n\_1)** Use those two lemmas about multiplication that we just checked to prove the following theorem. Hint: recall that 1 is S O.

Theorem mult\_n\_1 :  $\forall p : \text{nat},$

$$p \times 1 = p.$$

Proof.

*Admitted.*

□

## 2.6 Proof by Case Analysis

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

Theorem plus\_1\_neq\_0\_firsttry :  $\forall n : \text{nat},$

$$(n + 1) =? 0 = \text{false}.$$

Proof.

`intros n.`

`simpl. Abort.`

The reason for this is that the definitions of both `eqb` 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 `eqb` is the compound expression  $n + 1$ ; neither can be simplified.

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

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

Theorem plus\_1\_neq\_0 :  $\forall n : \text{nat},$

$$(n + 1) =? 0 = \text{false}.$$

Proof.

`intros n. destruct n as [| n'] eqn:E.`

`- reflexivity.`

`- reflexivity. Qed.`

The `destruct` generates *two* subgoals, which we must then prove, separately, in order to get Coq to accept the theorem.

The annotation “`as [| n']`” is called an *intro pattern*. It tells Coq what variable names to introduce in each subgoal. In general, what goes between the square brackets is a *list of lists* of names, separated by `|`. In this case, the first component is empty, since the `O` constructor

is nullary (it doesn't have any arguments). The second component gives a single name,  $n'$ , since  $S$  is a unary constructor.

In each subgoal, Coq remembers the assumption about  $n$  that is relevant for this subgoal – either  $n = 0$  or  $n = S\ n'$  for some  $n'$ . The  $eqn:E$  annotation tells **destruct** to give the name  $E$  to this equation. Leaving off the  $eqn:E$  annotation causes Coq to elide these assumptions in the subgoals. This slightly streamlines proofs where the assumptions are not explicitly used, but it is better practice to keep them for the sake of documentation, as they can help keep you oriented when working with the subgoals.

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

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

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

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

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

```
Theorem negb_involutive : ∀ b : bool,
  negb (negb b) = b.
```

```
Proof.
```

```
  intros b. destruct b eqn:E.
  - 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. In fact, we can omit the **as** clause from *any* **destruct** and Coq will fill in variable names automatically.

This is generally considered bad style, since Coq often makes confusing choices of names when left to its own devices.

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

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

**Proof.**

```
intros b c. destruct b eqn:Eb.
- destruct c eqn:Ec.
  + reflexivity.
  + reflexivity.
- destruct c eqn:Ec.
  + reflexivity.
  + reflexivity.
```

**Qed.**

Each pair of calls to `reflexivity` corresponds to the subgoals that were generated after the execution of the `destruct c` line right above it.

Besides `-` and `+`, we can use `*` (asterisk) or any repetition of a bullet symbol (e.g. `-` or `***`) as a bullet. We can also enclose sub-proofs in curly braces:

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

**Proof.**

```
intros b c. destruct b eqn:Eb.
{ destruct c eqn:Ec.
  { reflexivity. }
  { reflexivity. } }
{ destruct c eqn:Ec.
  { reflexivity. }
  { reflexivity. } }
```

**Qed.**

Since curly braces mark both the beginning and the end of a proof, they can be used for multiple subgoal levels, as this example shows. Furthermore, curly braces allow us to reuse the same bullet shapes at multiple levels in a proof. The choice of braces, bullets, or a combination of the two is purely a matter of taste.

**Theorem** `andb3_exchange` :

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

**Proof.**

```
intros b c d. destruct b eqn:Eb.
- destruct c eqn:Ec.
  { destruct d eqn:Ed.
    - reflexivity.
    - reflexivity. }
```

```

    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
- destruct c eqn:Ec.
  { destruct d eqn:Ed.
    - reflexivity.
    - reflexivity. }
  { destruct d eqn:Ed.
    - reflexivity.
    - reflexivity. }

```

Qed.

**Exercise: 2 stars, standard (andb\_true\_elim2)** Prove the following claim, marking cases (and subcases) with bullets when you use `destruct`. Hint: delay introducing the hypothesis until after you have an opportunity to simplify it.

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

Proof.

*Admitted.*

□

Before closing the chapter, let's mention one final convenience. As you may have noticed, many proofs perform case analysis on a variable right after introducing it:

```
intros x y. destruct y as [y eqn:E].
```

This pattern is so common that Coq provides a shorthand for it: we can perform case analysis on a variable when introducing it by using an `intro` pattern instead of a variable name. For instance, here is a shorter proof of the `plus_1_neq_0` theorem above. (You'll also note one downside of this shorthand: we lose the equation recording the assumption we are making in each subgoal, which we previously got from the `eqn:E` annotation.)

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

Proof.

```

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

```

If there are no constructor arguments that need names, we can just write `||` to get the case analysis.

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

Proof.

```

intros [|].
- reflexivity.

```

```

- reflexivity.
- reflexivity.
- reflexivity.
Qed.

```

**Exercise: 1 star, standard (zero\_nbeq\_plus\_1)** Theorem `zero_nbeq_plus_1` :  $\forall n : \text{nat}, 0 =? (n + 1) = \text{false}$ .

Proof.

*Admitted.*

□

## 2.6.1 More on Notation (Optional)

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

Recall the notation definitions for infix plus and times:

```

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

```

```

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

```

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

Each notation symbol is also associated with a *notation scope*. Coq tries to guess what scope is meant from context, so when it sees `S(O×O)` it guesses `nat_scope`, but when it sees the product type `bool×bool` (which we'll see in later chapters) it guesses `type_scope`. Occasionally, it is necessary to help it out with percent-notation by writing `(x×y)%nat`, and sometimes in what Coq prints it will use `%nat` to indicate what scope a notation is in.

Notation scopes also apply to numeral notation (3, 4, 5, 42, etc.), so you may sometimes see `0%nat`, which means `O` (the natural number 0 that we're using in this chapter), or `0%Z`, which means the integer zero (which comes from a different part of the standard library).

Pro tip: Coq's notation mechanism is not especially powerful. Don't expect too much from it.

## 2.6.2 Fixpoints and Structural Recursion (Optional)

Here is a copy of the definition of addition:

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

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

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

**Exercise: 2 stars, standard, optional (decreasing)** To get a concrete sense of this, find a way to write a sensible `Fixpoint` definition (of a simple function on numbers, say) that *does* terminate on all inputs, but that Coq will reject because of this restriction. (If you choose to turn in this optional exercise as part of a homework assignment, make sure you comment out your solution so that it doesn’t cause Coq to reject the whole file!)

## 2.7 More Exercises

**Exercise: 1 star, standard (identity\_fn\_applied\_twice)** Use the tactics you have learned so far to prove the following theorem about boolean functions.

Theorem `identity_fn_applied_twice` :

```
  ∀ (f : bool → bool),  
  (∀ (x : bool), f x = x) →  
  ∀ (b : bool), f (f b) = b.
```

Proof.

*Admitted.*

□

**Exercise: 1 star, standard (negation\_fn\_applied\_twice)** Now state and prove a theorem `negation_fn_applied_twice` similar to the previous one but where the second hypothesis says that the function  $f$  has the property that  $f\ x = \text{negb } x$ .

Definition `manual_grade_for_negation_fn_applied_twice` : `option (nat × string)` := `None`.

(The last definition is used by the autograder.)

□

**Exercise: 3 stars, standard, optional (andb\_eq\_orb)** Prove the following theorem. (Hint: This one can be a bit tricky, depending on how you approach it. You will probably need both `destruct` and `rewrite`, but destructing everything in sight is not the best way.)

**Theorem** `andb_eq_orb` :

$\forall (b\ c : \text{bool}),$   
 $(\text{andb } b\ c = \text{orb } b\ c) \rightarrow$   
 $b = c.$

**Proof.**

*Admitted.*

□

**Exercise: 3 stars, standard (binary)** We can generalize our unary representation of natural numbers to the more efficient binary representation by treating a binary number as a sequence of constructors `B0` and `B1` (representing 0s and 1s), terminated by a `Z`. For comparison, in the unary representation, a number is a sequence of `S` constructors terminated by an `O`.

For example:

decimal    binary    unary  
0    Z    O  
1    B1    Z  
2    B0    (B1 Z) S  
3    B1    (B1 Z) S (S O)  
4    B0    (B0 (B1 Z)) S (S (S (S O)))  
5    B1    (B0 (B1 Z)) S (S (S (S (S O))))  
6    B0    (B1 (B1 Z)) S (S (S (S (S (S O)))))  
7    B1    (B1 (B1 Z)) S (S (S (S (S (S (S O))))))  
8    B0    (B0 (B0 (B1 Z)) S (S (S (S (S (S (S (S O)))))))

Note that the low-order bit is on the left and the high-order bit is on the right – the opposite of the way binary numbers are usually written. This choice makes them easier to manipulate.

**Inductive** `bin` : Type :=

| Z  
| B0 (n : bin)  
| B1 (n : bin).

Complete the definitions below of an increment function `incr` for binary numbers, and a function `bin_to_nat` to convert binary numbers to unary numbers.

**Fixpoint** `incr (m:bin) : bin`

. *Admitted.*

**Fixpoint** `bin_to_nat (m:bin) : nat`

. *Admitted.*

The following “unit tests” of your increment and binary-to-unary functions should pass after you have defined those functions correctly. Of course, unit tests don’t fully demonstrate the correctness of your functions! We’ll return to that thought at the end of the next chapter.



Example test\_bin\_incr1 :  $(incr\ (B1\ Z)) = B0\ (B1\ Z)$ .

*Admitted.*

Example test\_bin\_incr2 :  $(incr\ (B0\ (B1\ Z))) = B1\ (B1\ Z)$ .

*Admitted.*

Example test\_bin\_incr3 :  $(incr\ (B1\ (B1\ Z))) = B0\ (B0\ (B1\ Z))$ .

*Admitted.*

Example test\_bin\_incr4 :  $bin\_to\_nat\ (B0\ (B1\ Z)) = 2$ .

*Admitted.*

Example test\_bin\_incr5 :

$bin\_to\_nat\ (incr\ (B1\ Z)) = 1 + bin\_to\_nat\ (B1\ Z)$ .

*Admitted.*

Example test\_bin\_incr6 :

$bin\_to\_nat\ (incr\ (incr\ (B1\ Z))) = 2 + bin\_to\_nat\ (B1\ Z)$ .

*Admitted.*

□

## 2.8 Testing Your Solutions

Each SF chapter comes with a test file containing scripts that check whether you have solved the required exercises. If you're using SF as part of a course, your instructors will likely be running these test files to autograde your solutions. You can also use these test files, if you like, to make sure you haven't missed anything.

Important: This step is *optional*: if you've completed all the non-optional exercises and Coq accepts your answers, this already shows that you are in good shape.

The test file for this chapter is *BasicsTest.v*. To run it, make sure you have saved *Basics.v* to disk. Then do this:

```
coqc -Q . LF Basics.v coqc -Q . LF BasicsTest.v
```

If you accidentally deleted an exercise or changed its name, then *make BasicsTest.vo* will fail with an error that tells you the name of the missing exercise. Otherwise, you will get a lot of useful output:

- First will be all the output produced by *Basics.v* itself. At the end of that you will see *COQC BasicsTest.v*.
- Second, for each required exercise, there is a report that tells you its point value (the number of stars or some fraction thereof if there are multiple parts to the exercise), whether its type is ok, and what assumptions it relies upon.

If the *type* is not *ok*, it means you proved the wrong thing: most likely, you accidentally modified the theorem statement while you were proving it. The autograder won't give you any points for that, so make sure to correct the theorem.

The *assumptions* are any unproved theorems which your solution relies upon. “Closed under the global context” is a fancy way of saying “none”: you have solved the exercise. (Hooray!) On the other hand, a list of axioms means you haven’t fully solved the exercise. (But see below regarding “Allowed Axioms.”) If the exercise name itself is in the list, that means you haven’t solved it; probably you have *Admitted* it.

- Third, you will see the maximum number of points in standard and advanced versions of the assignment. That number is based on the number of stars in the non-optional exercises.
- Fourth, you will see a list of “Allowed Axioms”. These are unproved theorems that your solution is permitted to depend upon. You’ll probably see something about *functional\_extensionality* for this chapter; we’ll cover what that means in a later chapter.
- Finally, you will see a summary of whether you have solved each exercise. Note that summary does not include the critical information of whether the type is ok (that is, whether you accidentally changed the theorem statement): you have to look above for that information.

Exercises that are manually graded will also show up in the output. But since they have to be graded by a human, the test script won’t be able to tell you much about them.

# Chapter 3

## Library LF.Induction

### 3.1 Induction: Proof by Induction

### 3.2 Separate Compilation

Before getting started on this chapter, we need to import all of our definitions from the previous chapter:

From *LF* Require Export Basics.

For this `Require Export` command to work, Coq needs to be able to find a compiled version of *Basics.v*, called *Basics.vo*, in a directory associated with the prefix *LF*. This file is analogous to the *.class* files compiled from *.java* source files and the *.o* files compiled from *.c* files.

First create a file named *\_CoqProject* containing the following line (if you obtained the whole volume “Logical Foundations” as a single archive, a *\_CoqProject* should already exist and you can skip this step):

```
-Q . LF
```

This maps the current directory (“.”, which contains *Basics.v*, *Induction.v*, etc.) to the prefix (or “logical directory”) “*LF*”. PG and CoqIDE read *\_CoqProject* automatically, so they know to where to look for the file *Basics.vo* corresponding to the library *LF.Basics*.

Once *\_CoqProject* is thus created, there are various ways to build *Basics.vo*:

- In Proof General: The compilation can be made to happen automatically when you submit the `Require` line above to PG, by setting the emacs variable *coq-compile-before-require* to *t*. You can also use the menu option “Coq -> Auto Compilation -> Compile Before Require”.
- In CoqIDE: Open *Basics.v*; then, in the “Compile” menu, click on “Compile Buffer”.
- From the command line: Generate a *Makefile* using the *coq\_makefile* utility, that comes installed with Coq (if you obtained the whole volume as a single archive, a *Makefile* should already exist and you can skip this step):

```
coq_makefile -f _CoqProject *.v -o Makefile
```

Note: You should rerun that command whenever you add or remove Coq files to the directory.

Now you can compile *Basics.v* by running *make* with the corresponding *.vo* file as a target:

```
make Basics.vo
```

All files in the directory can be compiled by giving no arguments:

```
make
```

Under the hood, *make* uses the Coq compiler, *coqc*. You can also run *coqc* directly:

```
coqc -Q . LF Basics.v
```

But *make* also calculates dependencies between source files to compile them in the right order, so *make* should generally be preferred over explicit *coqc*.

If you have trouble (e.g., if you get complaints about missing identifiers later in the file), it may be because the “load path” for Coq is not set up correctly. The `Print LoadPath.` command may be helpful in sorting out such issues.

In particular, if you see a message like

*Compiled library Foo makes inconsistent assumptions over library Bar*

check whether you have multiple installations of Coq on your machine. It may be that commands (like *coqc*) that you execute in a terminal window are getting a different version of Coq than commands executed by Proof General or CoqIDE.

- Another common reason is that the library *Bar* was modified and recompiled without also recompiling *Foo* which depends on it. Recompile *Foo*, or everything if too many files are affected. (Using the third solution above: *make clean; make*.)

One more tip for CoqIDE users: If you see messages like *Error: Unable to locate library Basics*, a likely reason is inconsistencies between compiling things *within CoqIDE* vs *using coqc from the command line*. This typically happens when there are two incompatible versions of *coqc* installed on your system (one associated with CoqIDE, and one associated with *coqc* from the terminal). The workaround for this situation is compiling using CoqIDE only (i.e. choosing “make” from the menu), and avoiding using *coqc* directly at all.

### 3.3 Proof by Induction

We can prove that 0 is a neutral element for  $+$  on the left using just `reflexivity`. But the proof that it is also a neutral element on the *right* ...

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

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

Proof.

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

And reasoning by cases using `destruct n` doesn't get us much further: the branch of the case analysis where we assume  $n = 0$  goes through fine, but in the branch where  $n = S\ n'$  for some  $n'$  we get stuck in exactly the same way.

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

Proof.

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

We could use `destruct n'` to get one step further, but, since  $n$  can be arbitrarily large, we'll never get all the there if we just go on like this.

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

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

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

In Coq, the steps are the same: 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: one where we must show  $P(0)$  and another where we must show  $P(n') \rightarrow P(S\ n')$ . Here's how this works for the theorem at hand:

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

Proof.

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

Like `destruct`, the `induction` tactic takes an `as...` clause that specifies the names of the variables to be introduced in the subgoals. Since there are two subgoals, the `as...` clause has two parts, separated by `|`. (Strictly speaking, we can omit the `as...` clause and Coq will

choose names for us. In practice, this is a bad idea, as Coq's automatic choices tend to be confusing.)

In the first subgoal,  $n$  is replaced by 0. No new variables are introduced (so the first part of the **as...** is empty), and the goal becomes  $0 = 0 + 0$ , which follows by simplification.

In the second subgoal,  $n$  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  $n'$ ). These two names are specified in the second part of the **as...** clause. The goal in this case becomes  $S\ n' = (S\ n') + 0$ , which simplifies to  $S\ n' = S\ (n' + 0)$ , which in turn follows from  $IHn'$ .

Theorem minus\_n\_n :  $\forall\ n,$

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

Proof.

**intros**  $n$ . **induction**  $n$  **as** [ $n'$   $IHn'$ ].

-

**simpl.** **reflexivity.**

-

**simpl.** **rewrite**  $\rightarrow IHn'$ . **reflexivity.** **Qed.**

(The use of the **intros** tactic in these proofs is actually redundant. When applied to a goal that contains quantified variables, the **induction** tactic will automatically move them into the context as needed.)

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

Theorem mult\_0\_r :  $\forall\ n : \mathbf{nat},$

$n \times 0 = 0$ .

Proof.

*Admitted.*

Theorem plus\_n\_Sm :  $\forall\ n\ m : \mathbf{nat},$

$S\ (n + m) = n + (S\ m)$ .

Proof.

*Admitted.*

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

$n + m = m + n$ .

Proof.

*Admitted.*

Theorem plus\_assoc :  $\forall\ n\ m\ p : \mathbf{nat},$

$n + (m + p) = (n + m) + p$ .

Proof.

*Admitted.*

□

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

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

Use induction to prove this simple fact about double:

Lemma double\_plus :  $\forall n, \text{double } n = n + n$ .

Proof.

*Admitted.*

□

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

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

Proof.

*Admitted.*

□

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

Definition manual\_grade\_for\_destruct\_induction : `option (nat × string)` := `None`.

□

## 3.4 Proofs Within Proofs

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

Theorem mult\_0\_plus' :  $\forall n\ m : \text{nat},$   
 $(0 + n) \times m = n \times m.$

Proof.

```

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

```

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

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 not very smart about *where* it applies the rewrite. There are three uses of  $+$  here, and it turns out that doing `rewrite → plus_comm` will affect only the *outer* one...

```

Theorem plus_rearrange_firsttry : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).

```

Proof.

```

  intros n m p q.
  rewrite → plus_comm.

```

Abort.

To use `plus_comm` at the point where we need 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 it 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).
  { rewrite → plus_comm. reflexivity. }
  rewrite → H. reflexivity. Qed.

```

## 3.5 Formal vs. Informal Proof

“\_Informal proofs are algorithms; formal proofs are code.”

What constitutes a successful proof of a mathematical claim? The question has challenged philosophers for millennia, but a rough and ready definition 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 – an unassailable argument for the truth of  $P$ . That is, a proof is an act of communication.

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 that  $P$  can be mechanically derived from a certain set of formal logical rules, and the proof is a recipe that guides the program in checking this fact. 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, and will thus necessarily be *informal*. Here, the criteria for success are less clearly specified. A “valid” 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. Some readers 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 other readers, 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, since it is easier for them to fill in the details for themselves than to wade through a written presentation of them. 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 – at least 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 completely forget about 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' IHn']. reflexivity.
  simpl. rewrite → IHn'. reflexivity. Qed.
```

Coq is perfectly happy with this. For a human, however, it is difficult to make much sense of it. We can use comments and bullets to show the structure a little more clearly...

```
Theorem plus_assoc'' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
```

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

... and if you're used to Coq you may be able to 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.

A (pedantic) mathematician might write the proof something 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 that

$$0 + (m + p) = (0 + m) + p.$$

This follows directly from the definition of  $+$ .

- Next, suppose  $n = S\ n'$ , where

$$n' + (m + p) = (n' + m) + p.$$

We must now show that

$$(S\ n') + (m + p) = ((S\ n') + m) + p.$$

By the definition of  $+$ , this follows from

$$S\ (n' + (m + p)) = S\ ((n' + m) + p),$$

which is immediate from the induction hypothesis. *Qed.*

The overall form of the proof is basically similar, and of course this is no accident: 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).

**Exercise: 2 stars, advanced, especially useful (plus\_comm\_informal)** Translate your solution for `plus_comm` into an informal proof:

Theorem: Addition is commutative.

Proof:

Definition `manual_grade_for_plus_comm_informal` : `option (nat × string)` := `None`.

□

**Exercise: 2 stars, standard, optional (eqb\_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 = n =? n` for any  $n$ .

Proof: □

## 3.6 More Exercises

**Exercise: 3 stars, standard, especially useful (mult\_comm)** Use `assert` to help prove `plus_swap`. You don't need to use induction yet.

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

Proof.

*Admitted.*

Now prove commutativity of multiplication. You will probably want to define and prove a “helper” theorem to be used in the proof of this one. Hint: what is  $n \times (1 + k)$ ?

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

Proof.

*Admitted.*

□

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

Check `leb`.

Theorem `leb_refl` :  $\forall n : \mathbf{nat},$   
`true` =  $(n \leq n).$

Proof.

*Admitted.*

Theorem `zero_nbeq_S` :  $\forall n : \mathbf{nat},$   
 $0 = (S\ n) = \text{false}.$

Proof.

*Admitted.*

Theorem `andb_false_r` :  $\forall b : \mathbf{bool},$   
`andb`  $b\ \text{false} = \text{false}.$

Proof.

*Admitted.*

Theorem `plus_ble_compat_l` :  $\forall n\ m\ p : \mathbf{nat},$   
 $n \leq m = \text{true} \rightarrow (p + n) \leq (p + m) = \text{true}.$

Proof.

*Admitted.*

Theorem `S_nbeq_0` :  $\forall n : \mathbf{nat},$   
 $(S\ n) = 0 = \text{false}.$

Proof.

*Admitted.*

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

Proof.

*Admitted.*

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

orb

(andb b c)

(orb (negb b)

(negb c))

= true.

Proof.

*Admitted.*

Theorem mult\_plus\_distr\_r :  $\forall n\ m\ p : \mathbf{nat}$ ,

$(n + m) \times p = (n \times p) + (m \times p)$ .

Proof.

*Admitted.*

Theorem mult\_assoc :  $\forall n\ m\ p : \mathbf{nat}$ ,

$n \times (m \times p) = (n \times m) \times p$ .

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard, optional (eqb\_refl)** Theorem eqb\_refl :  $\forall n : \mathbf{nat}$ ,

true =  $(n =? n)$ .

Proof.

*Admitted.*

□

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

Use the `replace` tactic to do a proof of `plus_swap'`, just like `plus_swap` but without needing `assert`.

Theorem plus\_swap' :  $\forall n\ m\ p : \mathbf{nat}$ ,

$n + (m + p) = m + (n + p)$ .

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard, especially useful (binary\_commute)** Recall the `incr` and `bin_to_nat` functions that you wrote for the *binary* exercise in the Basics chapter. Prove that the following diagram commutes:

$$\begin{array}{ccc} \text{incr bin} & \xrightarrow{\quad\quad\quad} & \text{bin} \mid \mid \text{bin\_to\_nat} \mid \mid \text{bin\_to\_nat} \mid \mid \text{v v nat} \xrightarrow{\quad\quad\quad} \\ \text{nat S} & & \end{array}$$

That is, incrementing a binary number and then converting it to a (unary) natural number yields the same result as first converting it to a natural number and then incrementing. Name your theorem *bin\_to\_nat\_pres\_incr* (“pres” for “preserves”).

Before you start working on this exercise, copy the definitions of `incr` and `bin_to_nat` from your solution to the *binary* exercise here so that this file can be graded on its own. If you want to change your original definitions to make the property easier to prove, feel free to do so!

Definition `manual_grade_for_binary_commute` : **option** (**nat**×**string**) := **None**.

□

**Exercise: 5 stars, advanced (binary\_inverse)** This is a further continuation of the previous exercises about binary numbers. You may find you need to go back and change your earlier definitions to get things to work here.

(a) First, write a function to convert natural numbers to binary numbers.

Fixpoint `nat_to_bin` (*n*:**nat**) : **bin**

. *Admitted*.

Prove that, if we start with any **nat**, convert it to binary, and convert it back, we get the same **nat** we started with. (Hint: If your definition of `nat_to_bin` involved any extra functions, you may need to prove a subsidiary lemma showing how such functions relate to `nat_to_bin`.)

Theorem `nat_bin_nat` :  $\forall n, \text{bin\_to\_nat} (\text{nat\_to\_bin } n) = n$ .

Proof.

*Admitted*.

Definition `manual_grade_for_binary_inverse_a` : **option** (**nat**×**string**) := **None**.

(b) One might naturally expect that we should also prove the opposite direction – that starting with a binary number, converting to a natural, and then back to binary should yield the same number we started with. However, this is not the case! Explain (in a comment) what the problem is.

Definition `manual_grade_for_binary_inverse_b` : **option** (**nat**×**string**) := **None**.

(c) Define a normalization function – i.e., a function *normalize* going directly from **bin** to **bin** (i.e., *not* by converting to **nat** and back) such that, for any binary number **b**, converting **b** to a natural and then back to binary yields (*normalize b*). Prove it. (Warning: This part is a bit tricky – you may end up defining several auxiliary lemmas. One good way to find out what you need is to start by trying to prove the main statement, see where you get stuck,

and see if you can find a lemma – perhaps requiring its own inductive proof – that will allow the main proof to make progress.) Don't define this using `nat_to_bin` and `bin_to_nat`!

Definition `manual_grade_for_binary_inverse_c` : **option** (**nat**×**string**) := **None**.

□

# Chapter 4

## Library LF.Lists

### 4.1 Lists: Working with Structured Data

From *LF* Require Export Induction.  
Module NATLIST.

### 4.2 Pairs of Numbers

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

```
Inductive natprod : Type :=  
| pair (n1 n2 : nat).
```

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

```
Check (pair 3 5) : natprod.
```

Here are simple functions for extracting the first and second components of a pair.

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

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

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

Since pairs will be used heavily in what follows, 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.

Compute (fst (3,5)).

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

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

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

Note that pattern-matching on a pair (with parentheses:  $(x, y)$ ) is not to be confused with the “multiple pattern” syntax (with no parentheses:  $x, y$ ) that we have seen previously. The above examples illustrate pattern matching on a pair with elements  $x$  and  $y$ , whereas, for example, the definition of minus in Basics performs pattern matching on the values  $n$  and  $m$ :

```
Fixpoint minus (n m : nat) : nat := match n, m with | O , _ => O | S _, O => n | S n', S m' => minus n' m' end.
```

The distinction is minor, but it is worth knowing that they are not the same. For instance, the following definitions are ill-formed:

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

```
Definition bad_minus (n m : nat) : nat := match n, m with | (O , _) => O | (S _, O ) => n | (S n', S m') => bad_minus n' m' end.
```

Now let’s try to prove a few simple facts about pairs.

If we state properties of pairs in a slightly peculiar way, we can sometimes complete their proofs with just reflexivity (and its built-in simplification):

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

Proof.

reflexivity. Qed.

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

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

Proof.

simpl. Abort.



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

Theorem `surjective_pairing` :  $\forall (p : \mathbf{natprod})$ ,  
 $p = (\text{fst } p, \text{snd } p)$ .

Proof.

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

Notice that, unlike its behavior with `nats`, where it generates two subgoals, `destruct` generates just one subgoal here. That's because `natprods` can only be constructed in one way.

**Exercise: 1 star, standard (`snd_fst_is_swap`)** Theorem `snd_fst_is_swap` :  $\forall (p : \mathbf{natprod})$ ,

$(\text{snd } p, \text{fst } p) = \text{swap\_pair } p$ .

Proof.

*Admitted.*

□

**Exercise: 1 star, standard, optional (`fst_swap_is_snd`)** Theorem `fst_swap_is_snd` :  $\forall (p : \mathbf{natprod})$ ,

$\text{fst } (\text{swap\_pair } p) = \text{snd } p$ .

Proof.

*Admitted.*

□

## 4.3 Lists of Numbers

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

Inductive `natlist` : Type :=

| `nil`  
| `cons (n : nat) (l : natlist)`.

For example, here is a three-element list:

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

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

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

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

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

It is not necessary to understand the details of these declarations, but here is roughly what’s going on in case you are interested. The “**right associativity**” annotation tells Coq how to parenthesize expressions involving multiple 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])`.

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

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

## Repeat

Next let’s look at several functions for constructing and manipulating lists. First, the **repeat** function takes a number `n` and a `count` and returns a list of length `count` in which every element is `n`.

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

## Length

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

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

## Append

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

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

Since `app` will be used extensively, it is again convenient to have an infix operator for it.

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

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

Proof. reflexivity. Qed.

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

Proof. reflexivity. Qed.

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

Proof. reflexivity. Qed.

## Head and Tail

Here are two smaller examples of programming with lists. The `hd` function returns the first element (the “head”) of the list, while `tl` returns everything but the first element (the “tail”). Since the empty list has no first element, we 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 tl (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\_tl: tl [1;2;3] = [2;3].

Proof. reflexivity. Qed.

## Exercises

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

```
Fixpoint nonzeros (l:natlist) : natlist
. Admitted.
```

```
Example test_nonzeros:
  nonzeros [0;1;0;2;3;0;0] = [1;2;3].
Admitted.
```

```
Fixpoint oddmembers (l:natlist) : natlist
. Admitted.
```

```
Example test_oddmembers:
  oddmembers [0;1;0;2;3;0;0] = [1;3].
Admitted.
```

```
Definition countoddmembers (l:natlist) : nat
. Admitted.
```

```
Example test_countoddmembers1:
  countoddmembers [1;0;3;1;4;5] = 4.
Admitted.
```

```
Example test_countoddmembers2:
  countoddmembers [0;2;4] = 0.
Admitted.
```

```
Example test_countoddmembers3:
  countoddmembers nil = 0.
Admitted.
```

□

**Exercise: 3 stars, advanced (alternate)** Complete the following definition of `alternate`, which interleaves 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 involves defining a new kind of pairs, but this is not the only way.)

```
Fixpoint alternate (l1 l2 : natlist) : natlist
. Admitted.
```

```
Example test_alternate1:
  alternate [1;2;3] [4;5;6] = [1;4;2;5;3;6].
```

*Admitted.*

Example test\_alternate2:

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

*Admitted.*

Example test\_alternate3:

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

*Admitted.*

Example test\_alternate4:

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

*Admitted.*

□

## Bags via Lists

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

Definition `bag := natlist`.

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

Fixpoint `count (v : nat) (s : bag) : nat`

*. Admitted.*

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 – using `max` instead of `sum` – which is why we don't call this operation *union*.) 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 one or more functions that have already been defined.

Definition `sum : bag → bag → bag`

*. Admitted.*

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

*Admitted.*

Definition add ( $v : \text{nat}$ ) ( $s : \text{bag}$ ) : bag  
. Admitted.

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 ( $v : \text{nat}$ ) ( $s : \text{bag}$ ) : bool  
. Admitted.

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

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

□

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

When remove\_one is applied to a bag without the number to remove, it should return the same bag unchanged. (This exercise is optional, but students following the advanced track will need to fill in the definition of remove\_one for a later exercise.)

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

Example test\_remove\_one1:  
count 5 (remove\_one 5 [2;1;5;4;1]) = 0.  
Admitted.

Example test\_remove\_one2:  
count 5 (remove\_one 5 [2;1;4;1]) = 0.  
Admitted.

Example test\_remove\_one3:  
count 4 (remove\_one 5 [2;1;4;5;1;4]) = 2.  
Admitted.

Example test\_remove\_one4:  
count 5 (remove\_one 5 [2;1;5;4;5;1;4]) = 1.  
Admitted.

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

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**

. *Admitted.*

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: 2 stars, standard, especially useful (add\_inc\_count)** Adding a value to a bag should increase the value's count by one. State that as a theorem and prove it.

Definition manual\_grade\_for\_add\_inc\_count : **option** (**nat** × **string**) := **None**.

□

## 4.4 Reasoning About Lists

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

Theorem nil\_app :  $\forall l : \text{natlist},$

$[] ++ l = l.$

Proof. **reflexivity**. Qed.

...because the  $[]$  is substituted into the “scrutinee” (the expression whose value is being “scrutinized” by the match) in the definition of **app**, allowing the match itself to be simplified.

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

Theorem tl\_length\_pred :  $\forall l : \text{natlist},$

**pred** (length  $l$ ) = length (tl  $l$ ).

Proof.

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

-

**reflexivity**.

-

**reflexivity**. Qed.

Here, the nil case works because we've chosen to define  $\text{tl nil} = \text{nil}$ . 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. We'll see how to do this next.

(Micro-Sermon: As we get deeper into this material, simply *reading* proof scripts will not get you very far! It is important to step through the details of each one using Coq and think about what each step achieves. Otherwise it is more or less guaranteed that the exercises will make no sense when you get to them. 'Nuff said.)

### 4.4.1 Induction on Lists

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

This last 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 always be broken down into smaller ones, eventually reaching `nil`, these two arguments together establish the truth of  $P$  for all lists  $l$ . Here's a concrete example:

**Theorem** `app_assoc` :  $\forall l1\ l2\ l3 : \mathbf{natlist},$   
 $(l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).$

**Proof.**

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

Notice that, as when doing induction on natural numbers, the `as...` clause provided to the `induction` tactic gives a name to the induction hypothesis corresponding to the smaller list  $l1'$  in the `cons` case.



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

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

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

*Proof:* By induction on  $l1$ .

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

## Reversing a List

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

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

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

Proof. `reflexivity. Qed.`

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

Proof. `reflexivity. Qed.`

For something a bit more challenging than the proofs we’ve seen so far, let’s prove that reversing a list does not change its length. Our first attempt gets stuck in the successor case...

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

```

length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  -
    reflexivity.
  -

    simpl.
    rewrite ← IHl'.

```

Abort.

So let's take the equation relating ++ and length that would have enabled us to make progress at the point where we got stuck and state it as a separate lemma.

```

Theorem app_length : ∀ l1 l2 : natlist,
  length (l1 ++ l2) = (length l1) + (length l2).

```

```

Proof.
  intros l1 l2. induction l1 as [| n l1' IHl1'].
  -
    reflexivity.
  -

    simpl. rewrite → IHl1'. reflexivity. Qed.

```

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

Now we can complete the original proof.

```

Theorem rev_length : ∀ l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  -
    reflexivity.
  -

    simpl. rewrite → app_length.
    simpl. rewrite → IHl'. rewrite plus_comm.
    reflexivity.

```

Qed.

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

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

*Proof:* By induction on  $l1$ .

- First, suppose  $l1 = []$ . We must show

$\text{length } (\square ++ l2) = \text{length } \square + \text{length } l2,$

which follows directly from the definitions of **length**, **++**, and **plus**.

- Next, suppose  $l1 = n::l1'$ , with

$\text{length } (l1' ++ l2) = \text{length } l1' + \text{length } l2.$

We must show

$\text{length } ((n::l1') ++ l2) = \text{length } (n::l1') + \text{length } l2.$

This follows directly from the definitions of **length** and **++** together with the induction hypothesis.  $\square$

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

*Proof:* By induction on  $l$ .

- First, suppose  $l = []$ . We must show

$\text{length } (\text{rev } []) = \text{length } [],$

which follows directly from the definitions of **length** and **rev**.

- Next, suppose  $l = n::l'$ , with

$\text{length } (\text{rev } l') = \text{length } l'.$

We must show

$\text{length } (\text{rev } (n :: l')) = \text{length } (n :: l').$

By the definition of **rev**, this follows from

$\text{length } ((\text{rev } l') ++ n) = S (\text{length } l')$

which, by the previous lemma, is the same as

$\text{length } (\text{rev } l') + \text{length } n = S (\text{length } l').$

This follows directly from the induction hypothesis and the definition of **length**.  $\square$

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

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

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

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

### 4.4.2 Search

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

Coq’s `Search` command is quite helpful with this. Let’s say you’ve forgotten the name of a theorem about `rev`. The command `Search rev` will cause Coq to display a list of all theorems involving `rev`.

`Search rev.`

Or say you’ve forgotten the name of the theorem showing that plus is commutative. You can use a pattern to search for all theorems involving the equality of two additions.

`Search (_ + _ = _ + _).`

You’ll see a lot of results there, nearly all of them from the standard library. To restrict the results, you can search inside a particular module:

`Search (_ + _ = _ + _) inside Induction.`

You can also make the search more precise by using variables in the search pattern instead of wildcards:

`Search (?x + ?y = ?y + ?x).`

The question mark in front of the variable is needed to indicate that it is a variable in the search pattern, rather than a variable that is expected to be in scope currently.

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

Your IDE likely has its own functionality to help with searching. For example, in Proof-General, you can run `Search` with `C-c C-a C-a`, and paste its response into your buffer with `C-c C-;`.

### 4.4.3 List Exercises, Part 1

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

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

Proof.

*Admitted.*

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

Proof.

*Admitted.*

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

Proof.

*Admitted.*

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

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

**Proof.**

*Admitted.*

An exercise about your implementation of `nonzeros`:

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

**Proof.**

*Admitted.*

□

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

**Fixpoint** `eqblist` (`l1 l2` : `natlist`) : `bool`

. *Admitted.*

**Example** `test_eqblist1` :

$(\text{eqblist } \text{nil } \text{nil} = \text{true}).$

*Admitted.*

**Example** `test_eqblist2` :

$\text{eqblist } [1;2;3] [1;2;3] = \text{true}.$

*Admitted.*

**Example** `test_eqblist3` :

$\text{eqblist } [1;2;3] [1;2;4] = \text{false}.$

*Admitted.*

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

$\text{true} = \text{eqblist } l\ l.$

**Proof.**

*Admitted.*

□

#### 4.4.4 List Exercises, Part 2

Here are a couple of little theorems to prove about your definitions about bags above.

**Exercise: 1 star, standard (`count_member_nonzero`)** **Theorem** `count_member_nonzero`  
:  $\forall (s : \text{bag}),$

$1 \leq? (\text{count } 1 (1 :: s)) = \text{true}.$

Proof.

*Admitted.*

□

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

Theorem `leb_n_Sn` :  $\forall n$ ,

$n \leq? (\text{S } n) = \text{true}$ .

Proof.

`intros n. induction n as [| n' IHn'].`

-

`simpl. reflexivity.`

-

`simpl. rewrite IHn'. reflexivity. Qed.`

Before doing the next exercise, make sure you've filled in the definition of `remove_one` above.

**Exercise: 3 stars, advanced (`remove_does_not_increase_count`)** Theorem `remove_does_not_increa`

$\forall (s : \text{bag})$ ,

$(\text{count } 0 (\text{remove\_one } 0 s)) \leq? (\text{count } 0 s) = \text{true}$ .

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard, optional (`bag_count_sum`)** Write down an interesting theorem `bag_count_sum` about bags involving the functions `count` and `sum`, and prove it using Coq. (You may find that the difficulty of the proof depends on how you defined `count`!)

**Exercise: 4 stars, advanced (`rev_injective`)** Prove that the `rev` function is injective. There is a hard way and an easy way to do this.

Theorem `rev_injective` :  $\forall (l1 \ l2 : \text{natlist})$ ,

$\text{rev } l1 = \text{rev } l2 \rightarrow l1 = l2$ .

Proof.

*Admitted.*

□

## 4.5 Options

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

Fixpoint `nth_bad` ( $l : \text{natlist}$ ) ( $n : \text{nat}$ ) : `nat` :=

```

match l with
| nil ⇒ 42
| a :: l' ⇒ match n with
| 0 ⇒ a
| S n' ⇒ nth_bad l' n'
end
end.

```

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

```

Inductive natoption : Type :=
| Some (n : nat)
| None.

```

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

```

Fixpoint nth_error (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ match n with
  | 0 ⇒ Some a
  | S n' ⇒ nth_error l' n'
  end
end.

```

Example `test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.`

Proof. `reflexivity. Qed.`

Example `test_nth_error2 : nth_error [4;5;6;7] 3 = Some 7.`

Proof. `reflexivity. Qed.`

Example `test_nth_error3 : nth_error [4;5;6;7] 9 = None.`

Proof. `reflexivity. Qed.`

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

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

```

Fixpoint nth_error' (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ if n =? 0 then Some a
  else nth_error' l' (pred n)

```

end.

Coq's conditionals are exactly like those found in any other language, with one small generalization. Since the **bool** type is not built in, Coq actually supports 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, standard (hd\_error)** Using the same idea, fix the **hd** function from earlier so we don't have to pass a default element for the **nil** case.

```
Definition hd_error (l : natlist) : natoption  
  . Admitted.
```

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

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

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

□

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

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

**Proof.**

*Admitted.*

□

End NATLIST.

## 4.6 Partial Maps

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



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

```
Inductive id : Type :=
| id (n : nat).
```

Internally, an **id** is just a number. Introducing a separate type by wrapping each `nat` with the tag `id` makes definitions more readable and gives us more flexibility.

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

```
Definition eqb_id (x1 x2 : id) :=
  match x1, x2 with
  | id n1, id n2 => n1 == n2
  end.
```

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

Proof.

*Admitted.*

□

Now we define the type of partial maps:

```
Module PARTIALMAP.
```

```
Export NatList.
```

```
Inductive partial_map : Type :=
| empty
| record (i : id) (v : nat) (m : partial_map).
```

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

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

```
Definition update (d : partial_map)
  (x : id) (value : nat)
  : partial_map :=
  record x value d.
```

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

```
Fixpoint find (x : id) (d : partial_map) : natoption :=
  match d with
  | empty => None
  | record y v d' => if eqb_id x y
    then Some v
    else find x d'
```

end.

**Exercise: 1 star, standard (update\_eq)** Theorem update\_eq :

$\forall (d : \text{partial\_map}) (x : \text{id}) (v : \text{nat}),$   
find  $x$  (update  $d$   $x$   $v$ ) = Some  $v$ .

Proof.

*Admitted.*

□

**Exercise: 1 star, standard (update\_neq)** Theorem update\_neq :

$\forall (d : \text{partial\_map}) (x \ y : \text{id}) (o : \text{nat}),$   
eqb\_id  $x \ y$  = false  $\rightarrow$  find  $x$  (update  $d$   $y$   $o$ ) = find  $x$   $d$ .

Proof.

*Admitted.*

□ End PARTIALMAP.

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

Inductive **baz** : Type :=

| Baz1 ( $x : \text{baz}$ )  
| Baz2 ( $y : \text{baz}$ ) ( $b : \text{bool}$ ).

How *many* elements does the type **baz** have? (Explain in words, in a comment.)

Definition manual\_grade\_for\_baz\_num\_elts : **option** (**nat** × **string**) := **None**.

□

# Chapter 5

## Library LF.Poly

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

Set *Warnings* "-notation-overridden,-parsing".  
From *LF* Require Export Lists.

### 5.2 Polymorphism

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

#### 5.2.1 Polymorphic Lists

For the last chapter, we've been working with lists containing just numbers. Obviously, interesting programs also need to be able to manipulate lists with elements from other types – 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  
  | bool_cons (b : bool) (l : 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.) and all their properties (`rev_length`, `app_assoc`, 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
| cons (x : X) (l : 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 function header on the first line, and the occurrences of **natlist** in the types of the constructors have been replaced by **list X**.

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

Check **list** : Type  $\rightarrow$  Type.

The parameter **X** in the definition of **list** automatically becomes a parameter to the constructors **nil** and **cons** – that is, **nil** and **cons** are now polymorphic constructors; when we use them, we must now provide a first argument that is the type of the list they are building. For example, **nil nat** constructs the empty list of type **nat**.

Check (nil **nat**) : list **nat**.

Similarly, **cons nat** adds an element of type **nat** to a list of type **list nat**. Here is an example of forming a list containing just the natural number 3.

Check (cons **nat** 3 (nil **nat**)) : list **nat**.

What might the type of **nil** be? We can read off the type **list X** from the definition, but this omits the binding for **X** which is the parameter to **list**. **Type  $\rightarrow$  list X** does not explain the meaning of **X**. **(X : Type)  $\rightarrow$  list X** comes closer. Coq’s notation for this situation is  $\forall X : \text{Type}, \text{list } X$ .

Check **nil** :  $\forall X : \text{Type}, \text{list } X$ .

Similarly, the type of **cons** from the definition looks like **X  $\rightarrow$  list X  $\rightarrow$  list X**, but using this convention to explain the meaning of **X** results in the type  $\forall X, X \rightarrow \text{list } X \rightarrow \text{list } X$ .

Check **cons** :  $\forall X : \text{Type}, X \rightarrow \text{list } X \rightarrow \text{list } X$ .

(A side note on notations: In .v files, the “forall” quantifier is spelled out in letters. In the generated HTML files and in the way various IDEs show .v files, depending on the settings of their display controls,  $\forall$  is usually typeset as the standard mathematical “upside down A,” though you’ll still see the spelled-out “forall” in a few places. This is just a quirk of typesetting – there is no difference in meaning.)

Having to supply a type argument for every single use of a list constructor would be rather burdensome; we will soon see ways of reducing this annotation burden.

Check (cons **nat** 2 (cons **nat** 1 (nil **nat**)))  
: list **nat**.

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

```

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

```

As with `nil` and `cons`, we can use `repeat` by applying it first to a type and then to an element of this type (and a number):

```

Example test_repeat1 :
  repeat nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).

```

Proof. `reflexivity`. Qed.

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

```

Example test_repeat2 :
  repeat bool false 1 = cons bool false (nil bool).

```

Proof. `reflexivity`. Qed.

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

```

Module MUMBLEGRUMBLE.

```

```

Inductive mumble : Type :=
  | a
  | b (x : mumble) (y : nat)
  | c.

```

```

Inductive grumble (X:Type) : Type :=
  | d (m : mumble)
  | e (x : X).

```

Which of the following are well-typed elements of `grumble X` for some type `X`? (Add YES or NO to each line.)

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

End MUMBLEGRUMBLE.

Definition manual\_grade\_for\_mumble\_grumble : option (nat×string) := None.

□

## Type Annotation Inference

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

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

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

```
Check repeat'
: ∀ X : Type, X → nat → list X.
```

```
Check repeat
: ∀ X : Type, X → nat → list X.
```

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

This powerful facility means we don’t always have to write explicit type annotations everywhere, although explicit type annotations can still be quite useful as documentation and sanity checks, so we will continue to use them much of the time.

## Type Argument Synthesis

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

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

This may sound similar to type annotation inference – and, indeed, the two procedures rely on the same underlying mechanisms. Instead of simply omitting the types of some arguments to a function, like

repeat' X x count : list X :=  
 we can also replace the types with holes  
 repeat' (X : \_) (x : \_) (count : \_) : list X :=  
 to tell Coq to attempt to infer the missing information.  
 Using holes, the **repeat** function can be written like this:

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

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

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

...we can use holes to write this:

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

## Implicit Arguments

In fact, we can go further and even avoid writing `_`'s in most cases by telling Coq *always* to infer the type argument(s) of a given function.

The *Arguments* directive specifies the name of the function (or constructor) and then lists its argument names, with curly braces around any arguments to be treated as implicit. (If some arguments of a definition don't have a name, as is often the case for constructors, they can be marked with a wildcard pattern `_`.)

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

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

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

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

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

(Note that we didn't even have to provide a type argument to the recursive call to `repeat`'. Indeed, it would be invalid to provide one, because Coq is not expecting it.)

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

```
Inductive list' {X:Type} : Type :=
| nil'
| cons' (x : X) (l : list').
```

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

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

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

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

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

```
Example test_rev1 :
  rev (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
```

Proof. reflexivity. Qed.

```
Example test_rev2:
  rev (cons true nil) = cons true nil.
```

Proof. reflexivity. Qed.

```
Example test_length1: length (cons 1 (cons 2 (cons 3 nil))) = 3.
```

Proof. reflexivity. Qed.



## Supplying Type Arguments Explicitly

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

```
Fail Definition mynil := nil.
```

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

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

```
Definition mynil : list nat := nil.
```

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

```
Check @nil : ∀ X : Type, list X.
```

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

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

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

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

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

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

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

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

## Exercises

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

**Theorem** `app_nil_r` :  $\forall (X:\text{Type}), \forall l:\text{list } X,$   
 $l ++ [] = l.$

**Proof.**

*Admitted.*

**Theorem** `app_assoc` :  $\forall A (l\ m\ n:\text{list } A),$

$l ++ m ++ n = (l ++ m) ++ n.$

Proof.

*Admitted.*

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

Proof.

*Admitted.*

□

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

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

Proof.

*Admitted.*

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

Proof.

*Admitted.*

□

## 5.2.2 Polymorphic Pairs

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

Inductive **prod** ( $X\ Y : \text{Type}$ ) :  $\text{Type} :=$   
 $| \text{pair } (x : X) (y : Y).$

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

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

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

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

Notation " $X * Y$ " := `(prod X Y) : type_scope`.

(The annotation : *type\_scope* tells Coq that this abbreviation should only be used when parsing types, not when parsing expressions. This avoids a clash with the multiplication symbol.)

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, while  $X \times Y$  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 other functional languages, it is often called *zip*; we call it **combine** for consistency with Coq's standard library.

```
Fixpoint combine {X Y : Type} (lx : list X) (ly : list Y)
  : list (X × Y) :=
  match lx, ly with
  | [], _ => []
  | _, [] => []
  | x :: tx, y :: ty => (x, y) :: (combine tx ty)
  end.
```

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

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

□

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

Fill in the definition of **split** below. Make sure it passes the given unit test.

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

Example `test_split`:

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

Proof.

*Admitted.*

□

### 5.2.3 Polymorphic Options

Our last polymorphic type for now is *polymorphic options*, which generalize **natoption** from the previous chapter. (We put the definition inside a module because the standard library already defines **option** and it's this one that we want to use below.)

Module OPTIONPLAYGROUND.

Inductive **option** (X:Type) : Type :=

| Some (x : X)

| None.

Arguments Some {X} \_.

Arguments None {X}.

End OPTIONPLAYGROUND.

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

Fixpoint nth\_error {X : Type} (l : list X) (n : nat)  
: **option** X :=

match l with

| nil ⇒ **None**

| a :: l' ⇒ match n with

| 0 ⇒ **Some** a

| S n' ⇒ nth\_error l' n'

end

end.

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

Proof. reflexivity. Qed.

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

Proof. reflexivity. Qed.

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

Proof. reflexivity. Qed.

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

Definition hd\_error {X : Type} (l : list X) : **option** X

. *Admitted.*

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

Check @hd\_error :  $\forall X : \text{Type}, \text{list } X \rightarrow \text{option } X$ .

Example test\_hd\_error1 : hd\_error [1;2] = Some 1.

Admitted.

Example test\_hd\_error2 : hd\_error [[1];[2]] = Some [1].

Admitted.

□

## 5.3 Functions as Data

Like most modern programming languages – especially other “functional” languages, including OCaml, Haskell, Racket, Scala, Clojure, etc. – Coq treats functions as first-class citizens, allowing them to be passed as arguments to other functions, returned as results, stored in data structures, etc.

### 5.3.1 Higher-Order Functions

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

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

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

Check @doit3times :  $\forall X : \text{Type}, (X \rightarrow X) \rightarrow X \rightarrow X$ .

Example test\_doit3times: doit3times minustwo 9 = 3.

Proof. reflexivity. Qed.

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

Proof. reflexivity. Qed.

### 5.3.2 Filter

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

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

```

    else filter test t
end.

```

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

Example `test_filter1`: `filter evenb [1;2;3;4] = [2;4]`.

Proof. reflexivity. Qed.

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

Example `test_filter2`:

```

    filter length_is_1
      [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].

```

Proof. reflexivity. Qed.

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

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

Example `test_countoddmembers'1`: `countoddmembers' [1;0;3;1;4;5] = 4.`

Proof. reflexivity. Qed.

Example `test_countoddmembers'2`: `countoddmembers' [0;2;4] = 0.`

Proof. reflexivity. Qed.

Example `test_countoddmembers'3`: `countoddmembers' nil = 0.`

Proof. reflexivity. Qed.

### 5.3.3 Anonymous Functions

It is arguably a little sad, in the example just above, to be forced to define the function `length_is_1` and give it a name just to be able to pass it as an argument to `filter`, since we will probably never use it again. Moreover, this is not an isolated example: when using higher-order functions, we often want to pass as arguments “one-off” functions that we will never use again; having to give each of these functions a name would be tedious.

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

Example `test_anon_fun'`:

```

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

```

Proof. reflexivity. Qed.

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

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

Example `test_filter2'`:

```

    filter (fun l => (length l) =? 1)
      [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.

```

**Exercise: 2 stars, standard (filter\_even\_gt7)** Use `filter` (instead of `Fixpoint`) to write a Coq function `filter_even_gt7` that takes a list of natural numbers as input and returns a list of just those that are even and greater than 7.

```

Definition filter_even_gt7 (l : list nat) : list nat
. Admitted.

```

```

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, standard (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 predicate of type  $X \rightarrow \mathbf{bool}$  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
. Admitted.

```

```

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.
□

```

### 5.3.4 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  $l = [n1, n2, n3, \dots]$  and returns the list  $[f\ n1, f\ n2, f\ n3, \dots]$ , where  $f$  has been applied to each element of  $l$  in turn. For example:

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

Proof. reflexivity. Qed.

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

Example test\_map2:

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

Proof. reflexivity. Qed.

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

Example test\_map3:

```
map (fun n ⇒ [evenb n;oddb n]) [2;1;2;5]
= [[true;false];[false;true];[true;false];[false;true]].
```

Proof. reflexivity. Qed.

## Exercises

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

Theorem `map_rev` :  $\forall (X\ Y : \text{Type}) (f : X \rightarrow Y) (l : \text{list } X)$ ,

```
map f (rev l) = rev (map f l).
```

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard, especially useful (flat\_map)** The function `map` maps a **list**  $X$  to a **list**  $Y$  using a function of type  $X \rightarrow Y$ . We can define a similar function, `flat_map`, which maps a **list**  $X$  to a **list**  $Y$  using a function  $f$  of type  $X \rightarrow \text{list } Y$ . 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` { $X\ Y : \text{Type}$ } ( $f : X \rightarrow \text{list } Y$ ) ( $l : \text{list } X$ )  
: (**list**  $Y$ )

. *Admitted.*

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

### 5.3.5 Fold

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

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

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

Intuitively, the behavior of the `fold` operation is to insert a given binary operator  $f$  between every pair of elements in a given list. For example, `fold plus [1;2;3;4]` intuitively means  $1+2+3+4$ . To make this precise, we also need a “starting element” that serves as the initial second input to  $f$ . So, for example,

```
fold plus 1;2;3;4 0
yields
1 + (2 + (3 + (4 + 0))).
```

Some more examples:

Check (fold andb) : list bool → bool → bool.

Example fold\_example1 :

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

Proof. reflexivity. Qed.

Example fold\_example2 :  
 fold andb [true;true;false;true] true = false.

Proof. reflexivity. Qed.

Example fold\_example3 :  
 fold app [[1];[];[2;3];[4]] [] = [1;2;3;4].

Proof. reflexivity. Qed.

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

Definition manual\_grade\_for\_fold\_types\_different : option (nat × string) := None.

□

### 5.3.6 Functions That Construct Functions

Most of the higher-order functions we have talked about so far take functions as arguments. Let's look at some examples that involve *returning* functions as the results of other functions. To begin, here is a function that takes a value `x` (drawn from some type `X`) and returns a function from `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.

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 : nat → nat → nat.

Each `→` in this expression is actually a *binary* operator on types. This operator is *right-associative*, so the type of `plus` is really a shorthand for `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 : nat → nat.

Example test\_plus3 : plus3 4 = 7.

Proof. reflexivity. Qed.  
 Example test\_plus3' : doit3times plus3 0 = 9.  
 Proof. reflexivity. Qed.  
 Example test\_plus3'' : doit3times (plus 3) 0 = 9.  
 Proof. reflexivity. Qed.

## 5.4 Additional Exercises

Module EXERCISES.

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

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

Example test\_fold\_length1 : fold\_length [4;7;0] = 3.

Proof. reflexivity. Qed.

Prove the correctness of fold\_length. (Hint: It may help to know that reflexivity simplifies expressions a bit more aggressively than simpl does – i.e., you may find yourself in a situation where simpl does nothing but reflexivity solves the goal.)

Theorem fold\_length\_correct :  $\forall X (l : \text{list } X),$   
 fold\_length l = length l.

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard (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  
 . *Admitted.*

Write down a theorem *fold\_map\_correct* in Coq stating that fold\_map is correct, and prove it. (Hint: again, remember that reflexivity simplifies expressions a bit more aggressively than simpl.)

Definition manual\_grade\_for\_fold\_map : option (nat × string) := None.

□

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

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

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

We can define currying as follows:

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

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

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

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

Example `test_map1'`: `map (plus 3) [2;0;2] = [5;3;5]`.

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

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

**Check** `@prod_curry`.

**Check** `@prod_uncurry`.

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

**Proof.**

*Admitted.*

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

**Proof.**

*Admitted.*

□

**Exercise: 2 stars, advanced (`nth_error_informal`)** Recall the definition of the `nth_error` function:

Fixpoint `nth_error`  $\{X : \text{Type}\} (l : \text{list } X) (n : \text{nat}) : \text{option } X := \text{match } l \text{ with } | [] => \text{None} | a :: l' => \text{if } n = ?\ 0 \text{ then } \text{Some } a \text{ else } \text{nth\_error } l' (\text{pred } n) \text{ end}.$

Write an informal proof of the following theorem:

forall  $X\ l\ n, \text{length } l = n \rightarrow @\text{nth\_error } X\ l\ n = \text{None}$

Definition manual\_grade\_for\_informal\_proof : option (nat × string) := None.

□

The following exercises explore an alternative way of defining natural numbers, using the so-called *Church numerals*, named after mathematician Alonzo Church. We can represent a natural number  $n$  as a function that takes a function  $f$  as a parameter and returns  $f$  iterated  $n$  times.

Module CHURCH.

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

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

Definition one : cnat :=

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

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

Definition two : cnat :=

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

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

Definition zero : cnat :=

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

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

Definition three : cnat := @doit3times.

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

**Exercise: 1 star, advanced (church\_succ)** Successor of a natural number: given a Church numeral  $n$ , the successor `succ n` is a function that iterates its argument once more than  $n$ . Definition `succ (n : cnat) : cnat`

. *Admitted.*

Example `succ_1 : succ zero = one`.

Proof. *Admitted.*

Example `succ_2 : succ one = two`.

Proof. *Admitted.*

Example `succ_3 : succ two = three`.

Proof. *Admitted.*

□

**Exercise: 1 star, advanced (church\_plus)** Addition of two natural numbers: Definition  
plus (*n m* : cnat) : cnat  
  . *Admitted.*

Example plus\_1 : *plus* zero one = one.

Proof. *Admitted.*

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

Proof. *Admitted.*

Example plus\_3 :

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

Proof. *Admitted.*

□

**Exercise: 2 stars, advanced (church\_mult)** Multiplication: Definition mult (*n m* :  
cnat) : cnat  
  . *Admitted.*

Example mult\_1 : *mult* one one = one.

Proof. *Admitted.*

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

Proof. *Admitted.*

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

Proof. *Admitted.*

□

**Exercise: 2 stars, advanced (church\_exp)** Exponentiation:

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

Definition exp (*n m* : cnat) : cnat  
  . *Admitted.*

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

Proof. *Admitted.*

Example exp\_2 : *exp* three zero = one.

Proof. *Admitted.*

Example exp\_3 : *exp* three two = *plus* (*mult* two (*mult* two two)) one.

Proof. *Admitted.*

□

End CHURCH.

End EXERCISES.

# Chapter 6

## Library LF.Tactics

### 6.1 Tactics: More Basic Tactics

This chapter introduces several additional proof strategies and tactics that allow us to begin proving more interesting properties of functional programs.

We will see:

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

Set *Warnings* "-notation-overridden,-parsing".

From *LF* Require Export Poly.

### 6.2 The apply Tactic

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

Theorem silly1 :  $\forall (n\ m\ o\ p : \text{nat})$ ,

$n = m \rightarrow$   
 $[n; o] = [n; p] \rightarrow$   
 $[n; o] = [m; p]$ .

Proof.

```
intros n m o p eq1 eq2.  
rewrite ← eq1.
```

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

```
apply eq2. Qed.
```

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

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

Proof.

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

Typically, when we use `apply H`, the statement  $H$  will begin with a  $\forall$  that introduces some *universally quantified 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, standard, optional (silly\_ex)** Complete the following proof using only `intros` and `apply`.

```
Theorem silly_ex :
  (∀ n, evenb n = true → oddb (S n) = true) →
  evenb 2 = true →
  oddb 3 = 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 = (n =? 5) →
```



`(S (S n)) =? 7 = true.`

Proof.

`intros n H.`

Here we cannot use `apply` directly, but we can use the `symmetry` tactic, which switches the left and right sides of an equality in the goal.

`symmetry.`

`simpl.` (This `simpl` is optional, since `apply` will perform simplification first, if needed.)  
`apply H. Qed.`

**Exercise: 3 stars, standard (`apply_exercise1`)** *Hint:* You can use `apply` with previously defined lemmas, not just hypotheses in the context. You may find earlier lemmas like `app_nil_r`, `app_assoc`, `rev_app_distr`, `rev_involutive`, etc. helpful. Also, remember that `Search` is your friend (though it may not find earlier lemmas if they were posed as optional problems and you chose not to finish the proofs).

Theorem `rev_exercise1` :  $\forall (l \ l' : \text{list } \text{nat}),$

$l = \text{rev } l' \rightarrow$

$l' = \text{rev } l.$

Proof.

*Admitted.*

□

**Exercise: 1 star, standard, optional (`apply_rewrite`)** Briefly explain the difference between the tactics `apply` and `rewrite`. What are the situations where both can usefully be applied?

## 6.3 The `apply` with `Tactic`

The following silly example uses two rewrites in a row to get from `[a;b]` to `[e;f]`.

Example `trans_eq_example` :  $\forall (a \ b \ c \ d \ e \ f : \text{nat}),$

$[a;b] = [c;d] \rightarrow$

$[c;d] = [e;f] \rightarrow$

$[a;b] = [e;f].$

Proof.

`intros a b c d e f eq1 eq2.`

`rewrite  $\rightarrow$  eq1. rewrite  $\rightarrow$  eq2. reflexivity. Qed.`

Since this is a common pattern, we might like to pull it out as a lemma that records, once and for all, the fact that equality is transitive.

Theorem `trans_eq` :  $\forall (X:\text{Type}) (n \ m \ o : X),$

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

Proof.

```

intros X n m o eq1 eq2. rewrite → eq1. rewrite → eq2.
reflexivity. Qed.

```

Now, we should be able to use `trans_eq` to prove the above example. However, to do this we need a slight refinement of the `apply` tactic.

```

Example trans_eq_example' : ∀ (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.

```

If we simply tell Coq `apply trans_eq` at this point, it can tell (by matching the goal against the conclusion of the lemma) that it should instantiate `X` with `[nat]`, `n` with `[a,b]`, and `o` with `[e,f]`. However, the matching process doesn't determine an instantiation for `m`: we have to supply one explicitly by adding "`with (m:=[c,d])`" to the invocation of `apply`.

```

apply trans_eq with (m:=[c;d]).
apply eq1. apply eq2. Qed.

```

(Actually, we usually don't have to include the name `m` in the `with` clause; Coq is often smart enough to figure out which variable we are instantiating. We could instead write `apply trans_eq with [c;d].`)

Coq also has a tactic `transitivity` that accomplishes the same purpose as applying `trans_eq`. The tactic requires us to state the instantiation we want, just like `apply with` does.

```

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.
transitivity [c;d].
apply eq1. apply eq2. Qed.

```

**Exercise: 3 stars, standard, optional (trans\_eq\_exercise)** Example trans\_eq\_exercise

```

: ∀ (n m o p : nat),
  m = (minustwo o) →
  (n + p) = m →
  (n + p) = (minustwo o).

```

Proof.

*Admitted.*

□

## 6.4 The injection and discriminate Tactics

Recall the definition of natural numbers:

Inductive nat : Type := | O | S (n : nat).

It is obvious from this definition that every number has one of two forms: either it is the constructor `O` or it is built by applying the constructor `S` to another number. But there is more here than meets the eye: implicit in the definition are two more facts:

- The constructor `S` is *injective*, or *one-to-one*. That is, if `S n = S m`, it must be that `n = m`.
- The constructors `O` and `S` are *disjoint*. That is, `O` is not equal to `S n` for any `n`.

Similar principles apply to all inductively defined types: all constructors are injective, and the values built from distinct constructors are never equal. For lists, the `cons` constructor is injective and `nil` is different from every non-empty list. For booleans, `true` and `false` are different. (Since `true` and `false` take no arguments, their injectivity is neither here nor there.) And so on.

For example, we can prove the injectivity of `S` by using the `pred` function defined in *Basics.v*.

Theorem S\_injective :  $\forall (n\ m : \text{nat}),$

`S n = S m`  $\rightarrow$

`n = m`.

Proof.

`intros n m H1.`

`assert (H2: n = pred (S n)). { reflexivity. }`

`rewrite H2. rewrite H1. reflexivity.`

Qed.

This technique can be generalized to any constructor by writing the equivalent of `pred` – i.e., writing a function that “undoes” one application of the constructor. As a more convenient alternative, Coq provides a tactic called `injection` that allows us to exploit the injectivity of any constructor. Here is an alternate proof of the above theorem using `injection`:

Theorem S\_injective' :  $\forall (n\ m : \text{nat}),$

`S n = S m`  $\rightarrow$

`n = m`.

Proof.

`intros n m H.`

By writing `injection H` as `Hmn` at this point, we are asking Coq to generate all equations that it can infer from `H` using the injectivity of constructors (in the present example, the equation `n = m`). Each such equation is added as a hypothesis (with the name `Hmn` in this case) into the context.

`injection H as Hnm. apply Hnm.`

Qed.

Here's a more interesting example that shows how `injection` can derive multiple equations at once.

Theorem `injection_ex1` :  $\forall (n\ m\ o : \text{nat}),$

$[n; m] = [o; o] \rightarrow$

$[n] = [m].$

Proof.

`intros n m o H.`

`injection H as H1 H2.`

`rewrite H1. rewrite H2. reflexivity.`

Qed.

Alternatively, if you just say `injection H` with no `as` clause, then all the equations will be turned into hypotheses at the beginning of the goal.

Theorem `injection_ex2` :  $\forall (n\ m\ o : \text{nat}),$

$[n; m] = [o; o] \rightarrow$

$[n] = [m].$

Proof.

`intros n m o H.`

`injection H.`

`intros H1 H2. rewrite H1. rewrite H2. reflexivity.`

Qed.

**Exercise: 3 stars, standard (`injection_ex3`)** Example `injection_ex3` :  $\forall (X : \text{Type}) (x$

$y\ z : X) (l\ j : \text{list } X),$

$x :: y :: l = z :: j \rightarrow$

$j = z :: l \rightarrow$

$x = y.$

Proof.

*Admitted.*

□

So much for injectivity of constructors. What about disjointness?

The principle of disjointness says that two terms beginning with different constructors (like `O` and `S`, or `true` and `false`) can never be equal. This means that, any time we find ourselves in a context where we've *assumed* that two such terms are equal, we are justified in concluding anything we want, since the assumption is nonsensical.

The `discriminate` tactic embodies this principle: It is used on a hypothesis involving an equality between different constructors (e.g., `S n = O`), and it solves the current goal immediately. Here is an example:

Theorem `eqb_0_l` :  $\forall n,$

$0 =? n = \text{true} \rightarrow n = 0.$

Proof.

```
intros n.
```

We can proceed by case analysis on  $n$ . The first case is trivial.

```
destruct n as [| n'] eqn:E.
```

```
-
```

```
  intros H. reflexivity.
```

However, the second one doesn't look so simple: assuming  $0 = ? (S\ n') = \text{true}$ , we must show  $S\ n' = 0$ ! The way forward is to observe that the assumption itself is nonsensical:

```
-
```

```
  simpl.
```

If we use `discriminate` on this hypothesis, Coq confirms that the subgoal we are working on is impossible and removes it from further consideration.

```
  intros H. discriminate H.
```

Qed.

This is an instance of a logical principle known as the *principle of explosion*, which asserts that a contradictory hypothesis entails anything (even false things!).

Theorem `discriminate_ex1` :  $\forall (n : \text{nat})$ ,

$S\ n = 0 \rightarrow$

$2 + 2 = 5$ .

Proof.

```
  intros n contra. discriminate contra. Qed.
```

Theorem `discriminate_ex2` :  $\forall (n\ m : \text{nat})$ ,

$\text{false} = \text{true} \rightarrow$

$[n] = [m]$ .

Proof.

```
  intros n m contra. discriminate contra. Qed.
```

If you find the principle of explosion confusing, remember that these proofs are *not* showing that the conclusion of the statement holds. Rather, they are showing that, *if* the nonsensical situation described by the premise did somehow arise, *then* the nonsensical conclusion would also follow, because we'd be living in an inconsistent universe where every statement is true. We'll explore the principle of explosion in more detail in the next chapter.

**Exercise: 1 star, standard (`discriminate_ex3`)** Example `discriminate_ex3` :

$\forall (X : \text{Type}) (x\ y\ z : X) (l\ j : \text{list } X)$ ,

$x :: y :: l = [] \rightarrow$

$x = z$ .

Proof.

*Admitted.*

□

The injectivity of constructors allows us to reason that  $\forall (n\ m : \mathbf{nat}), S\ n = S\ m \rightarrow n = m$ . The converse of this implication is an instance of a more general fact about both constructors and functions, which we will find convenient in a few places below:

**Theorem** `f_equal` :  $\forall (A\ B : \mathbf{Type}) (f : A \rightarrow B) (x\ y : A),$   
 $x = y \rightarrow f\ x = f\ y.$

**Proof.** `intros A B f x y eq. rewrite eq. reflexivity. Qed.`

**Theorem** `eq_implies_succ_equal` :  $\forall (n\ m : \mathbf{nat}),$   
 $n = m \rightarrow S\ n = S\ m.$

**Proof.** `intros n m H. apply f_equal. apply H. Qed.`

There is also a tactic named ‘`f_equal`’ that can prove such theorems. Given a goal of the form  $f\ a1 \dots an = g\ b1 \dots bn$ , the tactic `f_equal` will produce subgoals of the form  $f = g$ ,  $a1 = b1$ , ...,  $an = bn$ . At the same time, any of these subgoals that are simple enough (e.g., immediately provable by `reflexivity`) will be automatically discharged by `f_equal`.

**Theorem** `eq_implies_succ_equal'` :  $\forall (n\ m : \mathbf{nat}),$   
 $n = m \rightarrow S\ n = S\ m.$

**Proof.** `intros n m H. f_equal. apply H. Qed.`

## 6.5 Using Tactics on Hypotheses

By default, most tactics work on the goal formula and leave the context unchanged. However, most tactics also have a variant that performs a similar operation on a statement in the context.

For example, the tactic “`simpl in H`” performs simplification on the hypothesis  $H$  in the context.

**Theorem** `S_inj` :  $\forall (n\ m : \mathbf{nat}) (b : \mathbf{bool}),$   
 $(S\ n) =? (S\ m) = b \rightarrow$   
 $n =? m = b.$

**Proof.**

`intros n m b H. simpl in H. apply H. Qed.`

Similarly, `apply L in H` matches some conditional statement  $L$  (of the form  $X \rightarrow Y$ , say) against a hypothesis  $H$  in the context. However, unlike ordinary `apply` (which rewrites a goal matching  $Y$  into a subgoal  $X$ ), `apply L in H` matches  $H$  against  $X$  and, if successful, replaces it with  $Y$ .

In other words, `apply L in H` gives us a form of “forward reasoning”: from  $X \rightarrow Y$  and a hypothesis matching  $X$ , it produces a hypothesis matching  $Y$ . By contrast, `apply L` is “backward reasoning”: it says that if we know  $X \rightarrow Y$  and we are trying to prove  $Y$ , it suffices to prove  $X$ .

Here is a variant of a proof from above, using forward reasoning throughout instead of backward reasoning.

**Theorem** `silly3'` :  $\forall (n : \mathbf{nat}),$

```

(n =? 5 = true → (S (S n)) =? 7 = true) →
true = (n =? 5) →
true = ((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.

The informal proofs that you’ve seen in math or computer science classes probably tended to use forward reasoning. In general, idiomatic use of Coq favors backward reasoning, but in some situations the forward style can be easier to think about.

## 6.6 Varying the Induction Hypothesis

Sometimes it is important to control the exact form of the induction hypothesis when carrying out inductive proofs in Coq. In particular, we sometimes need to be careful about which of the assumptions we move (using `intros`) from the goal to the context before invoking the `induction` tactic. For example, suppose we want to show that `double` is injective – i.e., that it maps different arguments to different results:

Theorem `double_injective`: forall n m, double n = double m -> n = m.

The way we start this proof is a bit delicate: if we begin 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` :  $\forall n\ m,$

```

double n = double m →
n = m.

```

Proof.

```

intros n m. induction n as [| n' IHn'].
- simpl. intros eq. destruct m as [| m'] eqn:E.
  + reflexivity.
  + discriminate eq.
- intros eq. destruct m as [| m'] eqn:E.
  + discriminate eq.
  + apply f_equal.

```

At this point, the induction hypothesis (`IHn'`) does *not* give us  $n' = m'$  – there is an extra `S` in the way – so the goal is not provable.

Abort.

What went wrong?

The problem is that, at the point we invoke the induction hypothesis, we have already introduced  $m$  into the context – intuitively, we have told Coq, “Let’s consider some particular  $n$  and  $m$ ...” and we now have to prove that, if  $\text{double } n = \text{double } m$  for *those particular*  $n$  and  $m$ , then  $n = m$ .

The next tactic, `induction n` says to Coq: We are going to show the goal by induction on  $n$ . That is, we are going to prove, for *all*  $n$ , that the proposition

- $P\ n = \text{“if double } n = \text{double } m, \text{ then } n = m\text{”}$

holds, by showing

- $P\ 0$   
(i.e., “if  $\text{double } 0 = \text{double } m$  then  $0 = m$ ”) and
- $P\ n \rightarrow P\ (\text{S } n)$   
(i.e., “if  $\text{double } n = \text{double } m$  then  $n = m$ ” implies “if  $\text{double } (\text{S } n) = \text{double } m$  then  $\text{S } n = m$ ”).

If we look closely at the second statement, it is saying something rather strange: that, for a *particular*  $m$ , if we know

- “if  $\text{double } n = \text{double } m$  then  $n = m$ ”

then we can prove

- “if  $\text{double } (\text{S } n) = \text{double } m$  then  $\text{S } n = m$ ”.

To see why this is strange, let’s think of a particular (arbitrary, but fixed)  $m$  – say, 5. The statement is then saying that, if we know

- $Q = \text{“if double } n = 10 \text{ then } n = 5\text{”}$

then we can prove

- $R = \text{“if double } (\text{S } n) = 10 \text{ then } \text{S } n = 5\text{”}.$

But knowing  $Q$  doesn’t give us any help at all with proving  $R$ ! If we tried to prove  $R$  from  $Q$ , we would start with something like “Suppose  $\text{double } (\text{S } n) = 10$ ...” but then we’d be stuck: knowing that  $\text{double } (\text{S } n)$  is 10 tells us nothing helpful about whether  $\text{double } n$  is 10 (indeed, it strongly suggests that  $\text{double } n$  is *not* 10!!), so  $Q$  is useless.

Trying to carry out this proof by induction on  $n$  when  $m$  is already in the context doesn’t work because we are then trying to prove a statement involving *every*  $n$  but just a *single*  $m$ .



A successful 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' IHn'].
- simpl. intros m eq. destruct m as [| m'] eqn:E.
  + reflexivity.
  + discriminate eq.
- simpl.
```

Notice that both the goal and the induction hypothesis are different this time: the goal asks us to prove something more general (i.e., to prove the statement for *every*  $m$ ), but the IH is correspondingly more flexible, allowing us to choose whichever  $m$  we like when we apply the IH.

```
intros m eq.
```

Now we've chosen a particular  $m$  and introduced the assumption that `double n = double m`. Since we are doing a case analysis on  $n$ , we also need a case analysis on  $m$  to keep the two “in sync.”

```
destruct m as [| m'] eqn:E.
+
```

The 0 case is trivial:

```
discriminate eq.
+
  apply f_equal.
```

At this point, since we are in the second branch of the `destruct m`, the  $m'$  mentioned in the context is the predecessor of the  $m$  we started out talking about. Since we are also in the S branch of the induction, this is perfect: if we instantiate the generic  $m$  in the IH with the current  $m'$  (this instantiation is performed automatically by the `apply` in the next step), then `IHn'` gives us exactly what we need to finish the proof.

```
apply IHn'. simpl in eq. injection eq as goal. apply goal. Qed.
```

What you should take away from all this is that we need to be careful, when using induction, that we are not trying to prove something too specific: When proving a property involving two variables  $n$  and  $m$  by induction on  $n$ , it is sometimes crucial to leave  $m$  generic.

The following exercise follows the same pattern.

**Exercise: 2 stars, standard (`eqb_true`)** Theorem `eqb_true` :  $\forall n m,$   
 $n = ? m = \text{true} \rightarrow n = m.$

Proof.

*Admitted.*

□

**Exercise: 2 stars, advanced (eqb\_true\_informal)** Give a careful informal proof of `eqb_true`, being as explicit as possible about quantifiers.

Definition `manual_grade_for_informal_proof` : **option** (**nat** × **string**) := **None**.

□

**Exercise: 3 stars, standard, especially useful (plus\_n\_n\_injective)** In addition to being careful about how you use `intros`, practice using “in” variants in this proof. (Hint: use `plus_n_Sm`.) Theorem `plus_n_n_injective` :  $\forall n\ m,$

$$\begin{aligned} n + n = m + m &\rightarrow \\ n = m. \end{aligned}$$

Proof.

*Admitted.*

□

The strategy of doing fewer `intros` before an `induction` to obtain a more general IH doesn't always work by itself; sometimes some *rearrangement* of quantified variables is needed. Suppose, for example, that we wanted to prove `double_injective` by induction on  $m$  instead of  $n$ .

Theorem `double_injective_take2_FAILED` :  $\forall n\ m,$   
 $\text{double } n = \text{double } m \rightarrow$   
 $n = m.$

Proof.

```
intros n m. induction m as [| m' IHm'].
- simpl. intros eq. destruct n as [| n'] eqn:E.
  + reflexivity.
  + discriminate eq.
- intros eq. destruct n as [| n'] eqn:E.
  + discriminate eq.
  + apply f_equal.
```

Abort.

The problem is that, to do induction on  $m$ , we must first introduce  $n$ . (And 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 works, but it's not nice: We don't want to have to twist the statements of lemmas to fit the needs of a particular strategy for proving them! Rather we want to state them in the clearest and most natural way.

What we can do instead is to first introduce all the quantified variables and then *re-generalize* one or more of them, selectively taking variables out of the context and putting

them back at the beginning of the goal. The `generalize dependent` tactic does this.

```
Theorem double_injective_take2 : ∀ n m,
  double n = double m →
  n = m.
```

Proof.

```
  intros n m.
  generalize dependent n.
  induction m as [| m' IHm'].
- simpl. intros n eq. destruct n as [| n'] eqn:E.
  + reflexivity.
  + discriminate eq.
- intros n eq. destruct n as [| n'] eqn:E.
  + discriminate eq.
  + apply f_equal.
    apply IHm'. injection eq as goal. apply goal. 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  $\text{double } n = \text{double } m$ , then  $n = m$ .

*Proof:* Let  $m$  be a **nat**. We prove by induction on  $m$  that, for any  $n$ , if  $\text{double } n = \text{double } m$  then  $n = m$ .

- First, suppose  $m = 0$ , and suppose  $n$  is a number such that  $\text{double } n = \text{double } m$ . We must show that  $n = 0$ .

Since  $m = 0$ , by the definition of `double` we have  $\text{double } n = 0$ . There are two cases to consider for  $n$ . If  $n = 0$  we are done, since  $m = 0 = n$ , as required. Otherwise, if  $n = S \ n'$  for some  $n'$ , we derive a contradiction: by the definition of `double`, we can calculate  $\text{double } n = S \ (S \ (\text{double } n'))$ , but this contradicts the assumption that  $\text{double } n = 0$ .

- Second, suppose  $m = S \ m'$  and that  $n$  is again a number such that  $\text{double } n = \text{double } m$ . We must show that  $n = S \ m'$ , with the induction hypothesis that for every number  $s$ , if  $\text{double } s = \text{double } m'$  then  $s = m'$ .

By the fact that  $m = S \ m'$  and the definition of `double`, we have  $\text{double } n = S \ (S \ (\text{double } m'))$ . There are two cases to consider for  $n$ .

If  $n = 0$ , then by definition  $\text{double } n = 0$ , a contradiction.

Thus, we may assume that  $n = S \ n'$  for some  $n'$ , and again by the definition of `double` we have  $S \ (S \ (\text{double } n')) = S \ (S \ (\text{double } m'))$ , which implies by injectivity that  $\text{double } n' = \text{double } m'$ . Instantiating the induction hypothesis with  $n'$  thus allows us to conclude that  $n' = m'$ , and it follows immediately that  $S \ n' = S \ m'$ . Since  $S \ n' = n$  and  $S \ m' = m$ , this is just what we wanted to show.  $\square$

**Exercise: 3 stars, standard, especially useful (gen\_dep\_practice)** Prove this by induction on  $l$ .

Theorem `nth_error_after_last`:  $\forall (n : \text{nat}) (X : \text{Type}) (l : \text{list } X),$   
     $\text{length } l = n \rightarrow$   
     $\text{nth\_error } l \ n = \text{None}.$

Proof.

*Admitted.*

□

## 6.7 Unfolding Definitions

It sometimes happens that we need to manually unfold a name that has been introduced by a **Definition** so that we can manipulate the expression it denotes. For example, if we define...

**Definition** `square`  $n := n \times n.$

... and try to prove a simple fact about `square`...

Lemma `square_mult` :  $\forall n \ m, \text{square } (n \times m) = \text{square } n \times \text{square } m.$

Proof.

`intros n m.`

`simpl.`

... we appear to be stuck: `simpl` doesn't simplify anything, and since we haven't proved any other facts about `square`, there is nothing we can `apply` or `rewrite` with.

To make progress, we can manually **unfold** the definition of `square`:

`unfold square.`

Now we have plenty to work with: both sides of the equality are expressions involving multiplication, and we have lots of facts about multiplication at our disposal. In particular, we know that it is commutative and associative, and from these it is not hard to finish the proof.

`rewrite mult_assoc.`

`assert (H :  $n \times m \times n = n \times n \times m$ ).`

`{ rewrite mult_comm. apply mult_assoc. }`

`rewrite H. rewrite mult_assoc. reflexivity.`

**Qed.**

At this point, some discussion of unfolding and simplification is in order.

We already have observed that tactics like `simpl`, `reflexivity`, and `apply` will often unfold the definitions of functions automatically when this allows them to make progress. For example, if we define `foo m` to be the constant 5...

**Definition** `foo` ( $x : \text{nat}$ ) := 5.

.... then the `simpl` in the following proof (or the `reflexivity`, if we omit the `simpl`) will unfold `foo m` to `(fun x => 5) m` and then further simplify this expression to just 5.

```
Fact silly_fact_1 : ∀ m, foo m + 1 = foo (m + 1) + 1.
```

```
Proof.
```

```
  intros m.
```

```
  simpl.
```

```
  reflexivity.
```

```
Qed.
```

However, this automatic unfolding is somewhat conservative. For example, if we define a slightly more complicated function involving a pattern match...

```
Definition bar x :=
```

```
  match x with
```

```
  | O => 5
```

```
  | S _ => 5
```

```
end.
```

...then the analogous proof will get stuck:

```
Fact silly_fact_2_FAILED : ∀ m, bar m + 1 = bar (m + 1) + 1.
```

```
Proof.
```

```
  intros m.
```

```
  simpl. Abort.
```

The reason that `simpl` doesn't make progress here is that it notices that, after tentatively unfolding `bar m`, it is left with a match whose scrutinee, `m`, is a variable, so the `match` cannot be simplified further. It is not smart enough to notice that the two branches of the `match` are identical, so it gives up on unfolding `bar m` and leaves it alone. Similarly, tentatively unfolding `bar (m+1)` leaves a `match` whose scrutinee is a function application (that cannot itself be simplified, even after unfolding the definition of `+`), so `simpl` leaves it alone.

At this point, there are two ways to make progress. One is to use `destruct m` to break the proof into two cases, each focusing on a more concrete choice of `m` (`O` vs `S _`). In each case, the `match` inside of `bar` can now make progress, and the proof is easy to complete.

```
Fact silly_fact_2 : ∀ m, bar m + 1 = bar (m + 1) + 1.
```

```
Proof.
```

```
  intros m.
```

```
  destruct m eqn:E.
```

```
  - simpl. reflexivity.
```

```
  - simpl. reflexivity.
```

```
Qed.
```

This approach works, but it depends on our recognizing that the `match` hidden inside `bar` is what was preventing us from making progress.

A more straightforward way forward is to explicitly tell Coq to unfold `bar`.

```
Fact silly_fact_2' : ∀ m, bar m + 1 = bar (m + 1) + 1.
```

Proof.

```
intros m.  
unfold bar.
```

Now it is apparent that we are stuck on the `match` expressions on both sides of the `=`, and we can use `destruct` to finish the proof without thinking too hard.

```
destruct m eqn:E.  
- reflexivity.  
- reflexivity.
```

Qed.

## 6.8 Using destruct on Compound Expressions

We have seen many examples where `destruct` is used to perform case analysis of the value of some variable. 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 n =? 3 then false  
  else if n =? 5 then false  
  else false.
```

```
Theorem sillyfun_false : ∀ (n : nat),  
  sillyfun n = false.
```

Proof.

```
intros n. unfold sillyfun.  
destruct (n =? 3) eqn:E1.  
- reflexivity.  
- destruct (n =? 5) eqn:E2.  
  + reflexivity.  
  + reflexivity. Qed.
```

After unfolding `sillyfun` in the above proof, we find that we are stuck on `if (n =? 3) then ... else ...`. But either `n` is equal to 3 or it isn't, so we can use `destruct (eqb n 3)` to let us reason about the two cases.

In general, the `destruct` tactic can be used to perform case analysis of the results of arbitrary computations. If `e` is an expression whose type is some inductively defined type `T`, then, for each constructor `c` of `T`, `destruct e` generates a subgoal in which all occurrences of `e` (in the goal and in the context) are replaced by `c`.

**Exercise: 3 stars, standard (combine\_split)** Here is an implementation of the `split` function mentioned in chapter Poly:

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

```

      : (list X) × (list Y) :=
match l with
| [] ⇒ ([], [])
| (x, y) :: t ⇒
  match split t with
  | (lx, ly) ⇒ (x :: lx, y :: ly)
  end
end.

```

Prove that `split` and `combine` are inverses in the following sense:

**Theorem** `combine_split` :  $\forall X Y (l : \text{list } (X \times Y)) \ l1 \ l2,$   
`split l = (l1, l2) →`  
`combine l1 l2 = l.`

**Proof.**

*Admitted.*

□

The *eqn*: part of the `destruct` tactic is optional: So far, we've chosen to include it most of the time, just for the sake of documentation.

However, when `destructing` compound expressions, the information recorded by the *eqn*: can actually be critical: if we leave it out, then `destruct` can erase information we need to complete a proof.

For example, suppose we define a function `sillyfun1` like this:

**Definition** `sillyfun1` ( $n : \text{nat}$ ) : `bool` :=  
`if n =? 3 then true`  
`else if n =? 5 then true`  
`else false.`

Now suppose that we want to convince Coq that `sillyfun1 n` yields `true` only when  $n$  is odd. If we start the proof like this (with no *eqn*: on the `destruct`)...

**Theorem** `sillyfun1_odd_FAILED` :  $\forall (n : \text{nat}),$   
`sillyfun1 n = true →`  
`oddb n = true.`

**Proof.**

```

intros n eq. unfold sillyfun1 in eq.
destruct (n =? 3).

```

**Abort.**

... then we are stuck at this point because the context does not contain enough information to prove the goal! The problem is that the substitution performed by `destruct` is quite brutal – in this case, it throws away every occurrence of `n =? 3`, but we need to keep some memory of this expression and how it was destructed, because we need to be able to reason that, since `n =? 3 = true` in this branch of the case analysis, it must be that  $n = 3$ , from which it follows that  $n$  is odd.

What we want here is to substitute away all existing occurrences of  $n =? 3$ , but at the same time add an equation to the context that records which case we are in. This is precisely what the *eqn*: qualifier does.

Theorem `sillyfun1_odd` :  $\forall (n : \text{nat})$ ,  
     `sillyfun1`  $n = \text{true} \rightarrow$   
     `oddb`  $n = \text{true}$ .

Proof.

```
intros n eq. unfold sillyfun1 in eq.
destruct (n =? 3) eqn:Heqe3.
- apply eqb_true in Heqe3.
  rewrite → Heqe3. reflexivity.
-

destruct (n =? 5) eqn:Heqe5.
+
  apply eqb_true in Heqe5.
  rewrite → Heqe5. reflexivity.
+ discriminate eq. Qed.
```

**Exercise: 2 stars, standard (`destruct_eqn_practice`)** Theorem `bool_fn_applied_thrice` :

$\forall (f : \text{bool} \rightarrow \text{bool}) (b : \text{bool})$ ,  
 $f (f (f b)) = f b$ .

Proof.

*Admitted.*

□

## 6.9 Review

We've now seen many of Coq's most fundamental tactics. We'll introduce a few more in the coming chapters, and later on we'll see some more powerful *automation* tactics that make Coq help us with low-level details. But basically we've got what we need to get work done.

Here are the ones we've seen:

- **intros**: move hypotheses/variables from goal to context
- **reflexivity**: finish the proof (when the goal looks like  $e = e$ )
- **apply**: prove goal using a hypothesis, lemma, or constructor
- **apply... in  $H$** : apply a hypothesis, lemma, or constructor to a hypothesis in the context (forward reasoning)



- `apply... with...:` explicitly specify values for variables that cannot be determined by pattern matching
- `simpl:` simplify computations in the goal
- `simpl in H: ...` or a hypothesis
- `rewrite:` use an equality hypothesis (or lemma) to rewrite the goal
- `rewrite ... in H: ...` or a hypothesis
- `symmetry:` changes a goal of the form  $t=u$  into  $u=t$
- `symmetry in H:` changes a hypothesis of the form  $t=u$  into  $u=t$
- `transitivity y:` prove a goal  $x=z$  by proving two new subgoals,  $x=y$  and  $y=z$
- `unfold:` replace a defined constant by its right-hand side in the goal
- `unfold... in H: ...` or a hypothesis
- `destruct... as...:` case analysis on values of inductively defined types
- `destruct... eqn:...:` specify the name of an equation to be added to the context, recording the result of the case analysis
- `induction... as...:` induction on values of inductively defined types
- `injection:` reason by injectivity on equalities between values of inductively defined types
- `discriminate:` reason by disjointness of constructors on equalities between values of inductively defined types
- `assert (H: e)` (or `assert (e) as H`): introduce a “local lemma”  $e$  and call it  $H$
- `generalize dependent x:` move the variable  $x$  (and anything else that depends on it) from the context back to an explicit hypothesis in the goal formula
- `f_equal:` change a goal of the form  $f\ x = f\ y$  into  $x = y$

## 6.10 Additional Exercises

**Exercise: 3 stars, standard (eqb\_sym)** Theorem `eqb_sym` :  $\forall (n\ m : \text{nat}),$   
 $(n =? m) = (m =? n).$

Proof.

*Admitted.*

□

**Exercise: 3 stars, advanced, optional (eqb\_sym\_informal)** Give an informal proof of this lemma that corresponds to your formal proof above:

Theorem: For any **nats**  $n\ m$ ,  $(n =? m) = (m =? n)$ .

Proof:

**Exercise: 3 stars, standard, optional (eqb\_trans)** Theorem `eqb_trans` :  $\forall\ n\ m\ p$ ,

$n =? m = \text{true} \rightarrow$

$m =? p = \text{true} \rightarrow$

$n =? p = \text{true}.$

Proof.

*Admitted.*

□

**Exercise: 3 stars, advanced (split\_combine)** We proved, in an exercise above, that for all lists of pairs, `combine` is the inverse of `split`. How would you formalize the statement that `split` is the inverse of `combine`? When is this property true?

Complete the definition of `split_combine_statement` below with a property that states that `split` is the inverse of `combine`. Then, prove that the property holds. (Be sure to leave your induction hypothesis general by not doing `intros` on more things than necessary. Hint: what property do you need of  $l1$  and  $l2$  for `split (combine  $l1\ l2$ ) = ( $l1, l2$ )` to be true?)

Definition `split_combine_statement` : Prop

. *Admitted.*

Theorem `split_combine` : *split\_combine\_statement*.

Proof.

*Admitted.*

Definition `manual_grade_for_split_combine` : **option** (**nat**×**string**) := **None**.

□

**Exercise: 3 stars, advanced (filter\_exercise)** This one is a bit challenging. Pay attention to the form of your induction hypothesis.

Theorem `filter_exercise` :  $\forall\ (X : \text{Type})\ (test : X \rightarrow \text{bool})$   
 $(x : X)\ (l\ lf : \text{list } X),$

`filter test l = x` ::  $lf \rightarrow$

`test x = true`.

Proof.

*Admitted.*

□

**Exercise: 4 stars, advanced, especially useful (forall\_exists\_challenge)** Define two recursive *Fixpoints*, `forallb` and `existsb`. The first checks whether every element in a list satisfies a given predicate:

```
forallb oddb 1;3;5;7;9 = true
forallb negb false;false = true
forallb evenb 0;2;4;5 = false
forallb (eqb 5) [] = true
```

The second checks whether there exists an element in the list that satisfies a given predicate:

```
existsb (eqb 5) 0;2;3;6 = false
existsb (andb true) true;true;false = true
existsb oddb 1;0;0;0;0;3 = true
existsb evenb [] = false
```

Next, define a *nonrecursive* version of `existsb` – call it `existsb'` – using `forallb` and `negb`.

Finally, prove a theorem `existsb_existsb'` stating that `existsb'` and `existsb` have the same behavior.

```
Fixpoint forallb {X : Type} (test : X → bool) (l : list X) : bool
. Admitted.
```

Example `test_forallb_1 : forallb oddb [1;3;5;7;9] = true.`

Proof. *Admitted.*

Example `test_forallb_2 : forallb negb [false;false] = true.`

Proof. *Admitted.*

Example `test_forallb_3 : forallb evenb [0;2;4;5] = false.`

Proof. *Admitted.*

Example `test_forallb_4 : forallb (eqb 5) [] = true.`

Proof. *Admitted.*

```
Fixpoint existsb {X : Type} (test : X → bool) (l : list X) : bool
. Admitted.
```

Example `test_existsb_1 : existsb (eqb 5) [0;2;3;6] = false.`

Proof. *Admitted.*

Example `test_existsb_2 : existsb (andb true) [true;true;false] = true.`

Proof. *Admitted.*

Example `test_existsb_3 : existsb oddb [1;0;0;0;0;3] = true.`

Proof. *Admitted.*

Example `test_existsb_4 : existsb evenb [] = false.`

Proof. *Admitted.*

```
Definition existsb' {X : Type} (test : X → bool) (l : list X) : bool
. Admitted.
```

Theorem `existsb_existsb' : ∀ (X : Type) (test : X → bool) (l : list X),`

$\text{existsb } test\ l = \text{existsb}'\ test\ l.$

*Proof. Admitted.*

□

# Chapter 7

## Library LF.Logic

### 7.1 Logic: Logic in Coq

Set Warnings "-notation-overridden,-parsing".  
From LF Require Export Tactics.

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* ( $e1 = e2$ ), implications ( $P \rightarrow Q$ ), and quantified propositions ( $\forall x, P$ ). In this chapter, we will see how Coq can be used to carry out other familiar forms of logical reasoning.

Before diving into details, let's talk a bit about the status of mathematical statements in Coq. Recall that Coq is a *typed* language, which means that every sensible expression in its world has an associated type. Logical claims are no exception: any statement we might try to prove in Coq has a type, namely **Prop**, the type of *propositions*. We can see this with the **Check** command:

```
Check 3 = 3 : Prop.
```

```
Check  $\forall n\ m : \text{nat}, n + m = m + n : \text{Prop}.$ 
```

Note that *all* syntactically well-formed propositions have type **Prop** in Coq, regardless of whether they are true.

Simply *being* a proposition is one thing; being *provable* is something else!

```
Check 2 = 2 : Prop.
```

```
Check 3 = 2 : Prop.
```

```
Check  $\forall n : \text{nat}, n = 2 : \text{Prop}.$ 
```

Indeed, propositions not only have types: they are *first-class* entities that can be manipulated in all the same ways as any of the other things in Coq's world.

So far, we've seen one primary place that propositions can appear: in **Theorem** (and **Lemma** and **Example**) declarations.

```
Theorem plus_2_2_is_4 :
```

$2 + 2 = 4$ .

Proof. reflexivity. Qed.

But propositions can be used in many other ways. For example, we can give a name to a proposition using a **Definition**, just as we have given names to other kinds of expressions.

Definition plus\_claim : Prop :=  $2 + 2 = 4$ .

Check plus\_claim : Prop.

We can later use this name in any situation where a proposition is expected – for example, as the claim in a **Theorem** declaration.

Theorem plus\_claim\_is\_true :

plus\_claim.

Proof. reflexivity. Qed.

We can also write *parameterized* propositions – that is, functions that take arguments of some type and return a proposition.

For instance, the following function takes a number and returns a proposition asserting that this number is equal to three:

Definition is\_three (n : nat) : Prop :=

n = 3.

Check is\_three : nat → Prop.

In Coq, functions that return propositions are said to define *properties* of their arguments. For instance, here's a (polymorphic) property defining the familiar notion of an *injective function*.

Definition injective {A B} (f : A → B) :=

$\forall x\ y : A, f\ x = f\ y \rightarrow x = y$ .

Lemma succ\_inj : injective S.

Proof.

intros n m H. injection H as H1. apply H1.

Qed.

The equality operator = is also a function that returns a **Prop**.

The expression  $n = m$  is syntactic sugar for **eq**  $n\ m$  (defined in Coq's standard library using the **Notation** mechanism). Because **eq** can be used with elements of any type, it is also polymorphic:

Check @eq :  $\forall A : \text{Type}, A \rightarrow A \rightarrow \text{Prop}$ .

(Notice that we wrote @eq instead of eq: The type argument  $A$  to eq is declared as implicit, and we need to turn off the inference of this implicit argument to see the full type of eq.)

## 7.2 Logical Connectives

### 7.2.1 Conjunction

The *conjunction*, or *logical and*, of propositions  $A$  and  $B$  is written  $A \wedge B$ , representing the claim that both  $A$  and  $B$  are true.

Example and\_example :  $3 + 4 = 7 \wedge 2 \times 2 = 4$ .

To prove a conjunction, use the `split` tactic. It will generate two subgoals, one for each part of the statement:

Proof.

split.

- reflexivity.

- reflexivity.

Qed.

For any propositions  $A$  and  $B$ , if we assume that  $A$  is true and that  $B$  is true, we can conclude that  $A \wedge B$  is also true.

Lemma and\_intro :  $\forall A B : \text{Prop}, A \rightarrow B \rightarrow A \wedge B$ .

Proof.

intros  $A B HA HB$ . split.

- apply  $HA$ .

- apply  $HB$ .

Qed.

Since applying a theorem with hypotheses to some goal has the effect of generating as many subgoals as there are hypotheses for that theorem, we can apply `and_intro` to achieve the same effect as `split`.

Example and\_example' :  $3 + 4 = 7 \wedge 2 \times 2 = 4$ .

Proof.

apply and\_intro.

- reflexivity.

- reflexivity.

Qed.

**Exercise: 2 stars, standard (and\_exercise)** Example and\_exercise :

$\forall n m : \text{nat}, n + m = 0 \rightarrow n = 0 \wedge m = 0$ .

Proof.

*Admitted.*

□

So much for proving conjunctive statements. To go in the other direction – i.e., to *use* a conjunctive hypothesis to help prove something else – we employ the `destruct` tactic.

If the proof context contains a hypothesis  $H$  of the form  $A \wedge B$ , writing `destruct H as [HA HB]` will remove  $H$  from the context and add two new hypotheses:  $HA$ , stating that  $A$  is true, and  $HB$ , stating that  $B$  is true.

Lemma and\_example2 :

$\forall n\ m : \text{nat}, n = 0 \wedge m = 0 \rightarrow n + m = 0.$

Proof.

```
intros n m H.
destruct H as [Hn Hm].
rewrite Hn. rewrite Hm.
reflexivity.
```

Qed.

As usual, we can also destruct  $H$  right when we introduce it, instead of introducing and then destructing it:

Lemma and\_example2' :

$\forall n\ m : \text{nat}, n = 0 \wedge m = 0 \rightarrow n + m = 0.$

Proof.

```
intros n m [Hn Hm].
rewrite Hn. rewrite Hm.
reflexivity.
```

Qed.

You may wonder why we bothered packing the two hypotheses  $n = 0$  and  $m = 0$  into a single conjunction, since we could have also stated the theorem with two separate premises:

Lemma and\_example2'' :

$\forall n\ m : \text{nat}, n = 0 \rightarrow m = 0 \rightarrow n + m = 0.$

Proof.

```
intros n m Hn Hm.
rewrite Hn. rewrite Hm.
reflexivity.
```

Qed.

For this specific theorem, both formulations are fine. But it's important to understand how to work with conjunctive hypotheses because conjunctions often arise from intermediate steps in proofs, especially in larger developments. Here's a simple example:

Lemma and\_example3 :

$\forall n\ m : \text{nat}, n + m = 0 \rightarrow n \times m = 0.$

Proof.

```
intros n m H.
apply and_exercise in H.
destruct H as [Hn Hm].
rewrite Hn. reflexivity.
```

Qed.



Another common situation with conjunctions is that we know  $A \wedge B$  but in some context we need just  $A$  or just  $B$ . In such cases we can do a **destruct** (possibly as part of an **intros**) and use an underscore pattern `_` to indicate that the unneeded conjunct should just be thrown away.

Lemma proj1 :  $\forall P Q : \text{Prop},$

$P \wedge Q \rightarrow P.$

Proof.

**intros**  $P Q$   $HPQ$ .

**destruct**  $HPQ$  as [ $HP$   $_$ ].

**apply**  $HP$ . **Qed**.

**Exercise: 1 star, standard, optional (proj2)** Lemma proj2 :  $\forall P Q : \text{Prop},$

$P \wedge Q \rightarrow Q.$

Proof.

*Admitted.*

□

Finally, we sometimes need to rearrange the order of conjunctions and/or the grouping of multi-way conjunctions. The following commutativity and associativity theorems are handy in such cases.

Theorem and\_commut :  $\forall P Q : \text{Prop},$

$P \wedge Q \rightarrow Q \wedge P.$

Proof.

**intros**  $P Q$  [ $HP HQ$ ].

**split**.

- **apply**  $HQ$ .

- **apply**  $HP$ . **Qed**.

**Exercise: 2 stars, standard (and\_assoc)** (In the following proof of associativity, notice how the *nested intros* pattern breaks the hypothesis  $H : P \wedge (Q \wedge R)$  down into  $HP : P$ ,  $HQ : Q$ , and  $HR : R$ . Finish the proof from there.)

Theorem and\_assoc :  $\forall P Q R : \text{Prop},$

$P \wedge (Q \wedge R) \rightarrow (P \wedge Q) \wedge R.$

Proof.

**intros**  $P Q R$  [ $HP [HQ HR]$ ].

*Admitted.*

□

By the way, the infix notation  $\wedge$  is actually just syntactic sugar for **and**  $A B$ . That is, **and** is a Coq operator that takes two propositions as arguments and yields a proposition.

Check **and** :  $\text{Prop} \rightarrow \text{Prop} \rightarrow \text{Prop}.$

## 7.2.2 Disjunction

Another important connective is the *disjunction*, or *logical or*, of two propositions:  $A \vee B$  is true when either  $A$  or  $B$  is. (This infix notation stands for **or**  $A$   $B$ , where **or** :  $\text{Prop} \rightarrow \text{Prop} \rightarrow \text{Prop}$ .)

To use a disjunctive hypothesis in a proof, we proceed by case analysis (which, as with other data types like **nat**, can be done explicitly with **destruct** or implicitly with an **intros** pattern):

Lemma eq\_mult\_0 :

$\forall n\ m : \text{nat}, n = 0 \vee m = 0 \rightarrow n \times m = 0$ .

Proof.

intros n m [Hn | Hm].

-

rewrite Hn. reflexivity.

-

rewrite Hm. rewrite ← mult\_n\_0.  
reflexivity.

Qed.

Conversely, to show that a disjunction holds, it suffices to show that one of its sides holds. This is done via two tactics, **left** and **right**. As their names imply, the first one requires proving the left side of the disjunction, while the second requires proving its right side. Here is a trivial use...

Lemma or\_intro\_l :  $\forall A\ B : \text{Prop}, A \rightarrow A \vee B$ .

Proof.

intros A B HA.

left.

apply HA.

Qed.

... and here is a slightly more interesting example requiring both **left** and **right**:

Lemma zero\_or\_succ :

$\forall n : \text{nat}, n = 0 \vee n = S(\text{pred } n)$ .

Proof.

intros [|n'].

- left. reflexivity.

- right. reflexivity.

Qed.

**Exercise: 1 star, standard (mult\_eq\_0)** Lemma mult\_eq\_0 :

$\forall n\ m, n \times m = 0 \rightarrow n = 0 \vee m = 0$ .

Proof.

Admitted.

□

**Exercise: 1 star, standard (or\_commut)** Theorem `or_commut` :  $\forall P Q : \text{Prop}, P \vee Q \rightarrow Q \vee P$ .

Proof.

*Admitted.*

□

### 7.2.3 Falsehood and Negation

So far, we have mostly been concerned with proving that certain things are *true* – addition is commutative, appending lists is associative, etc. Of course, we may also be interested in negative results, demonstrating that some given proposition is *not* true. Such statements are expressed with the logical negation operator  $\neg$ .

To see how negation works, recall the *principle of explosion* from the `Tactics` chapter, which asserts that, if we assume a contradiction, then any other proposition can be derived. Following this intuition, we could define  $\neg P$  (“not  $P$ ”) as  $\forall Q, P \rightarrow Q$ .

Coq actually makes a slightly different (but equivalent) choice, defining  $\neg P$  as  $P \rightarrow \mathbf{False}$ , where **False** is a specific contradictory proposition defined in the standard library.

Module MYNOT.

Definition `not` ( $P:\text{Prop}$ ) :=  $P \rightarrow \mathbf{False}$ .

Notation “ $\sim x$ ” := (`not`  $x$ ) : *type\_scope*.

Check `not` :  $\text{Prop} \rightarrow \text{Prop}$ .

End MYNOT.

Since **False** is a contradictory proposition, the principle of explosion also applies to it. If we get **False** into the proof context, we can use `destruct` on it to complete any goal:

Theorem `ex_falso_quodlibet` :  $\forall (P:\text{Prop}),$

$\mathbf{False} \rightarrow P$ .

Proof.

`intros`  $P$  *contra*.

`destruct` *contra*. `Qed`.

The Latin *ex falso quodlibet* means, literally, “from falsehood follows whatever you like”; this is another common name for the principle of explosion.

**Exercise: 2 stars, standard, optional (not\_implies\_our\_not)** Show that Coq’s definition of negation implies the intuitive one mentioned above:

Fact `not_implies_our_not` :  $\forall (P:\text{Prop}),$

$\neg P \rightarrow (\forall (Q:\text{Prop}), P \rightarrow Q)$ .

Proof.

*Admitted.*

□

Inequality is a frequent enough example of negated statement that there is a special notation for it,  $x \neq y$ :

Notation “ $x <> y$ ” :=  $(\neg(x = y))$ .

We can use **not** to state that 0 and 1 are different elements of **nat**:

Theorem zero\_not\_one :  $0 \neq 1$ .

Proof.

The proposition  $0 \neq 1$  is exactly the same as  $\neg(0 = 1)$ , that is **not**  $(0 = 1)$ , which unfolds to  $(0 = 1) \rightarrow$  **False**. (We use **unfold not** explicitly here to illustrate that point, but generally it can be omitted.)     **unfold not**.

To prove an inequality, we may assume the opposite equality...     **intros contra**.

... and deduce a contradiction from it. Here, the equality **O** = **S O** contradicts the disjointness of constructors **O** and **S**, so **discriminate** takes care of it.     **discriminate contra**.

Qed.

It takes a little practice to get used to working with negation in Coq. Even though you can see perfectly well why a statement involving negation is true, it can be a little tricky at first to make Coq understand it! Here are proofs of a few familiar facts to get you warmed up.

Theorem not\_False :

$\neg$  **False**.

Proof.

**unfold not**. **intros H**. **destruct H**. Qed.

Theorem contradiction\_implies\_anything :  $\forall P Q : \text{Prop}$ ,

$(P \wedge \neg P) \rightarrow Q$ .

Proof.

**intros P Q [HP HNA]**. **unfold not** in **HNA**.

**apply HNA** in **HP**. **destruct HP**. Qed.

Theorem double\_neg :  $\forall P : \text{Prop}$ ,

$P \rightarrow \neg\neg P$ .

Proof.

**intros P H**. **unfold not**. **intros G**. **apply G**. **apply H**. Qed.

**Exercise: 2 stars, advanced (double\_neg\_inf)** Write an informal proof of double\_neg:

*Theorem:*  $P$  implies  $\neg\neg P$ , for any proposition  $P$ .

Definition manual\_grade\_for\_double\_neg\_inf : **option** (**nat**×**string**) := **None**.

□

**Exercise: 2 stars, standard, especially useful (contrapositive)** Theorem contrapositive

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

$(P \rightarrow Q) \rightarrow (\neg Q \rightarrow \neg P)$ .

Proof.

*Admitted.*

□

**Exercise: 1 star, standard (not\_both\_true\_and\_false)** Theorem not\_both\_true\_and\_false

:  $\forall P : \text{Prop},$

$\neg (P \wedge \neg P).$

Proof.

*Admitted.*

□

**Exercise: 1 star, advanced (informal\_not\_PNP)** Write an informal proof (in English) of the proposition  $\forall P : \text{Prop}, \neg(P \wedge \neg P).$

Definition manual\_grade\_for\_informal\_not\_PNP : **option** (**nat** × **string**) := **None**.

□

Since inequality involves a negation, it also requires a little practice to be able to work with it fluently. Here is one useful trick. If you are trying to prove a goal that is nonsensical (e.g., the goal state is `false = true`), apply `ex_falso_quodlibet` to change the goal to **False**. This makes it easier to use assumptions of the form  $\neg P$  that may be available in the context – in particular, assumptions of the form  $x \neq y$ .

Theorem not\_true\_is\_false :  $\forall b : \text{bool},$

$b \neq \text{true} \rightarrow b = \text{false}.$

Proof.

intros *b H*.

destruct *b* eqn:HE.

-

unfold **not** in *H*.

apply ex\_falso\_quodlibet.

apply *H*. reflexivity.

-

reflexivity.

Qed.

Since reasoning with `ex_falso_quodlibet` is quite common, Coq provides a built-in tactic, `ex_falso`, for applying it.

Theorem not\_true\_is\_false' :  $\forall b : \text{bool},$

$b \neq \text{true} \rightarrow b = \text{false}.$

Proof.

intros [| *H*]. -

unfold **not** in *H*.

*ex\_falso.* apply *H*. reflexivity.

- reflexivity.

Qed.

## 7.2.4 Truth

Besides **False**, Coq's standard library also defines **True**, a proposition that is trivially true. To prove it, we use the predefined constant `I` : **True**:

Lemma True\_is\_true : **True**.

Proof. apply I. Qed.

Unlike **False**, which is used extensively, **True** is used quite rarely, since it is trivial (and therefore uninteresting) to prove as a goal, and it carries no useful information as a hypothesis.

But it can be quite useful when defining complex Props using conditionals or as a parameter to higher-order Props. We will see examples later on.

## 7.2.5 Logical Equivalence

The handy “if and only if” connective, which asserts that two propositions have the same truth value, is simply the conjunction of two implications.

Module MYIFF.

Definition iff (P Q : Prop) := (P → Q) ∧ (Q → P).

Notation "P <-> Q" := (iff P Q)  
(at level 95, no associativity)  
: type\_scope.

End MYIFF.

Theorem iff\_sym : ∀ P Q : Prop,  
(P ↔ Q) → (Q ↔ P).

Proof.

intros P Q [HAB HBA].  
split.  
- apply HBA.  
- apply HAB. Qed.

Lemma not\_true\_iff\_false : ∀ b,  
b ≠ true ↔ b = false.

Proof.

intros b. split.  
- apply not\_true\_is\_false.  
-  
intros H. rewrite H. intros H'. discriminate H'.

Qed.

**Exercise: 1 star, standard, optional (iff\_properties)** Using the above proof that  $\leftrightarrow$  is symmetric (`iff_sym`) as a guide, prove that it is also reflexive and transitive.

Theorem `iff_refl` :  $\forall P : \text{Prop},$   
 $P \leftrightarrow P.$

Proof.

*Admitted.*

Theorem `iff_trans` :  $\forall P Q R : \text{Prop},$   
 $(P \leftrightarrow Q) \rightarrow (Q \leftrightarrow R) \rightarrow (P \leftrightarrow R).$

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard (or\_distributes\_over\_and)** Theorem `or_distributes_over_and`  
 $\forall P Q R : \text{Prop},$

$P \vee (Q \wedge R) \leftrightarrow (P \vee Q) \wedge (P \vee R).$

Proof.

*Admitted.*

□

## 7.2.6 Setoids and Logical Equivalence

Some of Coq’s tactics treat iff statements specially, avoiding the need for some low-level proof-state manipulation. In particular, `rewrite` and `reflexivity` can be used with iff statements, not just equalities. To enable this behavior, we have to import the Coq library that supports it:

From *Coq* Require Import `Setoids.Setoid`.

A “setoid” is a set equipped with an equivalence relation, that is, a relation that is reflexive, symmetric, and transitive. When two elements of a set are equivalent according to the relation, `rewrite` can be used to replace one element with the other. We’ve seen that already with the equality relation  $=$  in Coq: when  $x = y$ , we can use `rewrite` to replace  $x$  with  $y$ , or vice-versa.

Similarly, the logical equivalence relation  $\leftrightarrow$  is reflexive, symmetric, and transitive, so we can use it to replace one part of a proposition with another: if  $P \leftrightarrow Q$ , then we can use `rewrite` to replace  $P$  with  $Q$ , or vice-versa.

Here is a simple example demonstrating how these tactics work with iff. First, let’s prove a couple of basic iff equivalences.

Lemma `mult_0` :  $\forall n m, n \times m = 0 \leftrightarrow n = 0 \vee m = 0.$

Proof.

`split.`

- apply `mult_eq_0`.

- apply `eq_mult_0`.

Qed.

Theorem or\_assoc :

$\forall P Q R : \text{Prop}, P \vee (Q \vee R) \leftrightarrow (P \vee Q) \vee R.$

Proof.

```
intros P Q R. split.
- intros [H | [H | H]].
  + left. left. apply H.
  + left. right. apply H.
  + right. apply H.
- intros [[H | H] | H].
  + left. apply H.
  + right. left. apply H.
  + right. right. apply H.
```

Qed.

We can now use these facts with `rewrite` and `reflexivity` to give smooth proofs of statements involving equivalences. For example, here is a ternary version of the previous `mult_0` result:

Lemma mult\_0\_3 :

$\forall n m p, n \times m \times p = 0 \leftrightarrow n = 0 \vee m = 0 \vee p = 0.$

Proof.

```
intros n m p.
rewrite mult_0. rewrite mult_0. rewrite or_assoc.
reflexivity.
```

Qed.

The `apply` tactic can also be used with  $\leftrightarrow$ . When given an equivalence as its argument, `apply` tries to guess which direction of the equivalence will be useful.

Lemma apply\_iff\_example :

$\forall n m : \text{nat}, n \times m = 0 \rightarrow n = 0 \vee m = 0.$

Proof.

```
intros n m H. apply mult_0. apply H.
```

Qed.

## 7.2.7 Existential Quantification

Another important logical connective is *existential quantification*. To say that there is some  $x$  of type  $T$  such that some property  $P$  holds of  $x$ , we write  $\exists x : T, P$ . As with  $\forall$ , the type annotation  $: T$  can be omitted if Coq is able to infer from the context what the type of  $x$  should be.

To prove a statement of the form  $\exists x, P$ , we must show that  $P$  holds for some specific choice of value for  $x$ , known as the *witness* of the existential. This is done in two steps: First,



we explicitly tell Coq which witness  $t$  we have in mind by invoking the tactic  $\exists t$ . Then we prove that  $P$  holds after all occurrences of  $x$  are replaced by  $t$ .

Definition `even`  $x := \exists n : \text{nat}, x = \text{double } n$ .

Lemma `four_is_even` : `even 4`.

Proof.

`unfold even.  $\exists$  2. reflexivity.`

Qed.

Conversely, if we have an existential hypothesis  $\exists x, P$  in the context, we can destruct it to obtain a witness  $x$  and a hypothesis stating that  $P$  holds of  $x$ .

Theorem `exists_example_2` :  $\forall n,$

$(\exists m, n = 4 + m) \rightarrow$

$(\exists o, n = 2 + o).$

Proof.

`intros n [m Hm].  $\exists$  (2 + m).`

`apply Hm. Qed.`

**Exercise: 1 star, standard, especially useful (`dist_not_exists`)** Prove that “ $P$  holds for all  $x$ ” implies “there is no  $x$  for which  $P$  does not hold.” (Hint: `destruct H as [x E]` works on existential assumptions!)

Theorem `dist_not_exists` :  $\forall (X:\text{Type}) (P : X \rightarrow \text{Prop}),$

$(\forall x, P x) \rightarrow \neg (\exists x, \neg P x).$

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard (`dist_exists_or`)** Prove that existential quantification distributes over disjunction.

Theorem `dist_exists_or` :  $\forall (X:\text{Type}) (P Q : X \rightarrow \text{Prop}),$

$(\exists x, P x \vee Q x) \leftrightarrow (\exists x, P x) \vee (\exists x, Q x).$

Proof.

*Admitted.*

□

## 7.3 Programming with Propositions

The logical connectives that we have seen provide a rich vocabulary for defining complex propositions from simpler ones. To illustrate, let’s look at how to express the claim that an element  $x$  occurs in a list  $l$ . Notice that this property has a simple recursive structure:

- If  $l$  is the empty list, then  $x$  cannot occur in it, so the property “ $x$  appears in  $l$ ” is simply false.

- Otherwise,  $l$  has the form  $x' :: l'$ . In this case,  $x$  occurs in  $l$  if either it is equal to  $x'$  or it occurs in  $l'$ .

We can translate this directly into a straightforward recursive function taking an element and a list and returning a proposition (!):

```
Fixpoint ln {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x ∨ ln x l'
  end.
```

When `ln` is applied to a concrete list, it expands into a concrete sequence of nested disjunctions.

Example `ln_example_1 : ln 4 [1; 2; 3; 4; 5]`.

Proof.

```
simpl. right. right. right. left. reflexivity.
```

Qed.

Example `ln_example_2 :`

```
∀ n, ln n [2; 4] →
  ∃ n', n = 2 × n'.
```

Proof.

```
simpl.
intros n [H | [H | []]].
- ∃ 1. rewrite ← H. reflexivity.
- ∃ 2. rewrite ← H. reflexivity.
```

Qed.

(Notice the use of the empty pattern to discharge the last case *en passant*.)

We can also prove more generic, higher-level lemmas about `ln`.

Note, in the first, how `ln` starts out applied to a variable and only gets expanded when we do case analysis on this variable:

Theorem `ln_map :`

```
∀ (A B : Type) (f : A → B) (l : list A) (x : A),
  ln x l →
  ln (f x) (map f l).
```

Proof.

```
intros A B f l x.
induction l as [|x' l' IHL'].
-
  simpl. intros [].
-
  simpl. intros [H | H].
  + rewrite H. left. reflexivity.
```

+ right. apply *IHL'*. apply *H*.  
 Qed.

This way of defining propositions recursively, though convenient in some cases, also has some drawbacks. In particular, it is subject to Coq's usual restrictions regarding the definition of recursive functions, e.g., the requirement that they be “obviously terminating.” In the next chapter, we will see how to define propositions *inductively*, a different technique with its own set of strengths and limitations.

**Exercise: 3 stars, standard (In\_map\_iff)** Theorem *ln\_map\_iff* :

$\forall (A\ B : \text{Type}) (f : A \rightarrow B) (l : \text{list } A) (y : B),$   
 $\text{ln } y (\text{map } f\ l) \leftrightarrow$   
 $\exists x, f\ x = y \wedge \text{ln } x\ l.$

Proof.

intros *A B f l y*. split.  
 Admitted.  
 □

**Exercise: 2 stars, standard (In\_app\_iff)** Theorem *ln\_app\_iff* :  $\forall A\ l\ l' (a:A),$

$\text{ln } a\ (l++l') \leftrightarrow \text{ln } a\ l \vee \text{ln } a\ l'.$

Proof.

intros *A l*. induction *l* as [|*a' l'* IH].  
 Admitted.  
 □

**Exercise: 3 stars, standard, especially useful (All)** Recall that functions returning propositions can be seen as *properties* of their arguments. For instance, if *P* has type **nat**  $\rightarrow$  **Prop**, then *P n* states that property *P* holds of *n*.

Drawing inspiration from *ln*, write a recursive function *All* stating that some property *P* holds of all elements of a list *l*. To make sure your definition is correct, prove the *All\_ln* lemma below. (Of course, your definition should *not* just restate the left-hand side of *All\_ln*.)

Fixpoint *All* {*T* : Type} (*P* : *T*  $\rightarrow$  **Prop**) (*l* : list *T*) : **Prop**  
 . Admitted.

Theorem *All\_ln* :

$\forall T (P : T \rightarrow \text{Prop}) (l : \text{list } T),$   
 $(\forall x, \text{ln } x\ l \rightarrow P\ x) \leftrightarrow$   
 $\text{All } P\ l.$

Proof.

Admitted.  
 □

**Exercise: 2 stars, standard, optional (combine\_odd\_even)** Complete the definition of the `combine_odd_even` function below. It takes as arguments two properties of numbers,  $Podd$  and  $Peven$ , and it should return a property  $P$  such that  $P\ n$  is equivalent to  $Podd\ n$  when  $n$  is odd and equivalent to  $Peven\ n$  otherwise.

Definition `combine_odd_even` ( $Podd\ Peven : \mathbf{nat} \rightarrow \mathbf{Prop}$ ) :  $\mathbf{nat} \rightarrow \mathbf{Prop}$   
*. Admitted.*

To test your definition, prove the following facts:

Theorem `combine_odd_even_intro` :  
 $\forall (Podd\ Peven : \mathbf{nat} \rightarrow \mathbf{Prop}) (n : \mathbf{nat}),$   
 $(\text{oddb } n = \text{true} \rightarrow Podd\ n) \rightarrow$   
 $(\text{oddb } n = \text{false} \rightarrow Peven\ n) \rightarrow$   
 $\text{combine\_odd\_even } Podd\ Peven\ n.$

Proof.

*Admitted.*

Theorem `combine_odd_even_elim_odd` :  
 $\forall (Podd\ Peven : \mathbf{nat} \rightarrow \mathbf{Prop}) (n : \mathbf{nat}),$   
 $\text{combine\_odd\_even } Podd\ Peven\ n \rightarrow$   
 $\text{oddb } n = \text{true} \rightarrow$   
 $Podd\ n.$

Proof.

*Admitted.*

Theorem `combine_odd_even_elim_even` :  
 $\forall (Podd\ Peven : \mathbf{nat} \rightarrow \mathbf{Prop}) (n : \mathbf{nat}),$   
 $\text{combine\_odd\_even } Podd\ Peven\ n \rightarrow$   
 $\text{oddb } n = \text{false} \rightarrow$   
 $Peven\ n.$

Proof.

*Admitted.*

□

## 7.4 Applying Theorems to Arguments

One feature that distinguishes Coq from some other popular proof assistants (e.g., ACL2 and Isabelle) is that it treats *proofs* as first-class objects.

There is a great deal to be said about this, but it is not necessary to understand it all in detail in order to use Coq. This section gives just a taste, while a deeper exploration can be found in the optional chapters `ProofObjects` and `IndPrinciples`.

We have seen that we can use `Check` to ask Coq to print the type of an expression. We can also use it to ask what theorem a particular identifier refers to.

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

Coq checks the *statement* of the `plus_comm` theorem (or prints it for us, if we leave off the part beginning with the colon) in the same way that it checks the *type* of any term that we ask it to `Check`. Why?

The reason is that the identifier `plus_comm` actually refers to a *proof object*, which represents a logical derivation establishing of the truth of the statement  $\forall n\ m : \mathbf{nat},\ n + m = m + n$ . The type of this object is the proposition which it is a proof of.

Intuitively, this makes sense because the statement of a theorem tells us what we can use that theorem for, just as the type of a “computational” object tells us what we can do with that object – e.g., if we have a term of type  $\mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{nat}$ , we can give it two **nats** as arguments and get a **nat** back. Similarly, if we have an object of type  $n = m \rightarrow n + n = m + m$  and we provide it an “argument” of type  $n = m$ , we can derive  $n + n = m + m$ .

Operationally, this analogy goes even further: by applying a theorem as if it were a function, i.e., applying it to hypotheses with matching types, we can specialize its result without having to resort to intermediate assertions. For example, suppose we wanted to prove the following result:

Lemma `plus_comm3` :

$\forall x\ y\ z,\ x + (y + z) = (z + y) + x.$

It appears at first sight that we ought to be able to prove this by rewriting with `plus_comm` twice to make the two sides match. The problem, however, is that the second `rewrite` will undo the effect of the first.

Proof.

```
intros x y z.
rewrite plus_comm.
rewrite plus_comm.
```

Abort.

We saw similar problems back in Chapter [Induction](#), and saw one way to work around them by using `assert` to derive a specialized version of `plus_comm` that can be used to rewrite exactly where we want.

Lemma `plus_comm3_take2` :

$\forall x\ y\ z,\ x + (y + z) = (z + y) + x.$

Proof.

```
intros x y z.
rewrite plus_comm.
assert (H : y + z = z + y).
{ rewrite plus_comm. reflexivity. }
rewrite H.
reflexivity.
```

Qed.

A more elegant alternative is to apply `plus_comm` directly to the arguments we want to instantiate it with, in much the same way as we apply a polymorphic function to a type argument.

Lemma plus\_comm3\_take3 :  
 $\forall x y z, x + (y + z) = (z + y) + x.$

Proof.

```
intros x y z.
rewrite plus_comm.
rewrite (plus_comm y z).
reflexivity.
```

Qed.

Let's see another example of using a theorem like a function. The following theorem says: any list  $l$  containing some element must be nonempty.

Theorem in\_not\_nil :

$\forall A (x : A) (l : \text{list } A), \text{In } x l \rightarrow l \neq [].$

Proof.

```
intros A x l H. unfold not. intro Hl.
rewrite Hl in H.
simpl in H.
apply H.
```

Qed.

What makes this interesting is that one quantified variable ( $x$ ) does not appear in the conclusion ( $l \neq []$ ).

We should be able to use this theorem to prove the special case where  $x$  is 42. However, naively, the tactic `apply in_not_nil` will fail because it cannot infer the value of  $x$ .

Lemma in\_not\_nil\_42 :

$\forall l : \text{list nat}, \text{In } 42 l \rightarrow l \neq [].$

Proof.

```
intros l H.
Fail apply in_not_nil.
```

Abort.

There are several ways to work around this:

Use `apply ... with ...` Lemma in\_not\_nil\_42\_take2 :

$\forall l : \text{list nat}, \text{In } 42 l \rightarrow l \neq [].$

Proof.

```
intros l H.
apply in_not_nil with (x := 42).
apply H.
```

Qed.

Use `apply ... in ...` Lemma in\_not\_nil\_42\_take3 :

$\forall l : \text{list nat}, \text{In } 42 l \rightarrow l \neq [].$

Proof.

```
intros l H.
apply in_not_nil in H.
```



capturing informal mathematical concepts, but these are, by and large, about equally natural and easy to work with. For example, instead of saying that a natural number  $n$  belongs to the set of even numbers, we would say in Coq that `even  $n$`  holds, where `even : nat → Prop` is a property describing even numbers.

However, there are some cases where translating standard mathematical reasoning into Coq can be cumbersome or sometimes even impossible, unless we enrich the core logic with additional axioms.

We conclude this chapter with a brief discussion of some of the most significant differences between the two worlds.

### 7.5.1 Functional Extensionality

The equality assertions that we have seen so far mostly have concerned elements of inductive types (`nat`, `bool`, etc.). But, since Coq’s equality operator is polymorphic, we can use it at *any* type – in particular, we can write propositions claiming that two *functions* are equal to each other:

Example `function_equality_ex1` :

```
(fun x => 3 + x) = (fun x => (pred 4) + x).
```

Proof. `reflexivity`. Qed.

In common mathematical practice, two functions  $f$  and  $g$  are considered equal if they produce the same output on every input:

(forall x, f x = g x) -> f = g

This is known as the principle of *functional extensionality*.

Informally, an “extensional property” is one that pertains to an object’s observable behavior. Thus, functional extensionality simply means that a function’s identity is completely determined by what we can observe from it – i.e., the results we obtain after applying it.

However, functional extensionality is not part of Coq’s built-in logic. This means that some apparently “obvious” propositions are not provable.

Example `function_equality_ex2` :

```
(fun x => plus x 1) = (fun x => plus 1 x).
```

Proof.

Abort.

However, we can add functional extensionality to Coq’s core using the `Axiom` command.

```
Axiom functional_extensionality : ∀ {X Y: Type}
                                     {f g : X → Y},
  (∀ (x:X), f x = g x) → f = g.
```

Defining something as an `Axiom` has the same effect as stating a theorem and skipping its proof using `Admitted`, but it alerts the reader that this isn’t just something we’re going to come back and fill in later!

We can now invoke functional extensionality in proofs:

Example `function_equality_ex2` :



```
(fun x => plus x 1) = (fun x => plus 1 x).
```

Proof.

```
apply functional_extensionality. intros x.
```

```
apply plus_comm.
```

Qed.

Naturally, we must be careful when adding new axioms into Coq's logic, as this can render it *inconsistent* – that is, it may become possible to prove every proposition, including **False**,  $2+2=5$ , etc.!

Unfortunately, there is no simple way of telling whether an axiom is safe to add: hard work by highly trained mathematicians is often required to establish the consistency of any particular combination of axioms.

Fortunately, it is known that adding functional extensionality, in particular, *is* consistent.

To check whether a particular proof relies on any additional axioms, use the **Print Assumptions** command.

```
Print Assumptions function_equality_ex2.
```

**Exercise: 4 stars, standard (tr\_rev\_correct)** One problem with the definition of the list-reversing function `rev` that we have is that it performs a call to `app` on each step; running `app` takes time asymptotically linear in the size of the list, which means that `rev` is asymptotically quadratic. We can improve this with the following definitions:

```
Fixpoint rev_append {X} (l1 l2 : list X) : list X :=
  match l1 with
  | [] => l2
  | x :: l1' => rev_append l1' (x :: l2)
end.
```

```
Definition tr_rev {X} (l : list X) : list X :=
  rev_append l [].
```

This version of `rev` is said to be *tail-recursive*, because the recursive call to the function is the last operation that needs to be performed (i.e., we don't have to execute `++` after the recursive call); a decent compiler will generate very efficient code in this case.

Prove that the two definitions are indeed equivalent.

```
Theorem tr_rev_correct : ∀ X, @tr_rev X = @rev X.
```

Proof.

*Admitted.*

□

## 7.5.2 Propositions vs. Booleans

We've seen two different ways of expressing logical claims in Coq: with *booleans* (of type **bool**), and with *propositions* (of type **Prop**).

For instance, to claim that a number  $n$  is even, we can say either...

... that `evenb n` evaluates to `true`... Example `even_42_bool : evenb 42 = true`.  
 Proof. `reflexivity`. Qed.

... or that there exists some  $k$  such that  $n = \text{double } k$ . Example `even_42_prop : even 42`.  
 Proof. `unfold even`. `∃ 21`. `reflexivity`. Qed.

Of course, it would be pretty strange if these two characterizations of evenness did not describe the same set of natural numbers! Fortunately, we can prove that they do...

We first need two helper lemmas. Lemma `evenb_double : ∀ k, evenb (double k) = true`.  
 Proof.

```
intros k. induction k as [|k' IHk'].
- reflexivity.
- simpl. apply IHk'.
Qed.
```

**Exercise: 3 stars, standard (evenb\_double\_conv)** Lemma `evenb_double_conv : ∀ n, ∃ k,`

`n = if evenb n then double k else S (double k)`.  
 Proof.

*Admitted.*  
 $\square$

Now the main theorem: Theorem `even_bool_prop : ∀ n, evenb n = true ↔ even n`.

Proof.

```
intros n. split.
- intros H. destruct (evenb_double_conv n) as [k Hk].
  rewrite Hk. rewrite H. ∃ k. reflexivity.
- intros [k Hk]. rewrite Hk. apply evenb_double.
```

Qed.

In view of this theorem, we say that the boolean computation `evenb n` is *reflected* in the truth of the proposition  $\exists k, n = \text{double } k$ .

Similarly, to state that two numbers  $n$  and  $m$  are equal, we can say either

- (1) that `n =? m` returns `true`, or
- (2) that  $n = m$ .

Again, these two notions are equivalent.

Theorem `eqb_eq : ∀ n1 n2 : nat, n1 =? n2 = true ↔ n1 = n2`.

Proof.

```
intros n1 n2. split.
- apply eqb_true.
- intros H. rewrite H. rewrite ← eqb_refl. reflexivity.
```

Qed.

However, even when the boolean and propositional formulations of a claim are equivalent from a purely logical perspective, they are often not equivalent from the point of view of convenience for some specific purpose.

In the case of even numbers above, when proving the backwards direction of `even_bool_prop` (i.e., `evenb_double`, going from the propositional to the boolean claim), we used a simple induction on  $k$ . On the other hand, the converse (the `evenb_double_conv` exercise) required a clever generalization, since we can't directly prove  $(\text{evenb } n = \text{true}) \rightarrow \text{even } n$ .

For these examples, the propositional claims are more useful than their boolean counterparts, but this is not always the case. For instance, we cannot test whether a general proposition is true or not in a function definition; as a consequence, the following code fragment is rejected:

*Fail*

```
Definition is_even_prime n :=  
  if n = 2 then true  
  else false.
```

Coq complains that  $n = 2$  has type `Prop`, while it expects an element of `bool` (or some other inductive type with two elements). The reason has to do with the *computational* nature of Coq's core language, which is designed so that every function it can express is computable and total. One reason for this is to allow the extraction of executable programs from Coq developments. As a consequence, `Prop` in Coq does *not* have a universal case analysis operation telling whether any given proposition is true or false, since such an operation would allow us to write non-computable functions.

Beyond the fact that non-computable properties are impossible in general to phrase as boolean computations, even many *computable* properties are easier to express using `Prop` than `bool`, since recursive function definitions in Coq are subject to significant restrictions. For instance, the next chapter shows how to define the property that a regular expression matches a given string using `Prop`. Doing the same with `bool` would amount to writing a regular expression matching algorithm, which would be more complicated, harder to understand, and harder to reason about than a simple (non-algorithmic) definition of this property.

Conversely, an important side benefit of stating facts using booleans is enabling some proof automation through computation with Coq terms, a technique known as *proof by reflection*.

Consider the following statement:

Example `even_1000` : `even 1000`.

The most direct way to prove this is to give the value of  $k$  explicitly.

Proof. `unfold even. ∃ 500. reflexivity. Qed.`

The proof of the corresponding boolean statement is even simpler (because we don't have to invent the witness: Coq's computation mechanism does it for us!).

Example even\_1000' : evenb 1000 = true.

Proof. reflexivity. Qed.

What is interesting is that, since the two notions are equivalent, we can use the boolean formulation to prove the other one without mentioning the value 500 explicitly:

Example even\_1000'' : even 1000.

Proof. apply even\_bool\_prop. reflexivity. Qed.

Although we haven't gained much in terms of proof-script size in this case, larger proofs can often be made considerably simpler by the use of reflection. As an extreme example, a famous Coq proof of the even more famous *4-color theorem* uses reflection to reduce the analysis of hundreds of different cases to a boolean computation.

Another notable difference is that the negation of a “boolean fact” is straightforward to state and prove: simply flip the expected boolean result.

Example not\_even\_1001 : evenb 1001 = false.

Proof.

  reflexivity.

Qed.

In contrast, propositional negation can be more difficult to work with directly.

Example not\_even\_1001' : ~(even 1001).

Proof.

  rewrite ← even\_bool\_prop.

  unfold not.

  simpl.

  intro H.

  discriminate H.

Qed.

Equality provides a complementary example, where it is sometimes easier to work in the propositional world. Knowing that  $n =? m = \text{true}$  is generally of little direct help in the middle of a proof involving  $n$  and  $m$ ; however, if we convert the statement to the equivalent form  $n = m$ , we can rewrite with it.

Lemma plus\_eqb\_example :  $\forall n m p : \text{nat}$ ,

$n =? m = \text{true} \rightarrow n + p =? m + p = \text{true}$ .

Proof.

  intros n m p H.

  rewrite eqb\_eq in H.

  rewrite H.

  rewrite eqb\_eq.

  reflexivity.

Qed.

We won't discuss reflection any further here, but it serves as a good example showing the complementary strengths of booleans and general propositions, and being able to cross

back and forth between the boolean and propositional worlds will often be convenient in later chapters.

**Exercise: 2 stars, standard (logical\_connectives)** The following theorems relate the propositional connectives studied in this chapter to the corresponding boolean operations.

Theorem `andb_true_iff` :  $\forall b1\ b2:\mathbf{bool}$ ,  
 $b1 \ \&\&\ b2 = \mathbf{true} \leftrightarrow b1 = \mathbf{true} \wedge b2 = \mathbf{true}$ .

Proof.

*Admitted.*

Theorem `orb_true_iff` :  $\forall b1\ b2$ ,  
 $b1 \ ||\ b2 = \mathbf{true} \leftrightarrow b1 = \mathbf{true} \vee b2 = \mathbf{true}$ .

Proof.

*Admitted.*

□

**Exercise: 1 star, standard (eqb\_neq)** The following theorem is an alternate “negative” formulation of `eqb_eq` that is more convenient in certain situations. (We’ll see examples in later chapters.) Hint: `not_true_iff_false`.

Theorem `eqb_neq` :  $\forall x\ y : \mathbf{nat}$ ,  
 $x =? y = \mathbf{false} \leftrightarrow x \neq y$ .

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard (eqb\_list)** Given a boolean operator `eqb` for testing equality of elements of some type  $A$ , we can define a function `eqb_list` for testing equality of lists with elements in  $A$ . Complete the definition of the `eqb_list` function below. To make sure that your definition is correct, prove the lemma `eqb_list_true_iff`.

Fixpoint `eqb_list` { $A : \mathbf{Type}$ } ( $eqb : A \rightarrow A \rightarrow \mathbf{bool}$ )  
 ( $l1\ l2 : \mathbf{list}\ A$ ) :  $\mathbf{bool}$

. *Admitted.*

Theorem `eqb_list_true_iff` :

$\forall A\ (eqb : A \rightarrow A \rightarrow \mathbf{bool})$ ,  
 $(\forall a1\ a2, eqb\ a1\ a2 = \mathbf{true} \leftrightarrow a1 = a2) \rightarrow$   
 $\forall l1\ l2, eqb\_list\ eqb\ l1\ l2 = \mathbf{true} \leftrightarrow l1 = l2$ .

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard, especially useful (All\_forallb)** Recall the function `forallb`, from the exercise *forall\_exists\_challenge* in chapter Tactics:

```
Fixpoint forallb {X : Type} (test : X → bool) (l : list X) : bool :=
  match l with
  | [] ⇒ true
  | x :: l' ⇒ andb (test x) (forallb test l')
  end.
```

Prove the theorem below, which relates `forallb` to the `All` property defined above.

**Theorem** `forallb_true_iff` :  $\forall X \text{ test } (l : \text{list } X),$   
 $\text{forallb test } l = \text{true} \leftrightarrow \text{All } (\text{fun } x \Rightarrow \text{test } x = \text{true}) \text{ } l.$

**Proof.**

*Admitted.*

(Ungraded thought question) Are there any important properties of the function `forallb` which are not captured by this specification?

### 7.5.3 Classical vs. Constructive Logic

We have seen that it is not possible to test whether or not a proposition  $P$  holds while defining a Coq function. You may be surprised to learn that a similar restriction applies to *proofs*! In other words, the following intuitive reasoning principle is not derivable in Coq:

**Definition** `excluded_middle` :=  $\forall P : \text{Prop},$   
 $P \vee \neg P.$

To understand operationally why this is the case, recall that, to prove a statement of the form  $P \vee Q$ , we use the `left` and `right` tactics, which effectively require knowing which side of the disjunction holds. But the universally quantified  $P$  in `excluded_middle` is an *arbitrary* proposition, which we know nothing about. We don't have enough information to choose which of `left` or `right` to apply, just as Coq doesn't have enough information to mechanically decide whether  $P$  holds or not inside a function.

However, if we happen to know that  $P$  is reflected in some boolean term `b`, then knowing whether it holds or not is trivial: we just have to check the value of `b`.

**Theorem** `restricted_excluded_middle` :  $\forall P \text{ } b,$   
 $(P \leftrightarrow b = \text{true}) \rightarrow P \vee \neg P.$

**Proof.**

```
intros P || H.
- left. rewrite H. reflexivity.
- right. rewrite H. intros contra. discriminate contra.
```

**Qed.**

In particular, the excluded middle is valid for equations  $n = m$ , between natural numbers  $n$  and  $m$ .

**Theorem** `restricted_excluded_middle_eq` :  $\forall (n \text{ } m : \text{nat}),$

$n = m \vee n \neq m.$

Proof.

```
intros n m.  
apply (restricted_excluded_middle (n = m) (n =? m)).  
symmetry.  
apply eqb_eq.
```

Qed.

It may seem strange that the general excluded middle is not available by default in Coq, since it is a standard feature of familiar logics like ZFC. But there is a distinct advantage in not assuming the excluded middle: statements in Coq make stronger claims than the analogous statements in standard mathematics. Notably, when there is a Coq proof of  $\exists x, P x$ , it is always possible to explicitly exhibit a value of  $x$  for which we can prove  $P x$  – in other words, every proof of existence is *constructive*.

Logics like Coq's, which do not assume the excluded middle, are referred to as *constructive logics*.

More conventional logical systems such as ZFC, in which the excluded middle does hold for arbitrary propositions, are referred to as *classical*.

The following example illustrates why assuming the excluded middle may lead to non-constructive proofs:

*Claim:* There exist irrational numbers  $a$  and  $b$  such that  $a^b$  ( $a$  to the power  $b$ ) is rational.

*Proof:* It is not difficult to show that  $\sqrt{2}$  is irrational. If  $\sqrt{2}^{\sqrt{2}}$  is rational, it suffices to take  $a = b = \sqrt{2}$  and we are done. Otherwise,  $\sqrt{2}^{\sqrt{2}}$  is irrational. In this case, we can take  $a = \sqrt{2}^{\sqrt{2}}$  and  $b = \sqrt{2}$ , since  $a^b = \sqrt{2}^{\sqrt{2} \times \sqrt{2}} = \sqrt{2}^2 = 2$ .  $\square$

Do you see what happened here? We used the excluded middle to consider separately the cases where  $\sqrt{2}^{\sqrt{2}}$  is rational and where it is not, without knowing which one actually holds! Because of that, we finish the proof knowing that such  $a$  and  $b$  exist but we cannot determine what their actual values are (at least, not from this line of argument).

As useful as constructive logic is, it does have its limitations: There are many statements that can easily be proven in classical logic but that have only much more complicated constructive proofs, and there are some that are known to have no constructive proof at all! Fortunately, like functional extensionality, the excluded middle is known to be compatible with Coq's logic, allowing us to add it safely as an axiom. However, we will not need to do so here: the results that we cover can be developed entirely within constructive logic at negligible extra cost.

It takes some practice to understand which proof techniques must be avoided in constructive reasoning, but arguments by contradiction, in particular, are infamous for leading to non-constructive proofs. Here's a typical example: suppose that we want to show that there exists  $x$  with some property  $P$ , i.e., such that  $P x$ . We start by assuming that our conclusion is false; that is,  $\neg \exists x, P x$ . From this premise, it is not hard to derive  $\forall x, \neg P x$ . If we manage to show that this intermediate fact results in a contradiction, we arrive at

an existence proof without ever exhibiting a value of  $x$  for which  $P\ x$  holds!

The technical flaw here, from a constructive standpoint, is that we claimed to prove  $\exists x, P\ x$  using a proof of  $\neg \neg (\exists x, P\ x)$ . Allowing ourselves to remove double negations from arbitrary statements is equivalent to assuming the excluded middle, as shown in one of the exercises below. Thus, this line of reasoning cannot be encoded in Coq without assuming additional axioms.

**Exercise: 3 stars, standard (excluded\_middle\_irrefutable)** Proving the consistency of Coq with the general excluded middle axiom requires complicated reasoning that cannot be carried out within Coq itself. However, the following theorem implies that it is always safe to assume a decidability axiom (i.e., an instance of excluded middle) for any *particular* Prop  $P$ . Why? Because we cannot prove the negation of such an axiom. If we could, we would have both  $\neg (P \vee \neg P)$  and  $\neg \neg (P \vee \neg P)$  (since  $P$  implies  $\neg \neg P$ , by lemma `double_neg`, which we proved above), which would be a contradiction. But since we can't, it is safe to add  $P \vee \neg P$  as an axiom.

Succinctly: for any proposition  $P$ , *Coq is consistent*  $\implies$  *(Coq +  $P \vee \neg P$ ) is consistent*.

(Hint: you will need to come up with a clever assertion as the next step in the proof.)

Theorem `excluded_middle_irrefutable`:  $\forall (P:\text{Prop}),$   
 $\neg \neg (P \vee \neg P).$

Proof.

`unfold not. intros P H.`

*Admitted.*

□

**Exercise: 3 stars, advanced (not\_exists\_dist)** It is a theorem of classical logic that the following two assertions are equivalent:

$\sim (\exists x, \sim P\ x)$  forall  $x, P\ x$

The `dist_not_exists` theorem above proves one side of this equivalence. Interestingly, the other direction cannot be proved in constructive logic. Your job is to show that it is implied by the excluded middle.

Theorem `not_exists_dist` :  
`excluded_middle  $\rightarrow$`   
 $\forall (X:\text{Type}) (P : X \rightarrow \text{Prop}),$   
 $\neg (\exists x, \neg P\ x) \rightarrow (\forall x, P\ x).$

Proof.

*Admitted.*

□

**Exercise: 5 stars, standard, optional (classical\_axioms)** For those who like a challenge, here is an exercise taken from the Coq'Art book by Bertot and Casteran (p. 123). Each of the following four statements, together with `excluded_middle`, can be considered as



characterizing classical logic. We can't prove any of them in Coq, but we can consistently add any one of them as an axiom if we wish to work in classical logic.

Prove that all five propositions (these four plus `excluded_middle`) are equivalent.

Hint: Rather than considering all pairs of statements pairwise, prove a single circular chain of implications that connects them all.

Definition `peirce` :=  $\forall P Q : \text{Prop},$   
 $((P \rightarrow Q) \rightarrow P) \rightarrow P.$

Definition `double_negation_elimination` :=  $\forall P : \text{Prop},$   
 $\neg\neg P \rightarrow P.$

Definition `de_morgan_not_and_not` :=  $\forall P Q : \text{Prop},$   
 $\neg(\neg P \wedge \neg Q) \rightarrow P \vee Q.$

Definition `implies_to_or` :=  $\forall P Q : \text{Prop},$   
 $(P \rightarrow Q) \rightarrow (\neg P \vee Q).$

# Chapter 8

## Library LF.IndProp

### 8.1 IndProp: Inductively Defined Propositions

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export Logic.
From Coq Require Export Lia.
```

### 8.2 Inductively Defined Propositions

In the Logic chapter, we looked at several ways of writing propositions, including conjunction, disjunction, and existential quantification. In this chapter, we bring yet another new tool into the mix: *inductive definitions*.

*Note:* For the sake of simplicity, most of this chapter uses an inductive definition of “evenness” as a running example. You may find this confusing, since we already have a perfectly good way of defining evenness as a proposition ( $n$  is even if it is equal to the result of doubling some number); if so, rest assured that we will see many more compelling examples of inductively defined propositions toward the end of this chapter and in future chapters.

In past chapters, we have seen two ways of stating that a number  $n$  is even: We can say

- (1) `evenb  $n$  = true`, or
- (2)  $\exists k, n = \text{double } k$ .

Yet another possibility is to say that  $n$  is even if we can establish its evenness from the following rules:

- Rule `ev_0`: The number 0 is even.
- Rule `ev_SS`: If  $n$  is even, then  $S (S \ n)$  is even.

To illustrate how this new definition of evenness works, let’s imagine using it to show that 4 is even. By rule `ev_SS`, it suffices to show that 2 is even. This, in turn, is again

guaranteed by rule `ev_SS`, as long as we can show that 0 is even. But this last fact follows directly from the `ev_0` rule.

We will see many definitions like this one during the rest of the course. For purposes of informal discussions, it is helpful to have a lightweight notation that makes them easy to read and write. *Inference rules* are one such notation. (We'll use **ev** for the name of this property, since `even` is already used.)

---

```
(ev_0) ev 0
      ev n
```

---

```
(ev_SS) ev (S (S n))
```

---

Each of the textual rules that we started with is reformatted here as an inference rule; the intended reading is that, if the *premises* above the line all hold, then the *conclusion* below the line follows. For example, the rule `ev_SS` says that, if  $n$  satisfies **ev**, then `S (S n)` also does. If a rule has no premises above the line, then its conclusion holds unconditionally.

We can represent a proof using these rules by combining rule applications into a *proof tree*. Here's how we might transcribe the above proof that 4 is even:

---

```
(ev_0) ev 0
```

---

```
(ev_SS) ev 2
```

---

```
(ev_SS) ev 4
```

---

(Why call this a “tree”, rather than a “stack”, for example? Because, in general, inference rules can have multiple premises. We will see examples of this shortly.)

### 8.2.1 Inductive Definition of Evenness

Putting all of this together, we can translate the definition of evenness into a formal Coq definition using an `Inductive` declaration, where each constructor corresponds to an inference rule:

```
Inductive ev : nat → Prop :=
| ev_0 : ev 0
| ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

This definition is interestingly different from previous uses of `Inductive`. For one thing, we are defining not a `Type` (like `nat`) or a function yielding a `Type` (like `list`), but rather a function from `nat` to `Prop` – that is, a property of numbers. But what is really new is that, because the `nat` argument of `ev` appears to the *right* of the colon on the first line, it is allowed to take different values in the types of different constructors: 0 in the type of `ev_0` and `S (S n)` in the type of `ev_SS`. Accordingly, the type of each constructor must be specified explicitly (after a colon), and each constructor's type must have the form `ev n` for some natural number  $n$ .

In contrast, recall the definition of **list**:

Inductive list (X:Type) : Type := | nil | cons (x : X) (l : list X).

This definition introduces the X parameter *globally*, to the *left* of the colon, forcing the result of nil and cons to be the same (i.e., **list** X). Had we tried to bring **nat** to the left of the colon in defining **ev**, we would have seen an error:

```
Fail Inductive wrong_ev (n : nat) : Prop :=
| wrong_ev_0 : wrong_ev 0
| wrong_ev_SS (H: wrong_ev n) : wrong_ev (S (S n)).
```

In an Inductive definition, an argument to the type constructor on the left of the colon is called a “parameter”, whereas an argument on the right is called an “index” or “annotation.”

For example, in Inductive **list** (X : Type) := ..., the X is a parameter; in Inductive **ev** : **nat** → Prop := ..., the unnamed **nat** argument is an index.

We can think of the definition of **ev** as defining a Coq property **ev** : **nat** → Prop, together with “evidence constructors” **ev\_0** : **ev** 0 and **ev\_SS** : ∀ n, **ev** n → **ev** (S (S n)).

Such “evidence constructors” have the same status as proven theorems. In particular, we can use Coq’s **apply** tactic with the rule names to prove **ev** for particular numbers...

Theorem **ev\_4** : **ev** 4.

Proof. apply **ev\_SS**. apply **ev\_SS**. apply **ev\_0**. Qed.

... or we can use function application syntax:

Theorem **ev\_4'** : **ev** 4.

Proof. apply (ev\_SS 2 (ev\_SS 0 ev\_0)). Qed.

We can also prove theorems that have hypotheses involving **ev**.

Theorem **ev\_plus4** : ∀ n, **ev** n → **ev** (4 + n).

Proof.

intros n. simpl. intros Hn.

apply **ev\_SS**. apply **ev\_SS**. apply Hn.

Qed.

**Exercise: 1 star, standard (ev\_double)** Theorem **ev\_double** : ∀ n, **ev** (double n).

Proof.

*Admitted.*

□

## 8.3 Using Evidence in Proofs

Besides *constructing* evidence that numbers are even, we can also *destruct* such evidence, which amounts to reasoning about how it could have been built.

Introducing **ev** with an **Inductive** declaration tells Coq not only that the constructors `ev_0` and `ev_SS` are valid ways to build evidence that some number is **ev**, but also that these two constructors are the *only* ways to build evidence that numbers are **ev**.

In other words, if someone gives us evidence  $E$  for the assertion **ev**  $n$ , then we know that  $E$  must be one of two things:

- $E$  is `ev_0` (and  $n$  is `O`), or
- $E$  is `ev_SS`  $n'$   $E'$  (and  $n$  is `S (S`  $n'$ ), where  $E'$  is evidence for **ev**  $n'$ ).

This suggests that it should be possible to analyze a hypothesis of the form **ev**  $n$  much as we do inductively defined data structures; in particular, it should be possible to argue by *induction* and *case analysis* on such evidence. Let's look at a few examples to see what this means in practice.

### 8.3.1 Inversion on Evidence

Suppose we are proving some fact involving a number  $n$ , and we are given **ev**  $n$  as a hypothesis. We already know how to perform case analysis on  $n$  using **destruct** or **induction**, generating separate subgoals for the case where  $n = \text{O}$  and the case where  $n = \text{S } n'$  for some  $n'$ . But for some proofs we may instead want to analyze the evidence that **ev**  $n$  *directly*. As a tool, we can prove our characterization of evidence for **ev**  $n$ , using **destruct**.

Theorem `ev_inversion` :

$$\forall (n : \text{nat}), \text{ev } n \rightarrow (n = 0) \vee (\exists n', n = \text{S } (\text{S } n') \wedge \text{ev } n').$$

Proof.

```
intros n E.
destruct E as [ | n' E'] eqn:EE.
-
  left. reflexivity.
-
  right.  $\exists$  n'. split. reflexivity. apply E'.
```

Qed.

The following theorem can easily be proved using **destruct** on evidence.

Theorem `ev_minus2` :  $\forall n$ ,

$$\text{ev } n \rightarrow \text{ev } (\text{pred } (\text{pred } n)).$$

Proof.

```
intros n E.
destruct E as [| n' E'] eqn:EE.
- simpl. apply ev_0.
- simpl. apply E'.
```

Qed.

However, this variation cannot easily be handled with just `destruct`.

Theorem `evSS_ev` :  $\forall n,$   
`ev (S (S n))  $\rightarrow$  ev n.`

Intuitively, we know that evidence for the hypothesis cannot consist just of the `ev_0` constructor, since `O` and `S` are different constructors of the type `nat`; hence, `ev_SS` is the only case that applies. Unfortunately, `destruct` is not smart enough to realize this, and it still generates two subgoals. Even worse, in doing so, it keeps the final goal unchanged, failing to provide any useful information for completing the proof. **Proof.**

```
intros n E.
destruct E as [| n' E'] eqn:EE.
-
```

**Abort.**

What happened, exactly? Calling `destruct` has the effect of replacing all occurrences of the property argument by the values that correspond to each constructor. This is enough in the case of `ev_minus2` because that argument `n` is mentioned directly in the final goal. However, it doesn't help in the case of `evSS_ev` since the term that gets replaced (`S (S n)`) is not mentioned anywhere.

If we *remember* that term `S (S n)`, the proof goes through. (We'll discuss *remember* in more detail below.)

Theorem `evSS_ev_remember` :  $\forall n,$   
`ev (S (S n))  $\rightarrow$  ev n.`

**Proof.**

```
intros n E. remember (S (S n)) as k eqn:Hk. destruct E as [| n' E'] eqn:EE.
-

discriminate Hk.
-
```

```
injection Hk as Heq. rewrite  $\leftarrow$  Heq. apply E'.
```

**Qed.**

Alternatively, the proof is straightforward using our inversion lemma.

Theorem `evSS_ev` :  $\forall n,$  `ev (S (S n))  $\rightarrow$  ev n.`

**Proof.**

```
intros n H. apply ev_inversion in H.
destruct H as [H0|H1].
- discriminate H0.
- destruct H1 as [n' [Hnm Hev]]. injection Hnm as Heq.
  rewrite Heq. apply Hev.
```

**Qed.**

Note how both proofs produce two subgoals, which correspond to the two ways of proving

**ev**. The first subgoal is a contradiction that is discharged with **discriminate**. The second subgoal makes use of **injection** and **rewrite**. Coq provides a handy tactic called **inversion** that factors out that common pattern.

The **inversion** tactic can detect (1) that the first case ( $n = 0$ ) does not apply and (2) that the  $n'$  that appears in the **ev\_SS** case must be the same as  $n$ . It has an “**as**” variant similar to **destruct**, allowing us to assign names rather than have Coq choose them.

Theorem **evSS\_ev'** :  $\forall n,$   
 $\text{ev } (\text{S } (\text{S } n)) \rightarrow \text{ev } n.$

Proof.

```
intros n E.
inversion E as [| n' E' Heq].
apply E'.
```

Qed.

The **inversion** tactic can apply the principle of explosion to “obviously contradictory” hypotheses involving inductively defined properties, something that takes a bit more work using our inversion lemma. For example:

Theorem **one\_not\_even** :  $\neg \text{ev } 1.$

Proof.

```
intros H. apply ev_inversion in H.
destruct H as [| m [Hm _]].
- discriminate H.
- discriminate Hm.
```

Qed.

Theorem **one\_not\_even'** :  $\neg \text{ev } 1.$   
 intros H. inversion H. Qed.

**Exercise: 1 star, standard (inversion\_practice)** Prove the following result using **inversion**. (For extra practice, you can also prove it using the inversion lemma.)

Theorem **SSSSev\_\_even** :  $\forall n,$   
 $\text{ev } (\text{S } (\text{S } (\text{S } (\text{S } n)))) \rightarrow \text{ev } n.$

Proof.

*Admitted.*

□

**Exercise: 1 star, standard (ev5\_nonsense)** Prove the following result using **inversion**.

Theorem **ev5\_nonsense** :

$\text{ev } 5 \rightarrow 2 + 2 = 9.$

Proof.

*Admitted.*

□

The `inversion` tactic does quite a bit of work. For example, when applied to an equality assumption, it does the work of both `discriminate` and `injection`. In addition, it carries out the `intros` and `rewrites` that are typically necessary in the case of `injection`. It can also be applied, more generally, to analyze evidence for inductively defined propositions. As examples, we'll use it to reprove some theorems from chapter `Tactics`. (Here we are being a bit lazy by omitting the `as` clause from `inversion`, thereby asking Coq to choose names for the variables and hypotheses that it introduces.)

Theorem `inversion_ex1` :  $\forall (n\ m\ o : \text{nat}),$

$[n;\ m] = [o;\ o] \rightarrow$

$[n] = [m].$

Proof.

`intros n m o H. inversion H. reflexivity. Qed.`

Theorem `inversion_ex2` :  $\forall (n : \text{nat}),$

$S\ n = 0 \rightarrow$

$2 + 2 = 5.$

Proof.

`intros n contra. inversion contra. Qed.`

Here's how `inversion` works in general. Suppose the name  $H$  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 H` generates a subgoal in which  $H$  has been replaced by the exact, specific conditions under which this constructor could have been used to prove  $P$ . Some of these subgoals will be self-contradictory; `inversion` throws these away. The ones that are left represent the cases that must be proved to establish the original goal. For those, `inversion` adds all equations into the proof context that must hold of the arguments given to  $P$  (e.g.,  $S\ (S\ n') = n$  in the proof of `evSS_ev`).

The `ev_double` exercise above shows that our new notion of evenness is implied by the two earlier ones (since, by `even_bool_prop` in chapter `Logic`, we already know that those are equivalent to each other). To show that all three coincide, we just need the following lemma.

Lemma `ev_even_firsttry` :  $\forall\ n,$

$\text{ev}\ n \rightarrow \text{even}\ n.$

Proof.

We could try to proceed by case analysis or induction on  $n$ . But since `ev` is mentioned in a premise, this strategy would probably lead to a dead end, because (as we've noted before) the induction hypothesis will talk about  $n-1$  (which is *not* even!). Thus, it seems better to first try `inversion` on the evidence for `ev`. Indeed, the first case can be solved trivially. And we can seemingly make progress on the second case with a helper lemma.

`intros n E. inversion E as [EQ' | n' E' EQ'].`

-

$\exists\ 0. \text{reflexivity}.$

- `simpl.`



Unfortunately, the second case is harder. We need to show  $\exists k, S (S \ n') = \text{double } k$ , but the only available assumption is  $E'$ , which states that  $\mathbf{ev} \ n'$  holds. Since this isn't directly useful, it seems that we are stuck and that performing case analysis on  $E$  was a waste of time.

If we look more closely at our second goal, however, we can see that something interesting happened: By performing case analysis on  $E$ , we were able to reduce the original result to a similar one that involves a *different* piece of evidence for  $\mathbf{ev}$ : namely  $E'$ . More formally, we can finish our proof by showing that

exists  $k', n' = \text{double } k'$ ,

which is the same as the original statement, but with  $n'$  instead of  $n$ . Indeed, it is not difficult to convince Coq that this intermediate result suffices.

Unfortunately, now we are stuck. To make that apparent, let's move  $E'$  back into the goal from the hypotheses.

generalize dependent  $E'$ .

Now it is clear we are trying to prove another instance of the same theorem we set out to prove. This instance is with  $n'$ , instead of  $n$ , where  $n'$  is a smaller natural number than  $n$ . **Abort.**

### 8.3.2 Induction on Evidence

If this looks familiar, it is no coincidence: We've encountered similar problems in the **Induction** chapter, when trying to use case analysis to prove results that required induction. And once again the solution is... induction!

The behavior of **induction** on evidence is the same as its behavior on data: It causes Coq to generate one subgoal for each constructor that could have used to build that evidence, while providing an induction hypothesis for each recursive occurrence of the property in question.

To prove a property of  $n$  holds for all numbers for which  $\mathbf{ev} \ n$  holds, we can use induction on  $\mathbf{ev} \ n$ . This requires us to prove two things, corresponding to the two ways in which  $\mathbf{ev} \ n$  could have been constructed. If it was constructed by  $\mathbf{ev\_0}$ , then  $n=0$ , and the property must hold of 0. If it was constructed by  $\mathbf{ev\_SS}$ , then the evidence of  $\mathbf{ev} \ n$  is of the form  $\mathbf{ev\_SS} \ n' \ E'$ , where  $n = S (S \ n')$  and  $E'$  is evidence for  $\mathbf{ev} \ n'$ . In this case, the inductive hypothesis says that the property we are trying to prove holds for  $n'$ .

Let's try our current lemma again:

Lemma `ev_even` :  $\forall \ n,$

$\mathbf{ev} \ n \rightarrow \text{even } n$ .

**Proof.**

`intros  $n \ E$ .`

`induction  $E$  as [ $|n' \ E' \ IH$ ].`

-

$\exists 0$ . `reflexivity.`

-

```

    unfold even in IH.
    destruct IH as [k Hk].
    rewrite Hk.  $\exists$  (S k). simpl. reflexivity.
Qed.

```

Here, we can see that Coq produced an  $IH$  that corresponds to  $E'$ , the single recursive occurrence of **ev** in its own definition. Since  $E'$  mentions  $n'$ , the induction hypothesis talks about  $n'$ , as opposed to  $n$  or some other number.

The equivalence between the second and third definitions of evenness now follows.

**Theorem** `ev_even_iff` :  $\forall n$ ,

**ev**  $n \leftrightarrow$  even  $n$ .

**Proof.**

```

    intros n. split.
    - apply ev_even.
    - unfold even. intros [k Hk]. rewrite Hk. apply ev_double.

```

**Qed.**

As we will see in later chapters, induction on evidence is a recurring technique across many areas, and in particular when formalizing the semantics of programming languages, where many properties of interest are defined inductively.

The following exercises provide simple examples of this technique, to help you familiarize yourself with it.

**Exercise: 2 stars, standard (ev\_sum)** **Theorem** `ev_sum` :  $\forall n\ m$ , **ev**  $n \rightarrow$  **ev**  $m \rightarrow$  **ev**  $(n + m)$ .

**Proof.**

*Admitted.*

□

**Exercise: 4 stars, advanced, optional (ev'\_ev)** In general, there may be multiple ways of defining a property inductively. For example, here's a (slightly contrived) alternative definition for **ev**:

**Inductive** `ev'` : **nat**  $\rightarrow$  **Prop** :=

| `ev'_0` : **ev'** 0

| `ev'_2` : **ev'** 2

| `ev'_sum`  $n\ m$  ( $Hn$  : **ev'**  $n$ ) ( $Hm$  : **ev'**  $m$ ) : **ev'**  $(n + m)$ .

Prove that this definition is logically equivalent to the old one. To streamline the proof, use the technique (from Logic) of applying theorems to arguments, and note that the same technique works with constructors of inductively defined propositions.

**Theorem** `ev'_ev` :  $\forall n$ , **ev'**  $n \leftrightarrow$  **ev**  $n$ .

**Proof.**

*Admitted.*

□

**Exercise: 3 stars, advanced, especially useful (ev\_ev\_ev)** There are two pieces of evidence you could attempt to induct upon here. If one doesn't work, try the other.

Theorem `ev_ev_ev` :  $\forall n m,$   
 $\text{ev } (n+m) \rightarrow \text{ev } n \rightarrow \text{ev } m.$

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard, optional (ev\_plus\_plus)** This exercise can be completed without induction or case analysis. But, you will need a clever assertion and some tedious rewriting. Hint: is  $(n+m) + (n+p)$  even?

Theorem `ev_plus_plus` :  $\forall n m p,$   
 $\text{ev } (n+m) \rightarrow \text{ev } (n+p) \rightarrow \text{ev } (m+p).$

Proof.

*Admitted.*

□

## 8.4 Inductive Relations

A proposition parameterized by a number (such as `ev`) can be thought of as a *property* – i.e., it defines a subset of `nat`, namely those numbers for which the proposition is provable. In the same way, a two-argument proposition can be thought of as a *relation* – i.e., it defines a set of pairs for which the proposition is provable.

Module `PLAYGROUND`.

Just like properties, relations can be defined inductively. One useful example is the “less than or equal to” relation on numbers.

The following definition should be fairly intuitive. It says that there are two ways to give evidence that one number is less than or equal to another: either observe that they are the same number, or give evidence that the first is less than or equal to the predecessor of the second.

Inductive `le` : `nat`  $\rightarrow$  `nat`  $\rightarrow$  `Prop` :=  
 | `le_n` ( $n$  : `nat`) : `le`  $n$   $n$   
 | `le_S` ( $n$   $m$  : `nat`) ( $H$  : `le`  $n$   $m$ ) : `le`  $n$  (`S`  $m$ ).

Notation "`n <= m`" := (`le`  $n$   $m$ ).

Proofs of facts about  $\leq$  using the constructors `le_n` and `le_S` follow the same patterns as proofs about properties, like `ev` above. We can **apply** the constructors to prove  $\leq$  goals (e.g., to show that  $3 \leq 3$  or  $3 \leq 6$ ), and we can use tactics like `inversion` to extract information from  $\leq$  hypotheses in the context (e.g., to prove that  $(2 \leq 1) \rightarrow 2+2=5$ .)

Here are some sanity checks on the definition. (Notice that, although these are the same kind of simple “unit tests” as we gave for the testing functions we wrote in the first few

lectures, we must construct their proofs explicitly – `simpl` and `reflexivity` don't do the job, because the proofs aren't just a matter of simplifying computations.)

Theorem `test_le1` :

$3 \leq 3$ .

Proof.

`apply le_n. Qed.`

Theorem `test_le2` :

$3 \leq 6$ .

Proof.

`apply le_S. apply le_S. apply le_S. apply le_n. Qed.`

Theorem `test_le3` :

$(2 \leq 1) \rightarrow 2 + 2 = 5$ .

Proof.

`intros H. inversion H. inversion H2. Qed.`

The “strictly less than” relation  $n < m$  can now be defined in terms of `le`.

Definition `lt (n m : nat) := le (S n) m`.

Notation "`m < n`" := `(lt m n)`.

End PLAYGROUND.

Here are a few more simple relations on numbers:

Inductive `square_of : nat → nat → Prop` :=

| `sq n : square_of n (n × n)`.

Inductive `next_nat : nat → nat → Prop` :=

| `nn n : next_nat n (S n)`.

Inductive `next_ev : nat → nat → Prop` :=

| `ne_1 n (H : ev (S n)) : next_ev n (S n)`  
| `ne_2 n (H : ev (S (S n))) : next_ev n (S (S n))`.

**Exercise: 2 stars, standard, optional (total\_relation)** Define an inductive binary relation *total\_relation* that holds between every pair of natural numbers.

**Exercise: 2 stars, standard, optional (empty\_relation)** Define an inductive binary relation *empty\_relation* (on numbers) that never holds.

From the definition of `le`, we can sketch the behaviors of `destruct`, `inversion`, and `induction` on a hypothesis *H* providing evidence of the form `le e1 e2`. Doing `destruct H` will generate two cases. In the first case,  $e1 = e2$ , and it will replace instances of *e2* with *e1* in the goal and context. In the second case,  $e2 = S\ n'$  for some *n'* for which `le e1 n'` holds, and it will replace instances of *e2* with `S n'`. Doing `inversion H` will remove impossible cases and add generated equalities to the context for further use. Doing `induction H` will,

in the second case, add the induction hypothesis that the goal holds when  $e2$  is replaced with  $n'$ .

**Exercise: 3 stars, standard, optional (le\_exercises)** Here are a number of facts about the  $\leq$  and  $<$  relations that we are going to need later in the course. The proofs make good practice exercises.

Lemma `le_trans` :  $\forall m\ n\ o, m \leq n \rightarrow n \leq o \rightarrow m \leq o$ .

Proof.

*Admitted.*

Theorem `O_le_n` :  $\forall n,$

$0 \leq n$ .

Proof.

*Admitted.*

Theorem `n_le_m__Sn_le_Sm` :  $\forall n\ m,$

$n \leq m \rightarrow S\ n \leq S\ m$ .

Proof.

*Admitted.*

Theorem `Sn_le_Sm__n_le_m` :  $\forall n\ m,$

$S\ n \leq S\ m \rightarrow n \leq m$ .

Proof.

*Admitted.*

Theorem `le_plus_l` :  $\forall a\ b,$

$a \leq a + b$ .

Proof.

*Admitted.*

Theorem `plus_le` :  $\forall n1\ n2\ m,$

$n1 + n2 \leq m \rightarrow$

$n1 \leq m \wedge n2 \leq m$ .

Proof.

*Admitted.*

Hint: the next one may be easiest to prove by induction on  $n$ .

Theorem `add_le_cases` :  $\forall n\ m\ p\ q,$

$n + m \leq p + q \rightarrow n \leq p \vee m \leq q$ .

Proof.

*Admitted.*

Theorem `lt_S` :  $\forall n\ m,$

$n < m \rightarrow$

$n < S\ m$ .

Proof.

*Admitted.*

Theorem plus\_lt :  $\forall n1\ n2\ m,$   
 $n1 + n2 < m \rightarrow$   
 $n1 < m \wedge n2 < m.$

Proof.

*Admitted.*

Theorem leb\_complete :  $\forall n\ m,$   
 $n \leq m = \text{true} \rightarrow n \leq m.$

Proof.

*Admitted.*

Hint: The next one may be easiest to prove by induction on  $m$ .

Theorem leb\_correct :  $\forall n\ m,$   
 $n \leq m \rightarrow$   
 $n \leq m = \text{true}.$

Proof.

*Admitted.*

Hint: The next one can easily be proved without using induction.

Theorem leb\_true\_trans :  $\forall n\ m\ o,$   
 $n \leq m = \text{true} \rightarrow m \leq o = \text{true} \rightarrow n \leq o = \text{true}.$

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard, optional (leb\_iff)** Theorem leb\_iff :  $\forall n\ m,$   
 $n \leq m = \text{true} \leftrightarrow n \leq m.$

Proof.

*Admitted.*

□

Module R.

**Exercise: 3 stars, standard, especially useful (R\_provability)** 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 :  $\text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop} :=$   
| c1 : R 0 0 0  
| c2 m n o (H : R m n o) : R (S m) n (S o)  
| c3 m n o (H : R m n o) : R m (S n) (S o)  
| c4 m n o (H : R (S m) (S n) (S (S o))) : R m n o  
| c5 m n o (H : 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.

Definition manual\_grade\_for\_R\_provability : **option** (**nat**×**string**) := **None**.

□

**Exercise: 3 stars, standard, optional (R\_fact)** The relation **R** above actually encodes a familiar function. Figure out which function; then state and prove this equivalence in Coq?

Definition fR : **nat** → **nat** → **nat**

. *Admitted*.

Theorem R\_equiv\_fR :  $\forall m\ n\ o, \mathbf{R}\ m\ n\ o \leftrightarrow fR\ m\ n = o$ .

Proof.

*Admitted*.

□

End R.

**Exercise: 2 stars, advanced (subsequence)** A list is a *subsequence* of another list if all of the elements in the first list occur in the same order in the second list, possibly with some extra elements in between. For example,

1;2;3

is a subsequence of each of the lists

1;2;3 1;1;1;2;2;3 1;2;7;3 5;6;1;9;9;2;7;3;8

but it is *not* a subsequence of any of the lists

1;2 1;3 5;6;2;1;7;3;8.

- Define an inductive proposition **subseq** on **list nat** that captures what it means to be a subsequence. (Hint: You'll need three cases.)
- Prove **subseq\_refl** that subsequence is reflexive, that is, any list is a subsequence of itself.
- Prove **subseq\_app** that for any lists *l1*, *l2*, and *l3*, if *l1* is a subsequence of *l2*, then *l1* is also a subsequence of *l2* ++ *l3*.
- (Optional, harder) Prove **subseq\_trans** that subsequence is transitive – that is, if *l1* is a subsequence of *l2* and *l2* is a subsequence of *l3*, then *l1* is a subsequence of *l3*. Hint: choose your induction carefully!

Inductive **subseq** : list nat → list nat → Prop :=

.

Theorem subseq\_refl : ∀ (l : list nat), subseq l l.

Proof.

*Admitted.*

Theorem subseq\_app : ∀ (l1 l2 l3 : list nat),

subseq l1 l2 →

subseq l1 (l2 ++ l3).

Proof.

*Admitted.*

Theorem subseq\_trans : ∀ (l1 l2 l3 : list nat),

subseq l1 l2 →

subseq l2 l3 →

subseq l1 l3.

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard, optional (R\_provability2)** Suppose we give Coq the following definition:

Inductive R : nat -> list nat -> Prop := | c1 : R 0 [] | c2 n l (H: R n l) : R (S n) (n :: l) | c3 n l (H: R (S n) l) : R n l.

Which of the following propositions are provable?

- R 2 [1;0]
- R 1 [1;2;1;0]
- R 6 [3;2;1;0]

## 8.5 Case Study: Regular Expressions

The **ev** property provides a simple example for illustrating inductive definitions and the basic techniques for reasoning about them, but it is not terribly exciting – after all, it is equivalent to the two non-inductive definitions of evenness that we had already seen, and does not seem to offer any concrete benefit over them.

To give a better sense of the power of inductive definitions, we now show how to use them to model a classic concept in computer science: *regular expressions*.

Regular expressions are a simple language for describing sets of strings. Their syntax is defined as follows:



```

Inductive reg_exp (T : Type) : Type :=
| EmptySet
| EmptyStr
| Char (t : T)
| App (r1 r2 : reg_exp T)
| Union (r1 r2 : reg_exp T)
| Star (r : reg_exp T).

```

*Arguments* EmptySet {T}.

*Arguments* EmptyStr {T}.

*Arguments* Char {T} ..

*Arguments* App {T} - ..

*Arguments* Union {T} - ..

*Arguments* Star {T} ..

Note that this definition is *polymorphic*: Regular expressions in **reg\_exp**  $T$  describe strings with characters drawn from  $T$  – that is, lists of elements of  $T$ .

(We depart slightly from standard practice in that we do not require the type  $T$  to be finite. This results in a somewhat different theory of regular expressions, but the difference is not significant for our purposes.)

We connect regular expressions and strings via the following rules, which define when a regular expression *matches* some string:

- The expression **EmptySet** does not match any string.
- The expression **EmptyStr** matches the empty string `[]`.
- The expression **Char**  $x$  matches the one-character string `[x]`.
- If  $re1$  matches  $s1$ , and  $re2$  matches  $s2$ , then **App**  $re1$   $re2$  matches  $s1 ++ s2$ .
- If at least one of  $re1$  and  $re2$  matches  $s$ , then **Union**  $re1$   $re2$  matches  $s$ .
- Finally, if we can write some string  $s$  as the concatenation of a sequence of strings  $s = s_1 ++ \dots ++ s_k$ , and the expression  $re$  matches each one of the strings  $s_i$ , then **Star**  $re$  matches  $s$ .

In particular, the sequence of strings may be empty, so **Star**  $re$  always matches the empty string `[]` no matter what  $re$  is.

We can easily translate this informal definition into an **Inductive** one as follows. We use the notation  $s \sim re$  in place of **exp\_match**  $s$   $re$ ; by “reserving” the notation before defining the **Inductive**, we can use it in the definition!

**Reserved Notation** “ $s \sim re$ ” (at level 80).

```

Inductive exp_match {T} : list T → reg_exp T → Prop :=
| MEmpty : [] ~ EmptyStr

```

```

| MChar x : [x] =~ (Char x)
| MApp s1 re1 s2 re2
    (H1 : s1 =~ re1)
    (H2 : s2 =~ re2)
    : (s1 ++ s2) =~ (App re1 re2)
| MUnionL s1 re1 re2
    (H1 : s1 =~ re1)
    : s1 =~ (Union re1 re2)
| MUnionR re1 s2 re2
    (H2 : s2 =~ re2)
    : s2 =~ (Union re1 re2)
| MStar0 re : [] =~ (Star re)
| MStarApp s1 s2 re
    (H1 : s1 =~ re)
    (H2 : s2 =~ (Star re))
    : (s1 ++ s2) =~ (Star re)
where "s =~ re" := (exp_match s re).

```

Lemma quiz :  $\forall T (s:\text{list } T), \sim(s = \sim \text{EmptySet})$ .

Proof. intros  $T s Hc$ . inversion  $Hc$ . Qed.

Again, for readability, we display this definition using inference-rule notation.

---

(MEmpty)  $\square = \sim \text{EmptyStr}$

---

(MChar)  $x = \sim \text{Char } x$   
 $s1 = \sim \text{re1 } s2 = \sim \text{re2}$

---

(MApp)  $s1 ++ s2 = \sim \text{App re1 re2}$   
 $s1 = \sim \text{re1}$

---

(MUnionL)  $s1 = \sim \text{Union re1 re2}$   
 $s2 = \sim \text{re2}$

---

(MUnionR)  $s2 = \sim \text{Union re1 re2}$

---

(MStar0)  $\square = \sim \text{Star re}$   
 $s1 = \sim \text{re } s2 = \sim \text{Star re}$

---

(MStarApp)  $s1 ++ s2 = \sim \text{Star re}$

---

Notice that these rules are not *quite* the same as the informal ones that we gave at the beginning of the section. First, we don't need to include a rule explicitly stating that no string matches `EmptySet`; we just don't happen to include any rule that would have the effect

of some string matching `EmptySet`. (Indeed, the syntax of inductive definitions doesn't even *allow* us to give such a “negative rule.”)

Second, the informal rules for `Union` and `Star` correspond to two constructors each: `MUnionL` / `MUnionR`, and `MStar0` / `MStarApp`. The result is logically equivalent to the original rules but more convenient to use in Coq, since the recursive occurrences of `exp_match` are given as direct arguments to the constructors, making it easier to perform induction on evidence. (The `exp_match_ex1` and `exp_match_ex2` exercises below ask you to prove that the constructors given in the inductive declaration and the ones that would arise from a more literal transcription of the informal rules are indeed equivalent.)

Let's illustrate these rules with a few examples.

Example `reg_exp_ex1 : [1] =~ Char 1`.

Proof.

`apply MChar.`

Qed.

Example `reg_exp_ex2 : [1; 2] =~ App (Char 1) (Char 2)`.

Proof.

`apply (MApp [1]).`

`- apply MChar.`

`- apply MChar.`

Qed.

(Notice how the last example applies `MApp` to the string `[1]` directly. Since the goal mentions `[1; 2]` instead of `[1] ++ [2]`, Coq wouldn't be able to figure out how to split the string on its own.)

Using `inversion`, we can also show that certain strings do *not* match a regular expression:

Example `reg_exp_ex3 : ¬ ([1; 2] =~ Char 1)`.

Proof.

`intros H. inversion H.`

Qed.

We can define helper functions for writing down regular expressions. The `reg_exp_of_list` function constructs a regular expression that matches exactly the list that it receives as an argument:

```
Fixpoint reg_exp_of_list {T} (l : list T) :=
  match l with
  | [] => EmptyStr
  | x :: l' => App (Char x) (reg_exp_of_list l')
  end.
```

Example `reg_exp_ex4 : [1; 2; 3] =~ reg_exp_of_list [1; 2; 3]`.

Proof.

`simpl. apply (MApp [1]).`

`{ apply MChar. }`

```

  apply (MApp [2]).
  { apply MChar. }
  apply (MApp [3]).
  { apply MChar. }
  apply MEmpty.
Qed.

```

We can also prove general facts about **exp\_match**. For instance, the following lemma shows that every string  $s$  that matches  $re$  also matches **Star**  $re$ .

Lemma MStar1 :

$$\forall T \ s \ (re : \mathbf{reg\_exp} \ T) ,$$

$$s =^{\sim} re \rightarrow$$

$$s =^{\sim} \mathbf{Star} \ re.$$

Proof.

```

  intros T s re H.
  rewrite ← (app_nil_r _ s).
  apply MStarApp.
- apply H.
- apply MStar0.

```

Qed.

(Note the use of `app_nil_r` to change the goal of the theorem to exactly the same shape expected by `MStarApp`.)

**Exercise: 3 stars, standard (exp\_match\_ex1)** The following lemmas show that the informal matching rules given at the beginning of the chapter can be obtained from the formal inductive definition.

Lemma empty\_is\_empty :  $\forall T \ (s : \mathbf{list} \ T),$   
 $\neg (s =^{\sim} \mathbf{EmptySet}).$

Proof.

*Admitted.*

Lemma MUnion' :  $\forall T \ (s : \mathbf{list} \ T) \ (re1 \ re2 : \mathbf{reg\_exp} \ T),$   
 $s =^{\sim} re1 \vee s =^{\sim} re2 \rightarrow$   
 $s =^{\sim} \mathbf{Union} \ re1 \ re2.$

Proof.

*Admitted.*

The next lemma is stated in terms of the `fold` function from the `Poly` chapter: If  $ss : \mathbf{list} \ (\mathbf{list} \ T)$  represents a sequence of strings  $s_1, \dots, s_n$ , then `fold app ss []` is the result of concatenating them all together.

Lemma MStar' :  $\forall T \ (ss : \mathbf{list} \ (\mathbf{list} \ T)) \ (re : \mathbf{reg\_exp} \ T),$   
 $(\forall s, \text{In } s \ ss \rightarrow s =^{\sim} re) \rightarrow$   
 $\text{fold app } ss \ [] =^{\sim} \mathbf{Star} \ re.$

Proof.

*Admitted.*

□

**Exercise: 4 stars, standard, optional (reg\_exp\_of\_list\_spec)** Prove that `reg_exp_of_list` satisfies the following specification:

Lemma `reg_exp_of_list_spec` :  $\forall T (s1\ s2 : \text{list } T),$   
 $s1 =^{\sim} \text{reg\_exp\_of\_list } s2 \leftrightarrow s1 = s2.$

Proof.

*Admitted.*

□

Since the definition of **exp\_match** has a recursive structure, we might expect that proofs involving regular expressions will often require induction on evidence.

For example, suppose that we wanted to prove the following intuitive result: If a regular expression `re` matches some string `s`, then all elements of `s` must occur as character literals somewhere in `re`.

To state this theorem, we first define a function `re_chars` that lists all characters that occur in a regular expression:

```
Fixpoint re_chars {T} (re : reg_exp T) : list T :=
  match re with
  | EmptySet => []
  | EmptyStr => []
  | Char x => [x]
  | App re1 re2 => re_chars re1 ++ re_chars re2
  | Union re1 re2 => re_chars re1 ++ re_chars re2
  | Star re => re_chars re
end.
```

We can then phrase our theorem as follows:

Theorem `in_re_match` :  $\forall T (s : \text{list } T) (re : \text{reg\_exp } T) (x : T),$   
 $s =^{\sim} re \rightarrow$   
 $\text{In } x\ s \rightarrow$   
 $\text{In } x\ (\text{re\_chars } re).$

Proof.

```
intros T s re x Hmatch Hin.
induction Hmatch
as [| x'
   | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
   | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
   | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2].
-
simpl in Hin. destruct Hin.
```

```
-
  simpl. simpl in Hin.
  apply Hin.
```

```
-
  simpl.
```

Something interesting happens in the **MApp** case. We obtain *two* induction hypotheses: One that applies when  $x$  occurs in  $s1$  (which matches  $re1$ ), and a second one that applies when  $x$  occurs in  $s2$  (which matches  $re2$ ).

```
  simpl. rewrite ln_app_iff in *.
  destruct Hin as [Hin | Hin].
+
  left. apply (IH1 Hin).
+
  right. apply (IH2 Hin).
-
  simpl. rewrite ln_app_iff.
  left. apply (IH Hin).
-
  simpl. rewrite ln_app_iff.
  right. apply (IH Hin).
-
  destruct Hin.
-
  simpl.
```

Here again we get two induction hypotheses, and they illustrate why we need induction on evidence for **exp\_match**, rather than induction on the regular expression  $re$ : The latter would only provide an induction hypothesis for strings that match  $re$ , which would not allow us to reason about the case  $\ln x\ s2$ .

```
  rewrite ln_app_iff in Hin.
  destruct Hin as [Hin | Hin].
+
  apply (IH1 Hin).
+
  apply (IH2 Hin).
Qed.
```

**Exercise: 4 stars, standard (re\_not\_empty)** Write a recursive function `re_not_empty` that tests whether a regular expression matches some string. Prove that your function is correct.

```
Fixpoint re_not_empty {T : Type} (re : reg_exp T) : bool
. Admitted.
```

Lemma `re_not_empty_correct` :  $\forall T (re : \mathbf{reg\_exp} T),$   
 $(\exists s, s =^{\sim} re) \leftrightarrow re\_not\_empty\ re = \mathbf{true}.$

Proof.

*Admitted.*

□

### 8.5.1 The *remember* Tactic

One potentially confusing feature of the `induction` tactic is that it will let you try to perform an induction over a term that isn't sufficiently general. The effect of this is to lose information (much as `destruct` without an *eqn:* clause can do), and leave you unable to complete the proof. Here's an example:

Lemma `star_app`:  $\forall T (s1\ s2 : \mathbf{list} T) (re : \mathbf{reg\_exp} T),$   
 $s1 =^{\sim} \mathbf{Star}\ re \rightarrow$   
 $s2 =^{\sim} \mathbf{Star}\ re \rightarrow$   
 $s1 ++ s2 =^{\sim} \mathbf{Star}\ re.$

Proof.

`intros T s1 s2 re H1.`

Just doing an `inversion` on *H1* won't get us very far in the recursive cases. (Try it!). So we need induction (on evidence!). Here is a naive first attempt:

`generalize dependent s2.`

`induction H1`

`as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2`  
`|s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH`  
`|re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].`

But now, although we get seven cases (as we would expect from the definition of `exp_match`), we have lost a very important bit of information from *H1*: the fact that *s1* matched something of the form `Star re`. This means that we have to give proofs for *all* seven constructors of this definition, even though all but two of them (`MStar0` and `MStarApp`) are contradictory. We can still get the proof to go through for a few constructors, such as `MEmpy...`

-

`simpl. intros s2 H. apply H.`

... but most cases get stuck. For `MChar`, for instance, we must show that  
 $s2 =^{\sim} \mathbf{Char}\ x' \rightarrow x' :: s2 =^{\sim} \mathbf{Char}\ x',$   
 which is clearly impossible.

- `intros s2 H. simpl. Abort.`

The problem is that `induction` over a Prop hypothesis only works properly with hypotheses that are completely general, i.e., ones in which all the arguments are variables, as opposed to more complex expressions, such as `Star re`.

(In this respect, `induction` on evidence behaves more like `destruct-without-eqn`: than like `inversion`.)

An awkward way to solve this problem is “manually generalizing” over the problematic expressions by adding explicit equality hypotheses to the lemma:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re re' : reg_exp T),
  re' = Star re →
  s1 =~ re' →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
```

We can now proceed by performing induction over evidence directly, because the argument to the first hypothesis is sufficiently general, which means that we can discharge most cases by inverting the  $re' = \text{Star } re$  equality in the context.

This idiom is so common that Coq provides a tactic to automatically generate such equations for us, avoiding thus the need for changing the statements of our theorems.

**Abort.**

The tactic `remember e as x` causes Coq to (1) replace all occurrences of the expression  $e$  by the variable  $x$ , and (2) add an equation  $x = e$  to the context. Here’s how we can use it to show the above result:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
```

**Proof.**

```
intros T s1 s2 re H1.
remember (Star re) as re'.
```

We now have  $H_{egre'} : re' = \text{Star } re$ .

```
generalize dependent s2.
```

```
induction H1
```

```
as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
   |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
   |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

The  $H_{egre'}$  is contradictory in most cases, allowing us to conclude immediately.

```
- discriminate.
- discriminate.
- discriminate.
- discriminate.
- discriminate.
```

The interesting cases are those that correspond to `Star`. Note that the induction hypothesis  $IH2$  on the `MStarApp` case mentions an additional premise  $\text{Star } re'' = \text{Star } re$ , which results from the equality generated by `remember`.



```

-
  injection Heqre' as Heqre''. intros s H. apply H.
-
  injection Heqre' as Heqre''.
  intros s2 H1. rewrite ← app_assoc.
  apply MStarApp.
  + apply Hmatch1.
  + apply IH2.
    × rewrite Heqre''. reflexivity.
    × apply H1.

```

Qed.

**Exercise: 4 stars, standard, optional (exp\_match\_ex2)** The MStar'' lemma below (combined with its converse, the MStar' exercise above), shows that our definition of **exp\_match** for Star is equivalent to the informal one given previously.

Lemma MStar'' :  $\forall T (s : \text{list } T) (re : \text{reg\_exp } T),$   
 $s = \sim \text{Star } re \rightarrow$   
 $\exists ss : \text{list } (\text{list } T),$   
 $s = \text{fold app } ss []$   
 $\wedge \forall s', \text{in } s' ss \rightarrow s' = \sim re.$

Proof.

*Admitted.*

□

**Exercise: 5 stars, advanced (weak\_pumping)** One of the first really interesting theorems in the theory of regular expressions is the so-called *pumping lemma*, which states, informally, that any sufficiently long string  $s$  matching a regular expression  $re$  can be “pumped” by repeating some middle section of  $s$  an arbitrary number of times to produce a new string also matching  $re$ . (For the sake of simplicity in this exercise, we consider a slightly weaker theorem than is usually stated in courses on automata theory.)

To get started, we need to define “sufficiently long.” Since we are working in a constructive logic, we actually need to be able to calculate, for each regular expression  $re$ , the minimum length for strings  $s$  to guarantee “pumpability.”

Module PUMPING.

```

Fixpoint pumping_constant {T} (re : reg_exp T) : nat :=
  match re with
  | EmptySet ⇒ 1
  | EmptyStr ⇒ 1
  | Char _ ⇒ 2
  | App re1 re2 ⇒
    pumping_constant re1 + pumping_constant re2

```

```

| Union re1 re2 ⇒
  pumping_constant re1 + pumping_constant re2
| Star r ⇒ pumping_constant r
end.

```

You may find these lemmas about the pumping constant useful when proving the pumping lemma below.

Lemma pumping\_constant\_ge\_1 :

```

∀ T (re : reg_exp T),
  pumping_constant re ≥ 1.

```

Proof.

```

intros T re. induction re.
-
  apply le_n.
-
  apply le_n.
-
  apply le_S. apply le_n.
-
  simpl.
  apply le_trans with (n:=pumping_constant re1).
  apply IHre1. apply le_plus_l.
-
  simpl.
  apply le_trans with (n:=pumping_constant re1).
  apply IHre1. apply le_plus_l.
-
  simpl. apply IHre.

```

Qed.

Lemma pumping\_constant\_0\_false :

```

∀ T (re : reg_exp T),
  pumping_constant re = 0 → False.

```

Proof.

```

intros T re H.
assert (Hp1 : pumping_constant re ≥ 1).
{ apply pumping_constant_ge_1. }
inversion Hp1 as [Hp1' | p Hp1' Hp1''].
- rewrite H in Hp1'. discriminate Hp1'.
- rewrite H in Hp1''. discriminate Hp1''.

```

Qed.

Next, it is useful to define an auxiliary function that repeats a string (appends it to itself) some number of times.

```

Fixpoint napp {T} (n : nat) (l : list T) : list T :=
  match n with
  | 0 => []
  | S n' => l ++ napp n' l
  end.

```

This auxiliary lemma might also be useful in your proof of the pumping lemma.

Lemma napp\_plus:  $\forall T (n\ m : \text{nat}) (l : \text{list } T),$   
 $\text{napp } (n + m) l = \text{napp } n l ++ \text{napp } m l.$

Proof.

```

intros T n m l.
induction n as [|n IHn].
- reflexivity.
- simpl. rewrite IHn, app_assoc. reflexivity.

```

Qed.

Lemma napp\_star :

$\forall T\ m\ s1\ s2\ (re : \text{reg\_exp } T),$   
 $s1 \sim re \rightarrow s2 \sim \text{Star } re \rightarrow$   
 $\text{napp } m\ s1 ++ s2 \sim \text{Star } re.$

Proof.

```

intros T m s1 s2 re Hs1 Hs2.
induction m.
- simpl. apply Hs2.
- simpl. rewrite <- app_assoc.
  apply MStarApp.
  + apply Hs1.
  + apply IHm.

```

Qed.

The (weak) pumping lemma itself says that, if  $s \sim re$  and if the length of  $s$  is at least the pumping constant of  $re$ , then  $s$  can be split into three substrings  $s1 ++ s2 ++ s3$  in such a way that  $s2$  can be repeated any number of times and the result, when combined with  $s1$  and  $s3$  will still match  $re$ . Since  $s2$  is also guaranteed not to be the empty string, this gives us a (constructive!) way to generate strings matching  $re$  that are as long as we like.

Lemma weak\_pumping :  $\forall T (re : \text{reg\_exp } T) s,$

$s \sim re \rightarrow$   
 $\text{pumping\_constant } re \leq \text{length } s \rightarrow$   
 $\exists s1\ s2\ s3,$   
 $s = s1 ++ s2 ++ s3 \wedge$   
 $s2 \neq [] \wedge$   
 $\forall m, s1 ++ \text{napp } m\ s2 ++ s3 \sim re.$

You are to fill in the proof. Several of the lemmas about **le** that were in an optional

exercise earlier in this chapter may be useful. **Proof.**

```

intros T re s Hmatch.
induction Hmatch
as [ | x | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
   | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
   | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2 ].
-
  simpl. intros contra. inversion contra.
  Admitted.
□

```

**Exercise: 5 stars, advanced, optional (pumping)** Now here is the usual version of the pumping lemma. In addition to requiring that  $s2 \neq []$ , it also requires that  $\text{length } s1 + \text{length } s2 \leq \text{pumping\_constant } re$ .

Lemma pumping :  $\forall T (re : \text{reg\_exp } T) s,$   
 $s =^{\sim} re \rightarrow$   
 $\text{pumping\_constant } re \leq \text{length } s \rightarrow$   
 $\exists s1 s2 s3,$   
 $s = s1 ++ s2 ++ s3 \wedge$   
 $s2 \neq [] \wedge$   
 $\text{length } s1 + \text{length } s2 \leq \text{pumping\_constant } re \wedge$   
 $\forall m, s1 ++ \text{napp } m s2 ++ s3 =^{\sim} re.$

You may want to copy your proof of weak\_pumping below. **Proof.**

```

intros T re s Hmatch.
induction Hmatch
as [ | x | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
   | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
   | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2 ].
-
  simpl. intros contra. inversion contra.
  Admitted.

```

End PUMPING.

□

## 8.6 Case Study: Improving Reflection

We've seen in the Logic chapter that we often need to relate boolean computations to statements in Prop. But performing this conversion as we did it there can result in tedious proof scripts. Consider the proof of the following theorem:

Theorem filter\_not\_empty\_ln :  $\forall n l,$   
 $\text{filter } (\text{fun } x \Rightarrow n =? x) l \neq [] \rightarrow$

```

  in n l.
Proof.
  intros n l. induction l as [|m l' IHL'].
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (n =? m) eqn:H.
    +
      intros _. rewrite eqb_eq in H. rewrite H.
      left. reflexivity.
    +
      intros H'. right. apply IHL'. apply H'.
Qed.

```

In the first branch after `destruct`, we explicitly apply the `eqb_eq` lemma to the equation generated by destructing `n =? m`, to convert the assumption `n =? m = true` into the assumption `n = m`; then we had to `rewrite` using this assumption to complete the case.

We can streamline this by defining an inductive proposition that yields a better case-analysis principle for `n =? m`. Instead of generating an equation such as `(n =? m) = true`, which is generally not directly useful, this principle gives us right away the assumption we really need: `n = m`.

```

Inductive reflect (P : Prop) : bool → Prop :=
| ReflectT (H : P) : reflect P true
| ReflectF (H : ¬ P) : reflect P false.

```

The **reflect** property takes two arguments: a proposition  $P$  and a boolean  $b$ . Intuitively, it states that the property  $P$  is *reflected* in (i.e., equivalent to) the boolean  $b$ : that is,  $P$  holds if and only if  $b = \text{true}$ . To see this, notice that, by definition, the only way we can produce evidence for **reflect**  $P$  `true` is by showing  $P$  and then using the `ReflectT` constructor. If we invert this statement, this means that it should be possible to extract evidence for  $P$  from a proof of **reflect**  $P$  `true`. Similarly, the only way to show **reflect**  $P$  `false` is by combining evidence for  $\neg P$  with the `ReflectF` constructor.

It is easy to formalize this intuition and show that the statements  $P \leftrightarrow b = \text{true}$  and **reflect**  $P$   $b$  are indeed equivalent. First, the left-to-right implication:

```

Theorem iff_reflect : ∀ P b, (P ↔ b = true) → reflect P b.
Proof.

```

```

  intros P b H. destruct b.
  - apply ReflectT. rewrite H. reflexivity.
  - apply ReflectF. rewrite H. intros H'. discriminate.
Qed.

```

Now you prove the right-to-left implication:

**Exercise: 2 stars, standard, especially useful (reflect\_iff)** Theorem `reflect_iff` :  $\forall P b, \text{reflect } P b \rightarrow (P \leftrightarrow b = \text{true})$ .

Proof.

*Admitted.*

□

The advantage of **reflect** over the normal “if and only if” connective is that, by destructing a hypothesis or lemma of the form **reflect**  $P b$ , we can perform case analysis on  $b$  while at the same time generating appropriate hypothesis in the two branches ( $P$  in the first subgoal and  $\neg P$  in the second).

Lemma `eqbP` :  $\forall n m, \text{reflect } (n = m) (n =? m)$ .

Proof.

`intros n m. apply iff_reflect. rewrite eqb_eq. reflexivity.`

`Qed.`

A smoother proof of `filter_not_empty_ln` now goes as follows. Notice how the calls to `destruct` and `rewrite` are combined into a single call to `destruct`.

(To see this clearly, look at the two proofs of `filter_not_empty_ln` with Coq and observe the differences in proof state at the beginning of the first case of the `destruct`.)

Theorem `filter_not_empty_ln'` :  $\forall n l,$   
`filter (fun x  $\Rightarrow$   $n =? x$ ) l  $\neq$  []  $\rightarrow$`   
`ln n l.`

Proof.

`intros n l. induction l as [|m l' IHL'].`

-

`simpl. intros H. apply H. reflexivity.`

-

`simpl. destruct (eqbP n m) as [H | H].`

+

`intros _. rewrite H. left. reflexivity.`

+

`intros H'. right. apply IHL'. apply H'.`

`Qed.`

**Exercise: 3 stars, standard, especially useful (eqbP\_practice)** Use `eqbP` as above to prove the following:

Fixpoint `count n l` :=

`match l with`

`| []  $\Rightarrow$  0`

`| m :: l'  $\Rightarrow$  (if  $n =? m$  then 1 else 0) + count n l'`

`end.`

Theorem `eqbP_practice` :  $\forall n l,$

`count n l = 0  $\rightarrow$  ~ (ln n l).`

Proof.

*Admitted.*

□

This small example shows how reflection gives us a small gain in convenience; in larger developments, using **reflect** consistently can often lead to noticeably shorter and clearer proof scripts. We’ll see many more examples in later chapters and in *Programming Language Foundations*.

The use of the **reflect** property has been popularized by *SSReflect*, a Coq library that has been used to formalize important results in mathematics, including as the 4-color theorem and the Feit-Thompson theorem. The name *SSReflect* stands for *small-scale reflection*, i.e., the pervasive use of reflection to simplify small proof steps with boolean computations.

## 8.7 Additional Exercises

**Exercise: 3 stars, standard, especially useful (nostutter\_defn)** Formulating inductive definitions of properties is an important skill you’ll need in this course. Try to solve this exercise without any help at all.

We say that a list “stutters” if it repeats the same element consecutively. (This is different from not containing duplicates: the sequence `[1;4;1]` repeats the element 1 but does not stutter.) The property “**nostutter** mylist” means that mylist does not stutter. Formulate an inductive definition for **nostutter**.

Inductive **nostutter** {X:Type} : list X → Prop :=

.

Make sure each of these tests succeeds, but feel free to change the suggested proof (in comments) if the given one doesn’t work for you. Your definition might be different from ours and still be correct, in which case the examples might need a different proof. (You’ll notice that the suggested proofs use a number of tactics we haven’t talked about, to make them more robust to different possible ways of defining **nostutter**. You can probably just uncomment and use them as-is, but you can also prove each example with more basic tactics.)

Example test\_nostutter\_1: **nostutter** [3;1;4;1;5;6].

*Admitted.*

Example test\_nostutter\_2: **nostutter** (@nil **nat**).

*Admitted.*

Example test\_nostutter\_3: **nostutter** [5].

*Admitted.*

Example test\_nostutter\_4: **not** (**nostutter** [3;1;1;4]).

*Admitted.*

Definition manual\_grade\_for\_nostutter : **option** (**nat**×**string**) := **None**.

□

**Exercise: 4 stars, advanced (filter\_challenge)** Let’s prove that our definition of `filter` from the Poly chapter matches an abstract specification. Here is the specification, written out informally in English:

A list  $l$  is an “in-order merge” of  $l1$  and  $l2$  if it contains all the same elements as  $l1$  and  $l2$ , in the same order as  $l1$  and  $l2$ , but possibly interleaved. For example,

1;4;6;2;3

is an in-order merge of

1;6;2

and

4;3.

Now, suppose we have a set  $X$ , a function  $test: X \rightarrow \mathbf{bool}$ , and a list  $l$  of type  $\mathbf{list}\ X$ . 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  $\mathbf{filter}\ test\ l = l1$ .

Translate this specification into a Coq theorem and prove it. (You’ll need to begin by defining what it means for one list to be a merge of two others. Do this with an inductive relation, not a `Fixpoint`.)

Definition `manual_grade_for_filter_challenge` : **option** (**nat** $\times$ **string**) := **None**.

□

**Exercise: 5 stars, advanced, optional (filter\_challenge\_2)** A different way to characterize the behavior of `filter` goes like this: Among all subsequences of  $l$  with the property that  $test$  evaluates to `true` on all their members,  $\mathbf{filter}\ test\ l$  is the longest. Formalize this claim and prove it.

**Exercise: 4 stars, standard, optional (palindromes)** A palindrome is a sequence that reads the same backwards as forwards.

- Define an inductive proposition  $pal$  on  $\mathbf{list}\ X$  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 like

$c : \text{forall } l, l = \text{rev } l \rightarrow pal\ l$

may seem obvious, but will not work very well.)

- Prove ( $pal\_app\_rev$ ) that

$\text{forall } l, pal\ (l ++ \text{rev } l).$

- Prove ( $pal\_rev$ ) that

$\text{forall } l, pal\ l \rightarrow l = \text{rev } l.$

Definition `manual_grade_for_pal_pal_app_rev_pal_rev` : **option** (**nat** $\times$ **string**) := **None**.

□



**Exercise: 5 stars, standard, optional (palindrome\_converse)** Again, the converse direction is significantly more difficult, due to the lack of evidence. Using your definition of *pal* from the previous exercise, prove that

forall l, l = rev l -> pal l.

**Exercise: 4 stars, advanced, optional (NoDup)** Recall the definition of the *ln* property from the Logic chapter, which asserts that a value *x* appears at least once in a list *l*:

Your first task is to use *ln* to define a proposition *disjoint* *X l1 l2*, which should be provable exactly when *l1* and *l2* are lists (with elements of type *X*) that have no elements in common.

Next, use *ln* to define an inductive proposition *NoDup* *X l*, which should be provable exactly when *l* is a list (with elements of type *X*) where every member is different from every other. For example, *NoDup* **nat** [1;2;3;4] and *NoDup* **bool** [] should be provable, while *NoDup* **nat** [1;2;1] and *NoDup* **bool** [true;true] should not be.

Finally, state and prove one or more interesting theorems relating *disjoint*, *NoDup* and ++ (list append).

Definition manual\_grade\_for\_NoDup\_disjoint\_etc : option (nat×string) := None.

□

**Exercise: 4 stars, advanced, optional (pigeonhole\_principle)** The *pigeonhole principle* states a basic fact about counting: if we distribute more than *n* items into *n* pigeonholes, some pigeonhole must contain at least two items. As often happens, this apparently trivial fact about numbers requires non-trivial machinery to prove, but we now have enough...

First prove an easy useful lemma.

Lemma in\_split : ∀ (X:Type) (x:X) (l:list X),

ln x l →

∃ l1 l2, l = l1 ++ x :: l2.

Proof.

Admitted.

Now define a property **repeats** such that **repeats** *X l* asserts that *l* contains at least one repeated element (of type *X*).

Inductive repeats {X:Type} : list X → Prop :=

.

Definition manual\_grade\_for\_check\_repeats : option (nat×string) := None.

Now, here's a way to formalize the pigeonhole principle. Suppose list *l2* represents a list of pigeonhole labels, and list *l1* represents the labels assigned to a list of items. If there

are more items than labels, at least two items must have the same label – i.e., list  $l1$  must contain repeats.

This proof is much easier if you use the `excluded_middle` hypothesis to show that `ln` is decidable, i.e.,  $\forall x\ l, (\ln\ x\ l) \vee \neg (\ln\ x\ l)$ . However, it is also possible to make the proof go through *without* assuming that `ln` is decidable; if you manage to do this, you will not need the `excluded_middle` hypothesis.

**Theorem** `pigeonhole_principle`:  $\forall (X:\text{Type}) (l1\ l2:\text{list } X),$   
`excluded_middle`  $\rightarrow$   
 $(\forall x, \ln\ x\ l1 \rightarrow \ln\ x\ l2) \rightarrow$   
 $\text{length } l2 < \text{length } l1 \rightarrow$   
`repeats`  $l1$ .

**Proof.**

`intros X l1. induction l1 as [|x l' IHl1'].`  
*Admitted.*

□

### 8.7.1 Extended Exercise: A Verified Regular-Expression Matcher

We have now defined a match relation over regular expressions and polymorphic lists. We can use such a definition to manually prove that a given regex matches a given string, but it does not give us a program that we can run to determine a match automatically.

It would be reasonable to hope that we can translate the definitions of the inductive rules for constructing evidence of the match relation into cases of a recursive function that reflects the relation by recursing on a given regex. However, it does not seem straightforward to define such a function in which the given regex is a recursion variable recognized by Coq. As a result, Coq will not accept that the function always terminates.

Heavily-optimized regex matchers match a regex by translating a given regex into a state machine and determining if the state machine accepts a given string. However, regex matching can also be implemented using an algorithm that operates purely on strings and regexes without defining and maintaining additional datatypes, such as state machines. We'll implement such an algorithm, and verify that its value reflects the match relation.

We will implement a regex matcher that matches strings represented as lists of ASCII characters: `Require Import Coq.Strings.Ascii.`

**Definition** `string` := `list ascii`.

The Coq standard library contains a distinct inductive definition of strings of ASCII characters. However, we will use the above definition of strings as lists of ASCII characters in order to apply the existing definition of the match relation.

We could also define a regex matcher over polymorphic lists, not lists of ASCII characters specifically. The matching algorithm that we will implement needs to be able to test equality of elements in a given list, and thus needs to be given an equality-testing function. Generalizing the definitions, theorems, and proofs that we define for such a setting is a bit tedious, but workable.

The proof of correctness of the regex matcher will combine properties of the regex-matching function with properties of the `match` relation that do not depend on the matching function. We'll go ahead and prove the latter class of properties now. Most of them have straightforward proofs, which have been given to you, although there are a few key lemmas that are left for you to prove.

Each provable `Prop` is equivalent to **True**. Lemma `provable_equiv_true` :  $\forall (P : \text{Prop}), P \rightarrow (P \leftrightarrow \text{True})$ .

Proof.

```
intros.
split.
- intros. constructor.
- intros .. apply H.
```

Qed.

Each `Prop` whose negation is provable is equivalent to **False**. Lemma `not_equiv_false` :  $\forall (P : \text{Prop}), \neg P \rightarrow (P \leftrightarrow \text{False})$ .

Proof.

```
intros.
split.
- apply H.
- intros. destruct H0.
```

Qed.

`EmptySet` matches no string. Lemma `null_matches_none` :  $\forall (s : \text{string}), (s =^{\sim} \text{EmptySet}) \leftrightarrow \text{False}$ .

Proof.

```
intros.
apply not_equiv_false.
unfold not. intros. inversion H.
```

Qed.

`EmptyStr` only matches the empty string. Lemma `empty_matches_eps` :  $\forall (s : \text{string}), s =^{\sim} \text{EmptyStr} \leftrightarrow s = []$ .

Proof.

```
split.
- intros. inversion H. reflexivity.
- intros. rewrite H. apply MEmpty.
```

Qed.

`EmptyStr` matches no non-empty string. Lemma `empty_nomatch_ne` :  $\forall (a : \text{ascii}) s, (a :: s =^{\sim} \text{EmptyStr}) \leftrightarrow \text{False}$ .

Proof.

```
intros.
apply not_equiv_false.
unfold not. intros. inversion H.
```

Qed.

Char *a* matches no string that starts with a non-*a* character. Lemma `char_nomatch_char` :

$\forall (a\ b : \text{ascii})\ s,\ b \neq a \rightarrow (b :: s =^{\sim} \text{Char } a \leftrightarrow \text{False})$ .

Proof.

```
intros.
apply not_equiv_false.
unfold not.
intros.
apply H.
inversion H0.
reflexivity.
```

Qed.

If Char *a* matches a non-empty string, then the string's tail is empty. Lemma `char_eps_suffix` :  $\forall (a : \text{ascii})\ s,\ a :: s =^{\sim} \text{Char } a \leftrightarrow s = []$ .

Proof.

```
split.
- intros. inversion H. reflexivity.
- intros. rewrite H. apply MChar.
```

Qed.

App *re0 re1* matches string *s* iff  $s = s0 ++ s1$ , where *s0* matches *re0* and *s1* matches *re1*. Lemma `app_exists` :  $\forall (s : \text{string})\ re0\ re1,$

$s =^{\sim} \text{App } re0\ re1 \leftrightarrow$   
 $\exists s0\ s1,\ s = s0 ++ s1 \wedge s0 =^{\sim} re0 \wedge s1 =^{\sim} re1$ .

Proof.

```
intros.
split.
- intros. inversion H.  $\exists s1, s2$ . split.
   $\times$  reflexivity.
   $\times$  split. apply H3. apply H4.
- intros [ s0 | s1 | Happ | Hmat0 Hmat1 | ] | ].
  rewrite Happ. apply (MApp s0 _ s1 _ Hmat0 Hmat1).
```

Qed.

**Exercise: 3 stars, standard, optional (`app_ne`)** App *re0 re1* matches *a::s* iff *re0* matches the empty string and *a::s* matches *re1* or  $s = s0 ++ s1$ , where *a::s0* matches *re0* and *s1* matches *re1*.

Even though this is a property of purely the match relation, it is a critical observation behind the design of our regex matcher. So (1) take time to understand it, (2) prove it, and (3) look for how you'll use it later. Lemma `app_ne` :  $\forall (a : \text{ascii})\ s\ re0\ re1,$

$a :: s =^{\sim} (\text{App } re0\ re1) \leftrightarrow$   
 $([] =^{\sim} re0 \wedge a :: s =^{\sim} re1) \vee$   
 $\exists s0\ s1,\ s = s0 ++ s1 \wedge a :: s0 =^{\sim} re0 \wedge s1 =^{\sim} re1$ .

Proof.

*Admitted.*

□

$s$  matches `Union re0 re1` iff  $s$  matches  $re0$  or  $s$  matches  $re1$ . Lemma `union_disj` :  $\forall (s : \text{string}) re0 re1$ ,

$$s \sim \text{Union } re0 \ re1 \leftrightarrow s \sim re0 \vee s \sim re1.$$

Proof.

`intros. split.`

`- intros. inversion H.`

`+ left. apply H2.`

`+ right. apply H1.`

`- intros [ H | H ].`

`+ apply MUnionL. apply H.`

`+ apply MUnionR. apply H.`

Qed.

**Exercise: 3 stars, standard, optional (star\_ne)**  $a :: s$  matches `Star re` iff  $s = s0 ++ s1$ , where  $a :: s0$  matches  $re$  and  $s1$  matches `Star re`. Like `app_ne`, this observation is critical, so understand it, prove it, and keep it in mind.

Hint: you'll need to perform induction. There are quite a few reasonable candidates for `Prop`'s to prove by induction. The only one that will work is splitting the iff into two implications and proving one by induction on the evidence for  $a :: s \sim \text{Star } re$ . The other implication can be proved without induction.

In order to prove the right property by induction, you'll need to rephrase  $a :: s \sim \text{Star } re$  to be a `Prop` over general variables, using the *remember* tactic.

Lemma `star_ne` :  $\forall (a : \text{ascii}) s re$ ,

$$a :: s \sim \text{Star } re \leftrightarrow$$

$$\exists s0 s1, s = s0 ++ s1 \wedge a :: s0 \sim re \wedge s1 \sim \text{Star } re.$$

Proof.

*Admitted.*

□

The definition of our regex matcher will include two fixpoint functions. The first function, given regex  $re$ , will evaluate to a value that reflects whether  $re$  matches the empty string. The function will satisfy the following property: Definition `refl_matches_eps`  $m :=$

$$\forall re : \text{reg\_exp } \text{ascii}, \text{reflect } ([ ] \sim re) (m \ re).$$

**Exercise: 2 stars, standard, optional (match\_eps)** Complete the definition of `match_eps` so that it tests if a given regex matches the empty string: Fixpoint `match_eps` ( $re : \text{reg\_exp } \text{ascii}$ ) : `bool`

*. Admitted.*

□

**Exercise: 3 stars, standard, optional (match\_eps\_refl)** Now, prove that `match_eps` indeed tests if a given regex matches the empty string. (Hint: You'll want to use the reflection lemmas `ReflectT` and `ReflectF`.) Lemma `match_eps_refl : refl_matches_eps match_eps`.

Proof.

*Admitted.*

□

We'll define other functions that use `match_eps`. However, the only property of `match_eps` that you'll need to use in all proofs over these functions is `match_eps_refl`.

The key operation that will be performed by our regex matcher will be to iteratively construct a sequence of regex derivatives. For each character `a` and regex `re`, the derivative of `re` on `a` is a regex that matches all suffixes of strings matched by `re` that start with `a`. I.e., `re'` is a derivative of `re` on `a` if they satisfy the following relation:

**Definition** `is_der re (a : ascii) re' :=`  
 $\forall s, a :: s =^{\sim} re \leftrightarrow s =^{\sim} re'$ .

A function `d` derives strings if, given character `a` and regex `re`, it evaluates to the derivative of `re` on `a`. I.e., `d` satisfies the following property: **Definition** `derives d :=  $\forall a re, is\_der\ re\ a\ (d\ a\ re)$` .

**Exercise: 3 stars, standard, optional (derive)** Define `derive` so that it derives strings. One natural implementation uses `match_eps` in some cases to determine if key regex's match the empty string. Fixpoint `derive (a : ascii) (re : reg_exp ascii) : reg_exp ascii`

. *Admitted.*

□

The `derive` function should pass the following tests. Each test establishes an equality between an expression that will be evaluated by our regex matcher and the final value that must be returned by the regex matcher. Each test is annotated with the match fact that it reflects. Example `c := ascii_of_nat 99`.

Example `d := ascii_of_nat 100`.

"c" =<sup>~</sup> EmptySet: Example `test_der0 : match_eps (derive c (EmptySet)) = false`.

Proof.

*Admitted.*

"c" =<sup>~</sup> Char c: Example `test_der1 : match_eps (derive c (Char c)) = true`.

Proof.

*Admitted.*

"c" =<sup>~</sup> Char d: Example `test_der2 : match_eps (derive c (Char d)) = false`.

Proof.

*Admitted.*

"c" =<sup>~</sup> App (Char c) EmptyStr: Example `test_der3 : match_eps (derive c (App (Char c) EmptyStr)) = true`.

Proof.

*Admitted.*

`"c" =~ App EmptyStr (Char c): Example test_der4 : match_eps (derive c (App EmptyStr (Char c))) = true.`

**Proof.**

*Admitted.*

`"c" =~ Star c: Example test_der5 : match_eps (derive c (Star (Char c))) = true.`

**Proof.**

*Admitted.*

`"cd" =~ App (Char c) (Char d): Example test_der6 :  
match_eps (derive d (derive c (App (Char c) (Char d)))) = true.`

**Proof.**

*Admitted.*

`"cd" =~ App (Char d) (Char c): Example test_der7 :  
match_eps (derive d (derive c (App (Char d) (Char c)))) = false.`

**Proof.**

*Admitted.*

**Exercise: 4 stars, standard, optional (derive\_corr)** Prove that `derive` in fact always derives strings.

Hint: one proof performs induction on *re*, although you'll need to carefully choose the property that you prove by induction by generalizing the appropriate terms.

Hint: if your definition of `derive` applies `match_eps` to a particular regex *re*, then a natural proof will apply `match_eps_refl` to *re* and destruct the result to generate cases with assumptions that the *re* does or does not match the empty string.

Hint: You can save quite a bit of work by using lemmas proved above. In particular, to prove many cases of the induction, you can rewrite a **Prop** over a complicated regex (e.g.,  $s = \sim \text{Union } re0 \ re1$ ) to a Boolean combination of **Prop**'s over simple regex's (e.g.,  $s = \sim re0 \vee s = \sim re1$ ) using lemmas given above that are logical equivalences. You can then reason about these **Prop**'s naturally using `intro` and `destruct`. **Lemma** `derive_corr` : derives *derive*.

**Proof.**

*Admitted.*

□

We'll define the regex matcher using `derive`. However, the only property of `derive` that you'll need to use in all proofs of properties of the matcher is `derive_corr`.

A function *m* matches regexes if, given string *s* and regex *re*, it evaluates to a value that reflects whether *s* is matched by *re*. I.e., *m* holds the following property: **Definition** `matches_regex m : Prop :=`

$\forall (s : \text{string}) \ re, \text{reflect } (s = \sim re) (m \ s \ re).$

**Exercise: 2 stars, standard, optional (regex\_match)** Complete the definition of `regex_match` so that it matches regexes. **Fixpoint** `regex_match (s : string) (re : reg_exp ascii) : bool`

. *Admitted.*

□

**Exercise: 3 stars, standard, optional (regex\_refl)** Finally, prove that `regex_match` in fact matches regexes.

Hint: if your definition of `regex_match` applies `match_eps` to regex  $re$ , then a natural proof applies `match_eps_refl` to  $re$  and destructs the result to generate cases in which you may assume that  $re$  does or does not match the empty string.

Hint: if your definition of `regex_match` applies `derive` to character  $x$  and regex  $re$ , then a natural proof applies `derive_corr` to  $x$  and  $re$  to prove that  $x :: s =^{\sim} re$  given  $s =^{\sim} \text{derive } x \text{ } re$ , and vice versa. **Theorem** `regex_refl : matches_regex regex_match`.

**Proof.**

*Admitted.*

□



# Chapter 9

## Library LF.Maps

### 9.1 Maps: Total and Partial Maps

*Maps* (or *dictionaries*) are ubiquitous data structures both generally and in the theory of programming languages in particular; we’re going to need them in many places in the coming chapters. They also make a nice case study using ideas we’ve seen in previous chapters, including building data structures out of higher-order functions (from `Basics` and `Poly`) and the use of reflection to streamline proofs (from `IndProp`).

We’ll define two flavors of maps: *total* maps, which include a “default” element to be returned when a key being looked up doesn’t exist, and *partial* maps, which return an **option** to indicate success or failure. The latter is defined in terms of the former, using `None` as the default element.

### 9.2 The Coq Standard Library

One small digression before we begin...

Unlike the chapters we have seen so far, this one does not `Require Import` the chapter before it (and, transitively, all the earlier chapters). Instead, in this chapter and from now, on we’re going to import the definitions and theorems we need directly from Coq’s standard library stuff. You should not notice much difference, though, because we’ve been careful to name our own definitions and theorems the same as their counterparts in the standard library, wherever they overlap.

```
From Coq Require Import Arith.Arith.  
From Coq Require Import Bool.Bool.  
Require Export Coq.Strings.String.  
From Coq Require Import Logic.FunctionalExtensionality.  
From Coq Require Import Lists.List.  
Import ListNotations.
```

Documentation for the standard library can be found at <https://coq.inria.fr/library/>.

The `Search` command is a good way to look for theorems involving objects of specific types. See `Lists` for a reminder of how to use it.

If you want to find out how or where a notation is defined, the `Locate` command is useful. For example, where is the natural addition operation defined in the standard library?

```
Locate "+".
```

There are several uses for that notation, but only one for naturals.

```
Print Init.Nat.add.
```

## 9.3 Identifiers

First, we need a type for the keys that we use to index into our maps. In *Lists.v* we introduced a fresh type `id` for a similar purpose; here and for the rest of *Software Foundations* we will use the `string` type from Coq’s standard library.

To compare strings, we define the function `eqb_string`, which internally uses the function `string_dec` from Coq’s string library.

```
Definition eqb_string (x y : string) : bool :=
  if string_dec x y then true else false.
```

(The function `string_dec` comes from Coq’s string library. If you check the result type of `string_dec`, you’ll see that it does not actually return a `bool`, but rather a type that looks like  $\{x = y\} + \{x \neq y\}$ , called a **sumbol**, which can be thought of as an “evidence-carrying boolean.” Formally, an element of  $\{x = y\} + \{x \neq y\}$  is either a proof that  $x$  and  $y$  are equal or a proof that they are unequal, together with a tag indicating which. But for present purposes you can think of it as just a fancy `bool`.)

Now we need a few basic properties of string equality... `Theorem eqb_string_refl :  $\forall s : \text{string}, \text{true} = \text{eqb\_string } s \text{ } s$ .`

`Proof.`

```
  intros s. unfold eqb_string.
  destruct (string_dec s s) as [Hs_eq | Hs_not_eq].
  - reflexivity.
  - destruct Hs_not_eq. reflexivity.
```

`Qed.`

Two strings are equal according to `eqb_string` iff they are equal according to `=`. So `=` is reflected in `eqb_string`, in the sense of “reflection” as discussed in `IndProp`.

```
Theorem eqb_string_true_iff :  $\forall x \ y : \text{string},$ 
  eqb_string x y = true  $\leftrightarrow$  x = y.
```

`Proof.`

```
  intros x y.
  unfold eqb_string.
  destruct (string_dec x y) as [Hs_eq | Hs_not_eq].
  - rewrite Hs_eq. split. reflexivity. reflexivity.
```

```

- split.
+ intros contra. discriminate contra.
+ intros H. exfalso. apply Hs_not_eq. apply H.
Qed.

```

Similarly:

Theorem eqb\_string\_false\_iff :  $\forall x y : \text{string},$   
 $\text{eqb\_string } x y = \text{false} \leftrightarrow x \neq y.$

Proof.

```

intros x y. rewrite ← eqb_string_true_iff.
rewrite not_true_iff_false. reflexivity. Qed.

```

This corollary follows just by rewriting:

Theorem false\_eqb\_string :  $\forall x y : \text{string},$   
 $x \neq y \rightarrow \text{eqb\_string } x y = \text{false}.$

Proof.

```

intros x y. rewrite eqb_string_false_iff.
intros H. apply H. Qed.

```

## 9.4 Total Maps

Our main job in this chapter will be to build a definition of partial maps that is similar in behavior to the one we saw in the Lists chapter, plus accompanying lemmas about its behavior.

This time around, though, we’re going to use *functions*, rather than lists of key-value pairs, to build maps. The advantage of this representation is that it offers a more *extensional* view of maps, where two maps that respond to queries in the same way will be represented as literally the same thing (the very same function), rather than just “equivalent” data structures. This, in turn, simplifies proofs that use maps.

We build partial maps in two steps. First, we define a type of *total maps* that return a default value when we look up a key that is not present in the map.

Definition total\_map (A : Type) :=  $\text{string} \rightarrow A.$

Intuitively, a total map over an element type  $A$  is just a function that can be used to look up strings, yielding  $A$ s.

The function `t_empty` yields an empty total map, given a default element; this map always returns the default element when applied to any string.

Definition t\_empty {A : Type} (v : A) : total\_map A :=  
 $(\text{fun } _ \Rightarrow v).$

More interesting is the `update` function, which (as before) takes a map  $m$ , a key  $x$ , and a value  $v$  and returns a new map that takes  $x$  to  $v$  and takes every other key to whatever  $m$  does.

```

Definition t_update {A : Type} (m : total_map A)
  (x : string) (v : A) :=
  fun x' => if eqb_string x x' then v else m x'.

```

This definition is a nice example of higher-order programming: `t_update` takes a *function* `m` and yields a new function `fun x' => ...` that behaves like the desired map.

For example, we can build a map taking strings to **bools**, where "foo" and "bar" are mapped to `true` and every other key is mapped to `false`, like this:

```

Definition examplemap :=
  t_update (t_update (t_empty false) "foo" true)
    "bar" true.

```

Next, let's introduce some new notations to facilitate working with maps.

First, we will use the following notation to create an empty total map with a default value. Notation `"'_ ' !-> v" := (t_empty v)`  
(at level 100, right associativity).

```

Example example_empty := (_ !-> false).

```

We then introduce a convenient notation for extending an existing map with some bindings. Notation `"x ' !-> v ';' m" := (t_update m x v)`  
(at level 100, *v* at *next level*, right associativity).

The `examplemap` above can now be defined as follows:

```

Definition examplemap' :=
  ( "bar" !-> true;
    "foo" !-> true;
    _ !-> false
  ).

```

This completes the definition of total maps. Note that we don't need to define a `find` operation because it is just function application!

```

Example update_example1 : examplemap' "baz" = false.

```

Proof. reflexivity. Qed.

```

Example update_example2 : examplemap' "foo" = true.

```

Proof. reflexivity. Qed.

```

Example update_example3 : examplemap' "quux" = false.

```

Proof. reflexivity. Qed.

```

Example update_example4 : examplemap' "bar" = true.

```

Proof. reflexivity. Qed.

To use maps in later chapters, we'll need several fundamental facts about how they behave.

Even if you don't work the following exercises, make sure you thoroughly understand the statements of the lemmas!

(Some of the proofs require the functional extensionality axiom, which is discussed in the Logic chapter.)

**Exercise: 1 star, standard, optional (t\_apply\_empty)** First, the empty map returns its default element for all keys:

Lemma t\_apply\_empty :  $\forall (A : \text{Type}) (x : \text{string}) (v : A),$   
 $(\_ \text{!-} \rightarrow v) \text{ x} = v.$

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard, optional (t\_update\_eq)** Next, if we update a map  $m$  at a key  $x$  with a new value  $v$  and then look up  $x$  in the map resulting from the `update`, we get back  $v$ :

Lemma t\_update\_eq :  $\forall (A : \text{Type}) (m : \text{total\_map } A) x v,$   
 $(x \text{!-} \rightarrow v ; m) \text{ x} = v.$

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard, optional (t\_update\_neq)** On the other hand, if we update a map  $m$  at a key  $x1$  and then look up a *different* key  $x2$  in the resulting map, we get the same result that  $m$  would have given:

Theorem t\_update\_neq :  $\forall (A : \text{Type}) (m : \text{total\_map } A) x1 x2 v,$   
 $x1 \neq x2 \rightarrow$   
 $(x1 \text{!-} \rightarrow v ; m) \text{ x2} = m \text{ x2}.$

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard, optional (t\_update\_shadow)** If we update a map  $m$  at a key  $x$  with a value  $v1$  and then update again with the same key  $x$  and another value  $v2$ , the resulting map behaves the same (gives the same result when applied to any key) as the simpler map obtained by performing just the second `update` on  $m$ :

Lemma t\_update\_shadow :  $\forall (A : \text{Type}) (m : \text{total\_map } A) x v1 v2,$   
 $(x \text{!-} \rightarrow v2 ; x \text{!-} \rightarrow v1 ; m) = (x \text{!-} \rightarrow v2 ; m).$

Proof.

*Admitted.*

□

For the final two lemmas about total maps, it's convenient to use the reflection idioms introduced in chapter `lndProp`. We begin by proving a fundamental *reflection lemma* relating the equality proposition on strings with the boolean function `eqb_string`.

**Exercise: 2 stars, standard, optional (`eqb_stringP`)** Use the proof of `eqbP` in chapter `lndProp` as a template to prove the following:

Lemma `eqb_stringP` :  $\forall x\ y : \text{string},$   
 $\text{reflect } (x = y) (\text{eqb\_string } x\ y).$

Proof.

*Admitted.*

□

Now, given strings `x1` and `x2`, we can use the tactic `destruct (eqb_stringP x1 x2)` to simultaneously perform case analysis on the result of `eqb_string x1 x2` and generate hypotheses about the equality (in the sense of `=`) of `x1` and `x2`.

**Exercise: 2 stars, standard (`t_update_same`)** With the example in chapter `lndProp` as a template, use `eqb_stringP` to prove the following theorem, which states that if we update a map to assign key `x` the same value as it already has in `m`, then the result is equal to `m`:

Theorem `t_update_same` :  $\forall (A : \text{Type}) (m : \text{total\_map } A) x,$   
 $(x \text{ !-> } m\ x ; m) = m.$

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard, especially useful (`t_update_permute`)** Use `eqb_stringP` to prove one final property of the `update` function: If we update a map `m` at two distinct keys, it doesn't matter in which order we do the updates.

Theorem `t_update_permute` :  $\forall (A : \text{Type}) (m : \text{total\_map } A)$   
 $v1\ v2\ x1\ x2,$

$x2 \neq x1 \rightarrow$   
 $(x1 \text{ !-> } v1 ; x2 \text{ !-> } v2 ; m)$   
 $=$   
 $(x2 \text{ !-> } v2 ; x1 \text{ !-> } v1 ; m).$

Proof.

*Admitted.*

□

## 9.5 Partial maps

Finally, we define *partial maps* on top of total maps. A partial map with elements of type `A` is simply a total map with elements of type `option A` and default element `None`.

Definition partial\_map (A : Type) := total\_map (option A).

Definition empty {A : Type} : partial\_map A :=  
t\_empty None.

Definition update {A : Type} (m : partial\_map A)  
(x : string) (v : A) :=  
(x !-> Some v ; m).

We introduce a similar notation for partial maps: Notation "x '|->' v ;' m" := (update m x v)

(at level 100, v at next level, right associativity).

We can also hide the last case when it is empty. Notation "x '|->' v" := (update empty x v)  
(at level 100).

Example examplemap :=  
("Church" |-> true ; "Turing" |-> false).

We now straightforwardly lift all of the basic lemmas about total maps to partial maps.

Lemma apply\_empty :  $\forall (A : \text{Type}) (x : \text{string}),$   
@empty A x = None.

Proof.

intros. unfold empty. rewrite t\_apply\_empty.  
reflexivity.

Qed.

Lemma update\_eq :  $\forall (A : \text{Type}) (m : \text{partial\_map } A) x v,$   
(x |-> v ; m) x = Some v.

Proof.

intros. unfold update. rewrite t\_update\_eq.  
reflexivity.

Qed.

Theorem update\_neq :  $\forall (A : \text{Type}) (m : \text{partial\_map } A) x1 x2 v,$   
 $x2 \neq x1 \rightarrow$   
(x2 |-> v ; m) x1 = m x1.

Proof.

intros A m x1 x2 v H.  
unfold update. rewrite t\_update\_neq. reflexivity.  
apply H. Qed.

Lemma update\_shadow :  $\forall (A : \text{Type}) (m : \text{partial\_map } A) x v1 v2,$   
(x |-> v2 ; x |-> v1 ; m) = (x |-> v2 ; m).

Proof.

intros A m x v1 v2. unfold update. rewrite t\_update\_shadow.  
reflexivity.

Qed.

Theorem update\_same :  $\forall (A : \text{Type}) (m : \text{partial\_map } A) x v,$   
 $m \ x = \text{Some } v \rightarrow$   
 $(x \mapsto v ; m) = m.$

Proof.

intros  $A \ m \ x \ v \ H$ . unfold update. rewrite  $\leftarrow H$ .  
 apply  $t\_update\_same$ .

Qed.

Theorem update\_permute :  $\forall (A : \text{Type}) (m : \text{partial\_map } A)$   
 $x1 \ x2 \ v1 \ v2,$

$x2 \neq x1 \rightarrow$   
 $(x1 \mapsto v1 ; x2 \mapsto v2 ; m) = (x2 \mapsto v2 ; x1 \mapsto v1 ; m).$

Proof.

intros  $A \ m \ x1 \ x2 \ v1 \ v2$ . unfold update.  
 apply  $t\_update\_permute$ .

Qed.

Finally, for partial maps we introduce a notion of map inclusion, stating that one map includes another:

Definition inclusion  $\{A : \text{Type}\} (m \ m' : \text{partial\_map } A) :=$   
 $\forall x \ v, m \ x = \text{Some } v \rightarrow m' \ x = \text{Some } v.$

We show that map update preserves map inclusion, that is:

Lemma inclusion\_update :  $\forall (A : \text{Type}) (m \ m' : \text{partial\_map } A)$   
 $x \ vx,$

inclusion  $m \ m' \rightarrow$   
 inclusion  $(x \mapsto vx ; m) (x \mapsto vx ; m').$

Proof.

unfold inclusion.  
 intros  $A \ m \ m' \ x \ vx \ H$ .  
 intros  $y \ vy$ .  
 destruct (eqb\_stringP  $x \ y$ ) as [ $Hxy \mid Hxy$ ].  
 - rewrite  $Hxy$ .  
 rewrite update\_eq. rewrite update\_eq. intro  $H1$ . apply  $H1$ .  
 - rewrite update\_neq. rewrite update\_neq.  
 + apply  $H$ .  
 + apply  $Hxy$ .  
 + apply  $Hxy$ .

Qed.

This property is useful for reasoning about the lambda-calculus, where maps are used to keep track of which program variables are defined at a given point.



# Chapter 10

## Library `LF.ProofObjects`

### 10.1 ProofObjects: The Curry-Howard Correspondence

`Set Warnings "-notation-overridden,-parsing".`  
`From LF Require Export IndProp.`

“\_Algorithms are the computational content of proofs.” Robert Harper

We have seen that Coq has mechanisms both for *programming*, using inductive data types like `nat` or `list` and functions over these types, and for *proving* properties of these programs, using inductive propositions (like `ev`), implication, universal quantification, and the like. So far, we have mostly treated these mechanisms as if they were quite separate, and for many purposes this is a good way to think. But we have also seen hints that Coq’s programming and proving facilities are closely related. For example, the keyword `Inductive` is used to declare both data types and propositions, and  $\rightarrow$  is used both to describe the type of functions on data and logical implication. This is not just a syntactic accident! In fact, programs and proofs in Coq are almost the same thing. In this chapter we will study how this works.

We have already seen the fundamental idea: provability in Coq is represented by concrete *evidence*. When we construct the proof of a basic proposition, we are actually building a tree of evidence, which can be thought of as a data structure.

If the proposition is an implication like  $A \rightarrow B$ , then its proof will be an evidence *transformer*: a recipe for converting evidence for A into evidence for B. So at a fundamental level, proofs are simply programs that manipulate evidence.

Question: If evidence is data, what are propositions themselves?

Answer: They are types!

Look again at the formal definition of the `ev` property.

`Print ev.`

Suppose we introduce an alternative pronunciation of “:”. Instead of “has type,” we can say “is a proof of.” For example, the second line in the definition of `ev` declares that `ev_0 : ev 0`. Instead of “`ev_0` has type `ev 0`,” we can say that “`ev_0` is a proof of `ev 0`.”

This pun between types and propositions – between `:` as “has type” and `:` as “is a proof of” or “is evidence for” – is called the *Curry-Howard correspondence*. It proposes a deep connection between the world of logic and the world of computation:

propositions  $\sim$  types   proofs  $\sim$  data values

See *Wadler 2015* (in *Bib.v*) for a brief history and up-to-date exposition.

Many useful insights follow from this connection. To begin with, it gives us a natural interpretation of the type of the `ev_SS` constructor:

**Check** `ev_SS`

```
: ∀ n,
  ev n →
  ev (S (S n)).
```

This can be read “`ev_SS` is a constructor that takes two arguments – a number  $n$  and evidence for the proposition `ev n` – and yields evidence for the proposition `ev (S (S n))`.”

Now let’s look again at a previous proof involving `ev`.

**Theorem** `ev_4` : `ev 4`.

**Proof.**

```
apply ev_SS. apply ev_SS. apply ev_0. Qed.
```

As with ordinary data values and functions, we can use the `Print` command to see the *proof object* that results from this proof script.

**Print** `ev_4`.

Indeed, we can also write down this proof object *directly*, without the need for a separate proof script:

```
Check (ev_SS 2 (ev_SS 0 ev_0))
: ev 4.
```

The expression `ev_SS 2 (ev_SS 0 ev_0)` can be thought of as instantiating the parameterized constructor `ev_SS` with the specific arguments 2 and 0 plus the corresponding proof objects for its premises `ev 2` and `ev 0`. Alternatively, we can think of `ev_SS` as a primitive “evidence constructor” that, when applied to a particular number, wants to be further applied to evidence that this number is even; its type,

forall n, ev n -> ev (S (S n)),

expresses this functionality, in the same way that the polymorphic type  $\forall X, \text{list } X$  expresses the fact that the constructor `nil` can be thought of as a function from types to empty lists with elements of that type.

We saw in the *Logic* chapter that we can use function application syntax to instantiate universally quantified variables in lemmas, as well as to supply evidence for assumptions that these lemmas impose. For instance:

**Theorem** `ev_4'` : `ev 4`.

**Proof.**

```
apply (ev_SS 2 (ev_SS 0 ev_0)).
```

**Qed.**

## 10.2 Proof Scripts

The *proof objects* we’ve been discussing lie at the core of how Coq operates. When Coq is following a proof script, what is happening internally is that it is gradually constructing a proof object – a term whose type is the proposition being proved. The tactics between **Proof** and **Qed** tell it how to build up a term of the required type. To see this process in action, let’s use the **Show Proof** command to display the current state of the proof tree at various points in the following tactic proof.

**Theorem** ev\_4'' : **ev** 4.

**Proof.**

```
Show Proof.  
  apply ev_SS.  
Show Proof.  
  apply ev_SS.  
Show Proof.  
  apply ev_0.  
Show Proof.
```

**Qed.**

At any given moment, Coq has constructed a term with a “hole” (indicated by `?Goal` here, and so on), and it knows what type of evidence is needed to fill this hole.

Each hole corresponds to a subgoal, and the proof is finished when there are no more subgoals. At this point, the evidence we’ve built is stored in the global context under the name given in the **Theorem** command.

Tactic proofs are useful and convenient, but they are not essential: in principle, we can always construct the required evidence by hand, as shown above. Then we can use **Definition** (rather than **Theorem**) to give a global name directly to this evidence.

```
Definition ev_4''' : ev 4 :=  
  ev_SS 2 (ev_SS 0 ev_0).
```

All these different ways of building the proof lead to exactly the same evidence being saved in the global environment.

```
Print ev_4.  
Print ev_4'.  
Print ev_4''.  
Print ev_4'''.
```

**Exercise: 2 stars, standard (eight\_is\_even)** Give a tactic proof and a proof object showing that **ev** 8.

**Theorem** ev\_8 : **ev** 8.

**Proof.**

*Admitted.*

**Definition** ev\_8' : **ev** 8

. *Admitted.*

□

## 10.3 Quantifiers, Implications, Functions

In Coq’s computational universe (where data structures and programs live), there are two sorts of values that have arrows in their types: *constructors* introduced by **Inductively** defined data types, and *functions*.

Similarly, in Coq’s logical universe (where we carry out proofs), there are two ways of giving evidence for an implication: constructors introduced by **Inductively** defined propositions, and... functions!

For example, consider this statement:

Theorem `ev_plus4` :  $\forall n, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ (4 + n)$ .

Proof.

```
intros n H. simpl.
apply ev_SS.
apply ev_SS.
apply H.
```

Qed.

What is the proof object corresponding to `ev_plus4`?

We’re looking for an expression whose *type* is  $\forall n, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ (4 + n)$  – that is, a *function* that takes two arguments (one number and a piece of evidence) and returns a piece of evidence!

Here it is:

Definition `ev_plus4'` :  $\forall n, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ (4 + n) :=$

```
fun (n : nat) => fun (H : ev n) =>
  ev_SS (S (S n)) (ev_SS n H).
```

Recall that `fun n => blah` means “the function that, given *n*, yields *blah*,” and that Coq treats  $4 + n$  and  $S (S (S (S n)))$  as synonyms. Another equivalent way to write this definition is:

```
Definition ev_plus4'' (n : nat) (H : ev n)
  : ev (4 + n) :=
  ev_SS (S (S n)) (ev_SS n H).
```

Check `ev_plus4''`

```
:  $\forall n : \mathbf{nat},$ 
   $\mathbf{ev} \ n \rightarrow$ 
   $\mathbf{ev} \ (4 + n)$ .
```

When we view the proposition being proved by `ev_plus4` as a function type, one interesting point becomes apparent: The second argument’s type, `ev n`, mentions the *value* of the first argument, *n*.

While such *dependent types* are not found in conventional programming languages, they can be useful in programming too, as the recent flurry of activity in the functional programming community demonstrates.

Notice that both implication ( $\rightarrow$ ) and quantification ( $\forall$ ) correspond to functions on evidence. In fact, they are really the same thing:  $\rightarrow$  is just a shorthand for a degenerate use of  $\forall$  where there is no dependency, i.e., no need to give a name to the type on the left-hand side of the arrow:

forall (x:nat), nat = forall (\_:nat), nat = nat -> nat

For example, consider this proposition:

**Definition** ev\_plus2 : Prop :=  
 $\forall n, \forall (E : \mathbf{ev} \ n), \mathbf{ev} \ (n + 2).$

A proof term inhabiting this proposition would be a function with two arguments: a number  $n$  and some evidence  $E$  that  $n$  is even. But the name  $E$  for this evidence is not used in the rest of the statement of `ev_plus2`, so it's a bit silly to bother making up a name for it. We could write it like this instead, using the dummy identifier `_` in place of a real name:

**Definition** ev\_plus2' : Prop :=  
 $\forall n, \forall (_ : \mathbf{ev} \ n), \mathbf{ev} \ (n + 2).$

Or, equivalently, we can write it in a more familiar way:

**Definition** ev\_plus2'' : Prop :=  
 $\forall n, \mathbf{ev} \ n \rightarrow \mathbf{ev} \ (n + 2).$

In general, “ $P \rightarrow Q$ ” is just syntactic sugar for “ $\forall (_:P), Q$ ”.

## 10.4 Programming with Tactics

If we can build proofs by giving explicit terms rather than executing tactic scripts, you may be wondering whether we can build *programs* using *tactics* rather than explicit terms. Naturally, the answer is yes!

**Definition** add1 : **nat**  $\rightarrow$  **nat**.

intro  $n$ .

Show Proof.

apply **S**.

Show Proof.

apply  $n$ . Defined.

Print add1.

Compute add1 2.

Notice that we terminate the **Definition** with a `.` rather than with `:=` followed by a term. This tells Coq to enter *proof scripting mode* to build an object of type **nat**  $\rightarrow$  **nat**. Also, we terminate the proof with **Defined** rather than **Qed**; this makes the definition *transparent*

so that it can be used in computation like a normally-defined function. (**Qed**-defined objects are opaque during computation.)

This feature is mainly useful for writing functions with dependent types, which we won't explore much further in this book. But it does illustrate the uniformity and orthogonality of the basic ideas in Coq.

## 10.5 Logical Connectives as Inductive Types

Inductive definitions are powerful enough to express most of the connectives we have seen so far. Indeed, only universal quantification (with implication as a special case) is built into Coq; all the others are defined inductively. We'll see these definitions in this section.

Module PROPS.

### 10.5.1 Conjunction

To prove that  $P \wedge Q$  holds, we must present evidence for both  $P$  and  $Q$ . Thus, it makes sense to define a proof object for  $P \wedge Q$  as consisting of a pair of two proofs: one for  $P$  and another one for  $Q$ . This leads to the following definition.

Module AND.

```
Inductive and (P Q : Prop) : Prop :=
| conj : P → Q → and P Q.
```

*Arguments* conj [P] [Q].

Notation "P ∧ Q" := (and P Q) : type\_scope.

Notice the similarity with the definition of the **prod** type, given in chapter Poly; the only difference is that **prod** takes **Type** arguments, whereas **and** takes **Prop** arguments.

Print prod.

This similarity should clarify why **destruct** and **intros** patterns can be used on a conjunctive hypothesis. Case analysis allows us to consider all possible ways in which  $P \wedge Q$  was proved – here just one (the **conj** constructor).

```
Theorem proj1' : ∀ P Q,
  P ∧ Q → P.
```

Proof.

```
  intros P Q HPQ. destruct HPQ as [HP HQ]. apply HP.
```

```
  Show Proof.
```

Qed.

Similarly, the **split** tactic actually works for any inductively defined proposition with exactly one constructor. In particular, it works for **and**:

```
Lemma and_comm : ∀ P Q : Prop, P ∧ Q ↔ Q ∧ P.
```

Proof.

```

intros P Q. split.
- intros [HP HQ]. split.
  + apply HQ.
  + apply HP.
- intros [HQ HP]. split.
  + apply HP.
  + apply HQ.
Qed.
End AND.

```

This shows why the inductive definition of **and** can be manipulated by tactics as we've been doing. We can also use it to build proofs directly, using pattern-matching. For instance:

```

Definition and_comm'_aux P Q (H : P ∧ Q) : Q ∧ P :=
  match H with
  | conj HP HQ ⇒ conj HQ HP
  end.

```

```

Definition and_comm' P Q : P ∧ Q ↔ Q ∧ P :=
  conj (and_comm'_aux P Q) (and_comm'_aux Q P).

```

**Exercise: 2 stars, standard (conj\_fact)** Construct a proof object for the following proposition.

```

Definition conj_fact : ∀ P Q R, P ∧ Q → Q ∧ R → P ∧ R
. Admitted.
□

```

## 10.5.2 Disjunction

The inductive definition of disjunction uses two constructors, one for each side of the disjunct:

Module OR.

```

Inductive or (P Q : Prop) : Prop :=
| or_introl : P → or P Q
| or_intror : Q → or P Q.

```

*Arguments* or\_introl [P] [Q].

*Arguments* or\_intror [P] [Q].

**Notation** "P ∨ Q" := (or P Q) : *type\_scope*.

This declaration explains the behavior of the **destruct** tactic on a disjunctive hypothesis, since the generated subgoals match the shape of the **or\_introl** and **or\_intror** constructors.

Once again, we can also directly write proof objects for theorems involving **or**, without resorting to tactics.

Definition inj\_l :  $\forall (P Q : \text{Prop}), P \rightarrow P \vee Q :=$   
 fun P Q HP  $\Rightarrow$  or\_introl HP.

Theorem inj\_l' :  $\forall (P Q : \text{Prop}), P \rightarrow P \vee Q$ .

Proof.

intros P Q HP. left. apply HP.

Qed.

Definition or\_elim :  $\forall (P Q R : \text{Prop}), (P \vee Q) \rightarrow (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R :=$   
 fun P Q R HPQ HPR HQR  $\Rightarrow$   
 match HPQ with  
 | or\_introl HP  $\Rightarrow$  HPR HP  
 | or\_intror HQ  $\Rightarrow$  HQR HQ  
 end.

Theorem or\_elim' :  $\forall (P Q R : \text{Prop}), (P \vee Q) \rightarrow (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R$ .

Proof.

intros P Q R HPQ HPR HQR.

destruct HPQ as [HP | HQ].

- apply HPR. apply HP.

- apply HQR. apply HQ.

Qed.

End OR.

**Exercise: 2 stars, standard (or\_commut')** Construct a proof object for the following proposition.

Definition or\_commut' :  $\forall P Q, P \vee Q \rightarrow Q \vee P$   
 . Admitted.  
 □

### 10.5.3 Existential Quantification

To give evidence for an existential quantifier, we package a witness  $x$  together with a proof that  $x$  satisfies the property  $P$ :

Module EX.

Inductive ex {A : Type} (P : A  $\rightarrow$  Prop) : Prop :=  
 | ex\_intro :  $\forall x : A, P x \rightarrow$  ex P.

Notation "'exists' x , p" :=

(ex (fun x  $\Rightarrow$  p))

(at level 200, right associativity) : type\_scope.

End EX.

This may benefit from a little unpacking. The core definition is for a type former **ex** that can be used to build propositions of the form **ex**  $P$ , where  $P$  itself is a *function* from



witness values in the type  $A$  to propositions. The `ex_intro` constructor then offers a way of constructing evidence for `ex P`, given a witness  $x$  and a proof of  $P x$ .

The notation in the standard library is a slight variant of the above, enabling syntactic forms such as  $\exists x y, P x y$ .

The more familiar form  $\exists x, P x$  desugars to an expression involving `ex`:

Check `ex (fun n => ev n) : Prop`.

Here's how to define an explicit proof object involving `ex`:

```
Definition some_nat_is_even :  $\exists n, \text{ev } n :=$ 
  ex_intro ev 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

**Exercise: 2 stars, standard (ex\_ev\_Sn)** Construct a proof object for the following proposition.

```
Definition ex_ev_Sn : ex (fun n => ev (S n))
  . Admitted.
□
```

## 10.5.4 True and False

The inductive definition of the **True** proposition is simple:

```
Inductive True : Prop :=
  | I : True.
```

It has one constructor (so every proof of **True** is the same, so being given a proof of **True** is not informative.)

**Exercise: 1 star, standard (p\_implies\_true)** Construct a proof object for the following proposition.

```
Definition p_implies_true :  $\forall P, P \rightarrow \text{True}$ 
  . Admitted.
□
```

**False** is equally simple – indeed, so simple it may look syntactically wrong at first glance!

```
Inductive False : Prop := .
```

That is, **False** is an inductive type with *no* constructors – i.e., no way to build evidence for it. For example, there is no way to complete the following definition such that it succeeds (rather than fails).

```
Fail Definition contra : False :=
  0 = 1.
```

But it is possible to destruct **False** by pattern matching. There can be no patterns that match it, since it has no constructors. So the pattern match also is so simple it may look syntactically wrong at first glance.

```

Definition false_implies_zero_eq_one : False → 0 = 1 :=
  fun contra ⇒ match contra with end.

```

Since there are no branches to evaluate, the **match** expression can be considered to have any type we want, including  $0 = 1$ . Indeed, it's impossible to ever cause the **match** to be evaluated, because we can never construct a value of type **False** to pass to the function.

**Exercise: 1 star, standard (ex\_falso\_quodlibet')** Construct a proof object for the following proposition.

```

Definition ex_falso_quodlibet' : ∀ P, False → P
. Admitted.
□

```

End PROPS.

## 10.6 Equality

Even Coq's equality relation is not built in. We can define it ourselves:

```

Module MYEQUALITY.

```

```

Inductive eq {X:Type} : X → X → Prop :=
| eq_refl : ∀ x, eq x x.

```

```

Notation "x == y" := (eq x y)
(at level 70, no associativity)
: type_scope.

```

The way to think about this definition (which is just a slight variant of the standard library's) 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 **eq\_refl** to a type  $X$  and a single value  $x : X$ , which yields evidence that  $x$  is equal to  $x$ .

Other types of the form **eq**  $x$   $y$  where  $x$  and  $y$  are not the same are thus uninhabited.

We can use **eq\_refl** to construct evidence that, for example,  $2 = 2$ . Can we also use it to construct evidence that  $1 + 1 = 2$ ? Yes, we can. Indeed, it is the very same piece of evidence!

The reason is that Coq treats as “the same” any two terms that are *convertible* according to a simple set of computation rules.

These rules, which are similar to those used by **Compute**, include evaluation of function application, inlining of definitions, and simplification of **matches**.

Lemma four:  $2 + 2 == 1 + 3$ .

Proof.

apply eq\_refl.

Qed.

The `reflexivity` tactic that we have used to prove equalities up to now is essentially just shorthand for `apply eq_refl`.

In tactic-based proofs of equality, the conversion rules are normally hidden in uses of `simpl` (either explicit or implicit in other tactics such as `reflexivity`).

But you can see them directly at work in the following explicit proof objects:

```
Definition four' : 2 + 2 == 1 + 3 :=
  eq_refl 4.
```

```
Definition singleton : ∀ (X:Type) (x:X), [] ++ [x] == x :: [] :=
  fun (X:Type) (x:X) => eq_refl [x].
```

**Exercise: 2 stars, standard (equality\_\_leibniz\_equality)** The inductive definition of equality implies *Leibniz equality*: what we mean when we say “ $x$  and  $y$  are equal” is that every property on  $P$  that is true of  $x$  is also true of  $y$ .

```
Lemma equality__leibniz_equality : ∀ (X : Type) (x y: X),
  x == y → ∀ P:X→Prop, P x → P y.
```

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard, optional (leibniz\_equality\_\_equality)** Show that, in fact, the inductive definition of equality is *equivalent* to Leibniz equality. Hint: the proof is quite short; about all you need to do is to invent a clever property  $P$  to instantiate the antecedent.

```
Lemma leibniz_equality__equality : ∀ (X : Type) (x y: X),
  (∀ P:X→Prop, P x → P y) → x == y.
```

Proof.

*Admitted.*

□

End MYEQUALITY.

## 10.6.1 Inversion, Again

We’ve seen `inversion` used with both equality hypotheses and hypotheses about inductively defined propositions. Now that we’ve seen that these are actually the same thing, we’re in a position to take a closer look at how `inversion` behaves.

In general, the `inversion` tactic...

- takes a hypothesis  $H$  whose type  $P$  is inductively defined, and
- for each constructor  $C$  in  $P$ ’s definition,

- generates a new subgoal in which we assume  $H$  was built with  $C$ ,
- adds the arguments (premises) of  $C$  to the context of the subgoal as extra hypotheses,
- matches the conclusion (result type) of  $C$  against the current goal and calculates a set of equalities that must hold in order for  $C$  to be applicable,
- adds these equalities to the context (and, for convenience, rewrites them in the goal), and
- if the equalities are not satisfiable (e.g., they involve things like  $S\ n = 0$ ), 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 **eq\_refl** 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.

## 10.7 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:

First, since Coq types can themselves be expressions, the checker must normalize these (by using the computation rules) before comparing them.

Second, 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:

```
Fail Definition or_bogus :  $\forall P Q, P \vee Q \rightarrow P :=$ 
  fun (P Q : Prop) (A : P  $\vee$  Q)  $\Rightarrow$ 
    match A with
    | or_introl H  $\Rightarrow$  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.

Third, the checker must make sure that each recursive function 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:

```
Fail Fixpoint infinite_loop {X : Type} (n : nat) {struct n} : X :=
  infinite_loop n.
```

```
Fail Definition falso : False := infinite_loop 0.
```

Recursive function `infinite_loop` purports to return a value of any type `X` that you would like. (The **struct** annotation on the function tells Coq that it recurses on argument `n`, not `X`.) Were Coq to allow `infinite_loop`, then `falso` would be definable, thus giving evidence for **False**. So Coq rejects `infinite_loop`.

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 theorems whose proofs pass the type-checker can be used in further proof developments.

# Chapter 11

## Library `LF.IndPrinciples`

### 11.1 `IndPrinciples`: Induction Principles

With the Curry-Howard correspondence and its realization in Coq in mind, we can now take a deeper look at induction principles.

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export ProofObjects.
```

### 11.2 Basics

Every time we declare a new `Inductive` datatype, Coq automatically generates an *induction principle* for this type. This induction principle is a theorem like any other: If  $t$  is defined inductively, the corresponding induction principle is called  $t\_ind$ . Here is the one for natural numbers:

```
Check nat_ind :
  ∀ P : nat → Prop,
    P 0 →
    (∀ n : nat, P n → P (S n)) →
    ∀ n : nat, P n.
```

In English: Suppose  $P$  is a property of natural numbers (that is,  $P\ n$  is a `Prop` for every  $n$ ). To show that  $P\ n$  holds of all  $n$ , it suffices to show:

- $P$  holds of 0
- for any  $n$ , if  $P$  holds of  $n$ , then  $P$  holds of  $S\ n$ .

The `induction` tactic is a straightforward wrapper that, at its core, simply performs `apply t_ind`. To see this more clearly, let's experiment with directly using `apply nat_ind`, instead of the `induction` tactic, to carry out some proofs. Here, for example, is an alternate proof of a theorem that we saw in the `Induction` chapter.

Theorem `mult_0_r'` :  $\forall n:\text{nat},$   
 $n \times 0 = 0.$

Proof.

```

  apply nat_ind.
- reflexivity.
- simpl. intros n' IHn'. rewrite → IHn'.
  reflexivity. Qed.

```

This proof is basically the same as the earlier one, but a few minor differences are worth noting.

First, in the induction step of the proof (the `S` case), we have to do a little bookkeeping manually (the `intros`) that `induction` does automatically.

Second, we do not introduce  $n$  into the context before applying `nat_ind` – the conclusion of `nat_ind` is a quantified formula, and `apply` needs this conclusion to exactly match the shape of the goal state, including the quantifier. By contrast, the `induction` tactic works either with a variable in the context or a quantified variable in the goal.

Third, we had to manually supply the name of the induction principle with `apply`, but `induction` figures that out itself.

These conveniences make `induction` nicer to use in practice than applying induction principles like `nat_ind` directly. But it is important to realize that, modulo these bits of bookkeeping, applying `nat_ind` is what we are really doing.

**Exercise: 2 stars, standard (`plus_one_r'`)** Complete this proof without using the `induction` tactic.

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

Proof.

*Admitted.*

□

Coq generates induction principles for every datatype defined with `Inductive`, including those that aren't recursive. Although of course we don't need the proof technique of induction to prove properties of non-recursive datatypes, the idea of an induction principle still makes sense for them: it gives a way to prove that a property holds for all values of the type.

These generated principles follow a similar pattern. If we define a type  $t$  with constructors `c1 ... cn`, Coq generates a theorem with this shape:

```

t_ind : forall P : t -> Prop, ... case for c1 ... -> ... case for c2 ... -> ... ... case for cn ...
-> forall n : t, P n

```

The specific shape of each case depends on the arguments to the corresponding constructor.

Before trying to write down a general rule, let's look at some more examples. First, an example where the constructors take no arguments:

```

Inductive time : Type :=
| day

```

```
| night.
```

```
Check time_ind :
```

```
  ∀ P : time → Prop,  
    P day →  
    P night →  
    ∀ t : time, P t.
```

**Exercise: 1 star, standard, optional (rgb)** Write out the induction principle that Coq will generate for the following datatype. Write down your answer on paper or type it into a comment, and then compare it with what Coq prints.

```
Inductive rgb : Type :=
```

```
  | red  
  | green  
  | blue.
```

```
Check rgb_ind.
```

```
□
```

Here's another example, this time with one of the constructors taking some arguments.

```
Inductive natlist : Type :=
```

```
  | nnil  
  | ncons (n : nat) (l : natlist).
```

```
Check natlist_ind :
```

```
  ∀ P : natlist → Prop,  
    P nnil →  
    (∀ (n : nat) (l : natlist),  
      P l → P (ncons n l)) →  
    ∀ l : natlist, P l.
```

In general, the automatically generated induction principle for inductive type  $t$  is formed as follows:

- Each constructor  $c$  generates one case of the principle.
- If  $c$  takes no arguments, that case is:  
“ $P$  holds of  $c$ ”
- If  $c$  takes arguments  $x_1:a_1 \dots x_n:a_n$ , that case is:  
“For all  $x_1:a_1 \dots x_n:a_n$ , if  $P$  holds of each of the arguments of type  $t$ , then  $P$  holds of  $c \ x_1 \dots x_n$ ”

But that oversimplifies a little. An assumption about  $P$  holding of an argument  $x$  of type  $t$  actually occurs immediately after the quantification of  $x$ .



For example, suppose we had written the definition of **natlist** a little differently:

```
Inductive natlist' : Type :=
| nnil'
| nsnoc (l : natlist') (n : nat).
```

Now the induction principle case for *nsnoc1* is a bit different than the earlier case for *ncons*:

```
Check natlist'_ind :
  ∀ P : natlist' → Prop,
    P nnil' →
    (∀ l : natlist', P l → ∀ n : nat, P (nsnoc l n)) →
    ∀ n : natlist', P n.
```

**Exercise: 1 star, standard (booltree\_ind)** In the comment below, Write out the induction principle that Coq will generate for the following datatype.

```
Inductive booltree : Type :=
| bt_empty
| bt_leaf (b : bool)
| bt_branch (b : bool) (t1 t2 : booltree).
```

Definition manual\_grade\_for\_booltree\_ind : option (nat×string) := None.

□

**Exercise: 1 star, standard (toy\_ind)** Here is an induction principle for a toy type:

forall P : Toy -> Prop, (forall b : bool, P (con1 b)) -> (forall (n : nat) (t : Toy), P t -> P (con2 n t)) -> forall t : Toy, P t

Give an **Inductive** definition of **Toy**, such that the induction principle Coq generates is that given above:

```
Inductive Toy : Type :=
```

.

Definition manual\_grade\_for\_toy\_ind : option (nat×string) := None.

□

## 11.3 Polymorphism

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),  $P$   $\square$  -> (forall ( $x$  :  $X$ ) ( $l$  : list  $X$ ),  $P$   $l$  ->  $P$  ( $x :: l$ )) -> forall  $l$  : list  $X$ ,  $P$   $l$

Note that the *whole* induction principle is parameterized on  $X$ . That is, *list\_ind* can be thought of as a polymorphic function that, when applied to a type  $X$ , gives us back an induction principle specialized to the type **list**  $X$ .

**Exercise: 1 star, standard, optional (tree)** Write out the induction principle that Coq will generate for the following datatype. Compare your answer with what Coq prints.

**Inductive tree** ( $X$ :Type) : Type :=

| leaf ( $x$  :  $X$ )  
| node ( $t1$   $t2$  : **tree**  $X$ ).

Check `tree_ind`.

$\square$

**Exercise: 1 star, standard, optional (mytype)** Find an inductive definition that gives rise to the following induction principle:

`mytype_ind` : forall ( $X$  : Type) ( $P$  : mytype  $X$  -> Prop), (forall  $x$  :  $X$ ,  $P$  (`constr1`  $X$   $x$ )) -> (forall  $n$  : nat,  $P$  (`constr2`  $X$   $n$ )) -> (forall  $m$  : mytype  $X$ ,  $P$   $m$  -> forall  $n$  : nat,  $P$  (`constr3`  $X$   $m$   $n$ )) -> forall  $m$  : mytype  $X$ ,  $P$   $m$   $\square$

**Exercise: 1 star, standard, 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  $X$   $Y$ ,  $P$   $f2$   $\square$

**Exercise: 1 star, standard, optional (foo')** Consider the following inductive definition:

**Inductive foo'** ( $X$ :Type) : Type :=

| C1 ( $l$  : list  $X$ ) ( $f$  : **foo'**  $X$ )  
| C2.

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'**  $X$ , -----  
 $\square$

## 11.4 Induction Hypotheses

Where does the phrase “induction hypothesis” fit into this story?

The induction principle for numbers

$\text{forall } P : \text{nat} \rightarrow \text{Prop}, P\ 0 \rightarrow (\text{forall } n : \text{nat}, P\ n \rightarrow P\ (S\ n)) \rightarrow \text{forall } n : \text{nat}, P\ n$

is a generic statement that holds for all propositions  $P$  (or rather, strictly speaking, for all families of propositions  $P$  indexed by a number  $n$ ). Each time we use this principle, we are choosing  $P$  to be a particular expression of type  $\text{nat} \rightarrow \text{Prop}$ .

We can make proofs by induction more explicit by giving this expression a name. For example, instead of stating the theorem `mult_0_r` as “ $\forall n, n \times 0 = 0$ ,” we can write it as “ $\forall n, P\_m0r\ n$ ”, where `P_m0r` is defined as...

**Definition** `P_m0r` ( $n:\text{nat}$ ) : `Prop` :=

$n \times 0 = 0$ .

... or equivalently:

**Definition** `P_m0r'` :  $\text{nat} \rightarrow \text{Prop}$  :=

`fun`  $n \Rightarrow n \times 0 = 0$ .

Now it is easier to see where `P_m0r` appears in the proof.

**Theorem** `mult_0_r''` :  $\forall n:\text{nat},$

`P_m0r`  $n$ .

**Proof.**

`apply` `nat_ind`.

- `reflexivity`.

-

`intros`  $n$  `IHn`.

`unfold` `P_m0r` `in` `IHn`. `unfold` `P_m0r`. `simpl`. `apply` `IHn`. `Qed`.

This extra naming step isn’t something that we do in normal proofs, but it is useful to do it explicitly for an example or two, because it allows us to see exactly what the induction hypothesis is. If we prove  $\forall n, P\_m0r\ n$  by induction on  $n$  (using either `induction` or `apply nat_ind`), we see that the first subgoal requires us to prove `P_m0r` 0 (“ $P$  holds for zero”), while the second subgoal requires us to prove  $\forall n', P\_m0r\ n' \rightarrow P\_m0r\ (S\ n')$  (that is “ $P$  holds of  $S\ n'$  if it holds of  $n'$ ” or, more elegantly, “ $P$  is preserved by  $S$ ”). The *induction hypothesis* is the premise of this latter implication – the assumption that  $P$  holds of  $n'$ , which we are allowed to use in proving that  $P$  holds for  $S\ n'$ .

## 11.5 More on the induction Tactic

The `induction` tactic actually does even more low-level bookkeeping for us than we discussed above.

Recall the informal statement of the induction principle for natural numbers:

- If  $P\ n$  is some proposition involving a natural number  $n$ , and we want to show that  $P$  holds for *all* numbers  $n$ , we can reason like this:
  - show that  $P\ 0$  holds
  - show that, if  $P\ n'$  holds, then so does  $P\ (S\ n')$
  - conclude that  $P\ n$  holds for all  $n$ .

So, when we begin a proof with `intros n` and then `induction n`, we are first telling Coq to consider a *particular*  $n$  (by introducing it into the context) and then telling it to prove something about *all* numbers (by using induction).

What Coq actually does in this situation, internally, is to “re-generalize” the variable we perform induction on. For example, in our original proof that `plus` is associative...

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

Proof.

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

It also works to apply `induction` to a variable that is quantified in the goal.

Theorem `plus_comm'` :  $\forall\ n\ m : \text{nat}$ ,  
 $n + m = m + n$ .

Proof.

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

Note that `induction n` leaves  $m$  still bound in the goal – i.e., what we are proving inductively is a statement beginning with  $\forall\ m$ .

If we do `induction` on a variable that is quantified in the goal *after* some other quantifiers, the `induction` tactic will automatically introduce the variables bound by these quantifiers into the context.

Theorem `plus_comm''` :  $\forall\ n\ m : \text{nat}$ ,  
 $n + m = m + n$ .

Proof.

```
induction m as [| m']. - simpl. rewrite ← plus_n_O. reflexivity.
- simpl. rewrite ← IHm'.
  rewrite ← plus_n_Sm. reflexivity. Qed.
```

**Exercise: 1 star, standard, optional (plus\_explicit\_prop)** Rewrite both `plus_assoc'` and `plus_comm'` and their proofs in the same style as `mult_0_r''` above – that is, for each theorem, give an explicit **Definition** of the proposition being proved by induction, and state the theorem and proof in terms of this defined proposition.

## 11.6 Induction Principles for Propositions

Inductive definitions of propositions also cause Coq to generate induction principles. For example, recall our proposition `ev`, repeated here as `ev''`:

```
Inductive ev'' : nat → Prop :=
| ev_0 : ev'' 0
| ev_SS (n : nat) : ev'' n → ev'' (S (S n)).

Check ev''_ind :
  ∀ P : nat → Prop,
    P 0 →
    (∀ n : nat, ev'' n → P n → P (S (S n))) →
    ∀ n : nat, ev'' n → P n.
```

In English, `ev''_ind` says: Suppose  $P$  is a property of natural numbers. To show that  $P$   $n$  holds whenever  $n$  is even, it suffices to show:

- $P$  holds for 0,
- for any  $n$ , if  $n$  is even and  $P$  holds for  $n$ , then  $P$  holds for  $S (S n)$ .

As expected, we can apply `ev''_ind` directly instead of using `induction`. For example, we can use it to show that `ev'` (the slightly awkward alternate definition of evenness that we saw in an exercise in the `\chap{IndProp}` chapter) is equivalent to the cleaner inductive definition `ev''`:

```
Inductive ev' : nat → Prop :=
| ev'_0 : ev' 0
| ev'_2 : ev' 2
| ev'_sum n m (Hn : ev' n) (Hm : ev' m) : ev' (n + m).
```

Theorem `ev''_ev' : ∀ n, ev'' n → ev' n`.

Proof.

```
  apply ev''_ind.
-
  apply ev'_0.
-
  intros m Hm IH.
  apply (ev'_sum 2 m).
  + apply ev'_2.
```

+ apply IH.

Qed.

The precise form of an `Inductive` definition can affect the induction principle Coq generates.

```
Inductive le1 : nat → nat → Prop :=
  | le1_n : ∀ n, le1 n n
  | le1_S : ∀ n m, (le1 n m) → (le1 n (S m)).
```

```
Notation "m <=1 n" := (le1 m n) (at level 70).
```

This definition can be streamlined a little by observing that the left-hand argument  $n$  is the same everywhere in the definition, so we can actually make it a “general parameter” to the whole definition, rather than an argument to each constructor.

```
Inductive le2 (n:nat) : nat → Prop :=
  | le2_n : le2 n n
  | le2_S m (H : le2 n m) : le2 n (S m).
```

```
Notation "m <=2 n" := (le2 m n) (at level 70).
```

The second one is better, even though it looks less symmetric. Why? Because it gives us a simpler induction principle.

Check le1\_ind :

```
∀ P : nat → nat → Prop,
  (∀ n : nat, P n n) →
  (∀ n m : nat, n <=1 m → P n m → P n (S m)) →
  ∀ n n0 : nat, n <=1 n0 → P n n0.
```

Check le2\_ind :

```
∀ (n : nat) (P : nat → Prop),
  P n →
  (∀ m : nat, n <=2 m → P m → P (S m)) →
  ∀ n0 : nat, n <=2 n0 → P n0.
```

## 11.7 Another Form of Induction Principles on Propositions (Optional)

The induction principle that Coq generated for `ev''` was parameterized on a natural number  $n$ . It could have additionally been parameterized on the evidence that  $n$  was even, which would have led to this induction principle:

```
forall P : (forall n : nat, ev'' n -> Prop), P O ev_0 -> (forall (m : nat) (E : ev'' m), P m
E -> P (S (S m)) (ev_SS m E)) -> forall (n : nat) (E : ev'' n), P n E
... because:
```

- Since `ev''` is indexed by a number  $n$  (every `ev''` object  $E$  is a piece of evidence that some particular number  $n$  is even), the proposition  $P$  is parameterized by both  $n$  and

$E$  – that is, the induction principle can be used to prove assertions involving both an even number and the evidence that it is even.

- Since there are two ways of giving evidence of evenness (`even` has two constructors), applying the induction principle generates two subgoals:
  - We must prove that  $P$  holds for `O` and `ev_0`.
  - We must prove that, whenever  $m$  is an even number and  $E$  is an evidence of its evenness, if  $P$  holds of  $m$  and  $E$ , then it also holds of `S (S m)` and `ev_SS m E`.
- If these subgoals can be proved, then the induction principle tells us that  $P$  is true for *all* even numbers  $n$  and evidence  $E$  of their evenness.

This is more flexibility than we normally need or want: it is giving us a way to prove logical assertions where the assertion involves properties of some piece of *evidence* of evenness, while all we really care about is proving properties of *numbers* that are even – we are interested in assertions about numbers, not about evidence. It would therefore be more convenient to have an induction principle for proving propositions  $P$  that are parameterized just by  $n$  and whose conclusion establishes  $P$  for all even numbers  $n$ :

forall P : nat -> Prop, ... -> forall n : nat, even n -> P n

That is why Coq actually generates the induction principle `ev''_ind` that we saw before.

## 11.8 Formal vs. Informal Proofs by Induction

Question: What is the relation between a formal proof of a proposition  $P$  and an informal proof of the same proposition  $P$ ?

Answer: The latter should *teach* the reader how to produce the former.

Question: How much detail is needed??

Unfortunately, there is no single right answer; rather, there is a range of choices.

At one end of the spectrum, we can essentially give the reader the whole formal proof (i.e., the “informal” proof will amount to just transcribing the formal one into words). This may give the reader the ability to reproduce the formal one for themselves, but it probably doesn’t *teach* them anything much.

At the other end of the spectrum, we can say “The theorem is true and you can figure out why for yourself if you think about it hard enough.” This is also not a good teaching strategy, because often writing the proof requires one or more significant insights into the thing we’re proving, and most readers will give up before they rediscover all the same insights as we did.

In the middle is the golden mean – a proof that includes all of the essential insights (saving the reader the hard work that we went through to find the proof in the first place) plus high-level suggestions for the more routine parts to save the reader from spending too much time reconstructing these (e.g., what the IH says and what must be shown in each case of an inductive proof), but not so much detail that the main ideas are obscured.

Since we've spent much of this chapter looking “under the hood” at formal proofs by induction, now is a good moment to talk a little about *informal* proofs by induction.

In the real world of mathematical communication, written proofs range from extremely longwinded and pedantic to extremely brief and telegraphic. Although the ideal is somewhere in between, while one is getting used to the style it is better to start out at the pedantic end. Also, during the learning phase, it is probably helpful to have a clear standard to compare against. With this in mind, we offer two templates – one for proofs by induction over *data* (i.e., where the thing we're doing induction on lives in **Type**) and one for proofs by induction over *evidence* (i.e., where the inductively defined thing lives in **Prop**).

### 11.8.1 Induction Over an Inductively Defined Set

*Template:*

- *Theorem:* <Universally quantified proposition of the form “For all  $n:S$ ,  $P(n)$ ,” where  $S$  is some inductively defined set.>

*Proof:* By induction on  $n$ .

<one case for each constructor  $c$  of  $S$ ...>

- Suppose  $n = c \ a1 \ \dots \ ak$ , where <...and here we state the IH for each of the  $a$ 's that has type  $S$ , if any>. We must show <...and here we restate  $P(c \ a1 \ \dots \ ak)$ >. <go on and prove  $P(n)$  to finish the case...>
- <other cases similarly...>  $\square$

*Example:*

- *Theorem:* For all sets  $X$ , lists  $l : \text{list } X$ , and numbers  $n$ , if  $\text{length } l = n$  then  $\text{index } (S \ n) \ l = \text{None}$ .

*Proof:* By induction on  $l$ .

- Suppose  $l = []$ . We must show, for all numbers  $n$ , that, if  $\text{length } [] = n$ , then  $\text{index } (S \ n) \ [] = \text{None}$ .

This follows immediately from the definition of *index*.

- Suppose  $l = x :: l'$  for some  $x$  and  $l'$ , where  $\text{length } l' = n'$  implies  $\text{index } (S \ n') \ l' = \text{None}$ , for any number  $n'$ . We must show, for all  $n$ , that, if  $\text{length } (x::l') = n$  then  $\text{index } (S \ n) \ (x::l') = \text{None}$ .

Let  $n$  be a number with  $\text{length } l = n$ . Since

$\text{length } l = \text{length } (x::l') = S \ (\text{length } l')$ ,

it suffices to show that

$\text{index } (S \ (\text{length } l')) \ l' = \text{None}$ .

But this follows directly from the induction hypothesis, picking  $n'$  to be  $\text{length } l'$ .

$\square$



## 11.8.2 Induction Over an Inductively Defined Proposition

Since inductively defined proof objects are often called “derivation trees,” this form of proof is also known as *induction on derivations*.

*Template:*

- *Theorem:* <Proposition of the form “ $Q \rightarrow P$ ,” where  $Q$  is some inductively defined proposition (more generally, “For all  $x\ y\ z$ ,  $Q\ x\ y\ z \rightarrow P\ x\ y\ z$ ”)>

*Proof:* By induction on a derivation of  $Q$ . <Or, more generally, “Suppose we are given  $x$ ,  $y$ , and  $z$ . We show that  $Q\ x\ y\ z$  implies  $P\ x\ y\ z$ , by induction on a derivation of  $Q\ x\ y\ z$ ”...>

<one case for each constructor  $c$  of  $Q$ ...>

- Suppose the final rule used to show  $Q$  is  $c$ . Then <...and here we state the types of all of the  $a$ ’s together with any equalities that follow from the definition of the constructor and the IH for each of the  $a$ ’s that has type  $Q$ , if there are any>. We must show <...and here we restate  $P$ >.  
<go on and prove  $P$  to finish the case...>
- <other cases similarly...>  $\square$

*Example*

- *Theorem:* The  $\leq$  relation is transitive – i.e., for all numbers  $n$ ,  $m$ , and  $o$ , if  $n \leq m$  and  $m \leq o$ , then  $n \leq o$ .

*Proof:* By induction on a derivation of  $m \leq o$ .

- Suppose the final rule used to show  $m \leq o$  is  $\text{le\_n}$ . 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  $\text{le\_S}$ . 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  $\text{le\_S}$ ,  $n \leq S\ o'$ .  $\square$

## 11.9 Explicit Proof Objects for Induction (Optional)

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 again the induction principle on naturals that Coq generates for us automatically from the Inductive declaration for **nat**.

Check **nat\_ind** :

```

∀ P : nat → Prop,
  P 0 →
  (∀ n : nat, P n → P (S n)) →
  ∀ n : nat, P n.

```

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 rewrite that more tidily as follows: `Fixpoint build_proof`

```

(P : nat → Prop)
(evPO : P 0)
(evPS : ∀ n : nat, P n → P (S n))
(n : nat) : P n :=
match n with
| 0 ⇒ evPO
| S k ⇒ evPS k (build_proof P evPO evPS k)
end.

```

Definition `nat_ind_tidy := build_proof`.

We can read `build_proof` as follows: Suppose we have evidence `evPO` that  $P$  holds on 0, and evidence `evPS` that  $\forall n:\text{nat}, P\ n \rightarrow P\ (S\ n)$ . Then we can prove that  $P$  holds of an arbitrary `nat`  $n$  using recursive function `build_proof`, which pattern matches on  $n$ :

- If  $n$  is 0, `build_proof` returns `evPO` to show that  $P\ n$  holds.
- If  $n$  is `S k`, `build_proof` applies itself recursively on  $k$  to obtain evidence that  $P\ k$  holds; then it applies `evPS` on that evidence to show that  $P\ (S\ n)$  holds.

Recursive function `build_proof` thus pattern matches against  $n$ , recursing all the way down to 0, and building up a proof as it returns.

The actual `nat_ind` that Coq generates uses a recursive function  $F$  defined with `fix` instead of `Fixpoint`.

We can adapt this approach to proving `nat_ind` to help prove *non-standard* induction principles too. As a motivating example, suppose that we want to prove the following lemma, directly relating the `ev` predicate we defined in `IndProp` to the `evenb` function defined in `Basics`.

Lemma `evenb_ev : ∀ n : nat, evenb n = true → ev'' n`.

Proof.

```

induction n; intros.
- apply ev_0.
- destruct n.
  + simpl in H. inversion H.
  + simpl in H.

```

apply ev\_SS.  
 Abort.

Attempts to prove this by standard induction on  $n$  fail in the case for  $S (S n)$ , because the induction hypothesis only tells us something about  $S n$ , which is useless. There are various ways to hack around this problem; for example, we *can* use ordinary induction on  $n$  to prove this (try it!):

**Lemma** *evenb\_ev'* :  $\forall n : \mathbf{nat}, (\text{evenb } n = \text{true} \rightarrow \mathbf{ev} \ n) \wedge (\text{evenb } (S \ n) = \text{true} \rightarrow \mathbf{ev} \ (S \ n))$ .

But we can make a much better proof by defining and proving a non-standard induction principle that goes “by twos”:

**Definition** *nat\_ind2* :  
 $\forall (P : \mathbf{nat} \rightarrow \text{Prop}),$   
 $P \ 0 \rightarrow$   
 $P \ 1 \rightarrow$   
 $(\forall n : \mathbf{nat}, P \ n \rightarrow P \ (S(S \ n))) \rightarrow$   
 $\forall n : \mathbf{nat}, P \ n :=$   
 fun  $P \Rightarrow$  fun  $P0 \Rightarrow$  fun  $P1 \Rightarrow$  fun  $PSS \Rightarrow$   
     fix  $f \ (n:\mathbf{nat}) :=$  match  $n$  with  
       0  $\Rightarrow P0$   
      | 1  $\Rightarrow P1$   
      |  $S \ (S \ n') \Rightarrow PSS \ n' \ (f \ n')$   
     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.

The `induction ... using tactic` variant gives a convenient way to utilize a non-standard induction principle like this.

**Lemma** *evenb\_ev* :  $\forall n, \text{evenb } n = \text{true} \rightarrow \mathbf{ev}'' \ n$ .

**Proof.**

  intros.  
 induction  $n$  as [| |  $n'$ ] using nat\_ind2.  
 - apply ev\_0.  
 - simpl in  $H$ .  
   inversion  $H$ .  
 - simpl in  $H$ .  
   apply ev\_SS.  
   apply  $IHn'$ .  
   apply  $H$ .

**Qed.**

# Chapter 12

## Library LF.Rel

### 12.1 Rel: Properties of Relations

This short (and optional) chapter develops some basic definitions and a few theorems about binary relations in Coq. The key definitions are repeated where they are actually used (in the *Smallstep* chapter of *Programming Language Foundations*), so readers who are already comfortable with these ideas can safely skim or skip this chapter. However, relations are also a good source of exercises for developing facility with Coq’s basic reasoning facilities, so it may be useful to look at this material just after the `IndProp` chapter.

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export IndProp.
```

### 12.2 Relations

A binary *relation* on a set  $X$  is a family of propositions parameterized by two elements of  $X$  – i.e., a proposition about pairs of elements of  $X$ .

**Definition** `relation (X: Type) := X → X → Prop`.

Confusingly, the Coq standard library hijacks the generic term “relation” for this specific instance of the idea. To maintain consistency with the library, we will do the same. So, henceforth the Coq identifier `relation` will always refer to a binary relation between some set and itself, whereas the English word “relation” can refer either to the specific Coq concept or the more general concept of a relation between any number of possibly different sets. The context of the discussion should always make clear which is meant.

An example relation on `nat` is `le`, the less-than-or-equal-to relation, which we usually write  $n1 \leq n2$ .

```
Print le.
Check le : nat → nat → Prop.
Check le : relation nat.
```

(Why did we write it this way instead of starting with `Inductive le : relation nat...`? Because we wanted to put the first `nat` to the left of the `:`, which makes Coq generate a somewhat nicer induction principle for reasoning about  $\leq$ .)

## 12.3 Basic Properties

As anyone knows who has taken an undergraduate discrete math course, there is a lot to be said about relations in general, including ways of classifying relations (as reflexive, transitive, etc.), theorems that can be proved generically about certain sorts of relations, constructions that build one relation from another, etc. For example...

### Partial Functions

A relation **R** on a set **X** is a *partial function* if, for every  $x$ , there is at most one  $y$  such that **R**  $x$   $y$  – i.e., **R**  $x$   $y1$  and **R**  $x$   $y2$  together imply  $y1 = y2$ .

Definition `partial_function`  $\{X: \text{Type}\}$  ( $R: \text{relation } X$ ) :=

$\forall x\ y1\ y2 : X, R\ x\ y1 \rightarrow R\ x\ y2 \rightarrow y1 = y2$ .

For example, the `next_nat` relation defined earlier is a partial function.

Print `next_nat`.

Check `next_nat : relation nat`.

Theorem `next_nat_partial_function` :  
`partial_function next_nat`.

Proof.

```
unfold partial_function.
intros x y1 y2 H1 H2.
inversion H1. inversion H2.
reflexivity. Qed.
```

However, the  $\leq$  relation on numbers is not a partial function. (Assume, for a contradiction, that  $\leq$  is a partial function. But then, since  $0 \leq 0$  and  $0 \leq 1$ , it follows that  $0 = 1$ . This is nonsense, so our assumption was contradictory.)

Theorem `le_not_a_partial_function` :  
 $\neg (\text{partial\_function } \text{le})$ .

Proof.

```
unfold not. unfold partial_function. intros Hc.
assert (0 = 1) as Nonsense. {
  apply Hc with (x := 0).
  - apply le_n.
  - apply le_S. apply le_n. }
discriminate Nonsense. Qed.
```

**Exercise: 2 stars, standard, optional (total\_relation\_not\_partial)** Show that the *total\_relation* defined in (an exercise in) `IndProp` is not a partial function.

**Exercise: 2 stars, standard, optional (empty\_relation\_partial)** Show that the *empty\_relation* defined in (an exercise in) `IndProp` is a partial function.

## Reflexive Relations

A *reflexive* relation on a set  $X$  is one for which every element of  $X$  is related to itself.

Definition `reflexive`  $\{X: \text{Type}\} (R: \text{relation } X) :=$

$\forall a : X, R \ a \ a.$

Theorem `le_reflexive` :

`reflexive` **le**.

Proof.

`unfold reflexive. intros n. apply le_n. Qed.`

## Transitive Relations

A relation  $R$  is *transitive* if  $R \ a \ c$  holds whenever  $R \ a \ b$  and  $R \ b \ c$  do.

Definition `transitive`  $\{X: \text{Type}\} (R: \text{relation } X) :=$

$\forall a \ b \ c : X, (R \ a \ b) \rightarrow (R \ b \ c) \rightarrow (R \ a \ c).$

Theorem `le_trans` :

`transitive` **le**.

Proof.

`intros n m o Hnm Hmo.`

`induction Hmo.`

`- apply Hnm.`

`- apply le_S. apply IH Hmo. 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, standard, optional (le\_trans\_hard\_way)** 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, standard, optional (lt\_trans)** Prove the same thing again by induction on  $o$ .

Theorem **lt\_trans''** :

transitive **lt**.

Proof.

```
unfold lt. unfold transitive.
intros n m o Hnm Hmo.
induction o as [| o'].
  Admitted.
□
```

The transitivity of **le**, in turn, can be used to prove some facts that will be useful later (e.g., for the proof of antisymmetry below)...

Theorem **le\_Sn\_le** :  $\forall n m, \text{S } n \leq m \rightarrow n \leq m$ .

Proof.

```
intros n m H. apply le_trans with (S n).
- apply le_S. apply le_n.
- apply H.
```

Qed.

**Exercise: 1 star, standard, optional (le\_S\_n)** Theorem **le\_S\_n** :  $\forall n m, (\text{S } n \leq \text{S } m) \rightarrow (n \leq m)$ .

Proof.

```
Admitted.
□
```

**Exercise: 2 stars, standard, optional (le\_Sn\_n\_inf)** Provide an informal proof of the following theorem:

Theorem: For every  $n$ ,  $\neg (\text{S } n \leq n)$

A formal proof of this is an optional exercise below, but try writing an informal proof without doing the formal proof first.

Proof:

**Exercise: 1 star, standard, optional (le\_Sn\_n)** 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, let's look at a few other common ones...

## Symmetric and Antisymmetric Relations

A relation **R** is *symmetric* if **R** a b implies **R** b a.

**Definition** symmetric {X: Type} (R: relation X) :=  
 $\forall a\ b : X, (R\ a\ b) \rightarrow (R\ b\ a).$

**Exercise: 2 stars, standard, optional (le\_not\_symmetric)** Theorem le\_not\_symmetric :  
 $\neg (\text{symmetric } le).$

Proof.

*Admitted.*

□

A relation **R** is *antisymmetric* if **R** a b and **R** b a together imply  $a = b$  – that is, if the only “cycles” in **R** are trivial ones.

**Definition** antisymmetric {X: Type} (R: relation X) :=  
 $\forall a\ b : X, (R\ a\ b) \rightarrow (R\ b\ a) \rightarrow a = b.$

**Exercise: 2 stars, standard, optional (le\_antisymmetric)** Theorem le\_antisymmetric :  
 $\text{antisymmetric } le.$

Proof.

*Admitted.*

□

**Exercise: 2 stars, standard, optional (le\_step)** Theorem le\_step :  $\forall n\ m\ p,$

$n < m \rightarrow$

$m \leq S\ p \rightarrow$

$n \leq p.$

Proof.

*Admitted.*

□



## Equivalence Relations

A relation is an *equivalence* if it's reflexive, symmetric, and transitive.

Definition equivalence  $\{X:\text{Type}\} (R: \text{relation } X) :=$   
 $(\text{reflexive } R) \wedge (\text{symmetric } R) \wedge (\text{transitive } R).$

## Partial Orders and Preorders

A relation is a *partial order* when it's reflexive, *anti*-symmetric, and transitive. In the Coq standard library it's called just “order” for short.

Definition order  $\{X:\text{Type}\} (R: \text{relation } X) :=$   
 $(\text{reflexive } R) \wedge (\text{antisymmetric } R) \wedge (\text{transitive } R).$

A preorder is almost like a partial order, but doesn't have to be antisymmetric.

Definition preorder  $\{X:\text{Type}\} (R: \text{relation } X) :=$   
 $(\text{reflexive } R) \wedge (\text{transitive } R).$

Theorem le\_order :  
order **le**.

Proof.

```
unfold order. split.  
- apply le_reflexive.  
- split.  
  + apply le_antisymmetric.  
  + apply le_trans. Qed.
```

## 12.4 Reflexive, Transitive Closure

The *reflexive, transitive closure* of a relation **R** is the smallest relation that contains **R** and that is both reflexive and transitive. Formally, it is defined like this in the Relations module of the Coq standard library:

Inductive clos\_refl\_trans  $\{A: \text{Type}\} (R: \text{relation } A) : \text{relation } A :=$   
| rt\_step  $x \ y \ (H : R \ x \ y) : \text{clos\_refl\_trans } R \ x \ y$   
| rt\_refl  $x : \text{clos\_refl\_trans } R \ x \ x$   
| rt\_trans  $x \ y \ z$   
   $(Hxy : \text{clos\_refl\_trans } R \ x \ y)$   
   $(Hyx : \text{clos\_refl\_trans } R \ y \ z) :$   
   $\text{clos\_refl\_trans } R \ x \ z.$

For example, the reflexive and transitive closure of the **next\_nat** relation coincides with the **le** relation.

Theorem next\_nat\_closure\_is\_le :  $\forall \ n \ m,$   
 $(n \leq m) \leftrightarrow ((\text{clos\_refl\_trans next\_nat}) \ n \ m).$

Proof.

```

intros n m. split.
-
  intro H. induction H.
  + apply rt_refl.
  +
    apply rt_trans with m. apply IHle. apply rt_step.
    apply nn.
-
  intro H. induction H.
  + inversion H. apply le_S. apply le_n.
  + apply le_n.
  +
    apply le_trans with y.
    apply IHclos_refl_trans1.
    apply IHclos_refl_trans2. Qed.

```

The above definition of reflexive, transitive closure is natural: it says, explicitly, that the reflexive and transitive closure of **R** is the least relation that includes **R** and that is closed under rules of reflexivity and transitivity. But it turns out that this definition is not very convenient for doing proofs, since the “nondeterminism” of the **rt\_trans** rule can sometimes lead to tricky inductions. Here is a more useful definition:

```

Inductive clos_refl_trans_1n {A : Type}
  (R : relation A) (x : A)
  : A → Prop :=
| rtln_refl : clos_refl_trans_1n R x x
| rtln_trans (y z : A)
  (Hxy : R x y) (Hrest : clos_refl_trans_1n R y z) :
  clos_refl_trans_1n R x z.

```

Our new definition of reflexive, transitive closure “bundles” the **rt\_step** and **rt\_trans** rules into the single rule **step**. The left-hand premise of this step is a single use of **R**, leading to a much simpler induction principle.

Before we go on, we should check that the two definitions do indeed define the same relation...

First, we prove two lemmas showing that **clos\_refl\_trans\_1n** mimics the behavior of the two “missing” **clos\_refl\_trans** constructors.

Lemma **rsc\_R** :  $\forall (X:\text{Type}) (R:\text{relation } X) (x\ y : X),$   
 $R\ x\ y \rightarrow \text{clos\_refl\_trans\_1n } R\ x\ y.$

Proof.

```

intros X R x y H.
apply rtln_trans with y. apply H. apply rtln_refl. Qed.

```

**Exercise: 2 stars, standard, optional (rsc\_trans)** Lemma rsc\_trans :

$\forall (X:\text{Type}) (R:\text{relation } X) (x\ y\ z : X),$   
    **clos\_refl\_trans\_1n**  $R\ x\ y \rightarrow$   
    **clos\_refl\_trans\_1n**  $R\ y\ z \rightarrow$   
    **clos\_refl\_trans\_1n**  $R\ x\ z.$

Proof.

*Admitted.*

□

Then we use these facts to prove that the two definitions of reflexive, transitive closure do indeed define the same relation.

**Exercise: 3 stars, standard, optional (rtc\_rsc\_coincide)** Theorem rtc\_rsc\_coincide :

$\forall (X:\text{Type}) (R:\text{relation } X) (x\ y : X),$   
    **clos\_refl\_trans**  $R\ x\ y \leftrightarrow$  **clos\_refl\_trans\_1n**  $R\ x\ y.$

Proof.

*Admitted.*

□

# Chapter 13

## Library LF.Imp

### 13.1 Imp: Simple Imperative Programs

In this chapter, we take a more serious look at how to use Coq to study other things. Our case study is a *simple imperative programming language* called Imp, embodying a tiny core fragment of conventional mainstream languages such as C and Java. Here is a familiar mathematical function written in Imp.

```
Z := X; Y := 1; while ~(Z = 0) do Y := Y * Z; Z := Z - 1 end
```

We concentrate here on defining the *syntax* and *semantics* of Imp; later chapters in *Programming Language Foundations* (*Software Foundations*, volume 2) develop a theory of *program equivalence* and introduce *Hoare Logic*, a widely used logic for reasoning about imperative programs.

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Bool.Bool.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.Arith.
From Coq Require Import Arith.EqNat.
From Coq Require Import Lia.
From Coq Require Import Lists.List.
From Coq Require Import Strings.String.
Import ListNotations.
From LF Require Import Maps.
```

### 13.2 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.

### 13.2.1 Syntax

Module AEXP.

These two definitions specify the *abstract syntax* of arithmetic and boolean expressions.

Inductive **aexp** : Type :=

| ANum (*n* : nat)  
| APlus (*a1 a2* : aexp)  
| AMinus (*a1 a2* : aexp)  
| AMult (*a1 a2* : aexp).

Inductive **bexp** : Type :=

| BTrue  
| BFalse  
| BEq (*a1 a2* : aexp)  
| BLe (*a1 a2* : aexp)  
| BNot (*b* : bexp)  
| BAnd (*b1 b2* : bexp).

In this chapter, we'll mostly 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 lexical analyzer and parser that can perform this translation. You do *not* need to understand that chapter to understand this one, but if you haven't already taken a course where these techniques are covered (e.g., a compilers course) you may want to skim it.

For comparison, here's a conventional BNF (Backus-Naur Form) grammar defining the same abstract syntax:

*a* := nat | *a* + *a* | *a* - *a* | *a* \* *a*  
*b* := true | false | *a* = *a* | *a* <= *a* | ~ *b* | *b* && *b*

Compared to the Coq version above...

- The BNF is more informal – for example, it gives some suggestions about the surface syntax of expressions (like the fact that the addition operation is written with an infix +) while leaving other aspects of lexical analysis and parsing (like the relative precedence of +, -, and ×, the use of parens to group subexpressions, etc.) unspecified. Some additional information – and human intelligence – would be required to turn this description into a formal definition, e.g., for implementing a compiler.

The Coq version consistently omits all this information and concentrates on the abstract syntax only.

- Conversely, the BNF version is lighter and easier to read. Its informality makes it flexible, a big 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 kind of BNF they're using because there is no need to: a rough-and-ready informal understanding is all that's important.

It's good to be comfortable with both sorts of notations: informal ones for communicating between humans and formal ones for carrying out implementations and proofs.

### 13.2.2 Evaluation

*Evaluating* an arithmetic expression produces a number.

```
Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | APlus a1 a2 ⇒ (aeval a1) + (aeval a2)
  | AMinus a1 a2 ⇒ (aeval a1) - (aeval a2)
  | AMult a1 a2 ⇒ (aeval a1) × (aeval a2)
  end.
```

Example test\_aeval1:

```
aeval (APlus (ANum 2) (ANum 2)) = 4.
```

Proof. reflexivity. Qed.

Similarly, evaluating a boolean expression yields a boolean.

```
Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ (aeval a1) =? (aeval a2)
  | BLe a1 a2 ⇒ (aeval a1) <=? (aeval a2)
  | BNot b1 ⇒ negb (beval b1)
  | BAnd b1 b2 ⇒ andb (beval b1) (beval b2)
  end.
```

### 13.2.3 Optimization

We haven't defined very much yet, but we can already get some mileage out of the definitions. Suppose we define a function that takes an arithmetic expression and slightly simplifies it, changing every occurrence of  $0 + e$  (i.e.,  $(APlus (ANum 0) e)$  into just  $e$ .

```
Fixpoint optimize_0plus (a : aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | APlus (ANum 0) e2 ⇒ optimize_0plus e2
  | APlus e1 e2 ⇒ APlus (optimize_0plus e1) (optimize_0plus e2)
```

```

| AMinus e1 e2 ⇒ AMinus (optimize_0plus e1) (optimize_0plus e2)
| AMult e1 e2 ⇒ AMult (optimize_0plus e1) (optimize_0plus e2)
end.

```

To make sure our optimization is doing the right thing we can test it on some examples and see if the output looks OK.

Example test\_optimize\_0plus:

```

optimize_0plus (APlus (ANum 2)
                    (APlus (ANum 0)
                          (APlus (ANum 0) (ANum 1))))
= APlus (ANum 2) (ANum 1).

```

Proof. reflexivity. Qed.

But if we want to be sure the optimization is correct – i.e., that evaluating an optimized expression gives the same result as the original – we should prove it.

Theorem optimize\_0plus\_sound:  $\forall a$ ,  
 $\text{aeval} (\text{optimize\_0plus } a) = \text{aeval } a$ .

Proof.

```

intros a. induction a.
- reflexivity.
- destruct a1 eqn:Ea1.
  + destruct n eqn:En.
    × simpl. apply IHa2.
    × simpl. rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
-
  simpl. rewrite IHa1. rewrite IHa2. reflexivity.
-
  simpl. rewrite IHa1. rewrite IHa2. reflexivity. Qed.

```

### 13.3 Coq Automation

The amount of repetition in this last proof is a little annoying. And if either the language of arithmetic expressions or the optimization being proved sound were significantly more complex, it would start 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.

### 13.3.1 Tacticals

*Tacticals* is Coq’s term for tactics that take other tactics as arguments – “higher-order tactics,” if you will.

#### The `try` Tactical

If  $T$  is a tactic, then `try  $T$`  is a tactic that is just like  $T$  except that, if  $T$  fails, `try  $T$`  *successfully* does nothing at all (rather than failing).

Theorem silly1 :  $\forall ae, aeval\ ae = aeval\ ae$ .

Proof. `try reflexivity. Qed.`

Theorem silly2 :  $\forall (P : Prop), P \rightarrow P$ .

Proof.

`intros  $P$   $HP$ .`

`try reflexivity.    apply  $HP$ . Qed.`

There is no real reason to use `try` in completely manual proofs like these, but it is very useful for doing automated proofs in conjunction with the `; tactical`, which we show next.

#### The `; tactical` (Simple Form)

In its most common form, the `; tactical` takes two tactics as arguments. The compound tactic  $T;T'$  first performs  $T$  and then performs  $T'$  on *each subgoal* generated by  $T$ .

For example, consider the following trivial lemma:

Lemma foo :  $\forall n, 0 <=? n = \text{true}$ .

Proof.

`intros.`

`destruct  $n$ .`

`- simpl. reflexivity.`

`- simpl. reflexivity.`

`Qed.`

We can simplify this proof using the `; tactical`:

Lemma foo' :  $\forall n, 0 <=? n = \text{true}$ .

Proof.

`intros.`



```

destruct n;

simpl;

reflexivity.
Qed.

```

Using `try` and `;` together, we can get rid of the repetition in the proof that was bothering us a little while ago.

Theorem `optimize_0plus_sound'`:  $\forall a,$   
`aeval (optimize_0plus a) = aeval a.`

Proof.

```

intros a.
induction a;

  try (simpl; rewrite IHa1; rewrite IHa2; reflexivity).
- reflexivity.
-
  destruct a1 eqn:Ea1;

    try (simpl; simpl in IHa1; rewrite IHa1;
      rewrite IHa2; reflexivity).
+ destruct n eqn:En;
  simpl; rewrite IHa2; reflexivity. Qed.

```

Coq experts often use this “...; `try`...” idiom after a tactic like `induction` to take care of many similar cases all at once. Naturally, this practice has an analog in informal proofs. For example, here is an informal proof of the optimization theorem that matches the structure of the formal one:

*Theorem:* For all arithmetic expressions `a`,  
`aeval (optimize_0plus a) = aeval a.`

*Proof:* By induction on `a`. Most cases follow directly from the IH. The remaining cases are as follows:

- Suppose `a = ANum n` for some `n`. We must show  
`aeval (optimize_0plus (ANum n)) = aeval (ANum n).`  
This is immediate from the definition of `optimize_0plus`.
- Suppose `a = APlus a1 a2` for some `a1` and `a2`. We must show  
`aeval (optimize_0plus (APlus a1 a2)) = aeval (APlus a1 a2).`

Consider the possible forms of `a1`. For most of them, `optimize_0plus` simply calls itself recursively for the subexpressions and rebuilds a new expression of the same form as `a1`; in these cases, the result follows directly from the IH.

The interesting case is when  $a1 = \text{ANum } n$  for some  $n$ . If  $n = 0$ , then

`optimize_0plus (APlus a1 a2) = optimize_0plus a2`

and the IH for  $a2$  is exactly what we need. On the other hand, if  $n = \text{S } n'$  for some  $n'$ , then again `optimize_0plus` simply calls itself recursively, and the result follows from the IH.  $\square$

However, this proof can still be improved: the first case (for  $a = \text{ANum } n$ ) is very trivial – even more trivial than the cases that we said simply followed from the IH – yet we have chosen to write it out in full. It would be better and clearer to drop it and just say, at the top, “Most cases are either immediate or direct from the IH. The only interesting case is the one for `APlus...`” We can make the same improvement in our formal proof too. Here’s how it looks:

**Theorem** `optimize_0plus_sound''`:  $\forall a,$   
`aeval (optimize_0plus a) = aeval a.`

**Proof.**

```
intros a.
induction a;

  try (simpl; rewrite IHa1; rewrite IHa2; reflexivity);

  try reflexivity.
-
  destruct a1; try (simpl; simpl in IHa1; rewrite IHa1;
    rewrite IHa2; reflexivity).
+ destruct n;
  simpl; rewrite IHa2; reflexivity. Qed.
```

## The ; Tactical (General Form)

The `;` tactical also has a more general form than the simple  $T;T'$  we’ve seen above. If  $T, T1, \dots, Tn$  are tactics, then

$T; T1 \mid T2 \mid \dots \mid Tn$

is a tactic that first performs  $T$  and then performs  $T1$  on the first subgoal generated by  $T$ , performs  $T2$  on the second subgoal, etc.

So  $T;T'$  is just special notation for the case when all of the  $Ti$ ’s are the same tactic; i.e.,  $T;T'$  is shorthand for:

$T; T' \mid T' \mid \dots \mid T'$

## The repeat Tactical

The `repeat` tactical takes another tactic and keeps applying this tactic until it fails to make progress. Here is an example showing that 10 is in a long list using `repeat`.

Theorem `ln10` : `ln 10 [1;2;3;4;5;6;7;8;9;10]`.

Proof.

`repeat (try (left; reflexivity); right).`

Qed.

The tactic `repeat`  $T$  never fails: if the tactic  $T$  doesn't apply to the original goal, then `repeat` still succeeds without changing the original goal (i.e., it repeats zero times).

Theorem `ln10'` : `ln 10 [1;2;3;4;5;6;7;8;9;10]`.

Proof.

`repeat simpl.`

`repeat (left; reflexivity).`

`repeat (right; try (left; reflexivity)).`

Qed.

The tactic `repeat`  $T$  also does not have any upper bound on the number of times it applies  $T$ . If  $T$  is a tactic that always succeeds (and makes progress), then `repeat`  $T$  will loop forever.

Theorem `repeat_loop` :  $\forall (m\ n : \text{nat}),$

$m + n = n + m.$

Proof.

`intros m n.`

*Admitted.*

While evaluation in Coq's term language, Gallina, is guaranteed to terminate, tactic evaluation is not! This does not affect Coq's logical consistency, however, since the job of `repeat` and other tactics is to guide Coq in constructing proofs; if the construction process diverges (i.e., it does not terminate), this simply means that we have failed to construct a proof, not that we have constructed a wrong one.

**Exercise: 3 stars, standard (`optimize_0plus_b_sound`)** Since the `optimize_0plus` transformation doesn't change the value of `aexps`, we should be able to apply it to all the `aexps` that appear in a `bexp` without changing the `bexp`'s value. Write a function that performs this transformation on `bexps` and prove it is sound. Use the tacticals we've just seen to make the proof as elegant as possible.

Fixpoint `optimize_0plus_b` ( $b : \text{bexp}$ ) : `bexp`

. *Admitted.*

Theorem `optimize_0plus_b_sound` :  $\forall b,$

$\text{beval} (\text{optimize\_0plus\_b } b) = \text{beval } b.$

Proof.

*Admitted.*

□

**Exercise: 4 stars, standard, optional (`optimize`)** *Design exercise:* The optimization implemented by our `optimize_0plus` function is only one of many possible optimizations on

arithmetic and boolean expressions. Write a more sophisticated optimizer and prove it correct. (You will probably find it easiest to start small – add just a single, simple optimization and its correctness proof – and build up to something more interesting incrementally.)

### 13.3.2 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 built-in language called **Ltac** with primitives that can examine and modify the proof state. The details are a bit too complicated to get into here (and it is generally agreed that **Ltac** is not the most beautiful part of Coq’s design!), but they can be found in the reference manual and other books on Coq, and there are many examples of **Ltac** definitions in the Coq standard library that you can use as examples.
- There is also an OCaml API, which can be used to build tactics that access Coq’s internal structures at a lower level, but this is seldom worth the trouble for ordinary Coq users.

The **Tactic Notation** mechanism is the easiest to come to grips with, and it offers plenty of power for many purposes. Here’s an example.

```
Tactic Notation "simpl_and_try" tactic(c) :=  
  simpl;  
  try c.
```

This defines a new tactical called *simpl\_and\_try* that takes one tactic *c* as an argument and is defined to be equivalent to the tactic `simpl; try c`. Now writing “*simpl\_and\_try* reflexivity.” in a proof will be the same as writing “`simpl; try reflexivity`.”

### 13.3.3 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 by William Pugh *Pugh* 1991 (in Bib.v).

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 ordering ( $\leq$ ), and
- the logical connectives  $\wedge$ ,  $\vee$ ,  $\neg$ , and  $\rightarrow$ ,

then invoking `omega` will either solve the goal or fail, meaning that the goal is actually false. (If the goal is *not* of this form, `omega` will also fail.)

Example `silly_presburger_example` :  $\forall m\ n\ o\ p,$

$m + n \leq n + o \wedge o + 3 = p + 3 \rightarrow$

$m \leq p.$

Proof.

`intros. lia.`

Qed.

Example `plus_comm_omega` :  $\forall m\ n,$

$m + n = n + m.$

Proof.

`intros. lia.`

Qed.

Example `plus_assoc_omega` :  $\forall m\ n\ p,$

$m + (n + p) = m + n + p.$

Proof.

`intros. lia.`

Qed.

(Note the `From Coq Require Import Lia.` at the top of the file.)

### 13.3.4 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`: For a variable  $x$ , 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  $e = x$  (where  $x$  is a variable).
- `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, solve the goal.
- `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 examples of all of these as we go along.

## 13.4 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 :
  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).
```

```
Module HYPOTHESISNAMES.
```

```
Inductive aevalR : aexp → nat → Prop :=
| E_ANum n :
  aevalR (ANum n) n
| E_APlus (e1 e2: aexp) (n1 n2: nat)
  (H1 : aevalR e1 n1)
  (H2 : aevalR e2 n2) :
  aevalR (APlus e1 e2) (n1 + n2)
| E_AMinus (e1 e2: aexp) (n1 n2: nat)
  (H1 : aevalR e1 n1)
  (H2 : aevalR e2 n2) :
  aevalR (AMinus e1 e2) (n1 - n2)
| E_AMult (e1 e2: aexp) (n1 n2: nat)
  (H1 : aevalR e1 n1)
  (H2 : aevalR e2 n2) :
  aevalR (AMult e1 e2) (n1 × n2).
```

This style gives us more control over the names that Coq chooses during proofs involving **aevalR**, at the cost of making the definition a little more verbose.

End HYPOTHESISNAMES.

It will be convenient to have an infix notation for **aevalR**. We'll write  $e ==> n$  to mean that arithmetic expression  $e$  evaluates to value  $n$ .

```
Notation "e '==>' n"
      := (aevalR e n)
      (at level 90, left associativity)
      : type_scope.
```

End AEVALR\_FIRST\_TRY.

As we saw in **lndProp** in our case study of regular expressions, Coq provides a way to use this notation in the definition of **aevalR** itself.

Reserved Notation "e '==>' n" (at level 90, 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.

### 13.4.1 Inference Rule Notation

In informal discussions, it is convenient to write the rules for **aevalR** and similar relations in the more readable graphical form of *inference rules*, where the premises above the line justify the conclusion below the line (we have already seen them in the **lndProp** chapter).

For example, the constructor **E\_APlus...**

| **E\_APlus** : forall (e1 e2: aexp) (n1 n2: nat), aevalR e1 n1 -> aevalR e2 n2 -> aevalR (APlus e1 e2) (n1 + n2)

...would be written like this as an inference rule:

$$e1 ==> n1 \quad e2 ==> n2$$


---

(**E\_APlus**) APlus e1 e2 ==> n1+n2

Formally, there is nothing deep about inference rules: they are just implications. You can read the rule name on the right as the name of the constructor and read each of the linebreaks between the premises above the line (as well as the line itself) as  $\rightarrow$ . All the variables mentioned in the rule ( $e1$ ,  $n1$ , etc.) are implicitly bound by universal quantifiers

at the beginning. (Such variables are often called *metavariables* to distinguish them from the variables of the language we are defining. At the moment, our arithmetic expressions don't include variables, but we'll soon be adding them.) The whole collection of rules is understood as being wrapped in an **Inductive** declaration. In informal prose, this is either elided or else indicated by saying something like “Let **aevalR** be the smallest relation closed under the following rules...”.

For example,  $\Rightarrow$  is the smallest relation closed under these rules:

---

(E_ANum)	$\text{ANum } n \Rightarrow n$
	$e1 \Rightarrow n1 \quad e2 \Rightarrow n2$

---

(E_APlus)	$\text{APlus } e1 \ e2 \Rightarrow n1+n2$
	$e1 \Rightarrow n1 \quad e2 \Rightarrow n2$

---

(E_AMinus)	$\text{AMinus } e1 \ e2 \Rightarrow n1-n2$
	$e1 \Rightarrow n1 \quad e2 \Rightarrow n2$

---

(E_AMult)	$\text{AMult } e1 \ e2 \Rightarrow n1*n2$
-----------	---

---

**Exercise: 1 star, standard, optional (beval\_rules)** Here, again, is the Coq definition of the **beval** function:

```
Fixpoint beval (e : bexp) : bool := match e with | BTrue => true | BFalse => false |
BEq a1 a2 => (aeval a1) ==? (aeval a2) | BLe a1 a2 => (aeval a1) <=? (aeval a2) | BNot
b => negb (beval b) | BAnd b1 b2 => andb (beval b1) (beval b2) end.
```

Write out a corresponding definition of boolean evaluation as a relation (in inference rule notation).

Definition manual\_grade\_for\_beval\_rules : **option** (nat×string) := **None**.

□

## 13.4.2 Equivalence of the Definitions

It is straightforward to prove that the relational and functional definitions of evaluation agree:

**Theorem** **aeval\_iff\_aevalR** :  $\forall a \ n,$   
 $(a \Rightarrow n) \leftrightarrow \text{aeval } a = n.$

**Proof.**

```
split.
-
  intros H.
  induction H; simpl.
+
  reflexivity.
```



```

+
  rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
+
  rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
+
  rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
-
  generalize dependent n.
  induction a;
    simpl; intros; subst.
+
  apply E_ANum.
+
  apply E_APlus.
    apply IHa1. reflexivity.
    apply IHa2. reflexivity.
+
  apply E_AMinus.
    apply IHa1. reflexivity.
    apply IHa2. reflexivity.
+
  apply E_AMult.
    apply IHa1. reflexivity.
    apply IHa2. reflexivity.
Qed.

```

We can make the proof quite a bit shorter by making more use of tacticals.

Theorem `aeval_iff_aevalR' :  $\forall a n,$`

`( $a ==> n$ )  $\leftrightarrow$  aeval  $a = n$ .`

Proof.

```

split.
-
  intros H; induction H; subst; reflexivity.
-
  generalize dependent n.
  induction a; simpl; intros; subst; constructor;
    try apply IHa1; try apply IHa2; reflexivity.

```

Qed.

**Exercise: 3 stars, standard (bevalR)** Write a relation **bevalR** in the same style as **aevalR**, and prove that it is equivalent to **beval**.

Reserved Notation " $e' ==> b$ " (at level 90, left associativity).

Inductive **bevalR**: `bexp  $\rightarrow$  bool  $\rightarrow$  Prop` :=

where "e '==>b' b" := (**bevalR** e b) : *type\_scope*

.

Lemma **beval\_iff\_bevalR** :  $\forall b \ bv,$

$b ==>b \ bv \leftrightarrow \text{beval } b = bv.$

Proof.

*Admitted.*

□

End AEXP.

### 13.4.3 Computational vs. Relational Definitions

For the definitions of evaluation for arithmetic and boolean expressions, the choice of whether to use functional or relational definitions is mainly a matter of taste: either way works.

However, there are circumstances where relational definitions of evaluation work much better than functional ones.

Module AEVALR\_DIVISION.

For example, suppose that we wanted to extend the arithmetic operations with division:

Inductive **aexp** : Type :=

| ANum (n : **nat**)  
 | APlus (a1 a2 : **aexp**)  
 | AMinus (a1 a2 : **aexp**)  
 | AMult (a1 a2 : **aexp**)  
 | ADiv (a1 a2 : **aexp**).

Extending the definition of **aeval** to handle this new operation would not be straightforward (what should we return as the result of ADiv (ANum 5) (ANum 0)?). But extending **aevalR** is very easy.

Reserved Notation "e '==>' n"

(at level 90, left associativity).

Inductive **aevalR** : **aexp** → **nat** → Prop :=

| E\_ANum (n : **nat**) :  
   (ANum n) ==> n  
 | E\_APlus (a1 a2 : **aexp**) (n1 n2 : **nat**) :  
   (a1 ==> n1) → (a2 ==> n2) → (APlus a1 a2) ==> (n1 + n2)  
 | E\_AMinus (a1 a2 : **aexp**) (n1 n2 : **nat**) :  
   (a1 ==> n1) → (a2 ==> n2) → (AMinus a1 a2) ==> (n1 - n2)  
 | E\_AMult (a1 a2 : **aexp**) (n1 n2 : **nat**) :  
   (a1 ==> n1) → (a2 ==> n2) → (AMult a1 a2) ==> (n1 × n2)  
 | E\_ADiv (a1 a2 : **aexp**) (n1 n2 n3 : **nat**) :  
   (a1 ==> n1) → (a2 ==> n2) → (n2 > 0) →

$(\text{mult } n2 \ n3 = n1) \rightarrow (\text{ADiv } a1 \ a2) ==> n3$

where "a '==>' n" := (aevalR a n) : type\_scope.

Notice that the evaluation relation has now become *partial*: There are some inputs for which it simply does not specify an output.

End AEVALR\_DIVISION.

Module AEVALR\_EXTENDED.

Or suppose that we want to extend the arithmetic operations by a nondeterministic number generator *any* that, when evaluated, may yield any number. Note that this is not the same as making a *probabilistic* choice among all possible numbers – we’re not specifying any particular probability distribution for the results, just saying what results are *possible*.

Reserved Notation "e '==>' n" (at level 90, left associativity).

Inductive aexp : Type :=

| AAny  
| ANum (n : nat)  
| APlus (a1 a2 : aexp)  
| AMinus (a1 a2 : aexp)  
| AMult (a1 a2 : aexp).

Again, extending aeval would be tricky, since now evaluation is *not* a deterministic function from expressions to numbers, but extending aevalR is no problem...

Inductive aevalR : aexp → nat → Prop :=

| E\_Any (n : nat) :  
  AAny ==> n  
| E\_ANum (n : nat) :  
  (ANum n) ==> n  
| E\_APlus (a1 a2 : aexp) (n1 n2 : nat) :  
  (a1 ==> n1) → (a2 ==> n2) → (APlus a1 a2) ==> (n1 + n2)  
| E\_AMinus (a1 a2 : aexp) (n1 n2 : nat) :  
  (a1 ==> n1) → (a2 ==> n2) → (AMinus a1 a2) ==> (n1 - n2)  
| E\_AMult (a1 a2 : aexp) (n1 n2 : nat) :  
  (a1 ==> n1) → (a2 ==> n2) → (AMult a1 a2) ==> (n1 × n2)

where "a '==>' n" := (aevalR a n) : type\_scope.

End AEVALR\_EXTENDED.

At this point you maybe wondering: which style should I use by default? In the examples we’ve just seen, relational definitions turned out to be more useful than functional ones. For situations like these, where the thing being defined is not easy to express as a function, or indeed where it is *not* a function, there is no real choice. But what about when both styles are workable?

One point in favor of relational definitions is that they can be more elegant and easier to understand.

Another is that Coq automatically generates nice inversion and induction principles from **Inductive** definitions.

On the other hand, functional definitions can often be more convenient:

- Functions are by definition deterministic and defined on all arguments; for a relation we have to *prove* these properties explicitly if we need them.
- With functions we can also take advantage of Coq’s computation mechanism to simplify expressions during proofs.

Furthermore, functions can be directly “extracted” from Gallina to executable code in OCaml or Haskell.

Ultimately, the choice often comes down to either the specifics of a particular situation or simply a question of taste. Indeed, in large Coq developments it is common to see a definition given in *both* functional and relational styles, plus a lemma stating that the two coincide, allowing further proofs to switch from one point of view to the other at will.

## 13.5 Expressions With Variables

Now we return 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.

### 13.5.1 States

Since we’ll want to look variables up to find out their current values, we’ll reuse maps from the **Maps** chapter, and **strings** will be used to represent variables in Imp.

A *machine state* (or just *state*) represents the current values of *all* variables at some point in the execution of a program.

For simplicity, we assume that the state is defined for *all* variables, even though any given program is only going to mention a finite number of them. The state captures all of the information stored in memory. For Imp programs, because each variable stores a natural number, we can represent the state as a mapping from strings to **nat**, and will use 0 as default value in the store. For more complex programming languages, the state might have more structure.

Definition `state := total_map nat`.

### 13.5.2 Syntax

We can add variables to the arithmetic expressions we had before by simply adding one more constructor:

```

Inductive aexp : Type :=
| ANum (n : nat)
| Ald (x : string)
| APlus (a1 a2 : aexp)
| AMinus (a1 a2 : aexp)
| AMult (a1 a2 : aexp).

```

Defining a few variable names as notational shorthands will make examples easier to read:

```

Definition W : string := "W".
Definition X : string := "X".
Definition Y : string := "Y".
Definition Z : string := "Z".

```

(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 the chapters developed to Imp, this overloading should not cause confusion.)

The definition of **bexps** is unchanged (except that it now refers to the new **aexps**):

```

Inductive bexp : Type :=
| BTrue
| BFalse
| BEq (a1 a2 : aexp)
| BLe (a1 a2 : aexp)
| BNot (b : bexp)
| BAnd (b1 b2 : bexp).

```

### 13.5.3 Notations

To make Imp programs easier to read and write, we introduce some notations and implicit coercions. You do not need to understand exactly what these declarations do. Briefly, though:

- The **Coercion** declaration stipulates that a function (or constructor) can be implicitly used by the type system to coerce a value of the input type to a value of the output type. For instance, the coercion declaration for **Ald** allows us to use plain strings when an **aexp** is expected; the string will implicitly be wrapped with **Ald**.
- *Declare Custom Entry* **com** tells Coq to create a new “custom grammar” for parsing Imp expressions and programs. The first notation declaration after this tells Coq that anything between  $\langle \{$  and  $\} \rangle$  should be parsed using the Imp grammar. Again, it is not necessary to understand the details, but it is important to recognize that we are defining *new* interpretations for some familiar operators like  $+$ ,  $-$ ,  $\times$ ,  $=$ ,  $\leq$ , etc., when they occur between  $\langle \{$  and  $\} \rangle$ .

Coercion Ald : string  $\rightarrow$  aexp.

Coercion ANum : nat  $\rightarrow$  aexp.

Declare Custom Entry com.

Declare Scope com\_scope.

Notation "<{ e }>" := e (at level 0, e custom com at level 99) : com\_scope.

Notation "( x )" := x (in custom com, x at level 99) : com\_scope.

Notation "x" := x (in custom com at level 0, x constr at level 0) : com\_scope.

Notation "f x .. y" := (.. (f x) .. y)  
(in custom com at level 0, only parsing,  
f constr at level 0, x constr at level 9,  
y constr at level 9) : com\_scope.

Notation "x + y" := (APlus x y) (in custom com at level 50, left associativity).

Notation "x - y" := (AMinus x y) (in custom com at level 50, left associativity).

Notation "x \* y" := (AMult x y) (in custom com at level 40, left associativity).

Notation "'true'" := true (at level 1).

Notation "'true'" := BTrue (in custom com at level 0).

Notation "'false'" := false (at level 1).

Notation "'false'" := BFalse (in custom com at level 0).

Notation "x <= y" := (BLe x y) (in custom com at level 70, no associativity).

Notation "x = y" := (BEq x y) (in custom com at level 70, no associativity).

Notation "x && y" := (BAnd x y) (in custom com at level 80, left associativity).

Notation "'~' b" := (BNot b) (in custom com at level 75, right associativity).

Open Scope com\_scope.

We can now write  $3 + (X \times 2)$  instead of `APlus 3 (AMult X 2)`, and `true && ~(X ≤ 4)` instead of `BAnd true (BNot (BLe X 4))`.

Definition example\_aexp : aexp := <{ 3 + (X × 2) }>.

Definition example\_bexp : bexp := <{ true && ¬(X ≤ 4) }>.

One downside of these and notation tricks – coercions in particular – is that they can make it a little harder for humans to calculate the types of expressions. If you ever find yourself confused, try doing `Set Printing Coercions` to see exactly what is going on.

Print example\_bexp.

Set Printing Coercions.

Print example\_bexp.

Unset Printing Coercions.

### 13.5.4 Evaluation

The arith and boolean evaluators are extended to handle variables in the obvious way, taking a state as an extra argument:

Fixpoint aeval (st : state) (a : aexp) : nat :=

```

match a with
| ANum n ⇒ n
| Ald x ⇒ st x
| <{a1 + a2}> ⇒ (aeval st a1) + (aeval st a2)
| <{a1 - a2}> ⇒ (aeval st a1) - (aeval st a2)
| <{a1 × a2}> ⇒ (aeval st a1) × (aeval st a2)
end.

```

```

Fixpoint beval (st : state) (b : bexp) : bool :=
  match b with
  | <{true}> ⇒ true
  | <{false}> ⇒ false
  | <{a1 = a2}> ⇒ (aeval st a1) =? (aeval st a2)
  | <{a1 ≤ a2}> ⇒ (aeval st a1) <=? (aeval st a2)
  | <{¬ b1}> ⇒ negb (beval st b1)
  | <{b1 && b2}> ⇒ andb (beval st b1) (beval st b2)
  end.

```

We specialize our notation for total maps to the specific case of states, i.e. using  $(\_ !-> 0)$  as empty state.

**Definition** `empty_st` :=  $(\_ !-> 0)$ .

Now we can add a notation for a “singleton state” with just one variable bound to a value. Notation “ $x \text{ '!->' } v$ ” :=  $(t\_update\ empty\_st\ x\ v)$  (at level 100).

**Example** `aexpl` :  
`aeval (X !-> 5) <{ (3 + (X × 2))}>`  
`= 13.`

**Proof.** reflexivity. Qed.

**Example** `bexpl` :  
`beval (X !-> 5) <{ true && ¬(X ≤ 4)>`  
`= true.`

**Proof.** reflexivity. Qed.

## 13.6 Commands

Now we are ready define the syntax and behavior of Imp *commands* (sometimes called *statements*).

### 13.6.1 Syntax

Informally, commands  $c$  are described by the following BNF grammar.

$c := \text{skip} \mid x := a \mid c ; c \mid \text{if } b \text{ then } c \text{ else } c \text{ end} \mid \text{while } b \text{ do } c \text{ end}$

Here is the formal definition of the abstract syntax of commands:

```

Inductive com : Type :=
| CSkip
| CAss (x : string) (a : aexp)
| CSeq (c1 c2 : com)
| Clf (b : bexp) (c1 c2 : com)
| CWhile (b : bexp) (c : com).

```

As for expressions, we can use a few `Notation` declarations to make reading and writing Imp programs more convenient.

```

Notation "'skip'" :=
  CSkip (in custom com at level 0) : com_scope.
Notation "x := y" :=
  (CAss x y)
  (in custom com at level 0, x constr at level 0,
   y at level 85, no associativity) : com_scope.
Notation "x ; y" :=
  (CSeq x y)
  (in custom com at level 90, right associativity) : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" :=
  (Clf x y z)
  (in custom com at level 89, x at level 99,
   y at level 99, z at level 99) : com_scope.
Notation "'while' x 'do' y 'end'" :=
  (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99) : com_scope.

```

For example, here is the factorial function again, written as a formal definition to Coq:

```

Definition fact_in_coq : com :=
  <{ Z := X;
    Y := 1;
    while ¬(Z = 0) do
      Y := Y × Z;
      Z := Z - 1
    end }>.

```

```
Print fact_in_coq.
```

## 13.6.2 Desugaring notations

Coq offers a rich set of features to manage the increasing complexity of the objects we work with, such as coercions and notations. However, their heavy usage can make for quite overwhelming syntax. It is often instructive to “turn off” those features to get a more elementary picture of things, using the following commands:

- `Unset Printing Notations` (undo with `Set Printing Notations`)



- `Set Printing Coercions` (undo with `Unset Printing Coercions`)
- `Set Printing All` (undo with `Unset Printing All`)

These commands can also be used in the middle of a proof, to elaborate the current goal and context.

```
Unset Printing Notations.
Print fact_in_coq.
Set Printing Notations.

Set Printing Coercions.
Print fact_in_coq.
Unset Printing Coercions.
```

### 13.6.3 The `Locate` command

#### Finding notations

When faced with unknown notation, use `Locate` with a *string* containing one of its symbols to see its possible interpretations. `Locate "&&"`.

```
Locate ";".
```

```
Locate "while".
```

#### Finding identifiers

When used with an identifier, the command `Locate` prints the full path to every value in scope with the same name. This is useful to troubleshoot problems due to variable shadowing.

```
Locate aexp.
```

### 13.6.4 More Examples

Assignment:

```
Definition plus2 : com :=
  <{ X := X + 2 }>.
```

```
Definition XtimesYinZ : com :=
  <{ Z := X × Y }>.
```

```
Definition subtract_slowly_body : com :=
  <{ Z := Z - 1 ;
    X := X - 1 }>.
```

## Loops

Definition `subtract_slowly : com :=`  
 <{ while  $\neg(X = 0)$  do  
 subtract\_slowly\_body  
 end }>.

Definition `subtract_3_from_5_slowly : com :=`  
 <{ X := 3 ;  
 Z := 5 ;  
 subtract\_slowly }>.

### An infinite loop:

Definition `loop : com :=`  
 <{ while true do  
 skip  
 end }>.

## 13.7 Evaluating Commands

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...

### 13.7.1 Evaluation as a Function (Failed Attempt)

Here's an attempt at defining an evaluation function for commands, omitting the *while* case.

The following declaration is needed to be able to use the notations in match patterns.

Fixpoint `ceval_fun_no_while (st : state) (c : com) : state :=`

```
match c with
| <{ skip }> =>
  st
| <{ x := a }> =>
  (x !-> (aeval st a) ; st)
| <{ c1 ; c2 }> =>
  let st' := ceval_fun_no_while st c1 in
  ceval_fun_no_while st' c2
| <{ if b then c1 else c2 end }> =>
  if (beval st b)
  then ceval_fun_no_while st c1
  else ceval_fun_no_while st c2
| <{ while b do c end }> =>
  st
```

end.

In a traditional functional programming language like OCaml or Haskell we could add the *while* case as follows:

Fixpoint *ceval\_fun* (st : state) (c : com) : state := match c with ... | while b do c end => if (beval st b) then *ceval\_fun* st (c ; while b do c end) else st end.

Coq doesn't accept such a definition ("Error: Cannot guess decreasing argument of fix") because the function we want to define is not guaranteed to terminate. Indeed, it *doesn't* always terminate: for example, the full version of the *ceval\_fun* function applied to the loop program above would never terminate. Since Coq is not just a functional programming language but also a consistent logic, any potentially non-terminating function needs to be rejected. Here is an (invalid!) 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 (*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, of *ceval\_fun* cannot be written in Coq – at least not without additional tricks and workarounds (see chapter `ImpCEvalFun` if you're curious about what those might be).

### 13.7.2 Evaluation as a Relation

Here's a better way: define **ceval** as a *relation* rather than a *function* – i.e., define it in `Prop` instead of `Type`, as we did for **aevalR** above.

This is an important change. Besides freeing us from awkward workarounds, it gives us a lot more flexibility in the definition. For example, if we add nondeterministic features like *any* to the language, we want the definition of evaluation to be nondeterministic – i.e., not only will it not be total, it will not even be a function!

We'll use the notation  $st = [c] => st'$  for the **ceval** relation:  $st = [c] => st'$  means that executing program *c* in a starting state *st* results in an ending state *st'*. This can be pronounced "c takes state *st* to *st'*".

#### Operational Semantics

Here is an informal definition of evaluation, presented as inference rules for readability:

---

(E\_Skip)  $st = skip => st$   
 $aeval\ st\ a = n$

---

(E\_Ass)  $st = x := a => (x !-> n ; st)$   
 $st = c1 => st'\ st' = c2 => st''$

---

(E\_Seq)  $st = c1;c2 => st''$   
 $beval\ st\ b = true\ st = c1 => st'$

---

---

(E\_IfTrue)  $st = \text{if } b \text{ then } c1 \text{ else } c2 \text{ end} \Rightarrow st'$   
 $\text{beval } st \ b = \text{false} \ st = c2 \Rightarrow st'$

---

(E\_IfFalse)  $st = \text{if } b \text{ then } c1 \text{ else } c2 \text{ end} \Rightarrow st'$   
 $\text{beval } st \ b = \text{false}$

---

(E\_WhileFalse)  $st = \text{while } b \text{ do } c \text{ end} \Rightarrow st$   
 $\text{beval } st \ b = \text{true} \ st = c \Rightarrow st' \ st' = \text{while } b \text{ do } c \text{ end} \Rightarrow st''$

---

(E\_WhileTrue)  $st = \text{while } b \text{ do } c \text{ end} \Rightarrow st''$

Here is the formal definition. Make sure you understand how it corresponds to the inference rules.

Reserved Notation

" $st' = [c] \Rightarrow st'$ "  
 (at level 40,  $c$  custom com at level 99,  
 $st$  constr,  $st'$  constr at next level).

Inductive **ceval** : **com**  $\rightarrow$  state  $\rightarrow$  state  $\rightarrow$  Prop :=

| E\_Skip :  $\forall st,$   
 $st = [\text{skip}] \Rightarrow st$   
 | E\_Ass :  $\forall st \ a \ n \ x,$   
 $\text{aeval } st \ a = n \rightarrow$   
 $st = [x := a] \Rightarrow (x \ !- \ n ; st)$   
 | E\_Seq :  $\forall c1 \ c2 \ st \ st' \ st'',$   
 $st = [c1] \Rightarrow st' \rightarrow$   
 $st' = [c2] \Rightarrow st'' \rightarrow$   
 $st = [c1 ; c2] \Rightarrow st''$   
 | E\_IfTrue :  $\forall st \ st' \ b \ c1 \ c2,$   
 $\text{beval } st \ b = \text{true} \rightarrow$   
 $st = [c1] \Rightarrow st' \rightarrow$   
 $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$   
 | E\_IfFalse :  $\forall st \ st' \ b \ c1 \ c2,$   
 $\text{beval } st \ b = \text{false} \rightarrow$   
 $st = [c2] \Rightarrow st' \rightarrow$   
 $st = [\text{if } b \text{ then } c1 \text{ else } c2 \text{ end}] \Rightarrow st'$   
 | E\_WhileFalse :  $\forall b \ st \ c,$   
 $\text{beval } st \ b = \text{false} \rightarrow$   
 $st = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st$   
 | E\_WhileTrue :  $\forall st \ st' \ st'' \ b \ c,$   
 $\text{beval } st \ b = \text{true} \rightarrow$   
 $st = [c] \Rightarrow st' \rightarrow$   
 $st' = [\text{while } b \text{ do } c \text{ end}] \Rightarrow st'' \rightarrow$

```
st = [ while b do c end ] => st''
```

```
where "st = [ c ] => st'" := (ceval c st st').
```

The cost of defining evaluation as a relation instead of a function is that we now need to construct *proofs* that some program evaluates to some result state, rather than just letting Coq's computation mechanism do it for us.

Example ceval\_example1:

```
empty_st = [
  X := 2;
  if (X ≤ 1)
    then Y := 3
    else Z := 4
  end
] => (Z !-> 4 ; X !-> 2).
```

Proof.

```
apply E_Seq with (X !-> 2).
-
  apply E_Ass. reflexivity.
-
  apply E_IfFalse.
  reflexivity.
  apply E_Ass. reflexivity.
```

Qed.

**Exercise: 2 stars, standard (ceval\_example2)** Example ceval\_example2:

```
empty_st = [
  X := 0;
  Y := 1;
  Z := 2
] => (Z !-> 2 ; Y !-> 1 ; X !-> 0).
```

Proof.

*Admitted.*

□

Set Printing Implicit.

Check @ceval\_example2.

**Exercise: 3 stars, standard, optional (pup\_to\_n)** Write an Imp program that sums the numbers from 1 to X (inclusive:  $1 + 2 + \dots + X$ ) in the variable Y. Your program should update the state as shown in theorem `pup_to_2_ceval`, which you can reverse-engineer to discover the program you should write. The proof of that theorem will be somewhat lengthy.

Definition pup\_to\_n : com

. *Admitted.*

Theorem pup\_to\_2\_ceval :

(X !-> 2) =  
     pup\_to\_n  
 ]=> (X !-> 0 ; Y !-> 3 ; X !-> 1 ; Y !-> 2 ; Y !-> 0 ; X !-> 2).

Proof.

*Admitted.*

□

### 13.7.3 Determinism of Evaluation

Changing from a computational to a relational definition of evaluation is a good move because it frees us from the artificial requirement that evaluation should be a total function. But it also raises a question: Is the second definition of evaluation really a partial *function*? Or is it possible that, beginning from the same state  $st$ , we could evaluate some command  $c$  in different ways to reach two different output states  $st'$  and  $st''$ ?

In fact, this cannot happen: **ceval** is a partial function:

Theorem ceval\_deterministic:  $\forall c\ st\ st1\ st2,$

$st = [c] \Rightarrow st1 \rightarrow$   
 $st = [c] \Rightarrow st2 \rightarrow$   
 $st1 = st2.$

Proof.

```

intros c st st1 st2 E1 E2.
generalize dependent st2.
induction E1; intros st2 E2; inversion E2; subst.
- reflexivity.
- reflexivity.
-
  rewrite (IHE1_1 st'0 H1) in *.
  apply IHE1_2. assumption.
-
  apply IHE1. assumption.
-
  rewrite H in H5. discriminate.
-
  rewrite H in H5. discriminate.
-
  apply IHE1. assumption.
-
  reflexivity.
-
  rewrite H in H2. discriminate.
```

```

-
  rewrite H in H4. discriminate.
-
  rewrite (IHE1_1 st'0 H3) in *.
  apply IHE1_2. assumption. Qed.

```

## 13.8 Reasoning About Imp Programs

We'll get deeper into more systematic and powerful techniques for reasoning about Imp programs in *Programming Language Foundations*, but we can get some distance just working with the bare definitions. This section explores some examples.

**Theorem** `plus2_spec` :  $\forall st\ n\ st'$ ,  
 $st\ X = n \rightarrow$   
 $st = [plus2] \Rightarrow st' \rightarrow$   
 $st'\ X = n + 2.$

**Proof.**

```
intros st n st' HX Heval.
```

Inverting *Heval* essentially forces Coq to expand one step of the **ceval** computation – in this case revealing that *st'* must be *st* extended with the new value of *X*, since `plus2` is an assignment.

```
inversion Heval. subst. clear Heval. simpl.
apply t_update_eq. Qed.
```

**Exercise: 3 stars, standard, optional** (`XtimesYinZ_spec`) State and prove a specification of `XtimesYinZ`.

**Definition** `manual_grade_for_XtimesYinZ_spec` : **option** (**nat** $\times$ **string**) := **None**.

□

**Exercise: 3 stars, standard, especially useful** (`loop_never_stops`) **Theorem** `loop_never_stops`

:  $\forall st\ st'$ ,  
 $\sim(st = [loop] \Rightarrow st').$

**Proof.**

```
intros st st' contra. unfold loop in contra.
remember <{ while true do skip end }> as loopdef
eqn:Heqloopdef.
```

Proceed by induction on the assumed derivation showing that *loopdef* terminates. Most of the cases are immediately contradictory (and so can be solved in one step with `discriminate`).

*Admitted.*

□

**Exercise: 3 stars, standard (no\_whiles\_eqv)** Consider the following function:

```

Fixpoint no_whiles (c : com) : bool :=
  match c with
  | <{ skip }> =>
    true
  | <{ _ := _ }> =>
    true
  | <{ c1 ; c2 }> =>
    andb (no_whiles c1) (no_whiles c2)
  | <{ if _ then ct else cf end }> =>
    andb (no_whiles ct) (no_whiles cf)
  | <{ while _ do _ end }> =>
    false
  end.

```

This predicate yields `true` just on programs that have no while loops. Using **Inductive**, write a property **no\_whilesR** such that **no\_whilesR** `c` is provable exactly when `c` is a program with no while loops. Then prove its equivalence with `no_whiles`.

**Inductive no\_whilesR: com → Prop :=**

.

**Theorem no\_whiles\_eqv:**

$\forall c, \text{no\_whiles } c = \text{true} \leftrightarrow \text{no\_whilesR } c.$

**Proof.**

*Admitted.*

□

**Exercise: 4 stars, standard (no\_whiles\_terminating)** Imp programs that don't involve while loops always terminate. State and prove a theorem *no\_whiles\_terminating* that says this.

Use either `no_whiles` or **no\_whilesR**, as you prefer.

**Definition manual\_grade\_for\_no\_whiles\_terminating : option (nat × string) := None.**

□

## 13.9 Additional Exercises

**Exercise: 3 stars, standard (stack\_compiler)** Old 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 written as



2 3 \* 3 4 2 - \* +

and evaluated like this (where we show the program being evaluated on the right and the contents of the stack on the left):

| 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 goal of this exercise is to write a small compiler that translates **aexps** into stack machine instructions.

The instruction set for our stack language will consist of the following instructions:

- **SPush**  $n$ : Push the number  $n$  on the stack.
- **SLoad**  $x$ : Load the identifier  $x$  from the store and push it on the stack
- **SPlus**: Pop the two top numbers from the stack, add them, and push the result onto the stack.
- **SMinus**: Similar, but subtract the first number from the second.
- **SMult**: Similar, but multiply.

Inductive **sinstr** : Type :=

| **SPush** ( $n$  : **nat**)  
| **SLoad** ( $x$  : **string**)  
| **SPlus**  
| **SMinus**  
| **SMult**.

Write a function to evaluate programs in the stack language. It should take as input a state, a stack represented as a list of numbers (top stack item is the head of the list), and a program represented as a list of instructions, and it should return the stack after executing the program. Test your function on the examples below.

Note that it is unspecified what to do when encountering an **SPlus**, **SMinus**, or **SMult** instruction if the stack contains fewer than two elements. In a sense, it is immaterial what we do, since a correct compiler will never emit such a malformed program. But for sake of later exercises, it would be best to skip the offending instruction and continue with the next one.

Fixpoint **s\_execute** ( $st$  : state) ( $stack$  : **list nat**)  
                  ( $prog$  : **list sinstr**)  
                  : **list nat**

. *Admitted*.

Check **s\_execute**.

Example **s\_execute1** :

**s\_execute** empty\_st []  
  [**SPush** 5; **SPush** 3; **SPush** 1; **SMinus**]

`= [2; 5].`

*Admitted.*

Example `s_execute2` :

`s_execute (X !-> 3) [3;4]`

`[SPush 4; SLoad X; SMult; SPlus]`

`= [15; 4].`

*Admitted.*

Next, write a function that compiles an **aexp** into a stack machine program. The effect of running the program should be the same as pushing the value of the expression on the stack.

Fixpoint `s_compile (e : aexp) : list sinstr`

*. Admitted.*

After you've defined `s_compile`, prove the following to test that it works.

Example `s_compile1` :

`s_compile <{ X - (2 × Y) }>`

`= [SLoad X; SPush 2; SLoad Y; SMult; SMinus].`

*Admitted.*

□

**Exercise: 3 stars, standard (execute\_app)** Execution can be decomposed in the following sense: executing stack program `p1 ++ p2` is the same as executing `p1`, taking the resulting stack, and executing `p2` from that stack. Prove that fact.

Theorem `execute_app` :  $\forall st\ p1\ p2\ stack,$

$s\_execute\ st\ stack\ (p1\ ++\ p2) = s\_execute\ st\ (s\_execute\ st\ stack\ p1)\ p2.$

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard (stack\_compiler\_correct)** Now we'll prove the correctness of the compiler implemented in the previous exercise. Begin by proving the following lemma. If it becomes difficult, consider whether your implementation of `s_execute` or `s_compile` could be simplified.

Lemma `s_compile_correct_aux` :  $\forall st\ e\ stack,$

$s\_execute\ st\ stack\ (s\_compile\ e) = aeval\ st\ e :: stack.$

Proof.

*Admitted.*

The main theorem should be a very easy corollary of that lemma.

Theorem `s_compile_correct` :  $\forall (st : state)\ (e : aexp),$

$s\_execute\ st\ []\ (s\_compile\ e) = [aeval\ st\ e].$

Proof.

*Admitted.*

□

**Exercise: 3 stars, standard, 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**. (N.b. This is only true because expression evaluation in **Imp** is rather simple. In a bigger language where evaluating an expression might diverge, the short-circuiting **BAnd** would *not* be equivalent to the original, since it would make more programs terminate.)

Module **BREAKIMP**.

**Exercise: 4 stars, advanced (break\_imp)** Imperative languages like C and Java often include a *break* or similar statement for interrupting the execution of loops. In this exercise we consider how to add *break* to **Imp**. First, we need to enrich the language of commands with an additional case.

Inductive **com** : Type :=

- | CSkip
- | CBreak
- | CAss (*x* : **string**) (*a* : **aexp**)
- | CSeq (*c1 c2* : **com**)
- | Clf (*b* : **bexp**) (*c1 c2* : **com**)
- | CWhile (*b* : **bexp**) (*c* : **com**).

Notation "'break'" := CBreak (in *custom com* at level 0).

Notation "'skip'" :=

CSkip (in *custom com* at level 0) : *com\_scope*.

Notation "x := y" :=

(CAss *x y*)  
(in *custom com* at level 0, *x* constr at level 0,  
*y* at level 85, no associativity) : *com\_scope*.

Notation "x ; y" :=

(CSeq *x y*)  
(in *custom com* at level 90, right associativity) : *com\_scope*.

Notation "'if' x 'then' y 'else' z 'end'" :=

(Clf *x y z*)  
(in *custom com* at level 89, *x* at level 99,  
*y* at level 99, *z* at level 99) : *com\_scope*.

Notation "'while' x 'do' y 'end'" :=

(CWhile  $x$   $y$ )

(in *custom com* at level 89,  $x$  at level 99,  $y$  at level 99) : *com\_scope*.

Next, we need to define the behavior of *break*. Informally, whenever *break* is executed in a sequence of commands, it stops the execution of that sequence and signals that the innermost enclosing loop should terminate. (If there aren't any enclosing loops, then the whole program simply terminates.) The final state should be the same as the one in which the *break* statement was executed.

One important point is what to do when there are multiple loops enclosing a given *break*. In those cases, *break* should only terminate the *innermost* loop. Thus, after executing the following...

X := 0; Y := 1; while ~(0 = Y) do while true do break end; X := 1; Y := Y - 1 end  
... the value of X should be 1, and not 0.

One way of expressing this behavior is to add another parameter to the evaluation relation that specifies whether evaluation of a command executes a *break* statement:

**Inductive result** : Type :=

| SContinue

| SBreak.

Reserved Notation " $st \text{ '=[ c ] => st' / s}$ "

(at level 40,  $c$  *custom com* at level 99,  $st'$  *constr* at *next level*).

Intuitively,  $st \text{ '=[ c ] => st' / s}$  means that, if  $c$  is started in state  $st$ , then it terminates in state  $st'$  and either signals that the innermost surrounding loop (or the whole program) should exit immediately ( $s = \text{SBreak}$ ) or that execution should continue normally ( $s = \text{SContinue}$ ).

The definition of the " $st \text{ '=[ c ] => st' / s}$ " relation is very similar to the one we gave above for the regular evaluation relation ( $st \text{ '=[ c ] => st'}$ ) – we just need to handle the termination signals appropriately:

- If the command is *skip*, then the state doesn't change and execution of any enclosing loop can continue normally.
- If the command is *break*, the state stays unchanged but we signal a SBreak.
- If the command is an assignment, then we update the binding for that variable in the state accordingly and signal that execution can continue normally.
- If the command is of the form *if b then c1 else c2 end*, then the state is updated as in the original semantics of Imp, except that we also propagate the signal from the execution of whichever branch was taken.
- If the command is a sequence  $c1 ; c2$ , we first execute  $c1$ . If this yields a SBreak, we skip the execution of  $c2$  and propagate the SBreak signal to the surrounding context; the resulting state is the same as the one obtained by executing  $c1$  alone. Otherwise, we

execute `c2` on the state obtained after executing `c1`, and propagate the signal generated there.

- Finally, for a loop of the form *while* `b` *do* `c` *end*, the semantics is almost the same as before. The only difference is that, when `b` evaluates to true, we execute `c` and check the signal that it raises. If that signal is `SContinue`, then the execution proceeds as in the original semantics. Otherwise, we stop the execution of the loop, and the resulting state is the same as the one resulting from the execution of the current iteration. In either case, since *break* only terminates the innermost loop, *while* signals `SContinue`.

Based on the above description, complete the definition of the **ceval** relation.

Inductive **ceval** : **com** → state → result → state → Prop :=  
 | E\_Skip : ∀ st,  
   st = [ CSkip ] => st / SContinue

where "st' = [ c ] => st' / s" := (**ceval** c st s st').

Now prove the following properties of your definition of **ceval**:

Theorem break\_ignore : ∀ c st st' s,  
   st = [ break; c ] => st' / s →  
   st = st'.

Proof.

*Admitted.*

Theorem while\_continue : ∀ b c st st' s,  
   st = [ while b do c end ] => st' / s →  
   s = SContinue.

Proof.

*Admitted.*

Theorem while\_stops\_on\_break : ∀ b c st st',  
   beval st b = true →  
   st = [ c ] => st' / SBreak →  
   st = [ while b do c end ] => st' / SContinue.

Proof.

*Admitted.*

□

**Exercise: 3 stars, advanced, optional (while\_break\_true)** Theorem while\_break\_true : ∀ b c st st',

  st = [ while b do c end ] => st' / SContinue →  
   beval st' b = true →  
   ∃ st'', st'' = [ c ] => st' / SBreak.

Proof.

*Admitted.*

□

**Exercise: 4 stars, advanced, optional (ceval\_deterministic)** Theorem `ceval_deterministic`:

$\forall (c:\text{com}) \ st \ st1 \ st2 \ s1 \ s2,$   
     $st = [c] \Rightarrow st1 \ / \ s1 \ \rightarrow$   
     $st = [c] \Rightarrow st2 \ / \ s2 \ \rightarrow$   
     $st1 = st2 \wedge s1 = s2.$

Proof.

*Admitted.*

□ End BREAKIMP.

**Exercise: 4 stars, standard, optional (add\_for\_loop)** Add C-style `for` loops to the language of commands, update the **ceval** definition to define the semantics of `for` loops, and add cases for `for` loops as needed so that all the proofs in this file are accepted by Coq.

A `for` loop should be parameterized by (a) a statement executed initially, (b) a test that is run on each iteration of the loop to determine whether the loop should continue, (c) a statement executed at the end of each loop iteration, and (d) a statement that makes up the body of the loop. (You don't need to worry about making up a concrete Notation for `for` loops, but feel free to play with this too if you like.)

# Chapter 14

## Library LF.ImpParser

### 14.1 ImpParser: Lexing and Parsing in Coq

The development of the Imp language in *Imp.v* completely ignores issues of concrete syntax – how an ascii string that a programmer might write gets translated into abstract syntax trees defined by the datatypes **aexp**, **bexp**, and **com**. In this chapter, we illustrate how the rest of the story can be filled in by building a simple lexical analyzer and parser using Coq’s functional programming facilities.

It is not important to understand all the details here (and accordingly, the explanations are fairly terse and there are no exercises). The main point is simply to demonstrate that it can be done. You are invited to look through the code – most of it is not very complicated, though the parser relies on some “monadic” programming idioms that may require a little work to make out – but most readers will probably want to just skim down to the Examples section at the very end to get the punchline.

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Strings.String.
From Coq Require Import Strings.Ascii.
From Coq Require Import Arith.Arith.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.EqNat.
From Coq Require Import Lists.List.
Import ListNotations.
From LF Require Import Maps Imp.
```

### 14.2 Internals

#### 14.2.1 Lexical Analysis

Definition isWhite (*c* : **ascii**) : **bool** :=

```

let n := nat_of_ascii c in
orb (orb (n =? 32)
      (n =? 9))
  (orb (n =? 10)
      (n =? 13)).

```

Notation "x '<=?' y" := (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.

Definition 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.

## 14.2.2 Parsing

### Options With Errors

An **option** type with error messages:

```

Inductive optionE (X:Type) : Type :=
| SomeE (x : X)
| NoneE (s : string).

```

*Arguments* SomeE {X}.

*Arguments* NoneE {X}.

Some syntactic sugar to make writing nested match-expressions on `optionE` more convenient.

```

Notation "' p <- e1 ;; e2"
:= (match e1 with

```

```

    | SomeE p ⇒ e2
    | NoneE err ⇒ NoneE err
  end)
(right associativity, p pattern, at level 60, e1 at next level).
Notation "'TRY' ' p <- e1 ;; e2 'OR' e3"
:= (match e1 with
   | SomeE p ⇒ e2
   | NoneE _ ⇒ e3
  end)
(right associativity, p pattern,
 at level 60, e1 at next level, e2 at next level).

```

## Generic Combinators for Building Parsers

Open Scope *string\_scope*.

Definition parser ( $T : \text{Type}$ ) :=  
**list** token → **optionE** ( $T \times \text{list token}$ ).

Fixpoint many\_helper { $T$ } ( $p : \text{parser } T$ ) acc steps xs :=  
 match steps, p xs with  
 | 0, \_ ⇒  
   NoneE "Too many recursive calls"  
 | \_, NoneE \_ ⇒  
   SomeE ((**rev** acc), xs)  
 | **S** steps', SomeE (t, xs') ⇒  
   many\_helper p (t :: acc) steps' xs'  
 end.

A (step-indexed) parser that expects zero or more  $ps$ :

Definition many { $T$ } ( $p : \text{parser } T$ ) (steps : **nat**) : parser (**list**  $T$ ) :=  
 many\_helper p [] steps.

A parser that expects a given token, followed by  $p$ :

Definition firstExpect { $T$ } ( $t : \text{token}$ ) ( $p : \text{parser } T$ )  
                                   : parser  $T$  :=  
 fun xs ⇒ match xs with  
   | x :: xs' ⇒  
     if **string\_dec** x t  
     then p xs'  
     else NoneE ("expected '" ++ t ++ "'."  
 | [] ⇒  
   NoneE ("expected '" ++ t ++ "'."  
 end.

A parser that expects a particular token:

Definition expect ( $t : \text{token}$ ) : parser **unit** :=  
firstExpect  $t$  (fun  $xs \Rightarrow \text{SomeE } (\text{tt}, xs)$ ).

## A Recursive-Descent Parser for Imp

Identifiers:

Definition parseIdentifier ( $xs : \text{list token}$ )  
: optionE (**string**  $\times$  **list** token) :=

```
match xs with
| []  $\Rightarrow$  NoneE "Expected identifier"
|  $x :: xs' \Rightarrow$ 
  if forallb isLowerAlpha (list_of_string x) then
    SomeE ( $x, xs'$ )
  else
    NoneE ("Illegal identifier:" ++ x ++ "'")
end.
```

Numbers:

Definition parseNumber ( $xs : \text{list token}$ )  
: optionE (**nat**  $\times$  **list** token) :=

```
match xs with
| []  $\Rightarrow$  NoneE "Expected number"
|  $x :: xs' \Rightarrow$ 
  if forallb isDigit (list_of_string x) then
    SomeE (fold_left
      (fun n d  $\Rightarrow$ 
        10  $\times$  n + (nat_of_ascii d -
          nat_of_ascii "0"%char))
      (list_of_string x)
      0,
      xs')
  else
    NoneE "Expected number"
end.
```

Parse arithmetic expressions

Fixpoint parsePrimaryExp ( $steps : \text{nat}$ )  
( $xs : \text{list token}$ )  
: optionE (**aexp**  $\times$  **list** token) :=

```
match steps with
| 0  $\Rightarrow$  NoneE "Too many recursive calls"
| S steps'  $\Rightarrow$ 
```

```

    TRY ' (i, rest) ← parseIdentifier xs ;;
        SomeE (Ald i, rest)
    OR
    TRY ' (n, rest) ← parseNumber xs ;;
        SomeE (ANum n, rest)
    OR
    ' (e, rest) ← firstExpect "(" (parseSumExp steps') xs ;;
    ' (u, rest') ← expect ")" rest ;;
    SomeE (e, rest')
end

with parseProductExp (steps:nat)
    (xs : list token) :=
    match steps with
    | 0 ⇒ NoneE "Too many recursive calls"
    | S steps' ⇒
        ' (e, rest) ← parsePrimaryExp steps' xs ;;
        ' (es, rest') ← many (firstExpect "*" (parsePrimaryExp steps'))
            steps' rest ;;
        SomeE (fold_left AMult es e, rest')
    end

with parseSumExp (steps:nat) (xs : list token) :=
    match steps with
    | 0 ⇒ NoneE "Too many recursive calls"
    | S steps' ⇒
        ' (e, rest) ← parseProductExp steps' xs ;;
        ' (es, rest') ←
            many (fun xs ⇒
                TRY ' (e, rest') ←
                    firstExpect "+"
                        (parseProductExp steps') xs ;;
                SomeE ( (true, e), rest')
                OR
                ' (e, rest') ←
                    firstExpect "-"
                        (parseProductExp steps') xs ;;
                SomeE ( (false, e), rest'))
            steps' rest ;;
        SomeE (fold_left (fun e0 term ⇒
            match term with
            | (true, e) ⇒ APlus e0 e

```

```

        | (false, e) ⇒ AMinus e 0 e
        end)
      es e,
    rest')
  end.

Definition parseAExp := parseSumExp.

Parsing boolean expressions:
Fixpoint parseAtomicExp (steps:nat)
  (xs : list token) :=
match steps with
| 0 ⇒ NoneE "Too many recursive calls"
| S steps' ⇒
  TRY ' (u, rest) ← expect "true" xs ;;
    SomeE (BTrue, rest)
  OR
  TRY ' (u, rest) ← expect "false" xs ;;
    SomeE (BFalse, rest)
  OR
  TRY ' (e, rest) ← firstExpect "~"
    (parseAtomicExp steps')
    xs ;;
    SomeE (BNot e, rest)
  OR
  TRY ' (e, rest) ← firstExpect "("
    (parseConjunctionExp steps')
    xs ;;
    ' (u, rest') ← expect ")" rest ;;
    SomeE (e, rest')
  OR
  ' (e, rest) ← parseProductExp steps' xs ;;
  TRY ' (e', rest') ← firstExpect "="
    (parseAExp steps') rest ;;
    SomeE (BEq e e', rest')
  OR
  TRY ' (e', rest') ← firstExpect "<="
    (parseAExp steps') rest ;;
    SomeE (BLe e e', rest')
  OR
  NoneE "Expected '=' or '<=' after arithmetic expression"
end

with parseConjunctionExp (steps:nat)

```

```

                                (xs : list token) :=
match steps with
| 0 ⇒ NoneE "Too many recursive calls"
| S steps' ⇒
  ' (e, rest) ← parseAtomicExp steps' xs ;;
  ' (es, rest') ← many (firstExpect "&&"
                        (parseAtomicExp steps'))
                        steps' rest ;;
  SomeE (fold_left BAnd es e, rest')
end.

Definition parseBExp := parseConjunctionExp.
Check parseConjunctionExp.

Definition testParsing {X : Type}
  (p : nat →
      list token →
      optionE (X × list token))
  (s : string) :=
let t := tokenize s in
p 100 t.

Parsing commands:

Fixpoint parseSimpleCommand (steps:nat)
  (xs : list token) :=

match steps with
| 0 ⇒ NoneE "Too many recursive calls"
| S steps' ⇒
  TRY ' (u, rest) ← expect "skip" xs ;;
  SomeE (<{skip}>, rest)
OR
  TRY ' (e, rest) ←
    firstExpect "if"
    (parseBExp steps') xs ;;
  ' (c, rest') ←
    firstExpect "then"
    (parseSequencedCommand steps') rest ;;
  ' (c', rest'') ←
    firstExpect "else"
    (parseSequencedCommand steps') rest' ;;
  ' (tt, rest''') ←
    expect "end" rest'' ;;
  SomeE(<{if e then c else c' end}>, rest''')
OR

```

```

    TRY ' (e, rest) ←
      firstExpect "while"
        (parseBExp steps') xs ;;
    ' (c, rest') ←
      firstExpect "do"
        (parseSequencedCommand steps') rest ;;
    ' (u, rest'') ←
      expect "end" rest' ;;
    SomeE (<{while e do c end}>, rest'')
  OR
  TRY ' (i, rest) ← parseIdentifier xs ;;
    ' (e, rest') ← firstExpect "!=" (parseAExp steps') rest ;;
    SomeE (<{i := e}>, rest')
  OR
    NoneE "Expecting a command"
end

with parseSequencedCommand (steps: nat)
  (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    ' (c, rest) ← parseSimpleCommand steps' xs ;;
    TRY ' (c', rest') ←
      firstExpect ";"
        (parseSequencedCommand steps') rest ;;
    SomeE (<{c ; c'}>, rest')
  OR
    SomeE (c, rest)
end.

Definition bignumber := 1000.

Definition parse (str : string) : optionE com :=
  let tokens := tokenize str in
  match parseSequencedCommand bignumber tokens with
  | SomeE (c, []) ⇒ SomeE c
  | SomeE (_, t :: _) ⇒ NoneE ("Trailing tokens remaining: " ++ t)
  | NoneE err ⇒ NoneE err
end.

```

## 14.3 Examples

Example eg1 : parse " if x = y + 1 + 2 - y \* 6 + 3 then x := x \* 1; y := 0 else skip end "

=

```
SomeE <{
  if ("x" = ("y" + 1 + 2 - "y" × 6 + 3)) then
    "x" := "x" × 1;
    "y" := 0
  else
    skip
  end }>.
```

Proof. cbv. reflexivity. Qed.

Example eg2 : parse " skip; z:=x\*y\*(x\*x); while x=x do if (z <= z\*z) && ~(x = 2) then x := z; y := z else skip end; skip end; x:=z "

=

```
SomeE <{
  skip;
  "z" := "x" × "y" × ("x" × "x");
  while ("x" = "x") do
    if ("z" ≤ "z" × "z") && ¬("x" = 2) then
      "x" := "z";
      "y" := "z"
    else
      skip
    end;
    skip
  end;
  "x" := "z" }>.
```

Proof. cbv. reflexivity. Qed.



# Chapter 15

## Library LF.ImpCEvalFun

### 15.1 ImpCEvalFun: An Evaluation Function for Imp

We saw in the `Imp` chapter how a naive approach to defining a function representing evaluation for `Imp` runs into difficulties. There, we adopted the solution of changing from a functional to a relational definition of evaluation. In this optional chapter, we consider strategies for getting the functional approach to work.

### 15.2 A Broken Evaluator

```
From Coq Require Import Lia.  
From Coq Require Import Arith.Arith.  
From LF Require Import Imp Maps.
```

Here was our first try at an evaluation function for commands, omitting *while*.

```
Fixpoint ceval_step1 (st : state) (c : com) : state :=  
  match c with  
  | <{ skip }> =>  
    st  
  | <{ l := a1 }> =>  
    (l !-> aeval st a1 ; st)  
  | <{ c1 ; c2 }> =>  
    let st' := ceval_step1 st c1 in  
    ceval_step1 st' c2  
  | <{ if b then c1 else c2 end }> =>  
    if (beval st b)  
    then ceval_step1 st c1  
    else ceval_step1 st c2  
  | <{ while b1 do c1 end }> =>  
    st
```

end.

As we remarked in chapter `Imp`, in a traditional functional programming language like ML or Haskell we could write the while case as follows:

```
| while b1 do c1 end => if (beval st b1) then ceval_step1 st <{ c1; while b1 do c1 end }> else st
```

Coq doesn't accept such a definition (*Error: Cannot guess decreasing argument of fix*) because the function we want to define is not guaranteed to terminate. Indeed, the changed `ceval_step1` function applied to the `loop` program from *Imp.v* would never terminate. Since Coq is not just a functional programming language, but also a consistent logic, any potentially non-terminating function needs to be rejected. Here is an invalid(!) Coq program showing what would go wrong if Coq allowed non-terminating recursive functions:

```
Fixpoint loop_false (n : nat) : False := loop_false n.
```

That is, propositions like **False** would become provable (e.g., `loop_false 0` would be a proof of **False**), which would be a disaster for Coq's logical consistency.

Thus, because it doesn't terminate on all inputs, the full version of `ceval_step1` cannot be written in Coq – at least not without one additional trick...

## 15.3 A Step-Indexed Evaluator

The trick we need is to pass an *additional* parameter to the evaluation function that tells it how long to run. Informally, we start the evaluator with a certain amount of “gas” in its tank, and we allow it to run until either it terminates in the usual way *or* it runs out of gas, at which point we simply stop evaluating and say that the final result is the empty memory. (We could also say that the result is the current state at the point where the evaluator runs out of gas – it doesn't really matter because the result is going to be wrong in either case!)

```
Fixpoint ceval_step2 (st : state) (c : com) (i : nat) : state :=
```

```
  match i with
  | 0 => empty_st
  | S i' =>
    match c with
    | <{ skip }> =>
      st
    | <{ l := a1 }> =>
      (l !-> aeval st a1 ; st)
    | <{ c1 ; c2 }> =>
      let st' := ceval_step2 st c1 i' in
      ceval_step2 st' c2 i'
    | <{ if b then c1 else c2 end }> =>
      if (beval st b)
      then ceval_step2 st c1 i'
      else ceval_step2 st c2 i'
```

```

| <{ while b1 do c1 end }> ⇒
  if (beval st b1)
  then let st' := ceval_step2 st c1 i' in
    ceval_step2 st' c i'
  else st
end
end.

```

*Note:* It is tempting to think that the index  $i$  here is counting the “number of steps of evaluation.” But if you look closely you’ll see that this is not the case: for example, in the rule for sequencing, the same  $i$  is passed to both recursive calls. Understanding the exact way that  $i$  is treated will be important in the proof of `ceval__ceval_step`, which is given as an exercise below.

One thing that is not so nice about this evaluator is that we can’t tell, from its result, whether it stopped because the program terminated normally or because it ran out of gas. Our next version returns an **option state** instead of just a **state**, so that we can distinguish between normal and abnormal termination.

```

Fixpoint ceval_step3 (st : state) (c : com) (i : nat)
  : option state :=
  match i with
  | 0 ⇒ None
  | S i' ⇒
    match c with
    | <{ skip }> ⇒
      Some st
    | <{ l := a1 }> ⇒
      Some (l !-> aeval st a1 ; st)
    | <{ c1 ; c2 }> ⇒
      match (ceval_step3 st c1 i') with
      | Some st' ⇒ ceval_step3 st' c2 i'
      | None ⇒ None
      end
    | <{ if b then c1 else c2 end }> ⇒
      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

```

```

end
end.

```

We can improve the readability of this version by introducing a bit of auxiliary notation to hide the plumbing involved in repeatedly matching against optional states.

Notation "'LETOPT' x <== e1 'IN' e2"

```

:= (match e1 with
    | Some x => e2
    | None => None
end)
(right associativity, at level 60).

```

Fixpoint ceval\_step (st : state) (c : com) (i : nat)  
: option state :=

```

match i with
| O => None
| S i' =>
    match c with
    | <{ skip }> =>
        Some st
    | <{ l := a1 }> =>
        Some (l !-> aeval st a1 ; st)
    | <{ c1 ; c2 }> =>
        LETOPT st' <== ceval_step st c1 i' IN
        ceval_step st' c2 i'
    | <{ if b then c1 else c2 end }> =>
        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, standard, especially useful (pup\_to\_n)** Write an Imp program that sums the numbers from 1 to X (inclusive:  $1 + 2 + \dots + X$ ) in the variable Y. Make sure your solution satisfies the test that follows.

Definition pup\_to\_n : com  
 . Admitted.

□

**Exercise: 2 stars, standard, optional (peven)** Write an Imp program that sets Z to 0 if X is even and sets Z to 1 otherwise. Use test\_ceval to test your program.

## 15.4 Relational vs. Step-Indexed Evaluation

As for arithmetic and boolean expressions, we'd hope that the two alternative definitions of evaluation would actually amount to the same thing in the end. This section shows that this is the case.

Theorem ceval\_step\_\_ceval:  $\forall c \ st \ st',$   
 $(\exists i, \text{ceval\_step } st \ c \ i = \text{Some } st') \rightarrow$   
 $st = [c] \Rightarrow st'.$

Proof.

```

intros c st st' H.
inversion H as [i E].
clear H.
generalize dependent st'.
generalize dependent st.
generalize dependent c.
induction i as [| i' ].
-
  intros c st st' H. discriminate H.
-
  intros c st st' H.
  destruct c;
    simpl in H; inversion H; subst; clear H.
  + apply E_Skip.
  + apply E_Ass. reflexivity.
  +
    destruct (ceval_step st c1 i') eqn:Hegr1.
    ×
    apply E_Seq with s.
    apply IHi'. rewrite Hegr1. reflexivity.
    apply IHi'. simpl in H1. assumption.
```

```

×
  discriminate H1.
+
destruct (beval st b) eqn:Hegr.
×
  apply E_IfTrue. rewrite Hegr. reflexivity.
  apply IHi'. assumption.
×
  apply E_IfFalse. rewrite Hegr. reflexivity.
  apply IHi'. assumption.
+ destruct (beval st b) eqn :Hegr.
×
  destruct (ceval_step st c i') eqn:Hegr1.
  {
    apply E_WhileTrue with s. rewrite Hegr.
    reflexivity.
    apply IHi'. rewrite Hegr1. reflexivity.
    apply IHi'. simpl in H1. assumption. }
  { discriminate H1. }
×
  injection H1 as H2. rewrite ← H2.
  apply E_WhileFalse. apply Hegr. Qed.

```

**Exercise: 4 stars, standard (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.

Definition `manual_grade_for_ceval_step__ceval_inf` : **option** (**nat**×**string**) := **None**.

□

Theorem `ceval_step_more`:  $\forall i1\ i2\ st\ st'\ c,$

$i1 \leq i2 \rightarrow$   
`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.`

```

-
  simpl in Hceval. discriminate Hceval.
-
  destruct i2 as [|i2']. inversion Hle.
  assert (Hle': i1' ≤ i2') by lia.

```

```

destruct c.
+
  simpl in Hceval. inversion Hceval.
  reflexivity.
+
  simpl in Hceval. inversion Hceval.
  reflexivity.
+
  simpl in Hceval. simpl.
  destruct (ceval_step st c1 i1') eqn:Hegst1'o.
  ×
    apply (IH i1' i2') in Hegst1'o; try assumption.
    rewrite Hegst1'o. simpl. simpl in Hceval.
    apply (IH i1' i2') in Hceval; try assumption.
  ×
    discriminate Hceval.
+
  simpl in Hceval. simpl.
  destruct (beval st b); apply (IH i1' i2') in Hceval;
    assumption.
+
  simpl in Hceval. simpl.
  destruct (beval st b); try assumption.
  destruct (ceval_step st c i1') eqn: Hegst1'o.
  ×
    apply (IH i1' i2') in Hegst1'o; try assumption.
    rewrite → Hegst1'o. simpl. simpl in Hceval.
    apply (IH i1' i2') in Hceval; try assumption.
  ×
    simpl in Hceval. discriminate Hceval. Qed.

```

**Exercise: 3 stars, standard, especially useful (ceval\_\_ceval\_step)** Finish the following proof. You'll need `ceval_step_more` in a few places, as well as some basic facts about  $\leq$  and `plus`.

Theorem `ceval__ceval_step`:  $\forall c \ st \ st',$   
 $st = [c] \Rightarrow st' \rightarrow$   
 $\exists i, \text{ceval\_step } st \ c \ i = \text{Some } st'.$

Proof.

```

intros c st st' Hce.
induction Hce.
Admitted.
□

```

Theorem `ceval_and_ceval_step_coincide`:  $\forall c\ st\ st',$   
 $st = [c] \Rightarrow st'$

$\leftrightarrow \exists i, \text{ceval\_step } st\ c\ i = \text{Some } st'.$

Proof.

`intros c st st'.`

`split. apply ceval__ceval_step. apply ceval_step__ceval.`

`Qed.`

## 15.5 Determinism of Evaluation Again

Using the fact that the relational and step-indexed definition of evaluation are the same, we can give a slicker proof that the evaluation *relation* is deterministic.

Theorem `ceval_deterministic'` :  $\forall c\ st\ st1\ st2,$

$st = [c] \Rightarrow st1 \rightarrow$

$st = [c] \Rightarrow st2 \rightarrow$

$st1 = st2.$

Proof.

`intros c st st1 st2 He1 He2.`

`apply ceval__ceval_step in He1.`

`apply ceval__ceval_step in He2.`

`inversion He1 as [i1 E1].`

`inversion He2 as [i2 E2].`

`apply ceval_step_more with (i2 := i1 + i2) in E1.`

`apply ceval_step_more with (i2 := i1 + i2) in E2.`

`rewrite E1 in E2. inversion E2. reflexivity.`

`lia. lia. Qed.`



# Chapter 16

## Library `LF.Extraction`

### 16.1 Extraction: Extracting ML from Coq

### 16.2 Basic Extraction

In its simplest form, extracting an efficient program from one written in Coq is completely straightforward.

First we say what language we want to extract into. Options are OCaml (the most mature), Haskell (mostly works), and Scheme (a bit out of date).

```
Require Coq.extraction.Extraction.
```

```
Extraction Language OCaml.
```

Now we load up the Coq environment with some definitions, either directly or by importing them from other modules.

```
From Coq Require Import Arith.Arith.
```

```
From Coq Require Import Init.Nat.
```

```
From Coq Require Import Arith.EqNat.
```

```
From LF Require Import ImpCEvalFun.
```

Finally, we tell Coq the name of a definition to extract and the name of a file to put the extracted code into.

```
Extraction "impl.ml" ceval_step.
```

When Coq processes this command, it generates a file *impl.ml* containing an extracted version of *ceval\_step*, together with everything that it recursively depends on. Compile the present *.v* file and have a look at *impl.ml* now.

### 16.3 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 eqb ⇒ "( = )".
```

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*.

## 16.4 A Complete Example

To use our extracted evaluator to run Imp programs, all we need to add is a tiny driver program that calls the evaluator and prints out the result.

For simplicity, we’ll print results by dumping out the first four memory locations in the final state.

Also, to make it easier to type in examples, let’s extract a parser from the *ImpParser* Coq module. To do this, we first need to set up the right correspondence between Coq strings and lists of OCaml characters.

```
Require Import ExtrOcamlBasic.
Require Import ExtrOcamlString.
```

We also need one more variant of booleans.

```
Extract Inductive sumbool ⇒ "bool" ["true" "false"].
```

The extraction is the same as always.

```
From LF Require Import Imp.
```

From *LF* Require Import ImpParser.

From *LF* Require Import Maps.

Extraction "imp.ml" *empty\_st 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.)

## 16.5 Discussion

Since we've proved that the *ceval\_step* function behaves the same as the **ceval** relation in an appropriate sense, the extracted program can be viewed as a *certified* Imp interpreter. Of course, the parser we're using is not certified, since we didn't prove anything about it!

## 16.6 Going Further

Further details about extraction can be found in the Extract chapter in *Verified Functional Algorithms* (*Software Foundations* volume 3).

# Chapter 17

## Library LF.Auto

### 17.1 Auto: More Automation

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Lia.
From LF Require Import Maps.
From LF Require Import Imp.
```

Up to now, we've used the more manual part of Coq's tactic facilities. In this chapter, we'll learn more about some of Coq's powerful automation features: proof search via the `auto` tactic, automated forward reasoning via the `Ltac` hypothesis matching machinery, and deferred instantiation of existential variables using `eapply` and `eauto`. Using these features together with `Ltac`'s scripting facilities will enable us to make our proofs startlingly short! Used properly, they can also make proofs more maintainable and robust to changes in underlying definitions. A deeper treatment of `auto` and `eauto` can be found in the *UseAuto* chapter in *Programming Language Foundations*.

There's another major category of automation we haven't discussed much yet, namely built-in decision procedures for specific kinds of problems: *lia* is one example, but there are others. This topic will be deferred for a while longer.

Our motivating example will be this proof, repeated with just a few small changes from the `Imp` chapter. We will simplify this proof in several stages.

```
Theorem ceval_deterministic:  $\forall c\ st\ st1\ st2,$ 
   $st = [c] \Rightarrow st1 \rightarrow$ 
   $st = [c] \Rightarrow st2 \rightarrow$ 
   $st1 = st2.$ 
```

**Proof.**

```
intros c st st1 st2 E1 E2;
generalize dependent st2;
induction E1; intros st2 E2; inversion E2; subst.
- reflexivity.
- reflexivity.
```

```

-
  rewrite (IHE1_1 st'0 H1) in *.
  apply IHE1_2. assumption.
-
  apply IHE1. assumption.
-
  rewrite H in H5. discriminate.
-
  rewrite H in H5. discriminate.
-
  apply IHE1. assumption.
-
  reflexivity.
-
  rewrite H in H2. discriminate.
-
  rewrite H in H4. discriminate.
-
  rewrite (IHE1_1 st'0 H3) in *.
  apply IHE1_2. assumption. Qed.

```

## 17.2 The auto Tactic

Thus far, our proof scripts mostly apply relevant hypotheses or lemmas by name, and one at a time.

Example auto\_example\_1 :  $\forall (P \ Q \ R: \text{Prop}),$

$(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R.$

Proof.

intros  $P \ Q \ R \ H1 \ H2 \ H3.$

apply  $H2.$  apply  $H1.$  assumption.

Qed.

The `auto` tactic frees us from this drudgery by *searching* for a sequence of applications that will prove the goal:

Example auto\_example\_1' :  $\forall (P \ Q \ R: \text{Prop}),$

$(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R.$

Proof.

auto.

Qed.

The `auto` tactic solves goals that are solvable by any combination of

- `intros` and

- `apply` (of hypotheses from the local context, by default).

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 larger example showing `auto`’s power:

Example `auto_example_2` :  $\forall P Q R S T U : \text{Prop}$ ,

$(P \rightarrow Q) \rightarrow$   
 $(P \rightarrow R) \rightarrow$   
 $(T \rightarrow R) \rightarrow$   
 $(S \rightarrow T \rightarrow U) \rightarrow$   
 $((P \rightarrow Q) \rightarrow (P \rightarrow S)) \rightarrow$   
 $T \rightarrow$   
 $P \rightarrow$   
 $U$ .

Proof. `auto`. Qed.

Proof search could, in principle, take an arbitrarily long time, so there are limits to how far `auto` will search by default.

Example `auto_example_3` :  $\forall (P Q R S T U : \text{Prop})$ ,

$(P \rightarrow Q) \rightarrow$   
 $(Q \rightarrow R) \rightarrow$   
 $(R \rightarrow S) \rightarrow$   
 $(S \rightarrow T) \rightarrow$   
 $(T \rightarrow U) \rightarrow$   
 $P \rightarrow$   
 $U$ .

Proof.

`auto`.

`auto 6`.

Qed.

When searching for potential proofs of the current goal, `auto` considers the hypotheses in the current context together with a *hint database* of other lemmas and constructors. Some common lemmas about equality and logical operators are installed in this hint database by default.

Example `auto_example_4` :  $\forall P Q R : \text{Prop}$ ,

$Q \rightarrow$   
 $(Q \rightarrow R) \rightarrow$   
 $P \vee (Q \wedge R)$ .

Proof. `auto`. Qed.

If we want to see which facts `auto` is using, we can use `info_auto` instead.

Example `auto_example_5`:  $2 = 2$ .

Proof.

```
info_auto.
Qed.
```

We can extend the hint database just for the purposes of one application of `auto` by writing “`auto using ...`”.

```
Lemma le_antisym : ∀ n m: nat, (n ≤ m ∧ m ≤ n) → n = m.
Proof. intros. lia. Qed.
```

```
Example auto_example_6 : ∀ n m p : nat,
  (n ≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
```

```
Proof.
  auto using le_antisym.
Qed.
```

Of course, in any given development there will probably be some specific constructors and lemmas that are used very often in proofs. We can add these to the global hint database by writing

```
Hint Resolve T : core.
```

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 : core.
```

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 : core.
```

where  $d$  is a defined symbol, so that `auto` knows to expand uses of  $d$ , thus enabling further possibilities for applying lemmas that it knows about.

It is also possible to define specialized hint databases that can be activated only when needed. See the Coq reference manual for details.

```
Hint Resolve le_antisym : core.
```

```
Example auto_example_6' : ∀ n m p : nat,
  (n ≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
```

```
Proof.
  auto. Qed.
```

```
Definition is_fortytwo x := (x = 42).
```

```
Example auto_example_7: ∀ x,
  (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
```

```
Proof.
  auto. Abort.
```

Hint `Unfold is_fortytwo : core.`

Example `auto_example_7' :  $\forall x,$   
 $(x \leq 42 \wedge 42 \leq x) \rightarrow \text{is\_fortytwo } x.$`

Proof.

`auto. Qed.`

Let's take a first pass over `ceval_deterministic` to simplify the proof script.

Theorem `ceval_deterministic' :  $\forall c \ st \ st1 \ st2,$`

`$st = [c] => st1 \rightarrow$`

`$st = [c] => st2 \rightarrow$`

`$st1 = st2.$`

Proof.

`intros c st st1 st2 E1 E2.`

`generalize dependent st2;`

`induction E1; intros st2 E2; inversion E2; subst; auto.`

-

`rewrite (IHE1_1 st'0 H1) in *.`

`auto.`

-

+

`rewrite H in H5. discriminate.`

-

+

`rewrite H in H5. discriminate.`

-

+

`rewrite H in H2. discriminate.`

-

`rewrite H in H4. discriminate.`

-

`rewrite (IHE1_1 st'0 H3) in *.`

`auto.`

`Qed.`

When we are using a particular tactic many times in a proof, we can use a variant of the `Proof` command to make that tactic into a default within the proof. Saying `Proof with t` (where `t` is an arbitrary tactic) allows us to use `t1...` as a shorthand for `t1;t` within the proof. As an illustration, here is an alternate version of the previous proof, using `Proof with auto`.

Theorem `ceval_deterministic'_alt :  $\forall c \ st \ st1 \ st2,$`

`$st = [c] => st1 \rightarrow$`

`$st = [c] => st2 \rightarrow$`

`$st1 = st2.$`



Proof with auto.

```

intros c st st1 st2 E1 E2;
generalize dependent st2;
induction E1;
    intros st2 E2; inversion E2; subst...
-
  rewrite (IHE1_1 st'0 H1) in *...
-
  +
    rewrite H in H5. discriminate.
-
  +
    rewrite H in H5. discriminate.
-
  +
    rewrite H in H2. discriminate.
-
  rewrite H in H4. discriminate.
-
  rewrite (IHE1_1 st'0 H3) in *...

```

Qed.

## 17.3 Searching For Hypotheses

The proof has become simpler, but there is still an annoying amount of repetition. Let's start by tackling the contradiction cases. Each of them occurs in a situation where we have both

H1: beval st b = false

and

H2: beval st b = true

as hypotheses. The contradiction is evident, but demonstrating it is a little complicated: we have to locate the two hypotheses *H1* and *H2* and do a `rewrite` following by a `discriminate`. We'd like to automate this process.

(In fact, Coq has a built-in tactic `congruence` that will do the job in this case. But we'll ignore the existence of this tactic for now, in order to demonstrate how to build forward search tactics by hand.)

As a first step, we can abstract out the piece of script in question by writing a little function in Ltac.

```
Ltac rwd H1 H2 := rewrite H1 in H2; discriminate.
```

Theorem ceval\_deterministic':  $\forall c \ st \ st1 \ st2,$

$st = [c] \Rightarrow st1 \rightarrow$

```

    st = [ c ] => st2 →
    st1 = st2.

```

Proof.

```

intros c st st1 st2 E1 E2.
generalize dependent st2;
induction E1; intros st2 E2; inversion E2; subst; auto.
-
  rewrite (IHE1_1 st'0 H1) in *.
  auto.
-
  +
    rwd H H5.
-
  +
    rwd H H5.
-
  +
    rwd H H2.
-
  rwd H H4.
-
  rewrite (IHE1_1 st'0 H3) in *.
  auto. Qed.

```

That was a bit better, but we really want Coq to discover the relevant hypotheses for us. We can do this by using the `match goal` facility of Ltac.

```

Ltac find_rwd :=
  match goal with
    H1: ?E = true,
    H2: ?E = false
  ⊢ _ ⇒ rwd H1 H2
end.

```

This `match goal` looks for two distinct hypotheses that have the form of equalities, with the same arbitrary expression  $E$  on the left and with conflicting boolean values on the right. If such hypotheses are found, it binds  $H1$  and  $H2$  to their names and applies the `rwd` tactic to  $H1$  and  $H2$ .

Adding this tactic to the ones that we invoke in each case of the induction handles all of the contradictory cases.

```

Theorem ceval_deterministic''': ∀ c st st1 st2,
  st = [ c ] => st1 →
  st = [ c ] => st2 →
  st1 = st2.

```

Proof.

```

intros c st st1 st2 E1 E2.
generalize dependent st2;
induction E1; intros st2 E2; inversion E2; subst; try find_rwd; auto.
-
  rewrite (IHE1_1 st'0 H1) in *.
  auto.
-
+
  rewrite (IHE1_1 st'0 H3) in *.
  auto. Qed.

```

Let's see about the remaining cases. Each of them involves applying rewriting an hypothesis after feeding it with the required condition. We can automate the task of finding the relevant hypotheses to rewrite with.

```

Ltac find_eqn :=
  match goal with
    H1:  $\forall x, ?P\ x \rightarrow ?L = ?R$ ,
    H2:  $?P\ ?X$ 
  | _ => rewrite (H1 X H2) in *
end.

```

The pattern  $\forall x, ?P\ x \rightarrow ?L = ?R$  matches any hypothesis of the form “for all  $x$ , *some property of  $x$*  implies *some equality*.” The property of  $x$  is bound to the pattern variable  $P$ , and the left- and right-hand sides of the equality are bound to  $L$  and  $R$ . The name of this hypothesis is bound to  $H1$ . Then the pattern  $?P\ ?X$  matches any hypothesis that provides evidence that  $P$  holds for some concrete  $X$ . If both patterns succeed, we apply the `rewrite` tactic (instantiating the quantified  $x$  with  $X$  and providing  $H2$  as the required evidence for  $P\ X$ ) in all hypotheses and the goal.

```

Theorem ceval_deterministic''':  $\forall c\ st\ st1\ st2$ ,
   $st = [c] \Rightarrow st1 \rightarrow$ 
   $st = [c] \Rightarrow st2 \rightarrow$ 
   $st1 = st2$ .

```

Proof.

```

intros c st st1 st2 E1 E2.
generalize dependent st2;
induction E1; intros st2 E2; inversion E2; subst; try find_rwd;
  try find_eqn; auto.

```

Qed.

The big payoff in this approach is that our proof script should be more robust in the face of modest changes to our language. To test this, let's try adding a *REPEAT* command to the language.

Module REPEAT.

```

Inductive com : Type :=
| CSkip
| CAss (x : string) (a : aexp)
| CSeq (c1 c2 : com)
| Clf (b : bexp) (c1 c2 : com)
| CWhile (b : bexp) (c : com)
| CRepeat (c : com) (b : bexp).

```

*REPEAT* behaves like *while*, except that the loop guard is checked *after* each execution of the body, with the loop repeating as long as the guard stays *false*. Because of this, the body will always execute at least once.

```

Notation "'repeat' x 'until' y 'end'" :=
  (CRepeat x y)
  (in custom com at level 0,
   x at level 99, y at level 99).

```

```

Notation "'skip'" :=
  CSkip (in custom com at level 0).

```

```

Notation "x := y" :=
  (CAss x y)
  (in custom com at level 0, x constr at level 0,
   y at level 85, no associativity).

```

```

Notation "x ; y" :=
  (CSeq x y)
  (in custom com at level 90, right associativity).

```

```

Notation "'if' x 'then' y 'else' z 'end'" :=
  (Clf x y z)
  (in custom com at level 89, x at level 99,
   y at level 99, z at level 99).

```

```

Notation "'while' x 'do' y 'end'" :=
  (CWhile x y)
  (in custom com at level 89, x at level 99, y at level 99).

```

```

Reserved Notation "st '[ c ]=> st'"
  (at level 40, c custom com at level 99, st' constr at next level).

```

```

Inductive ceval : com → state → state → Prop :=

```

```

| E_Skip : ∀ st,
  st = [ skip ] => st
| E_Ass : ∀ st a1 n x,
  aeval st a1 = n →
  st = [ x := a1 ] => (x !-> n ; st)
| E_Seq : ∀ c1 c2 st st' st'',
  st = [ c1 ] => st' →
  st' = [ c2 ] => st'' →

```

```

      st = [ c1 ; c2 ] => st''
| E_IfTrue : ∀ st st' b c1 c2,
  beval st b = true →
  st = [ c1 ] => st' →
  st = [ if b then c1 else c2 end ] => st'
| E_IfFalse : ∀ st st' b c1 c2,
  beval st b = false →
  st = [ c2 ] => st' →
  st = [ if b then c1 else c2 end ] => st'
| E_WhileFalse : ∀ b st c,
  beval st b = false →
  st = [ while b do c end ] => st
| E_WhileTrue : ∀ st st' st'' b c,
  beval st b = true →
  st = [ c ] => st' →
  st' = [ while b do c end ] => st'' →
  st = [ while b do c end ] => st''
| E_RepeatEnd : ∀ st st' b c,
  st = [ c ] => st' →
  beval st' b = true →
  st = [ repeat c until b end ] => st'
| E_RepeatLoop : ∀ st st' st'' b c,
  st = [ c ] => st' →
  beval st' b = false →
  st' = [ repeat c until b end ] => st'' →
  st = [ repeat c until b end ] => st''

```

where "st = [ c ] => st'" := (ceval c st st').

Our first attempt at the determinacy proof does not quite succeed: the `E_RepeatEnd` and `E_RepeatLoop` cases are not handled by our previous automation.

**Theorem** `ceval_deterministic`:  $\forall c \ st \ st1 \ st2,$

```

  st = [ c ] => st1 →
  st = [ c ] => st2 →
  st1 = st2.

```

**Proof.**

```

intros c st st1 st2 E1 E2.
generalize dependent st2;
induction E1;
  intros st2 E2; inversion E2; subst; try find_rwd; try find_eqn; auto.
-
  +
    find_rwd.

```

```
-
+
  find_rwd.
```

Qed.

Fortunately, to fix this, we just have to swap the invocations of *find\_eqn* and *find\_rwd*.

Theorem *ceval\_deterministic*:  $\forall c\ st\ st1\ st2,$

```
  st =[ c ]=> st1 →
  st =[ c ]=> st2 →
  st1 = st2.
```

Proof.

```
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1;
    intros st2 E2; inversion E2; subst; try find_eqn; try find_rwd; auto.
```

Qed.

End REPEAT.

These examples just give a flavor of what “hyper-automation” can achieve in Coq. The details of *match goal* are a bit tricky (and debugging scripts using it is, frankly, not very pleasant). But it is well worth adding at least simple uses to your proofs, both to avoid tedium and to “future proof” them.

## 17.4 Tactics *eapply* and *eauto*

To close the chapter, we’ll introduce one more convenient feature of Coq: its ability to delay instantiation of quantifiers. To motivate this feature, recall this example from the *Imp* chapter:

Example *ceval\_example1*:

```
empty_st = [
  X := 2;
  if (X ≤ 1)
    then Y := 3
    else Z := 4
  end
] => (Z !-> 4 ; X !-> 2).
```

Proof.

```
  apply E_Seq with (X !-> 2).
  - apply E_Ass. reflexivity.
  - apply E_IfFalse. reflexivity. apply E_Ass. reflexivity.
```

Qed.

In the first step of the proof, we had to explicitly provide a longish expression to help

Coq instantiate a “hidden” argument to the `E_Seq` constructor. This was needed because the definition of `E_Seq`...

`E_Seq : forall c1 c2 st st' st'', st = c1 => st' -> st' = c2 => st'' -> st = c1 ;; c2 => st''` is quantified over a variable, `st'`, that does not appear in its conclusion, so unifying its conclusion with the goal state doesn't help Coq find a suitable value for this variable. If we leave out the `with`, this step fails (“Error: Unable to find an instance for the variable `st''`”).

What's silly about this error is that the appropriate value for `st'` will actually become obvious in the very next step, where we apply `E_Ass`. If Coq could just wait until we get to this step, there would be no need to give the value explicitly. This is exactly what the `eapply` tactic gives us:

Example `ceval'_example1`:

```
empty_st = [
  X := 2;
  if (X ≤ 1)
    then Y := 3
    else Z := 4
  end
] => (Z !-> 4 ; X !-> 2).
```

Proof.

```
eapply E_Seq.    - apply E_Ass.      reflexivity.    - apply E_IfFalse. reflexivity.
apply E_Ass. reflexivity.
Qed.
```

The `eapply H` tactic behaves just like `apply H` except that, after it finishes unifying the goal state with the conclusion of `H`, it does not bother to check whether all the variables that were introduced in the process have been given concrete values during unification.

If you step through the proof above, you'll see that the goal state at position 1 mentions the *existential variable* `?st'` in both of the generated subgoals. The next step (which gets us to position 2) replaces `?st'` with a concrete value. This new value contains a new existential variable `?n`, which is instantiated in its turn by the following `reflexivity` step, position 3. When we start working on the second subgoal (position 4), we observe that the occurrence of `?st'` in this subgoal has been replaced by the value that it was given during the first subgoal.

Several of the tactics that we've seen so far, including `∃`, `constructor`, and `auto`, have similar variants. The `eauto` tactic works like `auto`, except that it uses `eapply` instead of `apply`. Tactic `info eauto` shows us which tactics `eauto` uses in its proof search.

Below is an example of `eauto`. Before using it, we need to give some hints to `auto` about using the constructors of `ceval` and the definitions of `state` and `total_map` as part of its proof search.

Hint Constructors `ceval` : *core*.

Hint Transparent state total\_map : *core*.

```
Example eauto_example : ∃ s',
  (Y !-> 1 ; X !-> 2) = [
```

```

      if (X ≤ Y)
        then Z := Y - X
        else Y := X + Z
      end
    ]=> s'.
Proof. info_eauto. Qed.

```

The `eauto` tactic works just like `auto`, except that it uses `eapply` instead of `apply`; `info_eauto` shows us which facts `eauto` uses.

Pro tip: One might think that, since `eapply` and `eauto` are more powerful than `apply` and `auto`, we should just use them all the time. Unfortunately, they are also significantly slower especially `eauto`. Coq experts tend to use `apply` and `auto` most of the time, only switching to the `e` variants when the ordinary variants don't do the job.

## 17.5 Constraints on Existential Variables

In order for `Qed` to succeed, all existential variables need to be determined by the end of the proof. Otherwise Coq will (rightly) refuse to accept the proof. Remember that the Coq tactics build proof objects, and proof objects containing existential variables are not complete.

```

Lemma silly1 : ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∀ x y : nat, P x y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.

```

```

Proof.
  intros P Q HP HQ. eapply HQ. apply HP.
Fail Qed.

```

Coq gives a warning after `apply HP`. (“All the remaining goals are on the shelf,” means that we’ve finished all our top-level proof obligations but along the way we’ve put some aside to be done later, and we have not finished those.) Trying to close the proof with `Qed` gives an error. `Abort`.

An additional constraint is that existential variables cannot be instantiated with terms containing ordinary variables that did not exist at the time the existential variable was created. (The reason for this technical restriction is that allowing such instantiation would lead to inconsistency of Coq’s logic.)

```

Lemma silly2 :
  ∀ (P : nat → nat → Prop) (Q : nat → Prop),
  (∃ y, P 42 y) →
  (∀ x y : nat, P x y → Q x) →
  Q 42.

```

```

Proof.
  intros P Q HP HQ. eapply HQ. destruct HP as [y HP'].

```



*Fail* `apply HP'`.

The error we get, with some details elided, is:  
cannot instantiate “?y” because “y” is not in its scope

In this case there is an easy fix: doing `destruct HP` *before* doing `eapply HQ`. `Abort`.

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.

The `apply HP'` in the last step unifies the existential variable in the goal with the variable `y`.

Note that the `assumption` tactic doesn't work in this case, since it cannot handle existential variables. However, Coq also provides an *eassumption* tactic that solves the goal if one of the premises matches the goal up to instantiations of existential variables. We can use it instead of `apply HP'` if we like.

Lemma `silly2_eassumption` : ∀ (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.

The `eauto` tactic will use `eapply` and *eassumption*, streamlining the proof even further.

Lemma `silly2_eauto` : ∀ (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']. eauto.
```

Qed.

# Chapter 18

## Library `LF.AltAuto`

### 18.1 `AltAuto`: More Automation

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Lia
                        Arith.
From LF Require Import IndProp.
```

Up to now, we’ve used the more manual part of Coq’s tactic facilities. In this chapter, we’ll learn more about some of Coq’s powerful automation features.

As a simple illustration of the benefits of automation, let’s consider another problem on regular expressions, which we formalized in `IndProp`. A given set of strings can be denoted by many different regular expressions. For example, `App EmptyString re` matches exactly the same strings as `re`. We can write a function that “optimizes” any regular expression into a potentially simpler one by applying this fact throughout the r.e. (Note that, for simplicity, the function does not optimize expressions that arise as the result of other optimizations.)

```
Fixpoint re_opt_e {T:Type} (re: reg_exp T) : reg_exp T :=
  match re with
  | App EmptyStr re2 => re_opt_e re2
  | App re1 re2 => App (re_opt_e re1) (re_opt_e re2)
  | Union re1 re2 => Union (re_opt_e re1) (re_opt_e re2)
  | Star re => Star (re_opt_e re)
  | _ => re
end.
```

We would like to show the equivalence of re’s with their “optimized” form. One direction of this equivalence looks like this (the other is similar).

```
Lemma re_opt_e_match : ∀ T (re: reg_exp T) s,
  s =~ re → s =~ re_opt_e re.
```

Proof.

```
  intros T re s M.
```

```

induction M
  as [| x'
      | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
      | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
      | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2].
- simpl. apply MEmpty.
- simpl. apply MChar.
- simpl.
  destruct re1.
  + apply MApp. apply IH1. apply IH2.
  + inversion Hmatch1. simpl. apply IH2.
  + apply MApp. apply IH1. apply IH2.
  + apply MApp. apply IH1. apply IH2.
  + apply MApp. apply IH1. apply IH2.
  + apply MApp. apply IH1. apply IH2.
- simpl. apply MUnionL. apply IH.
- simpl. apply MUnionR. apply IH.
- simpl. apply MStar0.
- simpl. apply MStarApp. apply IH1. apply IH2.
Qed.

```

## 18.2 Coq Automation

The amount of repetition in this last proof is rather annoying. And if we wanted to extend the optimization function to handle other, similar, rewriting opportunities, it would start 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.

## 18.3 Tacticals

*Tacticals* is Coq’s term for tactics that take other tactics as arguments – “higher-order tactics,” if you will.

### The `try` Tactical

If  $T$  is a tactic, then `try  $T$`  is a tactic that is just like  $T$  except that, if  $T$  fails, `try  $T$`  *successfully* does nothing at all (instead of failing).

Theorem silly1 :  $\forall n, 1 + n = S\ n$ .

Proof. try reflexivity. Qed.

Theorem silly2 :  $\forall (P : \text{Prop}), P \rightarrow P$ .

Proof.

intros  $P\ HP$ .

try reflexivity. apply  $HP$ . Qed.

There is no real reason to use `try` in completely manual proofs like these, but it is very useful for doing automated proofs in conjunction with the `; tactical`, which we show next.

## The ; Tactical (Simple Form)

In its most common form, the `; tactical` takes two tactics as arguments. The compound tactic  $T;T'$  first performs  $T$  and then performs  $T'$  on *each subgoal* generated by  $T$ .

For example, consider the following trivial lemma:

Lemma foo :  $\forall n, 0 \leq n = \text{true}$ .

Proof.

intros.

destruct  $n$  eqn: $E$ .

- simpl. reflexivity.

- simpl. reflexivity.

Qed.

We can simplify this proof using the `; tactical`:

Lemma foo' :  $\forall n, 0 \leq n = \text{true}$ .

Proof.

intros.

destruct  $n$ ;

simpl;

reflexivity.

Qed.

Using `try` and `; tactical` together, we can get rid of the repetition in the proof that was bothering us a little while ago.

Lemma re\_opt\_e\_match' :  $\forall T\ (re : \text{reg\_exp } T)\ s,$

$s =^{\sim} re \rightarrow s =^{\sim} \text{re\_opt\_e } re$ .

Proof.

intros  $T\ re\ s\ M$ .

induction  $M$

as  $||\ x'$

|  $s1\ re1\ s2\ re2\ Hmatch1\ IH1\ Hmatch2\ IH2$

|  $s1\ re1\ re2\ Hmatch\ IH\ |\ re1\ s2\ re2\ Hmatch\ IH$

```

      | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2];

    simpl.
  - apply MEmpty.
  - apply MChar.
  -
    destruct re1;

    try (apply MApp; try apply IH1; apply IH2).
    inversion Hmatch1. simpl. apply IH2.
  - apply MUnionL. apply IH.
  - apply MUnionR. apply IH.
  - apply MStar0.
  - apply MStarApp. apply IH1. apply IH2.
Qed.

```

### The ; Tactical (General Form)

The ; tactical also has a more general form than the simple  $T;T'$  we've seen above. If  $T$ ,  $T1$ , ...,  $Tn$  are tactics, then

$T; T1 \mid T2 \mid \dots \mid Tn$

is a tactic that first performs  $T$  and then performs  $T1$  on the first subgoal generated by  $T$ , performs  $T2$  on the second subgoal, etc.

So  $T;T'$  is just special notation for the case when all of the  $Ti$ 's are the same tactic; i.e.,  $T;T'$  is shorthand for:

$T; T' \mid T' \mid \dots \mid T'$

Lemma `re_opt_e_match''` :  $\forall T (re: \mathbf{reg\_exp} \ T) \ s,$   
 $s =^{\sim} re \rightarrow s =^{\sim} \mathbf{re\_opt\_e} \ re.$

Proof.

```

intros T re s M.
induction M
as [| x'
   | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
   | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
   | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2];

    simpl.
  - apply MEmpty.
  - apply MChar.
  -
    destruct re1;

    try (apply MApp; [apply IH1 | apply IH2]).      inversion Hmatch1. simpl. apply

```

*IH2.*

- apply MUnionL. apply *IH*.
- apply MUnionR. apply *IH*.
- apply MStar0.
- apply MStarApp; [apply *IH1* | apply *IH2*]. Qed.

## The repeat Tactical

The `repeat` tactical takes another tactic and keeps applying this tactic until it fails. Here is an example showing that 10 is in a long list using `repeat`.

Theorem `ln10` : `ln 10 [1;2;3;4;5;6;7;8;9;10]`.

Proof.

```
repeat (try (left; reflexivity); right).
```

Qed.

The tactic `repeat T` never fails: if the tactic `T` doesn't apply to the original goal, then `repeat` still succeeds without changing the original goal (i.e., it repeats zero times).

Theorem `ln10'` : `ln 10 [1;2;3;4;5;6;7;8;9;10]`.

Proof.

```
repeat (left; reflexivity).
repeat (right; try (left; reflexivity)).
```

Qed.

The tactic `repeat T` also does not have any upper bound on the number of times it applies `T`. If `T` is a tactic that always succeeds, then `repeat T` will loop forever (e.g., `repeat simpl` loops, since `simpl` always succeeds). While evaluation in Coq's term language, Gallina, is guaranteed to terminate, tactic evaluation is not! This does not affect Coq's logical consistency, however, since the job of `repeat` and other tactics is to guide Coq in constructing proofs; if the construction process diverges, this simply means that we have failed to construct a proof, not that we have constructed a wrong one.

**Exercise: 3 stars, standard (re\_opt)** Consider this more powerful version of the regular expression optimizer.

```
Fixpoint re_opt {T:Type} (re: reg_exp T) : reg_exp T :=
  match re with
  | App re1 EmptySet => EmptySet
  | App EmptyStr re2 => re_opt re2
  | App re1 EmptyStr => re_opt re1
  | App re1 re2 => App (re_opt re1) (re_opt re2)
  | Union EmptySet re2 => re_opt re2
  | Union re1 EmptySet => re_opt re1
  | Union re1 re2 => Union (re_opt re1) (re_opt re2)
  | Star EmptySet => EmptyStr
```

```

| Star EmptyStr  $\Rightarrow$  EmptyStr
| Star  $re \Rightarrow$  Star (re_opt re)
| EmptySet  $\Rightarrow$  EmptySet
| EmptyStr  $\Rightarrow$  EmptyStr
| Char  $x \Rightarrow$  Char  $x$ 
end.

```

Lemma re\_opt\_match :  $\forall T (re: \mathbf{reg\_exp} T) s,$   
 $s =^{\sim} re \rightarrow s =^{\sim} re\_opt re.$

Proof.

```

intros T re s M.
induction M
as [| x'
   | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
   | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
   | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2].
- simpl. apply MEmpty.
- simpl. apply MChar.
- simpl.
  destruct re1.
  + inversion IH1.
  + inversion IH1. simpl. destruct re2.
     $\times$  apply IH2.
     $\times$  apply IH2.
     $\times$  apply IH2.
     $\times$  apply IH2.
     $\times$  apply IH2.
     $\times$  apply IH2.
  + destruct re2.
     $\times$  inversion IH2.
     $\times$  inversion IH2. rewrite app_nil_r. apply IH1.
     $\times$  apply MApp. apply IH1. apply IH2.
     $\times$  apply MApp. apply IH1. apply IH2.
     $\times$  apply MApp. apply IH1. apply IH2.
     $\times$  apply MApp. apply IH1. apply IH2.
  + destruct re2.
     $\times$  inversion IH2.
     $\times$  inversion IH2. rewrite app_nil_r. apply IH1.
     $\times$  apply MApp. apply IH1. apply IH2.
     $\times$  apply MApp. apply IH1. apply IH2.
     $\times$  apply MApp. apply IH1. apply IH2.
     $\times$  apply MApp. apply IH1. apply IH2.
  + destruct re2.

```

```

    × inversion IH2.
    × inversion IH2. rewrite app_nil_r. apply IH1.
    × apply MApp. apply IH1. apply IH2.
    × apply MApp. apply IH1. apply IH2.
    × apply MApp. apply IH1. apply IH2.
    × apply MApp. apply IH1. apply IH2.
+ destruct re2.
    × inversion IH2.
    × inversion IH2. rewrite app_nil_r. apply IH1.
    × apply MApp. apply IH1. apply IH2.
    × apply MApp. apply IH1. apply IH2.
    × apply MApp. apply IH1. apply IH2.
    × apply MApp. apply IH1. apply IH2.
- simpl.
  destruct re1.
+ inversion IH.
+ destruct re2.
    × apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
+ destruct re2.
    × apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
+ destruct re2.
    × apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
+ destruct re2.
    × apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.

```



```

    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
+ destruct re2.
    × apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
- simpl.
  destruct re1.
+ apply IH.
+ destruct re2.
    × inversion IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
+ destruct re2.
    × inversion IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
+ destruct re2.
    × inversion IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
+ destruct re2.
    × inversion IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
+ destruct re2.
    × inversion IH.

```

```

      × apply MUnionR. apply IH.
      × apply MUnionR. apply IH.
      × apply MUnionR. apply IH.
      × apply MUnionR. apply IH.
      × apply MUnionR. apply IH.
- simpl.
  destruct re.
  + apply MEmpty.
  + apply MEmpty.
  + apply MStar0.
  + apply MStar0.
  + apply MStar0.
  + simpl.
    destruct re.
    × apply MStar0.
    × apply MStar0.
    × apply MStar0.
    × apply MStar0.
    × apply MStar0.
    × apply MStar0.
- simpl.
  destruct re.
  + inversion IH1.
  + inversion IH1. inversion IH2. apply MEmpty.
  + apply star_app.
    × apply MStar1. apply IH1.
    × apply IH2.
  + apply star_app.
    × apply MStar1. apply IH1.
    × apply IH2.
  + apply star_app.
    × apply MStar1. apply IH1.
    × apply IH2.
  + apply star_app.
    × apply MStar1. apply IH1.
    × apply IH2.

```

Qed.

Lemma `re_opt_match'` :  $\forall T (re: \text{reg\_exp } T) s,$   
 $s \sim re \rightarrow s \sim \text{re\_opt } re.$

Proof.

*Admitted.*

Definition `manual_grade_for_re_opt` : `option (nat×string)` := `None`.

□

### 18.3.1 A Few More Handy Tactics

By the way, here are some miscellaneous tactics that you may find convenient as we continue.

- **clear**  $H$ : Delete hypothesis  $H$  from the context.
- **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  $y$ .
- **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  $e = x$ .

We'll see examples as we go along.

### 18.3.2 Defining New Tactics

Coq also provides several ways of “programming” tactic scripts.

- Coq has a built-in language called **Ltac** with primitives that can examine and modify the proof state. The full details are a bit too complicated to get into here (and it is generally agreed that **Ltac** is not the most beautiful part of Coq's design!), but they can be found in the reference manual and other books on Coq. Simple use cases are not too difficult.
- 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.

Here is a simple **Ltac** example:

```
Ltac impl_and_try c := simpl; try c.
```

This defines a new tactical called *simpl\_and\_try* that takes one tactic  $c$  as an argument and is defined to be equivalent to the tactic **simpl; try**  $c$ . Now writing “*simpl\_and\_try* *reflexivity*.” in a proof will be the same as writing “**simpl; try** *reflexivity*.”

## 18.4 Decision Procedures

*Note to readers:* The `omega` tactic described here is now superseded by a more powerful tactic called `lia`. For the moment, you'll see uses of both in the text.

So far, the automation we have considered has primarily been useful for removing repetition. Another important category of automation consists of built-in decision procedures for specific kinds of problems. There are several of these, but the `omega` tactic is the most important to start with.

### 18.4.1 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 by William Pugh *Pugh* 1991 (in Bib.v).

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 ordering ( $\leq$ ), and
- the logical connectives  $\wedge$ ,  $\vee$ ,  $\neg$ , and  $\rightarrow$ ,

then invoking `omega` will either solve the goal or fail, meaning that the goal is actually false. (If the goal is *not* of this form, `omega` will also fail.)

Note that we needed the import `Require Import Omega` at the top of this file.

Example `silly-presburger-example` :  $\forall m\ n\ o\ p,$

$m + n \leq n + o \wedge o + 3 = p + 3 \rightarrow$

$m \leq p.$

Proof.

`intros. lia.`

Qed.

## 18.5 Search Tactics

Another very important category of automation tactics helps us construct proofs by *searching* for relevant facts. These tactics include the `auto` tactic for backwards reasoning, automated forward reasoning via the `Ltac` hypothesis matching machinery, and deferred instantiation of existential variables using `eapply` and `eauto`. Using these features together with `Ltac`'s scripting facilities will enable us to make our proofs startlingly short! Used properly, they can also make proofs more maintainable and robust to changes in underlying definitions. A deeper treatment of `auto` and `eauto` can be found in the *UseAuto* chapter in *Programming Language Foundations*.

### 18.5.1 The constructor tactic.

A simple first example of a search tactic is `constructor`, which tries 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`.

Example `constructor_example`:  $\forall (n:\text{nat}),$   
    `ev (n+n).`

Proof.

`induction n; simpl.`

`- constructor. - rewrite plus_comm. simpl. constructor. auto.`

Qed.

This saves us from needing to remember the names of our constructors. Warning: if more than one constructor can apply, `constructor` picks the first one (in the order in which they were defined in the `Inductive`) which is not necessarily the one we want!

### 18.5.2 The auto Tactic

Thus far, our proof scripts mostly apply relevant hypotheses or lemmas by name, and one at a time.

Example `auto_example_1`:  $\forall (P\ Q\ R:\text{Prop}),$   
     $(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R.$

Proof.

`intros P Q R H1 H2 H3.`

`apply H2. apply H1. assumption.`

Qed.

The `auto` tactic frees us from this drudgery by *searching* for a sequence of applications that will prove the goal:

Example `auto_example_1'`:  $\forall (P\ Q\ R:\text{Prop}),$   
     $(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R.$

Proof.

`auto.`

Qed.

The `auto` tactic solves goals that are solvable by any combination of

- `intros` and
- `apply` (of hypotheses from the local context, by default).

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 more interesting example showing `auto`’s power:

Example `auto_example_2`:  $\forall P\ Q\ R\ S\ T\ U : \text{Prop},$

```

(P → Q) →
(P → R) →
(T → R) →
(S → T → U) →
((P → Q) → (P → S)) →
T →
P →
U.

```

Proof. auto. Qed.

Proof search could, in principle, take an arbitrarily long time, so there are limits to how far `auto` will search by default.

Example `auto_example_3` :  $\forall (P\ Q\ R\ S\ T\ U : \text{Prop}),$

```

(P → Q) →
(Q → R) →
(R → S) →
(S → T) →
(T → U) →
P →
U.

```

Proof.

auto.

auto 6.

Qed.

When searching for potential proofs of the current goal, `auto` considers the hypotheses in the current context together with a *hint database* of other lemmas and constructors. Some common lemmas about equality and logical operators are installed in this hint database by default.

Example `auto_example_4` :  $\forall P\ Q\ R : \text{Prop},$

```

Q →
(Q → R) →
P ∨ (Q ∧ R).

```

Proof. auto. Qed.

If we want to see which facts `auto` is using, we can use `info_auto` instead.

Example `auto_example_5`:  $2 = 2.$

Proof.

info\_auto.

Qed.

We can extend the hint database just for the purposes of one application of `auto` by writing “`auto using ...`”.

Lemma `le_antisym` :  $\forall\ n\ m : \text{nat}, (n \leq m \wedge m \leq n) \rightarrow n = m.$

Proof. intros. lia. Qed.

```
Example auto_example_6 :  $\forall n\ m\ p : \mathbf{nat},$   
   $(n \leq p \rightarrow (n \leq m \wedge m \leq n)) \rightarrow$   
   $n \leq p \rightarrow$   
   $n = m.$ 
```

Proof.

intros.

auto using le\_antisym.

Qed.

Of course, in any given development there will probably be some specific constructors and lemmas that are used very often in proofs. We can add these to the global hint database by writing

Hint Resolve T : core.

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 : core.

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 : core.

where  $d$  is a defined symbol, so that `auto` knows to expand uses of  $d$ , thus enabling further possibilities for applying lemmas that it knows about.

It is also possible to define specialized hint databases that can be activated only when needed. See the Coq reference manual for more.

Hint Resolve le\_antisym : core.

```
Example auto_example_6' :  $\forall n\ m\ p : \mathbf{nat},$   
   $(n \leq p \rightarrow (n \leq m \wedge m \leq n)) \rightarrow$   
   $n \leq p \rightarrow$   
   $n = m.$ 
```

Proof.

intros.

auto. Qed.

Definition is\_fortytwo  $x := (x = 42).$

```
Example auto_example_7 :  $\forall x,$   
   $(x \leq 42 \wedge 42 \leq x) \rightarrow \text{is\_fortytwo } x.$ 
```

Proof.

auto. Abort.

Hint Unfold is\_fortytwo : core.

```
Example auto_example_7' :  $\forall x,$   
   $(x \leq 42 \wedge 42 \leq x) \rightarrow \text{is\_fortytwo } x.$ 
```

Proof. info\_auto. Qed.

**Exercise: 3 stars, advanced (pumping\_redux)** Use `auto`, `lia`, and any other useful tactics from this chapter to shorten your proof (or the “official” solution proof) of the weak Pumping Lemma exercise from `lndProp`. `Import Pumping`.

Lemma weak\_pumping :  $\forall T (re : \text{reg\_exp } T) s,$

$s =^{\sim} re \rightarrow$   
 $\text{pumping\_constant } re \leq \text{length } s \rightarrow$   
 $\exists s1\ s2\ s3,$   
 $s = s1 ++ s2 ++ s3 \wedge$   
 $s2 \neq [] \wedge$   
 $\forall m, s1 ++ \text{napp } m\ s2 ++ s3 =^{\sim} re.$

Proof.

*Admitted.*

Definition manual\_grade\_for\_pumping\_redux : `option (nat×string)` := `None`.

□

**Exercise: 3 stars, advanced, optional (pumping\_redux\_strong)** Use `auto`, `lia`, and any other useful tactics from this chapter to shorten your proof (or the “official” solution proof) of the stronger Pumping Lemma exercise from `lndProp`. `Import Pumping`.

Lemma pumping :  $\forall T (re : \text{reg\_exp } T) s,$

$s =^{\sim} re \rightarrow$   
 $\text{pumping\_constant } re \leq \text{length } s \rightarrow$   
 $\exists s1\ s2\ s3,$   
 $s = s1 ++ s2 ++ s3 \wedge$   
 $s2 \neq [] \wedge$   
 $\text{length } s1 + \text{length } s2 \leq \text{pumping\_constant } re \wedge$   
 $\forall m, s1 ++ \text{napp } m\ s2 ++ s3 =^{\sim} re.$

Proof.

*Admitted.*

Definition manual\_grade\_for\_pumping\_redux\_strong : `option (nat×string)` := `None`.

□

### 18.5.3 The `eapply` and `eauto` variants

To close the chapter, we’ll introduce one more convenient feature of Coq: its ability to delay instantiation of quantifiers. To motivate this feature, consider again this simple example:

Example trans\_example1:  $\forall a\ b\ c\ d,$

$a \leq b + b \times c \rightarrow$   
 $(1+c)*b \leq d \rightarrow$   
 $a \leq d.$

Proof.

`intros a b c d H1 H2.`  
`apply le_trans with (b+ b×c). + apply H1.`



```
+ simpl in H2. rewrite mult_comm. apply H2.
Qed.
```

In the first step of the proof, we had to explicitly provide a longish expression to help Coq instantiate a “hidden” argument to the `le_trans` constructor. This was needed because the definition of `le_trans`...

```
le_trans : forall m n o : nat, m <= n -> n <= o -> m <= o
```

is quantified over a variable,  $n$ , that does not appear in its conclusion, so unifying its conclusion with the goal state doesn’t help Coq find a suitable value for this variable. If we leave out the `with`, this step fails (“Error: Unable to find an instance for the variable  $n$ ”).

We already know one way to avoid an explicit `with` clause, namely to provide *H1* as the (first) explicit argument to `le_trans`. But here’s another way, using the `eapply` tactic:

```
Example trans_example1: ∀ a b c d,
  a ≤ b + b × c →
  (1+c)*b ≤ d →
  a ≤ d.
```

Proof.

```
intros a b c d H1 H2.
eapply le_trans.    + apply H1.    + simpl in H2. rewrite mult_comm. apply H2.
Qed.
```

The `eapply H` tactic behaves just like `apply H` except that, after it finishes unifying the goal state with the conclusion of *H*, it does not bother to check whether all the variables that were introduced in the process have been given concrete values during unification.

If you step through the proof above, you’ll see that the goal state at position 1 mentions the *existential variable*  $?n$  in both of the generated subgoals. The next step (which gets us to position 2) replaces  $?n$  with a concrete value. When we start working on the second subgoal (position 3), we observe that the occurrence of  $?n$  in this subgoal has been replaced by the value that it was given during the first subgoal.

Several of the tactics that we’ve seen so far, including  $\exists$ , `constructor`, and `auto`, have e... variants. For example, here’s a proof using `eauto`:

```
Example trans_example2: ∀ a b c d,
  a ≤ b + b × c →
  b + b × c ≤ d →
  a ≤ d.
```

Proof.

```
intros a b c d H1 H2.
info_eauto using le_trans.
Qed.
```

The `eauto` tactic works just like `auto`, except that it uses `eapply` instead of `apply`.

Pro tip: One might think that, since `eapply` and `eauto` are more powerful than `apply` and `auto`, it would be a good idea to use them all the time. Unfortunately, they are also significantly slower – especially `eauto`. Coq experts tend to use `apply` and `auto` most of the

time, only switching to the **e** variants when the ordinary variants don't do the job.

# Chapter 19

## Library LF.Postscript

### 19.1 Postscript

Congratulations: We’ve made it to the end!

### 19.2 Looking Back

We’ve covered quite a bit of ground so far. Here’s a quick review...

- *Functional programming*:
  - “declarative” programming style (recursion over immutable data structures, rather than looping over mutable arrays or pointer structures)
  - higher-order functions
  - polymorphism
- *Logic*, the mathematical basis for software engineering:
  - logic calculus
    - ————— ~ —————
  - software engineering mechanical/civil engineering
    - inductively defined sets and relations
    - inductive proofs
    - proof objects
- *Coq*, an industrial-strength proof assistant
  - functional core language

- core tactics
- automation

## 19.3 Looking Forward

If what you've seen so far has whetted your interest, you have two choices for further reading in the *Software Foundations* series:

- *Programming Language Foundations* (volume 2, by a set of authors similar to this book's) covers material that might be found in a graduate course on the theory of programming languages, including Hoare logic, operational semantics, and type systems.
- *Verified Functional Algorithms* (volume 3, by Andrew Appel) builds on the themes of functional programming and program verification in Coq, addressing a range of topics that might be found in a standard data structures course, with an eye to formal verification.

## 19.4 Resources

Here are some other good places to learn more...

- This book includes some optional chapters covering topics that you may find useful. Take a look at the table of contents and the chapter dependency diagram to find them.
- For questions about Coq, the `#coq` area of Stack Overflow (<https://stackoverflow.com/questions/tagged/coq>) is an excellent community resource.
- Here are some great books on functional programming
  - Learn You a Haskell for Great Good, by Miran Lipovaca *Lipovaca* 2011 (in Bib.v).
  - Real World Haskell, by Bryan O'Sullivan, John Goerzen, and Don Stewart *O'Sullivan* 2008 (in Bib.v)
  - ...and many other excellent books on Haskell, OCaml, Scheme, Racket, Scala, F sharp, etc., etc.
- And some further resources for Coq:
  - Certified Programming with Dependent Types, by Adam Chlipala *Chlipala* 2013 (in Bib.v).

- Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, by Yves Bertot and Pierre Casteran *Bertot* 2004 (in Bib.v).
- If you're interested in real-world applications of formal verification to critical software, see the Postscript chapter of *Programming Language Foundations*.
- For applications of Coq in building verified systems, the lectures and course materials for the 2017 DeepSpec Summer School are a great resource. <https://deepspec.org/event/dsss17/index>

# Chapter 20

## Library LF.Bib

### 20.1 Bib: Bibliography

### 20.2 Resources cited in this volume

*Bertot* 2004 Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, by Yves Bertot and Pierre Casteran. Springer-Verlag, 2004. <https://tinyurl.com/z3o7nqu>

*Chlipala* 2013 Certified Programming with Dependent Types, by Adam Chlipala. MIT Press. 2013. <https://tinyurl.com/zqdnyg2>

*Lipovaca* 2011 Learn You a Haskell for Great Good! A Beginner's Guide, by Miran Lipovaca, No Starch Press, April 2011. <http://learnyouahaskell.com>

*O'Sullivan* 2008 Bryan O'Sullivan, John Goerzen, and Don Stewart: Real world Haskell - code you can believe in. O'Reilly 2008. <http://book.realworldhaskell.org>

*Pugh* 1991 Pugh, William. "The Omega test: a fast and practical integer programming algorithm for dependence analysis." Proceedings of the 1991 ACM/IEEE conference on Supercomputing. ACM, 1991. <https://dl.acm.org/citation.cfm?id=125848>

*Wadler* 2015 Philip Wadler. "Propositions as types." Communications of the ACM 58, no. 12 (2015): 75-84. <https://dl.acm.org/citation.cfm?id=2699407>

# Chapter 21

## Library LF.PrefaceTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Preface.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Preface.
Import Check.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```



# Chapter 22

## Library LF.BasicsTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Basics.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Basics.
Import Check.
Goal True.
idtac "----- nandb -----".
```

```

idtac " ".
idtac "#> test_nandb4".
idtac "Possible points: 1".
check_type @test_nandb4 ((nandb true true = false)).
idtac "Assumptions:".
Abort.
Print Assumptions test_nandb4.
Goal True.
idtac " ".

idtac "————- andb3 —————".
idtac " ".

idtac "#> test_andb34".
idtac "Possible points: 1".
check_type @test_andb34 ((andb3 true true false = false)).
idtac "Assumptions:".
Abort.
Print Assumptions test_andb34.
Goal True.
idtac " ".

idtac "————- factorial —————".
idtac " ".

idtac "#> test_factorial2".
idtac "Possible points: 1".
check_type @test_factorial2 ((factorial 5 = 10 × 12)).
idtac "Assumptions:".
Abort.
Print Assumptions test_factorial2.
Goal True.
idtac " ".

idtac "————- ltb —————".
idtac " ".

idtac "#> test_ltb3".
idtac "Possible points: 1".
check_type @test_ltb3 (((4 <? 2) = false)).
idtac "Assumptions:".
Abort.
Print Assumptions test_ltb3.
Goal True.
idtac " ".

idtac "————- plus_id_exercise —————".

```

```

idtac " ".
idtac "#> plus_id_exercise".
idtac "Possible points: 1".
check_type @plus_id_exercise (( $\forall n m o : \text{nat}, n = m \rightarrow m = o \rightarrow n + m = m + o$ )).
idtac "Assumptions:".
Abort.
Print Assumptions plus_id_exercise.
Goal True.
idtac " ".

idtac "————- mult_n_1 —————".
idtac " ".

idtac "#> mult_n_1".
idtac "Possible points: 1".
check_type @mult_n_1 (( $\forall p : \text{nat}, p \times 1 = p$ )).
idtac "Assumptions:".
Abort.
Print Assumptions mult_n_1.
Goal True.
idtac " ".

idtac "————- andb_true_elim2 —————".
idtac " ".

idtac "#> andb_true_elim2".
idtac "Possible points: 2".
check_type @andb_true_elim2 (( $\forall b c : \text{bool}, b \ \&\& \ c = \text{true} \rightarrow c = \text{true}$ )).
idtac "Assumptions:".
Abort.
Print Assumptions andb_true_elim2.
Goal True.
idtac " ".

idtac "————- zero_nbeq_plus_1 —————".
idtac " ".

idtac "#> zero_nbeq_plus_1".
idtac "Possible points: 1".
check_type @zero_nbeq_plus_1 (( $\forall n : \text{nat}, (0 =? n + 1) = \text{false}$ )).
idtac "Assumptions:".
Abort.
Print Assumptions zero_nbeq_plus_1.
Goal True.
idtac " ".

idtac "————- identity_fn_applied_twice —————".

```

```

idtac " ".
idtac "#> identity_fn_applied_twice".
idtac "Possible points: 1".
check_type @identity_fn_applied_twice (
(∀ f : bool → bool,
  (∀ x : bool, f x = x) → ∀ b : bool, f (f b) = b)).
idtac "Assumptions:".
Abort.
Print Assumptions identity_fn_applied_twice.
Goal True.
idtac " ".
idtac "———— negation_fn_applied_twice —————".
idtac " ".
idtac "#> Manually graded: negation_fn_applied_twice".
idtac "Possible points: 1".
print_manual_grade manual_grade_for_negation_fn_applied_twice.
idtac " ".
idtac "———— binary —————".
idtac " ".
idtac "#> test_bin_incr1".
idtac "Possible points: 0.5".
check_type @test_bin_incr1 ((incr (B1 Z) = B0 (B1 Z))).
idtac "Assumptions:".
Abort.
Print Assumptions test_bin_incr1.
Goal True.
idtac " ".
idtac "#> test_bin_incr2".
idtac "Possible points: 0.5".
check_type @test_bin_incr2 ((incr (B0 (B1 Z)) = B1 (B1 Z))).
idtac "Assumptions:".
Abort.
Print Assumptions test_bin_incr2.
Goal True.
idtac " ".
idtac "#> test_bin_incr3".
idtac "Possible points: 0.5".
check_type @test_bin_incr3 ((incr (B1 (B1 Z)) = B0 (B0 (B1 Z)))).
idtac "Assumptions:".
Abort.
Print Assumptions test_bin_incr3.

```

```

Goal True.
idtac " ".

idtac "#> test_bin_incr4".
idtac "Possible points: 0.5".
check_type @test_bin_incr4 ((bin_to_nat (B0 (B1 Z)) = 2)).
idtac "Assumptions:".
Abort.
Print Assumptions test_bin_incr4.
Goal True.
idtac " ".

idtac "#> test_bin_incr5".
idtac "Possible points: 0.5".
check_type @test_bin_incr5 ((bin_to_nat (incr (B1 Z)) = 1 + bin_to_nat (B1 Z))).
idtac "Assumptions:".
Abort.
Print Assumptions test_bin_incr5.
Goal True.
idtac " ".

idtac "#> test_bin_incr6".
idtac "Possible points: 0.5".
check_type @test_bin_incr6 ((bin_to_nat (incr (incr (B1 Z))) = 2 + bin_to_nat (B1 Z))).
idtac "Assumptions:".
Abort.
Print Assumptions test_bin_incr6.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 14".
idtac "Max points - advanced: 14".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".

```

```

idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- test_nandb4 -----".
Print Assumptions test_nandb4.
idtac "----- test_andb34 -----".
Print Assumptions test_andb34.
idtac "----- test_factorial2 -----".
Print Assumptions test_factorial2.
idtac "----- test_ltb3 -----".
Print Assumptions test_ltb3.
idtac "----- plus_id_exercise -----".
Print Assumptions plus_id_exercise.
idtac "----- mult_n_1 -----".
Print Assumptions mult_n_1.
idtac "----- andb_true_elim2 -----".
Print Assumptions andb_true_elim2.
idtac "----- zero_nbeq_plus_1 -----".
Print Assumptions zero_nbeq_plus_1.
idtac "----- identity_fn_applied_twice -----".
Print Assumptions identity_fn_applied_twice.
idtac "----- negation_fn_applied_twice -----".
idtac "MANUAL".
idtac "----- test_bin_incr1 -----".
Print Assumptions test_bin_incr1.
idtac "----- test_bin_incr2 -----".
Print Assumptions test_bin_incr2.
idtac "----- test_bin_incr3 -----".
Print Assumptions test_bin_incr3.
idtac "----- test_bin_incr4 -----".
Print Assumptions test_bin_incr4.
idtac "----- test_bin_incr5 -----".
Print Assumptions test_bin_incr5.
idtac "----- test_bin_incr6 -----".
Print Assumptions test_bin_incr6.
idtac "".
idtac "***** Advanced *****".
Abort.

```

# Chapter 23

## Library LF.InductionTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Induction.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Induction.
Import Check.
Goal True.
idtac "————— basic_induction —————".
```

```

idtac " ".
idtac "#> mult_0_r".
idtac "Possible points: 0.5".
check_type @mult_0_r (( $\forall n : \text{nat}, n \times 0 = 0$ )).
idtac "Assumptions:".
Abort.
Print Assumptions mult_0_r.
Goal True.
idtac " ".

idtac "#> plus_n_Sm".
idtac "Possible points: 0.5".
check_type @plus_n_Sm (( $\forall n m : \text{nat}, S (n + m) = n + S m$ )).
idtac "Assumptions:".
Abort.
Print Assumptions plus_n_Sm.
Goal True.
idtac " ".

idtac "#> plus_comm".
idtac "Possible points: 0.5".
check_type @plus_comm (( $\forall n m : \text{nat}, n + m = m + n$ )).
idtac "Assumptions:".
Abort.
Print Assumptions plus_comm.
Goal True.
idtac " ".

idtac "#> plus_assoc".
idtac "Possible points: 0.5".
check_type @plus_assoc (( $\forall n m p : \text{nat}, n + (m + p) = n + m + p$ )).
idtac "Assumptions:".
Abort.
Print Assumptions plus_assoc.
Goal True.
idtac " ".

idtac "————- double_plus —————".
idtac " ".

idtac "#> double_plus".
idtac "Possible points: 2".
check_type @double_plus (( $\forall n : \text{nat}, \text{double } n = n + n$ )).
idtac "Assumptions:".
Abort.
Print Assumptions double_plus.

```



```

Goal True.
idtac " ".
idtac "————— plus_comm_informal —————".
idtac " ".
idtac "#> Manually graded: plus_comm_informal".
idtac "Advanced".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_plus_comm_informal.
idtac " ".
idtac "————— mult_comm —————".
idtac " ".
idtac "#> mult_comm".
idtac "Possible points: 3".
check_type @mult_comm (( $\forall m n : \text{nat}, m \times n = n \times m$ )).
idtac "Assumptions:".
Abort.
Print Assumptions mult_comm.
Goal True.
idtac " ".
idtac "————— binary_commute —————".
idtac " ".
idtac "#> Manually graded: binary_commute".
idtac "Possible points: 3".
print_manual_grade manual_grade_for_binary_commute.
idtac " ".
idtac "————— binary_inverse —————".
idtac " ".
idtac "#> Manually graded: binary_inverse_a".
idtac "Advanced".
idtac "Possible points: 4".
print_manual_grade manual_grade_for_binary_inverse_a.
idtac " ".
idtac "#> Manually graded: binary_inverse_b".
idtac "Advanced".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_binary_inverse_b.
idtac " ".
idtac "#> Manually graded: binary_inverse_c".
idtac "Advanced".
idtac "Possible points: 4".

```

```

print_manual_grade manual_grade_for_binary_inverse_c.
idtac " ".
idtac " ".
idtac "Max points - standard: 10".
idtac "Max points - advanced: 22".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- mult_0_r -----".
Print Assumptions mult_0_r.
idtac "----- plus_n_Sm -----".
Print Assumptions plus_n_Sm.
idtac "----- plus_comm -----".
Print Assumptions plus_comm.
idtac "----- plus_assoc -----".
Print Assumptions plus_assoc.
idtac "----- double_plus -----".
Print Assumptions double_plus.
idtac "----- mult_comm -----".
Print Assumptions mult_comm.
idtac "----- binary_commute -----".
idtac "MANUAL".
idtac "".
idtac "***** Advanced *****".
idtac "----- plus_comm_informal -----".
idtac "MANUAL".
idtac "----- binary_inverse_a -----".
idtac "MANUAL".

```

```
idtac "----- binary_inverse_b -----".  
idtac "MANUAL".  
idtac "----- binary_inverse_c -----".  
idtac "MANUAL".  
Abort.
```

# Chapter 24

## Library LF.ListsTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Lists.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Lists.
Import Check.
Goal True.
idtac "————— sndfst_is_swap —————".
```

```

idtac " ".
idtac "#> NatList.snd_fst_is_swap".
idtac "Possible points: 1".
check_type @NatList.snd_fst_is_swap (
(∀ p : NatList.natprod,
  NatList.pair (NatList.snd p) (NatList.fst p) = NatList.swap_pair p)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.snd_fst_is_swap.
Goal True.
idtac " ".

idtac "----- list_funs -----".
idtac " ".

idtac "#> NatList.test_nonzeros".
idtac "Possible points: 0.5".
check_type @NatList.test_nonzeros (
(NatList.nonzeros
  (NatList.cons 0
    (NatList.cons 1
      (NatList.cons 0
        (NatList.cons 2
          (NatList.cons 3 (NatList.cons 0 (NatList.cons 0 NatList.nil)))))) =
  NatList.cons 1 (NatList.cons 2 (NatList.cons 3 NatList.nil)))).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_nonzeros.
Goal True.
idtac " ".

idtac "#> NatList.test_oddmembers".
idtac "Possible points: 0.5".
check_type @NatList.test_oddmembers (
(NatList.oddmembers
  (NatList.cons 0
    (NatList.cons 1
      (NatList.cons 0
        (NatList.cons 2
          (NatList.cons 3 (NatList.cons 0 (NatList.cons 0 NatList.nil)))))) =
  NatList.cons 1 (NatList.cons 3 NatList.nil))).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_oddmembers.

```

```

Goal True.
idtac " ".

idtac "#> NatList.test_countoddmembers2".
idtac "Possible points: 0.5".
check_type @NatList.test_countoddmembers2 (
(NatList.countoddmembers
  (NatList.cons 0 (NatList.cons 2 (NatList.cons 4 NatList.nil))) = 0)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_countoddmembers2.
Goal True.
idtac " ".

idtac "#> NatList.test_countoddmembers3".
idtac "Possible points: 0.5".
check_type @NatList.test_countoddmembers3 ((NatList.countoddmembers NatList.nil = 0)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_countoddmembers3.
Goal True.
idtac " ".

idtac "----- alternate -----".
idtac " ".

idtac "#> NatList.test_alternate1".
idtac "Advanced".
idtac "Possible points: 1".
check_type @NatList.test_alternate1 (
(NatList.alternate
  (NatList.cons 1 (NatList.cons 2 (NatList.cons 3 NatList.nil)))
  (NatList.cons 4 (NatList.cons 5 (NatList.cons 6 NatList.nil))) =
  NatList.cons 1
  (NatList.cons 4
    (NatList.cons 2
      (NatList.cons 5 (NatList.cons 3 (NatList.cons 6 NatList.nil))))))).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_alternate1.
Goal True.
idtac " ".

idtac "#> NatList.test_alternate2".
idtac "Advanced".
idtac "Possible points: 1".

```

```

check_type @NatList.test_alternate2 (
(NatList.alternate (NatList.cons 1 NatList.nil)
  (NatList.cons 4 (NatList.cons 5 (NatList.cons 6 NatList.nil)))) =
  NatList.cons 1
  (NatList.cons 4 (NatList.cons 5 (NatList.cons 6 NatList.nil)))).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_alternate2.
Goal True.
idtac " ".

idtac "#> NatList.test_alternate4".
idtac "Advanced".
idtac "Possible points: 1".
check_type @NatList.test_alternate4 (
(NatList.alternate NatList.nil
  (NatList.cons 20 (NatList.cons 30 NatList.nil)) =
  NatList.cons 20 (NatList.cons 30 NatList.nil))).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_alternate4.
Goal True.
idtac " ".

idtac "————- bag_functions —————".
idtac " ".

idtac "#> NatList.test_count2".
idtac "Possible points: 0.5".
check_type @NatList.test_count2 (
(NatList.count 6
  (NatList.cons 1
    (NatList.cons 2
      (NatList.cons 3
        (NatList.cons 1 (NatList.cons 4 (NatList.cons 1 NatList.nil)))))) =
  0)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_count2.
Goal True.
idtac " ".

idtac "#> NatList.test_sum1".
idtac "Possible points: 0.5".
check_type @NatList.test_sum1 (

```

```

(NatList.count 1
  (NatList.sum
    (NatList.cons 1 (NatList.cons 2 (NatList.cons 3 NatList.nil)))
    (NatList.cons 1 (NatList.cons 4 (NatList.cons 1 NatList.nil)))) = 3)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_sum1.
Goal True.
idtac " ".

idtac "#> NatList.test_add1".
idtac "Possible points: 0.5".
check_type @NatList.test_add1 (
  (NatList.count 1
    (NatList.add 1
      (NatList.cons 1 (NatList.cons 4 (NatList.cons 1 NatList.nil)))) = 3)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_add1.
Goal True.
idtac " ".

idtac "#> NatList.test_add2".
idtac "Possible points: 0.5".
check_type @NatList.test_add2 (
  (NatList.count 5
    (NatList.add 1
      (NatList.cons 1 (NatList.cons 4 (NatList.cons 1 NatList.nil)))) = 0)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_add2.
Goal True.
idtac " ".

idtac "#> NatList.test_member1".
idtac "Possible points: 0.5".
check_type @NatList.test_member1 (
  (NatList.member 1
    (NatList.cons 1 (NatList.cons 4 (NatList.cons 1 NatList.nil)))) = true)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_member1.
Goal True.
idtac " ".

```



```

idtac "#> NatList.test_member2".
idtac "Possible points: 0.5".
check_type @NatList.test_member2 (
(NatList.member 2
  (NatList.cons 1 (NatList.cons 4 (NatList.cons 1 NatList.nil))) = false)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_member2.
Goal True.
idtac " ".

idtac "----- add_inc_count -----".
idtac " ".

idtac "#> Manually graded: NatList.add_inc_count".
idtac "Possible points: 2".
print_manual_grade NatList.manual_grade_for_add_inc_count.
idtac " ".

idtac "----- list_exercises -----".
idtac " ".

idtac "#> NatList.app_nil_r".
idtac "Possible points: 0.5".
check_type @NatList.app_nil_r (
(∀ l : NatList.natlist, NatList.app l NatList.nil = l)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.app_nil_r.
Goal True.
idtac " ".

idtac "#> NatList.rev_app_distr".
idtac "Possible points: 0.5".
check_type @NatList.rev_app_distr (
(∀ l1 l2 : NatList.natlist,
  NatList.rev (NatList.app l1 l2) =
  NatList.app (NatList.rev l2) (NatList.rev l1))).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.rev_app_distr.
Goal True.
idtac " ".

idtac "#> NatList.rev_involutive".
idtac "Possible points: 0.5".
check_type @NatList.rev_involutive (

```

```

(∀ l : NatList.natlist, NatList.rev (NatList.rev l) = l)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.rev_involutive.
Goal True.
idtac " ".

idtac "#> NatList.app_assoc4".
idtac "Possible points: 0.5".
check_type @NatList.app_assoc4 (
(∀ l1 l2 l3 l4 : NatList.natlist,
  NatList.app l1 (NatList.app l2 (NatList.app l3 l4)) =
  NatList.app (NatList.app (NatList.app l1 l2) l3) l4)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.app_assoc4.
Goal True.
idtac " ".

idtac "#> NatList.nonzeros_app".
idtac "Possible points: 1".
check_type @NatList.nonzeros_app (
(∀ l1 l2 : NatList.natlist,
  NatList.nonzeros (NatList.app l1 l2) =
  NatList.app (NatList.nonzeros l1) (NatList.nonzeros l2))).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.nonzeros_app.
Goal True.
idtac " ".

idtac "————— eqblist —————".
idtac " ".

idtac "#> NatList.eqblist_refl".
idtac "Possible points: 2".
check_type @NatList.eqblist_refl (
(∀ l : NatList.natlist, true = NatList.eqblist l l)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.eqblist_refl.
Goal True.
idtac " ".

idtac "————— count_member_nonzero —————".
idtac " ".

```

```

idtac "#> NatList.count_member_nonzero".
idtac "Possible points: 1".
check_type @NatList.count_member_nonzero (
(∀ s : NatList.bag, (1 <=? NatList.count 1 (NatList.cons 1 s)) = true)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.count_member_nonzero.
Goal True.
idtac " ".

idtac "————- remove_does_not_increase_count —————".
idtac " ".

idtac "#> NatList.remove_does_not_increase_count".
idtac "Advanced".
idtac "Possible points: 3".
check_type @NatList.remove_does_not_increase_count (
(∀ s : NatList.bag,
(NatList.count 0 (NatList.remove_one 0 s) <=? NatList.count 0 s) = true)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.remove_does_not_increase_count.
Goal True.
idtac " ".

idtac "————- rev_injective —————".
idtac " ".

idtac "#> NatList.rev_injective".
idtac "Advanced".
idtac "Possible points: 6".
check_type @NatList.rev_injective (
(∀ l1 l2 : NatList.natlist, NatList.rev l1 = NatList.rev l2 → l1 = l2)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.rev_injective.
Goal True.
idtac " ".

idtac "————- hd_error —————".
idtac " ".

idtac "#> NatList.test_hd_error1".
idtac "Possible points: 1".
check_type @NatList.test_hd_error1 ((NatList.hd_error NatList.nil = NatList.None)).
idtac "Assumptions:".
Abort.

```

```

Print Assumptions NatList.test_hd_error1.
Goal True.
idtac " ".
idtac "#> NatList.test_hd_error2".
idtac "Possible points: 1".
check_type @NatList.test_hd_error2 (
  (NatList.hd_error (NatList.cons 1 NatList.nil) = NatList.Some 1)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.test_hd_error2.
Goal True.
idtac " ".
idtac "————— eqb_id_refl —————".
idtac " ".
idtac "#> eqb_id_refl".
idtac "Possible points: 1".
check_type @eqb_id_refl (( $\forall x : \text{id}$ , true = eqb_id  $x\ x$ )).
idtac "Assumptions:".
Abort.
Print Assumptions eqb_id_refl.
Goal True.
idtac " ".
idtac "————— update_eq —————".
idtac " ".
idtac "#> PartialMap.update_eq".
idtac "Possible points: 1".
check_type @PartialMap.update_eq (
  ( $\forall (d : \text{PartialMap.partial\_map}) (x : \text{id}) (v : \text{nat})$ ,
    PartialMap.find  $x$  (PartialMap.update  $d\ x\ v$ ) = NatList.Some  $v$ )).
idtac "Assumptions:".
Abort.
Print Assumptions PartialMap.update_eq.
Goal True.
idtac " ".
idtac "————— update_neq —————".
idtac " ".
idtac "#> PartialMap.update_neq".
idtac "Possible points: 1".
check_type @PartialMap.update_neq (
  ( $\forall (d : \text{PartialMap.partial\_map}) (x\ y : \text{id}) (o : \text{nat})$ ,
    eqb_id  $x\ y$  = false  $\rightarrow$ 

```

```

PartialMap.find x (PartialMap.update d y o) = PartialMap.find x d)).
idtac "Assumptions:".
Abort.
Print Assumptions PartialMap.update_neq.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 19".
idtac "Max points - advanced: 31".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- NatList.snd_fst_is_swap -----".
Print Assumptions NatList.snd_fst_is_swap.
idtac "----- NatList.test_nonzeros -----".
Print Assumptions NatList.test_nonzeros.
idtac "----- NatList.test_oddmembers -----".
Print Assumptions NatList.test_oddmembers.
idtac "----- NatList.test_countoddmembers2 -----".
Print Assumptions NatList.test_countoddmembers2.
idtac "----- NatList.test_countoddmembers3 -----".
Print Assumptions NatList.test_countoddmembers3.
idtac "----- NatList.test_count2 -----".
Print Assumptions NatList.test_count2.
idtac "----- NatList.test_sum1 -----".
Print Assumptions NatList.test_sum1.
idtac "----- NatList.test_add1 -----".
Print Assumptions NatList.test_add1.

```

```

idtac "----- NatList.test_add2 -----".
Print Assumptions NatList.test_add2.
idtac "----- NatList.test_member1 -----".
Print Assumptions NatList.test_member1.
idtac "----- NatList.test_member2 -----".
Print Assumptions NatList.test_member2.
idtac "----- add_inc_count -----".
idtac "MANUAL".
idtac "----- NatList.app_nil_r -----".
Print Assumptions NatList.app_nil_r.
idtac "----- NatList.rev_app_distr -----".
Print Assumptions NatList.rev_app_distr.
idtac "----- NatList.rev_involutive -----".
Print Assumptions NatList.rev_involutive.
idtac "----- NatList.app_assoc4 -----".
Print Assumptions NatList.app_assoc4.
idtac "----- NatList.nonzeros_app -----".
Print Assumptions NatList.nonzeros_app.
idtac "----- NatList.eqblist_refl -----".
Print Assumptions NatList.eqblist_refl.
idtac "----- NatList.count_member_nonzero -----".
Print Assumptions NatList.count_member_nonzero.
idtac "----- NatList.test_hd_error1 -----".
Print Assumptions NatList.test_hd_error1.
idtac "----- NatList.test_hd_error2 -----".
Print Assumptions NatList.test_hd_error2.
idtac "----- eqb_id_refl -----".
Print Assumptions eqb_id_refl.
idtac "----- PartialMap.update_eq -----".
Print Assumptions PartialMap.update_eq.
idtac "----- PartialMap.update_neq -----".
Print Assumptions PartialMap.update_neq.
idtac "".
idtac "***** Advanced *****".
idtac "----- NatList.test_alternate1 -----".
Print Assumptions NatList.test_alternate1.
idtac "----- NatList.test_alternate2 -----".
Print Assumptions NatList.test_alternate2.
idtac "----- NatList.test_alternate4 -----".
Print Assumptions NatList.test_alternate4.
idtac "----- NatList.remove_does_not_increase_count -----".
Print Assumptions NatList.remove_does_not_increase_count.

```

```
idtac "————- NatList.rev_injective ————".  
Print Assumptions NatList.rev_injective.  
Abort.
```

# Chapter 25

## Library LF.PolyTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Poly.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  "]"
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Poly.
Import Check.
Goal True.
idtac "————— mumble_grumble —————".
```



```

idtac " ".
idtac "#> Manually graded: mumble_grumble".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_mumble_grumble.
idtac " ".
idtac "————— split —————".
idtac " ".
idtac "#> split".
idtac "Possible points: 1".
check_type @split (( $\forall X Y : \text{Type}, \text{list } (X \times Y) \rightarrow \text{list } X \times \text{list } Y$ )).
idtac "Assumptions:".
Abort.
Print Assumptions split.
Goal True.
idtac " ".
idtac "#> test_split".
idtac "Possible points: 1".
check_type @test_split (
  (@split nat bool [(1, false); (2, false)] = ([1; 2], [false; false]))).
idtac "Assumptions:".
Abort.
Print Assumptions test_split.
Goal True.
idtac " ".
idtac "————— filter_even_gt7 —————".
idtac " ".
idtac "#> test_filter_even_gt7_1".
idtac "Possible points: 1".
check_type @test_filter_even_gt7_1 (
  (filter_even_gt7 [1; 2; 6; 9; 10; 3; 12; 8] = [10; 12; 8])).
idtac "Assumptions:".
Abort.
Print Assumptions test_filter_even_gt7_1.
Goal True.
idtac " ".
idtac "#> test_filter_even_gt7_2".
idtac "Possible points: 1".
check_type @test_filter_even_gt7_2 ((filter_even_gt7 [5; 2; 6; 19; 129] = [ ])).
idtac "Assumptions:".
Abort.
Print Assumptions test_filter_even_gt7_2.

```

```

Goal True.
idtac " ".

idtac "----- partition -----".
idtac " ".

idtac "#> partition".
idtac "Possible points: 1".
check_type @partition (( $\forall X : \text{Type}, (X \rightarrow \text{bool}) \rightarrow \text{list } X \rightarrow \text{list } X \times \text{list } X$ )).
idtac "Assumptions:".
Abort.
Print Assumptions partition.
Goal True.
idtac " ".

idtac "#> test_partition1".
idtac "Possible points: 1".
check_type @test_partition1 ((@partition nat oddb [1; 2; 3; 4; 5] = ([1; 3; 5], [2; 4]))).
idtac "Assumptions:".
Abort.
Print Assumptions test_partition1.
Goal True.
idtac " ".

idtac "#> test_partition2".
idtac "Possible points: 1".
check_type @test_partition2 (
  (@partition nat (fun _ : nat  $\Rightarrow$  false) [5; 9; 0] = ([ ], [5; 9; 0]))).
idtac "Assumptions:".
Abort.
Print Assumptions test_partition2.
Goal True.
idtac " ".

idtac "----- map_rev -----".
idtac " ".

idtac "#> map_rev".
idtac "Possible points: 3".
check_type @map_rev (
  ( $\forall (X \ Y : \text{Type}) (f : X \rightarrow Y) (l : \text{list } X),$ 
    @map X Y f (@rev X l) = @rev Y (@map X Y f l))).
idtac "Assumptions:".
Abort.
Print Assumptions map_rev.
Goal True.
idtac " ".

```

```

idtac "----- flat_map -----".
idtac " ".
idtac "#> flat_map".
idtac "Possible points: 1".
check_type @flat_map (( $\forall X Y : \text{Type}, (X \rightarrow \text{list } Y) \rightarrow \text{list } X \rightarrow \text{list } Y$ )).
idtac "Assumptions:".
Abort.
Print Assumptions flat_map.
Goal True.
idtac " ".
idtac "#> test_flat_map1".
idtac "Possible points: 1".
check_type @test_flat_map1 (
  (@flat_map nat nat (fun n : nat  $\Rightarrow [n; n; n]$ ) [1; 5; 4] =
    [1; 1; 1; 5; 5; 5; 4; 4; 4])).
idtac "Assumptions:".
Abort.
Print Assumptions test_flat_map1.
Goal True.
idtac " ".
idtac "----- fold_types_different -----".
idtac " ".
idtac "#> Manually graded: fold_types_different".
idtac "Advanced".
idtac "Possible points: 1".
print_manual_grade manual_grade_for_fold_types_different.
idtac " ".
idtac "----- fold_length -----".
idtac " ".
idtac "#> Exercises.fold_length_correct".
idtac "Possible points: 2".
check_type @Exercises.fold_length_correct (
  ( $\forall (X : \text{Type}) (l : \text{list } X), @\text{Exercises.fold\_length } X \ l = @\text{length } X \ l$ )).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.fold_length_correct.
Goal True.
idtac " ".
idtac "----- fold_map -----".
idtac " ".
idtac "#> Manually graded: Exercises.fold_map".

```

```

idtac "Possible points: 3".
print_manual_grade Exercises.manual_grade_for_fold_map.
idtac " ".

idtac "----- currying -----".
idtac " ".

idtac "#> Exercises.uncurry_curry".
idtac "Advanced".
idtac "Possible points: 1".
check_type @Exercises.uncurry_curry (
  (∀ (X Y Z : Type) (f : X → Y → Z) (x : X) (y : Y),
    @Exercises.prod_curry X Y Z (@Exercises.prod_uncurry X Y Z f) x y = f x y)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.uncurry_curry.
Goal True.
idtac " ".

idtac "#> Exercises.curry_uncurry".
idtac "Advanced".
idtac "Possible points: 1".
check_type @Exercises.curry_uncurry (
  (∀ (X Y Z : Type) (f : X × Y → Z) (p : X × Y),
    @Exercises.prod_uncurry X Y Z (@Exercises.prod_curry X Y Z f) p = f p)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.curry_uncurry.
Goal True.
idtac " ".

idtac "----- nth_error_informal -----".
idtac " ".

idtac "#> Manually graded: Exercises.informal_proof".
idtac "Advanced".
idtac "Possible points: 2".
print_manual_grade Exercises.manual_grade_for_informal_proof.
idtac " ".

idtac "----- church_succ -----".
idtac " ".

idtac "#> Exercises.Church.succ_2".
idtac "Advanced".
idtac "Possible points: 0.5".
check_type @Exercises.Church.succ_2 (
  (Exercises.Church.succ Exercises.Church.one = Exercises.Church.two)).

```

```

idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.succ_2.
Goal True.
idtac " ".

idtac "#> Exercises.Church.succ_3".
idtac "Advanced".
idtac "Possible points: 0.5".
check_type @Exercises.Church.succ_3 (
(Exercises.Church.succ Exercises.Church.two = Exercises.Church.three)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.succ_3.
Goal True.
idtac " ".

idtac "_____ church_plus _____".
idtac " ".

idtac "#> Exercises.Church.plus_2".
idtac "Advanced".
idtac "Possible points: 0.5".
check_type @Exercises.Church.plus_2 (
(Exercises.Church.plus Exercises.Church.two Exercises.Church.three =
Exercises.Church.plus Exercises.Church.three Exercises.Church.two)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.plus_2.
Goal True.
idtac " ".

idtac "#> Exercises.Church.plus_3".
idtac "Advanced".
idtac "Possible points: 0.5".
check_type @Exercises.Church.plus_3 (
(Exercises.Church.plus
(Exercises.Church.plus Exercises.Church.two Exercises.Church.two)
Exercises.Church.three =
Exercises.Church.plus Exercises.Church.one
(Exercises.Church.plus Exercises.Church.three Exercises.Church.three))).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.plus_3.
Goal True.

```

```

idtac " ".
idtac "----- church_mult -----".
idtac " ".
idtac "#> Exercises.Church.mult_1".
idtac "Advanced".
idtac "Possible points: 0.5".
check_type @Exercises.Church.mult_1 (
(Exercises.Church.mult Exercises.Church.one Exercises.Church.one =
  Exercises.Church.one)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.mult_1.
Goal True.
idtac " ".
idtac "#> Exercises.Church.mult_2".
idtac "Advanced".
idtac "Possible points: 0.5".
check_type @Exercises.Church.mult_2 (
(Exercises.Church.mult Exercises.Church.zero
  (Exercises.Church.plus Exercises.Church.three Exercises.Church.three) =
    Exercises.Church.zero)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.mult_2.
Goal True.
idtac " ".
idtac "#> Exercises.Church.mult_3".
idtac "Advanced".
idtac "Possible points: 1".
check_type @Exercises.Church.mult_3 (
(Exercises.Church.mult Exercises.Church.two Exercises.Church.three =
  Exercises.Church.plus Exercises.Church.three Exercises.Church.three)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.mult_3.
Goal True.
idtac " ".
idtac "----- church_exp -----".
idtac " ".
idtac "#> Exercises.Church.exp_1".
idtac "Advanced".

```

```

idtac "Possible points: 0.5".
check_type @Exercises.Church.exp_1 (
  (Exercises.Church.exp Exercises.Church.two Exercises.Church.two =
    Exercises.Church.plus Exercises.Church.two Exercises.Church.two)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.exp_1.
Goal True.
idtac " ".

idtac "#> Exercises.Church.exp_2".
idtac "Advanced".
idtac "Possible points: 0.5".
check_type @Exercises.Church.exp_2 (
  (Exercises.Church.exp Exercises.Church.three Exercises.Church.zero =
    Exercises.Church.one)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.exp_2.
Goal True.
idtac " ".

idtac "#> Exercises.Church.exp_3".
idtac "Advanced".
idtac "Possible points: 1".
check_type @Exercises.Church.exp_3 (
  (Exercises.Church.exp Exercises.Church.three Exercises.Church.two =
    Exercises.Church.plus
      (Exercises.Church.mult Exercises.Church.two
        (Exercises.Church.mult Exercises.Church.two Exercises.Church.two))
    Exercises.Church.one)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.exp_3.
Goal True.
idtac " ".
idtac " ".

idtac "Max points - standard: 19".
idtac "Max points - advanced: 30".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".

```

```

idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- mumble_grumble -----".
idtac "MANUAL".
idtac "----- split -----".
Print Assumptions split.
idtac "----- test_split -----".
Print Assumptions test_split.
idtac "----- test_filter_even_gt7_1 -----".
Print Assumptions test_filter_even_gt7_1.
idtac "----- test_filter_even_gt7_2 -----".
Print Assumptions test_filter_even_gt7_2.
idtac "----- partition -----".
Print Assumptions partition.
idtac "----- test_partition1 -----".
Print Assumptions test_partition1.
idtac "----- test_partition2 -----".
Print Assumptions test_partition2.
idtac "----- map_rev -----".
Print Assumptions map_rev.
idtac "----- flat_map -----".
Print Assumptions flat_map.
idtac "----- test_flat_map1 -----".
Print Assumptions test_flat_map1.
idtac "----- Exercises.fold_length_correct -----".
Print Assumptions Exercises.fold_length_correct.
idtac "----- fold_map -----".
idtac "MANUAL".
idtac "".
idtac "***** Advanced *****".
idtac "----- fold_types_different -----".

```



```

idtac "MANUAL".
idtac "————- Exercises.uncurry_curry ————".
Print Assumptions Exercises.uncurry_curry.
idtac "————- Exercises.curry_uncurry ————".
Print Assumptions Exercises.curry_uncurry.
idtac "————- informal_proof ————".
idtac "MANUAL".
idtac "————- Exercises.Church.succ_2 ————".
Print Assumptions Exercises.Church.succ_2.
idtac "————- Exercises.Church.succ_3 ————".
Print Assumptions Exercises.Church.succ_3.
idtac "————- Exercises.Church.plus_2 ————".
Print Assumptions Exercises.Church.plus_2.
idtac "————- Exercises.Church.plus_3 ————".
Print Assumptions Exercises.Church.plus_3.
idtac "————- Exercises.Church.mult_1 ————".
Print Assumptions Exercises.Church.mult_1.
idtac "————- Exercises.Church.mult_2 ————".
Print Assumptions Exercises.Church.mult_2.
idtac "————- Exercises.Church.mult_3 ————".
Print Assumptions Exercises.Church.mult_3.
idtac "————- Exercises.Church.exp_1 ————".
Print Assumptions Exercises.Church.exp_1.
idtac "————- Exercises.Church.exp_2 ————".
Print Assumptions Exercises.Church.exp_2.
idtac "————- Exercises.Church.exp_3 ————".
Print Assumptions Exercises.Church.exp_3.
Abort.

```

# Chapter 26

## Library LF.TacticsTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Tactics.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Tactics.
Import Check.
Goal True.
idtac "————— apply_exercise1 —————".
```

```

idtac " ".
idtac "#> rev_exercise1".
idtac "Possible points: 3".
check_type @rev_exercise1 (( $\forall l\ l' : \text{list nat}, l = @\text{rev nat } l' \rightarrow l' = @\text{rev nat } l$ )).
idtac "Assumptions:".
Abort.
Print Assumptions rev_exercise1.
Goal True.
idtac " ".

idtac "———— injection_ex3 —————".
idtac " ".

idtac "#> injection_ex3".
idtac "Possible points: 3".
check_type @injection_ex3 (
( $\forall (X : \text{Type}) (x\ y\ z : X) (l\ j : \text{list } X),$ 
 $x :: y :: l = z :: j \rightarrow j = z :: l \rightarrow x = y$ )).
idtac "Assumptions:".
Abort.
Print Assumptions injection_ex3.
Goal True.
idtac " ".

idtac "———— discriminate_ex3 —————".
idtac " ".

idtac "#> discriminate_ex3".
idtac "Possible points: 1".
check_type @discriminate_ex3 (
( $\forall (X : \text{Type}) (x\ y\ z : X) (l : \text{list } X),$ 
 $\text{list } X \rightarrow x :: y :: l = [ ] \rightarrow x = z$ )).
idtac "Assumptions:".
Abort.
Print Assumptions discriminate_ex3.
Goal True.
idtac " ".

idtac "———— eqb_true —————".
idtac " ".

idtac "#> eqb_true".
idtac "Possible points: 2".
check_type @eqb_true (( $\forall n\ m : \text{nat}, (n =? m) = \text{true} \rightarrow n = m$ )).
idtac "Assumptions:".
Abort.
Print Assumptions eqb_true.

```

```

Goal True.
idtac " ".

idtac "————— eqb_true_informal —————".
idtac " ".

idtac "#> Manually graded: informal_proof".
idtac "Advanced".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_informal_proof.
idtac " ".

idtac "————— plus_n_n_injective —————".
idtac " ".

idtac "#> plus_n_n_injective".
idtac "Possible points: 3".
check_type @plus_n_n_injective (( $\forall n\ m : \text{nat}, n + n = m + m \rightarrow n = m$ )).
idtac "Assumptions:".
Abort.
Print Assumptions plus_n_n_injective.
Goal True.
idtac " ".

idtac "————— gen_dep_practice —————".
idtac " ".

idtac "#> nth_error_after_last".
idtac "Possible points: 3".
check_type @nth_error_after_last (
( $\forall (n : \text{nat}) (X : \text{Type}) (l : \text{list } X),$ 
  @length X l = n  $\rightarrow$  @nth_error X l n = @None X)).
idtac "Assumptions:".
Abort.
Print Assumptions nth_error_after_last.
Goal True.
idtac " ".

idtac "————— combine_split —————".
idtac " ".

idtac "#> combine_split".
idtac "Possible points: 3".
check_type @combine_split (
( $\forall (X\ Y : \text{Type}) (l : \text{list } (X \times Y)) (l1 : \text{list } X) (l2 : \text{list } Y),$ 
  @split X Y l = (l1 , l2)  $\rightarrow$  @combine X Y l1 l2 = l)).
idtac "Assumptions:".
Abort.

```

```

Print Assumptions combine_split.
Goal True.
idtac " ".

idtac "————— destruct_eqn_practice —————".
idtac " ".

idtac "#> bool_fn_applied_thrice".
idtac "Possible points: 2".
check_type @bool_fn_applied_thrice (
  (∀ (f : bool → bool) (b : bool), f (f (f b)) = f b)).
idtac "Assumptions:".
Abort.

Print Assumptions bool_fn_applied_thrice.
Goal True.
idtac " ".

idtac "————— eqb_sym —————".
idtac " ".

idtac "#> eqb_sym".
idtac "Possible points: 3".
check_type @eqb_sym ((∀ n m : nat, (n =? m) = (m =? n))).
idtac "Assumptions:".
Abort.

Print Assumptions eqb_sym.
Goal True.
idtac " ".

idtac "————— split_combine —————".
idtac " ".

idtac "#> Manually graded: split_combine".
idtac "Advanced".
idtac "Possible points: 3".
print_manual_grade manual_grade_for_split_combine.
idtac " ".

idtac "————— filter_exercise —————".
idtac " ".

idtac "#> filter_exercise".
idtac "Advanced".
idtac "Possible points: 3".
check_type @filter_exercise (
  (∀ (X : Type) (test : X → bool) (x : X) (l lf : list X),
    @filter X test l = x :: lf → test x = true)).
idtac "Assumptions:".

```

```

Abort.
Print Assumptions filter_exercise.
Goal True.
idtac " ".

idtac "————- forall_exists_challenge —————".
idtac " ".

idtac "#> existsb_existsb'".
idtac "Advanced".
idtac "Possible points: 6".
check_type @existsb_existsb' (
( $\forall (X : \text{Type}) (test : X \rightarrow \text{bool}) (l : \text{list } X),$ 
   $@existsb\ X\ test\ l = @existsb'\ X\ test\ l)$ ).
idtac "Assumptions:".
Abort.
Print Assumptions existsb_existsb'.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 23".
idtac "Max points - advanced: 37".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "————- rev_exercise1 ————".
Print Assumptions rev_exercise1.
idtac "————- injection_ex3 ————".
Print Assumptions injection_ex3.

```

```

idtac "——- discriminate_ex3 ——".
Print Assumptions discriminate_ex3.
idtac "——- eqb_true ——".
Print Assumptions eqb_true.
idtac "——- plus_n_n_injective ——".
Print Assumptions plus_n_n_injective.
idtac "——- nth_error_after_last ——".
Print Assumptions nth_error_after_last.
idtac "——- combine_split ——".
Print Assumptions combine_split.
idtac "——- bool_fn_applied_thrice ——".
Print Assumptions bool_fn_applied_thrice.
idtac "——- eqb_sym ——".
Print Assumptions eqb_sym.
idtac "".
idtac "***** Advanced *****".
idtac "——- informal_proof ——".
idtac "MANUAL".
idtac "——- split_combine ——".
idtac "MANUAL".
idtac "——- filter_exercise ——".
Print Assumptions filter_exercise.
idtac "——- existsb_existsb' ——".
Print Assumptions existsb_existsb'.
Abort.

```

# Chapter 27

## Library LF.LogicTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Logic.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Logic.
Import Check.
Goal True.
idtac "————— and_exercise —————".
```



```

idtac " ".
idtac "#> and_exercise".
idtac "Possible points: 2".
check_type @and_exercise (( $\forall n m : \text{nat}, n + m = 0 \rightarrow n = 0 \wedge m = 0$ )).
idtac "Assumptions:".
Abort.
Print Assumptions and_exercise.
Goal True.
idtac " ".

idtac "----- and_assoc -----".
idtac " ".

idtac "#> and_assoc".
idtac "Possible points: 2".
check_type @and_assoc (( $\forall P Q R : \text{Prop}, P \wedge Q \wedge R \rightarrow (P \wedge Q) \wedge R$ )).
idtac "Assumptions:".
Abort.
Print Assumptions and_assoc.
Goal True.
idtac " ".

idtac "----- mult_eq_0 -----".
idtac " ".

idtac "#> mult_eq_0".
idtac "Possible points: 1".
check_type @mult_eq_0 (( $\forall n m : \text{nat}, n \times m = 0 \rightarrow n = 0 \vee m = 0$ )).
idtac "Assumptions:".
Abort.
Print Assumptions mult_eq_0.
Goal True.
idtac " ".

idtac "----- or_commut -----".
idtac " ".

idtac "#> or_commut".
idtac "Possible points: 1".
check_type @or_commut (( $\forall P Q : \text{Prop}, P \vee Q \rightarrow Q \vee P$ )).
idtac "Assumptions:".
Abort.
Print Assumptions or_commut.
Goal True.
idtac " ".

idtac "----- double_neg_inf -----".

```

```

idtac " ".
idtac "#> Manually graded: double_neg_inf".
idtac "Advanced".
idtac "Possible points: 2".
print_manual_grade manual_grade_for_double_neg_inf.
idtac " ".
idtac "————- contrapositive —————".
idtac " ".
idtac "#> contrapositive".
idtac "Possible points: 2".
check_type @contrapositive (( $\forall P Q : \text{Prop}, (P \rightarrow Q) \rightarrow \neg Q \rightarrow \neg P$ )).
idtac "Assumptions:".
Abort.
Print Assumptions contrapositive.
Goal True.
idtac " ".
idtac "————- not_both_true_and_false —————".
idtac " ".
idtac "#> not_both_true_and_false".
idtac "Possible points: 1".
check_type @not_both_true_and_false (( $\forall P : \text{Prop}, \neg (P \wedge \neg P)$ )).
idtac "Assumptions:".
Abort.
Print Assumptions not_both_true_and_false.
Goal True.
idtac " ".
idtac "————- informal_not_PNP —————".
idtac " ".
idtac "#> Manually graded: informal_not_PNP".
idtac "Advanced".
idtac "Possible points: 1".
print_manual_grade manual_grade_for_informal_not_PNP.
idtac " ".
idtac "————- or_distributes_over_and —————".
idtac " ".
idtac "#> or_distributes_over_and".
idtac "Possible points: 3".
check_type @or_distributes_over_and (
( $\forall P Q R : \text{Prop}, P \vee Q \wedge R \leftrightarrow (P \vee Q) \wedge (P \vee R)$ )).
idtac "Assumptions:".

```

```

Abort.
Print Assumptions or_distributes_over_and.
Goal True.
idtac " ".
idtac "————— dist_not_exists —————".
idtac " ".
idtac "#> dist_not_exists".
idtac "Possible points: 1".
check_type @dist_not_exists (
  (∀ (X : Type) (P : X → Prop),
    (∀ x : X, P x) → ¬ (∃ x : X, ¬ P x))).
idtac "Assumptions:".
Abort.
Print Assumptions dist_not_exists.
Goal True.
idtac " ".
idtac "————— dist_exists_or —————".
idtac " ".
idtac "#> dist_exists_or".
idtac "Possible points: 2".
check_type @dist_exists_or (
  (∀ (X : Type) (P Q : X → Prop),
    (∃ x : X, P x ∨ Q x) ↔ (∃ x : X, P x) ∨ (∃ x : X, Q x))).
idtac "Assumptions:".
Abort.
Print Assumptions dist_exists_or.
Goal True.
idtac " ".
idtac "————— In_map_iff —————".
idtac " ".
idtac "#> In_map_iff".
idtac "Possible points: 3".
check_type @In_map_iff (
  (∀ (A B : Type) (f : A → B) (l : list A) (y : B),
    @In B y (@map A B f l) ↔ (∃ x : A, f x = y ∧ @In A x l))).
idtac "Assumptions:".
Abort.
Print Assumptions In_map_iff.
Goal True.
idtac " ".
idtac "————— In_app_iff —————".

```

```

idtac " ".
idtac "#> In_app_iff".
idtac "Possible points: 2".
check_type @In_app_iff (
(∀ (A : Type) (l l' : list A) (a : A),
  @In A a (l ++ l') ↔ @In A a l ∨ @In A a l')).
idtac "Assumptions:".
Abort.
Print Assumptions In_app_iff.
Goal True.
idtac " ".
idtac "_____ All _____".
idtac " ".
idtac "#> All_In".
idtac "Possible points: 3".
check_type @All_In (
(∀ (T : Type) (P : T → Prop) (l : list T),
  (∀ x : T, @In T x l → P x) ↔ @All T P l)).
idtac "Assumptions:".
Abort.
Print Assumptions All_In.
Goal True.
idtac " ".
idtac "_____ tr_rev_correct _____".
idtac " ".
idtac "#> tr_rev_correct".
idtac "Possible points: 6".
check_type @tr_rev_correct ((∀ X : Type, @tr_rev X = @rev X)).
idtac "Assumptions:".
Abort.
Print Assumptions tr_rev_correct.
Goal True.
idtac " ".
idtac "_____ evenb_double_conv _____".
idtac " ".
idtac "#> evenb_double_conv".
idtac "Possible points: 3".
check_type @evenb_double_conv (
(∀ n : nat,
  ∃ k : nat, n = (if evenb n then double k else S (double k)))).
idtac "Assumptions:".

```

```

Abort.
Print Assumptions evenb_double_conv.
Goal True.
idtac " ".

idtac "————- logical_connectives —————".
idtac " ".

idtac "#> andb_true_iff".
idtac "Possible points: 1".
check_type @andb_true_iff (
  (∀ b1 b2 : bool, b1 && b2 = true ↔ b1 = true ∧ b2 = true)).
idtac "Assumptions:".
Abort.
Print Assumptions andb_true_iff.
Goal True.
idtac " ".

idtac "#> orb_true_iff".
idtac "Possible points: 1".
check_type @orb_true_iff (
  (∀ b1 b2 : bool, b1 || b2 = true ↔ b1 = true ∨ b2 = true)).
idtac "Assumptions:".
Abort.
Print Assumptions orb_true_iff.
Goal True.
idtac " ".

idtac "————- eqb_neq —————".
idtac " ".

idtac "#> eqb_neq".
idtac "Possible points: 1".
check_type @eqb_neq ((∀ x y : nat, (x =? y) = false ↔ x ≠ y)).
idtac "Assumptions:".
Abort.
Print Assumptions eqb_neq.
Goal True.
idtac " ".

idtac "————- eqb_list —————".
idtac " ".

idtac "#> eqb_list_true_iff".
idtac "Possible points: 3".
check_type @eqb_list_true_iff (
  (∀ (A : Type) (eqb : A → A → bool),
    (∀ a1 a2 : A, eqb a1 a2 = true ↔ a1 = a2) →

```

```

   $\forall l1\ l2 : \text{list } A, @eqb\_list\ A\ eqb\ l1\ l2 = \text{true} \leftrightarrow l1 = l2$ )).
idtac "Assumptions:".
Abort.
Print Assumptions eqb_list_true_iff.
Goal True.
idtac " ".

idtac "----- All_forallb -----".
idtac " ".

idtac "#> forallb_true_iff".
idtac "Possible points: 2".
check_type @forallb_true_iff (
( $\forall (X : \text{Type}) (test : X \rightarrow \text{bool}) (l : \text{list } X),$ 
  @forallb  $X\ test\ l = \text{true} \leftrightarrow @All\ X\ (\text{fun } x : X \Rightarrow test\ x = \text{true})\ l$ )).
idtac "Assumptions:".
Abort.
Print Assumptions forallb_true_iff.
Goal True.
idtac " ".

idtac "----- excluded_middle_irrefutable -----".
idtac " ".

idtac "#> excluded_middle_irrefutable".
idtac "Possible points: 3".
check_type @excluded_middle_irrefutable (( $\forall P : \text{Prop}, \neg \neg (P \vee \neg P)$ )).
idtac "Assumptions:".
Abort.
Print Assumptions excluded_middle_irrefutable.
Goal True.
idtac " ".

idtac "----- not_exists_dist -----".
idtac " ".

idtac "#> not_exists_dist".
idtac "Advanced".
idtac "Possible points: 3".
check_type @not_exists_dist (
(excluded_middle  $\rightarrow$ 
   $\forall (X : \text{Type}) (P : X \rightarrow \text{Prop}),$ 
   $\neg (\exists x : X, \neg P\ x) \rightarrow \forall x : X, P\ x$ )).
idtac "Assumptions:".
Abort.
Print Assumptions not_exists_dist.
Goal True.

```

```

idtac " ".
idtac " ".
idtac "Max points - standard: 43".
idtac "Max points - advanced: 49".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- and_exercise -----".
Print Assumptions and_exercise.
idtac "----- and_assoc -----".
Print Assumptions and_assoc.
idtac "----- mult_eq_0 -----".
Print Assumptions mult_eq_0.
idtac "----- or_commut -----".
Print Assumptions or_commut.
idtac "----- contrapositive -----".
Print Assumptions contrapositive.
idtac "----- not_both_true_and_false -----".
Print Assumptions not_both_true_and_false.
idtac "----- or_distributes_over_and -----".
Print Assumptions or_distributes_over_and.
idtac "----- dist_not_exists -----".
Print Assumptions dist_not_exists.
idtac "----- dist_exists_or -----".
Print Assumptions dist_exists_or.
idtac "----- In_map_iff -----".
Print Assumptions In_map_iff.
idtac "----- In_app_iff -----".

```

```

Print Assumptions ln_app_iff.
idtac "----- All_In -----".
Print Assumptions All_In.
idtac "----- tr_rev_correct -----".
Print Assumptions tr_rev_correct.
idtac "----- evenb_double_conv -----".
Print Assumptions evenb_double_conv.
idtac "----- andb_true_iff -----".
Print Assumptions andb_true_iff.
idtac "----- orb_true_iff -----".
Print Assumptions orb_true_iff.
idtac "----- eqb_neq -----".
Print Assumptions eqb_neq.
idtac "----- eqb_list_true_iff -----".
Print Assumptions eqb_list_true_iff.
idtac "----- forallb_true_iff -----".
Print Assumptions forallb_true_iff.
idtac "----- excluded_middle_irrefutable -----".
Print Assumptions excluded_middle_irrefutable.
idtac "".
idtac "***** Advanced *****".
idtac "----- double_neg_inf -----".
idtac "MANUAL".
idtac "----- informal_not_PNP -----".
idtac "MANUAL".
idtac "----- not_exists_dist -----".
Print Assumptions not_exists_dist.
Abort.

```



# Chapter 28

## Library LF.IndPropTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import IndProp.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import IndProp.
Import Check.
Goal True.
idtac "————— ev_double —————".
```

```

idtac " ".
idtac "#> ev_double".
idtac "Possible points: 1".
check_type @ev_double (( $\forall n : \text{nat}, \text{ev} (\text{double } n)$ )).
idtac "Assumptions:".
Abort.
Print Assumptions ev_double.
Goal True.
idtac " ".

idtac "————- inversion_practice —————".
idtac " ".

idtac "#> SSSSev__even".
idtac "Possible points: 1".
check_type @SSSev__even (( $\forall n : \text{nat}, \text{ev} (\text{S} (\text{S} (\text{S} (\text{S } n)))) \rightarrow \text{ev } n$ )).
idtac "Assumptions:".
Abort.
Print Assumptions SSSSev__even.
Goal True.
idtac " ".

idtac "————- ev5_nonsense —————".
idtac " ".

idtac "#> ev5_nonsense".
idtac "Possible points: 1".
check_type @ev5_nonsense (( $\text{ev } 5 \rightarrow 2 + 2 = 9$ )).
idtac "Assumptions:".
Abort.
Print Assumptions ev5_nonsense.
Goal True.
idtac " ".

idtac "————- ev_sum —————".
idtac " ".

idtac "#> ev_sum".
idtac "Possible points: 2".
check_type @ev_sum (( $\forall n \ m : \text{nat}, \text{ev } n \rightarrow \text{ev } m \rightarrow \text{ev} (n + m)$ )).
idtac "Assumptions:".
Abort.
Print Assumptions ev_sum.
Goal True.
idtac " ".

idtac "————- ev_ev__ev —————".

```

```

idtac " ".
idtac "#> ev_ev__ev".
idtac "Advanced".
idtac "Possible points: 3".
check_type @ev_ev__ev (( $\forall n m : \text{nat}, \text{ev } (n + m) \rightarrow \text{ev } n \rightarrow \text{ev } m$ )).
idtac "Assumptions:".
Abort.
Print Assumptions ev_ev__ev.
Goal True.
idtac " ".

idtac "———— R_provability ————".
idtac " ".

idtac "#> Manually graded: R.R_provability".
idtac "Possible points: 3".
print_manual_grade R.manual_grade_for_R_provability.
idtac " ".

idtac "———— subsequence ————".
idtac " ".

idtac "#> subseq_refl".
idtac "Advanced".
idtac "Possible points: 1".
check_type @subseq_refl (( $\forall l : \text{list nat}, \text{subseq } l \ l$ )).
idtac "Assumptions:".
Abort.
Print Assumptions subseq_refl.
Goal True.
idtac " ".

idtac "#> subseq_app".
idtac "Advanced".
idtac "Possible points: 1".
check_type @subseq_app (
( $\forall l1 \ l2 \ l3 : \text{list nat}, \text{subseq } l1 \ l2 \rightarrow \text{subseq } l1 \ (l2 ++ l3)$ )).
idtac "Assumptions:".
Abort.
Print Assumptions subseq_app.
Goal True.
idtac " ".

idtac "———— exp_match_ex1 ————".
idtac " ".

idtac "#> empty_is_empty".

```

```

idtac "Possible points: 1".
check_type @empty_is_empty (( $\forall (T : \text{Type}) (s : \text{list } T), \neg (s \sim \text{@EmptySet } T))$ )).
idtac "Assumptions:".
Abort.
Print Assumptions empty_is_empty.
Goal True.
idtac " ".

idtac "#> MUnion'".
idtac "Possible points: 1".
check_type @MUnion' (
( $\forall (T : \text{Type}) (s : \text{list } T) (re1 \ re2 : \text{reg\_exp } T),$ 
 $s \sim re1 \vee s \sim re2 \rightarrow s \sim \text{@Union } T \ re1 \ re2$ )).
idtac "Assumptions:".
Abort.
Print Assumptions MUnion'.
Goal True.
idtac " ".

idtac "#> MStar'".
idtac "Possible points: 1".
check_type @MStar' (
( $\forall (T : \text{Type}) (ss : \text{list } (\text{list } T)) (re : \text{reg\_exp } T),$ 
 $(\forall s : \text{list } T, @ln (\text{list } T) \ s \ ss \rightarrow s \sim re) \rightarrow$ 
 $@fold (\text{list } T) (\text{list } T) (@app T) \ ss \ [] = \sim \text{@Star } T \ re$ )).
idtac "Assumptions:".
Abort.
Print Assumptions MStar'.
Goal True.
idtac " ".

idtac "————- re_not_empty —————".
idtac " ".

idtac "#> re_not_empty".
idtac "Possible points: 3".
check_type @re_not_empty (( $\forall T : \text{Type}, \text{reg\_exp } T \rightarrow \text{bool}$ )).
idtac "Assumptions:".
Abort.
Print Assumptions re_not_empty.
Goal True.
idtac " ".

idtac "#> re_not_empty_correct".
idtac "Possible points: 3".
check_type @re_not_empty_correct (

```

```

(∀ (T : Type) (re : reg_exp T),
  (∃ s : list T, s =~ re) ↔ @re_not_empty T re = true)).
idtac "Assumptions:".
Abort.
Print Assumptions re_not_empty_correct.
Goal True.
idtac " ".
idtac "———— weak_pumping —————".
idtac " ".
idtac "#> Pumping.weak_pumping".
idtac "Advanced".
idtac "Possible points: 10".
check_type @Pumping.weak_pumping (
  (∀ (T : Type) (re : reg_exp T) (s : list T),
    s =~ re →
    @Pumping.pumping_constant T re ≤ @length T s →
    ∃ s1 s2 s3 : list T,
      s = s1 ++ s2 ++ s3 ∧
      s2 ≠ [] ∧ (∀ m : nat, s1 ++ @Pumping.napp T m s2 ++ s3 =~ re))).
idtac "Assumptions:".
Abort.
Print Assumptions Pumping.weak_pumping.
Goal True.
idtac " ".
idtac "———— reflect_iff —————".
idtac " ".
idtac "#> reflect_iff".
idtac "Possible points: 2".
check_type @reflect_iff ((∀ (P : Prop) (b : bool), reflect P b → P ↔ b = true)).
idtac "Assumptions:".
Abort.
Print Assumptions reflect_iff.
Goal True.
idtac " ".
idtac "———— eqbP_practice —————".
idtac " ".
idtac "#> eqbP_practice".
idtac "Possible points: 3".
check_type @eqbP_practice (
  (∀ (n : nat) (l : list nat), count n l = 0 → ¬ @!n nat n l)).
idtac "Assumptions:".

```

```

Abort.
Print Assumptions eqbP_practice.
Goal True.
idtac " ".

idtac "————- nostutter_defn —————".
idtac " ".

idtac "#> Manually graded: nostutter".
idtac "Possible points: 3".
print_manual_grade manual_grade_for_nostutter.
idtac " ".

idtac "————- filter_challenge —————".
idtac " ".

idtac "#> Manually graded: filter_challenge".
idtac "Advanced".
idtac "Possible points: 6".
print_manual_grade manual_grade_for_filter_challenge.
idtac " ".

idtac " ".

idtac "Max points - standard: 25".
idtac "Max points - advanced: 46".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "————- ev_double ————".
Print Assumptions ev_double.
idtac "————- SSSSev__even ————".
Print Assumptions SSSSev__even.

```

```

idtac "----- ev5_nonsense -----".
Print Assumptions ev5_nonsense.
idtac "----- ev_sum -----".
Print Assumptions ev_sum.
idtac "----- R_provability -----".
idtac "MANUAL".
idtac "----- empty_is_empty -----".
Print Assumptions empty_is_empty.
idtac "----- MUnion' -----".
Print Assumptions MUnion'.
idtac "----- MStar' -----".
Print Assumptions MStar'.
idtac "----- re_not_empty -----".
Print Assumptions re_not_empty.
idtac "----- re_not_empty_correct -----".
Print Assumptions re_not_empty_correct.
idtac "----- reflect_iff -----".
Print Assumptions reflect_iff.
idtac "----- eqbP_practice -----".
Print Assumptions eqbP_practice.
idtac "----- nostutter -----".
idtac "MANUAL".
idtac "".
idtac "***** Advanced *****".
idtac "----- ev_ev_ev -----".
Print Assumptions ev_ev_ev.
idtac "----- subseq_refl -----".
Print Assumptions subseq_refl.
idtac "----- subseq_app -----".
Print Assumptions subseq_app.
idtac "----- Pumping.weak_pumping -----".
Print Assumptions Pumping.weak_pumping.
idtac "----- filter_challenge -----".
idtac "MANUAL".
Abort.

```

# Chapter 29

## Library LF.MapTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Maps.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Maps.
Import Check.
Goal True.
idtac "————— t_update_same —————".
```



```

idtac " ".
idtac "#> t_update_same".
idtac "Possible points: 2".
check_type @t_update_same (
  (∀ (A : Type) (m : total_map A) (x : string), (x !-> m x; m) = m)).
idtac "Assumptions:".
Abort.
Print Assumptions t_update_same.
Goal True.
idtac " ".

idtac "————— t_update_permute —————".
idtac " ".

idtac "#> t_update_permute".
idtac "Possible points: 3".
check_type @t_update_permute (
  (∀ (A : Type) (m : total_map A) (v1 v2 : A) (x1 x2 : string),
    x2 ≠ x1 → (x1 !-> v1; x2 !-> v2; m) = (x2 !-> v2; x1 !-> v1; m))).
idtac "Assumptions:".
Abort.
Print Assumptions t_update_permute.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 5".
idtac "Max points - advanced: 5".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".

```

```

idtac "***** Standard *****".
idtac "----- t_update_same -----".
Print Assumptions t_update_same.
idtac "----- t_update_permute -----".
Print Assumptions t_update_permute.
idtac "".
idtac "***** Advanced *****".
Abort.

```

# Chapter 30

## Library LF.ProofObjectsTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import ProofObjects.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import ProofObjects.
Import Check.
Goal True.
idtac "————— eight_is_even —————".
```

```

idtac " ".
idtac "#> ev_8".
idtac "Possible points: 1".
check_type @ev_8 ((ev 8)).
idtac "Assumptions:".
Abort.
Print Assumptions ev_8.
Goal True.
idtac " ".

idtac "#> ev_8'".
idtac "Possible points: 1".
check_type @ev_8' ((ev 8)).
idtac "Assumptions:".
Abort.
Print Assumptions ev_8'.
Goal True.
idtac " ".

idtac "_____ conj_fact _____".
idtac " ".

idtac "#> Props.conj_fact".
idtac "Possible points: 2".
check_type @Props.conj_fact (( $\forall P Q R : \text{Prop}, P \wedge Q \rightarrow Q \wedge R \rightarrow P \wedge R$ )).
idtac "Assumptions:".
Abort.
Print Assumptions Props.conj_fact.
Goal True.
idtac " ".

idtac "_____ or_commut' _____".
idtac " ".

idtac "#> Props.or_commut'".
idtac "Possible points: 2".
check_type @Props.or_commut' (( $\forall P Q : \text{Prop}, P \vee Q \rightarrow Q \vee P$ )).
idtac "Assumptions:".
Abort.
Print Assumptions Props.or_commut'.
Goal True.
idtac " ".

idtac "_____ ex_ev_Sn _____".
idtac " ".

idtac "#> Props.ex_ev_Sn".

```

```

idtac "Possible points: 2".
check_type @Props.ex_ev_Sn (( $\exists$  n : nat, ev (S n))).
idtac "Assumptions:".
Abort.
Print Assumptions Props.ex_ev_Sn.
Goal True.
idtac " ".
idtac "————— p_implies_true —————".
idtac " ".
idtac "#> Props.p_implies_true".
idtac "Possible points: 1".
check_type @Props.p_implies_true (( $\forall$  P : Type, P  $\rightarrow$  Props.True)).
idtac "Assumptions:".
Abort.
Print Assumptions Props.p_implies_true.
Goal True.
idtac " ".
idtac "————— ex_falso_quodlibet' —————".
idtac " ".
idtac "#> Props.ex_falso_quodlibet'".
idtac "Possible points: 1".
check_type @Props.ex_falso_quodlibet' (( $\forall$  P : Type, Props.False  $\rightarrow$  P)).
idtac "Assumptions:".
Abort.
Print Assumptions Props.ex_falso_quodlibet'.
Goal True.
idtac " ".
idtac "————— equality__leibniz_equality —————".
idtac " ".
idtac "#> MyEquality.equality__leibniz_equality".
idtac "Possible points: 2".
check_type @MyEquality.equality__leibniz_equality (
( $\forall$  (X : Type) (x y : X),
  @MyEquality.eq X x y  $\rightarrow$   $\forall$  P : X  $\rightarrow$  Prop, P x  $\rightarrow$  P y)).
idtac "Assumptions:".
Abort.
Print Assumptions MyEquality.equality__leibniz_equality.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 12".

```

```

idtac "Max points - advanced: 12".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- ev_8 -----".
Print Assumptions ev_8.
idtac "----- ev_8' -----".
Print Assumptions ev_8'.
idtac "----- Props.conj_fact -----".
Print Assumptions Props.conj_fact.
idtac "----- Props.or_commut' -----".
Print Assumptions Props.or_commut'.
idtac "----- Props.ex_ev_Sn -----".
Print Assumptions Props.ex_ev_Sn.
idtac "----- Props.p_implies_true -----".
Print Assumptions Props.p_implies_true.
idtac "----- Props.ex_falso_quodlibet' -----".
Print Assumptions Props.ex_falso_quodlibet'.
idtac "----- MyEquality.equality__leibniz_equality -----".
Print Assumptions MyEquality.equality__leibniz_equality.
idtac "".
idtac "***** Advanced *****".
Abort.

```

# Chapter 31

## Library LF.IndPrinciplesTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import IndPrinciples.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import IndPrinciples.
Import Check.
Goal True.
idtac "————— plus_one_r' —————".
```

```

idtac " ".
idtac "#> plus_one_r'".
idtac "Possible points: 2".
check_type @plus_one_r' (( $\forall n : \text{nat}, n + 1 = S\ n$ )).
idtac "Assumptions:".
Abort.
Print Assumptions plus_one_r'.
Goal True.
idtac " ".

idtac "----- booltree_ind -----".
idtac " ".

idtac "#> Manually graded: booltree_ind".
idtac "Possible points: 1".
print_manual_grade manual_grade_for_booltree_ind.
idtac " ".

idtac "----- toy_ind -----".
idtac " ".

idtac "#> Manually graded: toy_ind".
idtac "Possible points: 1".
print_manual_grade manual_grade_for_toy_ind.
idtac " ".

idtac " ".

idtac "Max points - standard: 4".
idtac "Max points - advanced: 4".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".

```



```
idtac "——- plus_one_r' ——".
Print Assumptions plus_one_r'.
idtac "——- booltree_ind ——".
idtac "MANUAL".
idtac "——- toy_ind ——".
idtac "MANUAL".
idtac "".
idtac "***** Advanced *****".
Abort.
```

# Chapter 32

## Library LF.RelTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Rel.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Rel.
Import Check.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

# Chapter 33

## Library LF.ImpTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Imp.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Imp.
Import Check.
Goal True.
idtac "————— optimize_0plus_b_sound —————".
```

```

idtac " ".
idtac "#> AExp.optimize_0plus_b_sound".
idtac "Possible points: 3".
check_type @AExp.optimize_0plus_b_sound (
(∀ b : AExp.bexp, AExp.beval (AExp.optimize_0plus_b b) = AExp.beval b)).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.optimize_0plus_b_sound.
Goal True.
idtac " ".
idtac "————— bevalR —————".
idtac " ".
idtac "#> AExp.beval_iff_bevalR".
idtac "Possible points: 3".
check_type @AExp.beval_iff_bevalR (
(∀ (b : AExp.bexp) (bv : bool), AExp.bevalR b bv ↔ AExp.beval b = bv)).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.beval_iff_bevalR.
Goal True.
idtac " ".
idtac "————— ceval_example2 —————".
idtac " ".
idtac "#> ceval_example2".
idtac "Possible points: 2".
check_type @ceval_example2 (
(empty_st = [ X := (ANum 0); Y := (ANum 1); Z := (ANum 2)
] => @Maps.t_update nat (@Maps.t_update nat (X !-> 0) Y 1) Z 2)).
idtac "Assumptions:".
Abort.
Print Assumptions ceval_example2.
Goal True.
idtac " ".
idtac "————— loop_never_stops —————".
idtac " ".
idtac "#> loop_never_stops".
idtac "Possible points: 3".
check_type @loop_never_stops ((∀ st st' : state, ¬ st = [ loop ] => st')).
idtac "Assumptions:".
Abort.
Print Assumptions loop_never_stops.

```

```

Goal True.
idtac " ".

idtac "————- no_whiles_eqv —————".
idtac " ".

idtac "#> no_whiles_eqv".
idtac "Possible points: 3".
check_type @no_whiles_eqv (( $\forall$  c : com, no_whiles c = true  $\leftrightarrow$  no_whilesR c)).
idtac "Assumptions:".
Abort.
Print Assumptions no_whiles_eqv.
Goal True.
idtac " ".

idtac "————- no_whiles_terminating —————".
idtac " ".

idtac "#> Manually graded: no_whiles_terminating".
idtac "Possible points: 6".
print_manual_grade manual_grade_for_no_whiles_terminating.
idtac " ".

idtac "————- stack_compiler —————".
idtac " ".

idtac "#> s_execute1".
idtac "Possible points: 1".
check_type @s_execute1 (
(s_execute empty_st (@nil nat)
  (SPush 5 :: SPush 3 :: SPush 1 :: SMinus :: @nil sinstr) =
  (2 :: 5 :: @nil nat)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions s_execute1.
Goal True.
idtac " ".

idtac "#> s_execute2".
idtac "Possible points: 0.5".
check_type @s_execute2 (
(s_execute (X !-> 3) (3 :: 4 :: @nil nat)
  (SPush 4 :: SLoad X :: SMult :: SPlus :: @nil sinstr) =
  (15 :: 4 :: @nil nat)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions s_execute2.
Goal True.

```

```

idtac " ".
idtac "#> s_compile1".
idtac "Possible points: 1.5".
check_type @s_compile1 (
(s_compile <{ (Ald X) - (ANum 2) × (Ald Y) }> =
  (SLoad X :: SPush 2 :: SLoad Y :: SMult :: SMinus :: @nil sinstr)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions s_compile1.
Goal True.
idtac " ".

idtac "————- execute_app —————".
idtac " ".

idtac "#> execute_app".
idtac "Possible points: 3".
check_type @execute_app (
(∀ (st : state) (p1 p2 : list sinstr) (stack : list nat),
  s_execute st stack (p1 ++ p2) = s_execute st (s_execute st stack p1) p2)).
idtac "Assumptions:".
Abort.
Print Assumptions execute_app.
Goal True.
idtac " ".

idtac "————- stack_compiler_correct —————".
idtac " ".

idtac "#> s_compile_correct_aux".
idtac "Possible points: 2.5".
check_type @s_compile_correct_aux (
(∀ (st : state) (e : aexp) (stack : list nat),
  s_execute st stack (s_compile e) = (aeval st e :: stack)%list)).
idtac "Assumptions:".
Abort.
Print Assumptions s_compile_correct_aux.
Goal True.
idtac " ".

idtac "#> s_compile_correct".
idtac "Possible points: 0.5".
check_type @s_compile_correct (
(∀ (st : state) (e : aexp),
  s_execute st (@nil nat) (s_compile e) = (aeval st e :: @nil nat)%list)).
idtac "Assumptions:".

```

```

Abort.
Print Assumptions s_compile_correct.
Goal True.
idtac " ".

idtac "————— break_imp —————".
idtac " ".

idtac "#> BreakImp.break_ignore".
idtac "Advanced".
idtac "Possible points: 1.5".
check_type @BreakImp.break_ignore (
  (∀ (c : BreakImp.com) (st st' : state) (s : BreakImp.result),
    BreakImp.ceval (BreakImp.CSeq BreakImp.CBreak c) st s st' → st = st')).
idtac "Assumptions:".

Abort.
Print Assumptions BreakImp.break_ignore.
Goal True.
idtac " ".

idtac "#> BreakImp.while_continue".
idtac "Advanced".
idtac "Possible points: 1.5".
check_type @BreakImp.while_continue (
  (∀ (b : bexp) (c : BreakImp.com) (st st' : state) (s : BreakImp.result),
    BreakImp.ceval (BreakImp.CWhile b c) st s st' → s = BreakImp.SContinue)).
idtac "Assumptions:".

Abort.
Print Assumptions BreakImp.while_continue.
Goal True.
idtac " ".

idtac "#> BreakImp.while_stops_on_break".
idtac "Advanced".
idtac "Possible points: 3".
check_type @BreakImp.while_stops_on_break (
  (∀ (b : bexp) (c : BreakImp.com) (st st' : state),
    beval st b = true →
      BreakImp.ceval c st BreakImp.SBreak st' →
      BreakImp.ceval (BreakImp.CWhile b c) st BreakImp.SContinue st')).
idtac "Assumptions:".

Abort.
Print Assumptions BreakImp.while_stops_on_break.
Goal True.
idtac " ".

```



```

idtac " ".
idtac "Max points - standard: 29".
idtac "Max points - advanced: 35".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "———- AExp.optimize_0plus_b_sound ———".
Print Assumptions AExp.optimize_0plus_b_sound.
idtac "———- AExp.beval_iff_bevalR ———".
Print Assumptions AExp.beval_iff_bevalR.
idtac "———- ceval_example2 ———".
Print Assumptions ceval_example2.
idtac "———- loop_never_stops ———".
Print Assumptions loop_never_stops.
idtac "———- no_whiles_eqv ———".
Print Assumptions no_whiles_eqv.
idtac "———- no_whiles_terminating ———".
idtac "MANUAL".
idtac "———- s_execute1 ———".
Print Assumptions s_execute1.
idtac "———- s_execute2 ———".
Print Assumptions s_execute2.
idtac "———- s_compile1 ———".
Print Assumptions s_compile1.
idtac "———- execute_app ———".
Print Assumptions execute_app.
idtac "———- s_compile_correct_aux ———".
Print Assumptions s_compile_correct_aux.

```

```

idtac "————- s_compile_correct ————".
Print Assumptions s_compile_correct.
idtac "".
idtac "***** Advanced *****".
idtac "————- BreakImp.break_ignore ————".
Print Assumptions BreakImp.break_ignore.
idtac "————- BreakImp.while_continue ————".
Print Assumptions BreakImp.while_continue.
idtac "————- BreakImp.while_stops_on_break ————".
Print Assumptions BreakImp.while_stops_on_break.
Abort.

```

# Chapter 34

## Library LF.ImpParserTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import ImpParser.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import ImpParser.
Import Check.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

# Chapter 35

## Library LF.ImpCEvalFunTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import ImpCEvalFun.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import ImpCEvalFun.
Import Check.
Goal True.
idtac "————— pup_to_n —————".
```

```

idtac " ".
idtac "#> pup_to_n".
idtac "Possible points: 2".
check_type @pup_to_n (Imp.com).
idtac "Assumptions:".
Abort.
Print Assumptions pup_to_n.
Goal True.
idtac " ".

idtac "————— ceval_step__ceval_inf —————".
idtac " ".

idtac "#> Manually graded: ceval_step__ceval_inf".
idtac "Possible points: 6".
print_manual_grade manual_grade_for_ceval_step__ceval_inf.
idtac " ".

idtac "————— ceval__ceval_step —————".
idtac " ".

idtac "#> ceval__ceval_step".
idtac "Possible points: 3".
check_type @ceval__ceval_step (
 $(\forall (c : \text{Imp.com}) (st\ st' : \text{Imp.state}),$ 
   $\text{Imp.ceval } c\ st\ st' \rightarrow$ 
 $\exists i : \text{nat}, \text{ceval\_step } st\ c\ i = \text{@Some Imp.state } st')$ 
 $\rangle$ 
).
idtac "Assumptions:".
Abort.
Print Assumptions ceval__ceval_step.
Goal True.
idtac " ".
idtac " ".

idtac "Max points - standard: 11".
idtac "Max points - advanced: 11".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".

```

```

idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "----- pup_to_n -----".
Print Assumptions pup_to_n.
idtac "----- ceval_step__ceval_inf -----".
idtac "MANUAL".
idtac "----- ceval__ceval_step -----".
Print Assumptions ceval__ceval_step.
idtac "".
idtac "***** Advanced *****".
Abort.

```

# Chapter 36

## Library LF.ExtractionTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Extraction.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Extraction.
Import Check.
Goal True.
idtac " ".
```



```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

# Chapter 37

## Library LF.AutoTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Auto.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Auto.
Import Check.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

# Chapter 38

## Library LF.AltAutoTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import AltAuto.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import AltAuto.
Import Check.
Goal True.
idtac "————— re_opt —————".
```

```

idtac " ".
idtac "#> Manually graded: re_opt".
idtac "Possible points: 3".
print_manual_grade manual_grade_for_re_opt.
idtac " ".

idtac "————- pumping_redux —————".
idtac " ".

idtac "#> Manually graded: pumping_redux".
idtac "Advanced".
idtac "Possible points: 3".
print_manual_grade manual_grade_for_pumping_redux.
idtac " ".

idtac " ".

idtac "Max points - standard: 3".
idtac "Max points - advanced: 6".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "————- re_opt ————".
idtac "MANUAL".
idtac "".
idtac "***** Advanced *****".
idtac "————- pumping_redux ————".
idtac "MANUAL".
Abort.

```

# Chapter 39

## Library LF.PostscriptTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Postscript.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Postscript.
Import Check.
Goal True.
idtac " ".
```

```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```

# Chapter 40

## Library LF.BibTest

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Export String.
From LF Require Import Bib.
Parameter MISSING: Type.
Module CHECK.
Ltac check_type A B :=
  match type of A with
  | context[MISSING] => idtac "Missing:" A
  | ?T => first [unify T B; idtac "Type: ok" | idtac "Type: wrong - should be (" B
  ")"]
  end.
Ltac print_manual_grade A :=
  match eval compute in A with
  | Some (- ?S ?C) =>
    idtac "Score:" S;
    match eval compute in C with
    | "%string" => idtac "Comment: None"
    | _ => idtac "Comment:" C
    end
  | None =>
    idtac "Score: Ungraded";
    idtac "Comment: None"
  end.
End CHECK.
From LF Require Import Bib.
Import Check.
Goal True.
idtac " ".
```



```

idtac "Max points - standard: 0".
idtac "Max points - advanced: 0".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "".
idtac "".
idtac "***** Summary *****".
idtac "".
idtac "Below is a summary of the automatically graded exercises that are incomplete.".
idtac "".
idtac "The output for each exercise can be any of the following:".
idtac " - 'Closed under the global context', if it is complete".
idtac " - 'MANUAL', if it is manually graded".
idtac " - A list of pending axioms, containing unproven assumptions. In this case".
idtac " the exercise is considered complete, if the axioms are all allowed.".
idtac "".
idtac "***** Standard *****".
idtac "".
idtac "***** Advanced *****".
Abort.

```