Software Foundations: LF

August 4, 2026

Contents

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 Rocq prover, 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 Rocq. The books are intended to be read alongside (or inside) an interactive session with Rocq. All the details in the text are fully formalized in Rocq, and most of the exercises are designed to be worked using Rocq.

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 Rocq prover.

1.3 Overview

Building reliable software is 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, 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:

(A) basic tools from *logic* for making and justifying precise claims about programs;

(B) the use of *proof assistants* (or *provers*) to construct rigorous logical arguments;

(C) *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 reasoning capabilities are still limited, they have matured tremendously in recent decades 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, F*, HOL4, Lean, and Rocq, among many others.

This course is based around Rocq, a proof assistant that has been under development since 1983 and has attracted a large community of users in both research and industry. Rocq provides a rich environment for interactive development of machine-checked formal reasoning. The kernel of the Rocq system is a simple proof-checker, which guarantees that only correct deduction steps are ever performed. On top of this kernel, the Rocq 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.

Rocq 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*, Rocq 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, FCF, and SSProve, which are frameworks for proving cryptographic algorithms secure. 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, Hoare Type Theory embeds reasoning about "pre-conditions" and "post-conditions" (an extension of the *Hoare Logic* we will see later in this course) in Rocq.

- 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 Rocq formalization, everything is checked, including the correctness of the computational part. More recently, an even more massive effort led to a Rocq formalization of the Feit-Thompson Theorem, the first major step in the classification of finite simple groups.

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, F*, and Rocq.

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, Rocq 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 Rocq are actually aspects of the very same underlying machinery – i.e., *proofs are programs*.

1.3.4 Rocq vs. Coq

Until 2025, the Rocq prover was known as Coq. According to the official webpage, "The name 'Coq' referenced the Calculus of Constructions (CoC), the foundational system it is based on, as well as one of its creators, Thierry Coquand. Additionally, it paid homage to the French national symbol, the rooster. The new name, 'the Rocq Prover', honors Inria Rocquencourt, the original site where the prover was developed. It also alludes to the mythological bird Roc (or Rokh), symbolizing strength and not so disconnected to a rooster. Furthermore, the name conveys a sense of solidity, and its unintended connection to music adds a pleasant resonance."

The current release of Software Foundations is still in a transitional state, and you will see references to both Coq and Rocq.

1.3.5 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

Rocq runs on Windows, Linux, and macOS. The files in this book have been tested with Rocq 9.0.0.

Recommended Installation Method: VSCode + Docker

The Visual Studio Code IDE can cooperate with the Docker virtualization platform to compile Rocq scripts without the need for any separate Rocq installation. This method is recommended for most Software Foundations readers.

- Install Docker from {https://www.docker.com/get-started/} or make sure your existing installation is up to date.
- Make sure Docker is running.
- Install VSCode from {https://code.visualstudio.com} and start it running.
- Install VSCode's Dev Containers Extension from {https://marketplace.visualstudio.com/items?itemN vscode-remote.remote-containers}

  (Note that this extension only works with the official version of VSCode, not with some VSCode forks like VsCodium.)

- Set up a directory for this SF volume by downloading the provided *.tgz* file. Besides the *.v* file for each chapter, this directory will contain a *.devcontainer* subdirectory with instructions for VSCode about where to find an appropriate Docker image and a *_CoqProject* file, whose presence triggers the VSCoq extension.

- In VSCode, use *File* > `Open` *Folder* to open the new directory. VSCode should ask you whether you want to run the project in the associated Docker container. (If it does not ask you, you can open the command palette by pressing F1 and run the command "Dev Containers: Reopen in Container".)

  This step may take some time.

- Check that VSCoq is working by double-clicking the file *Basics.v* from the list on the left (you should see a blinking cursor in the window that opens; if not you can click in that window to select it), and pressing *alt*+*downarrow* (on MacOS, *control*+**option**+*downarrow*) a few times. You should see the cursor move through the file and the region above the cursor get highlighted.

  - If VSCoq does not work and you receive an error indicating that *vscoqtop* was not found, open a new terminal in the container (you can do this by opening the command palette and running the command "Terminal: Create New Terminal") and run the command *which vscoqtop*. This should print the path to the VSCoq installation inside the container. Copy this path and paste it into the "VSCoq: Path" textbox in the VSCoq extension settings (accessible via the gear icon on the VSCoq extension page in VSCode), then reload your window.

- To see what other key bindings are available, press F1 and then type *Coq*:, or visit the VSCoq web pages: {https://github.com/rocq-prover/vsrocq}.

1.4.2 Alternative Installation Methods

If you prefer, there are several other ways to use Rocq. You will need:

- A current installation of Rocq, available from the Rocq home page ({https://rocq-prover.org/install}). The "Rocq Platform" offers the easiest installation experience for most people, especially on Windows.

- An IDE for interacting with Rocq. There are several choices:

  - *VsCoq* is an extension for Visual Studio Code that offers a simple interface via a familiar IDE. This option is the recommended default.

    VsCoq can be used as an ordinary IDE or it can be combined with Docker (see below) for a lightweight installation experience.

– *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", but generally all you need to do is *M-x package-list-packages*, then select the *proof-general* package from the list and hit the *i* key for install, then hit the *x* key for execute).

  There are only a few commands you need to know to use ProofGeneral effectively. They are:

  * C-c C-*n*: send the next command to Rocq.
  * C-c C-*u*: undo (retract) the most recently executed command.
  * C-c C-*RET*: submit everything up to the current cursor location to Rocq for processing.
  * C-c C-.: move the cursor to the end of the last command which has been processed by Rocq.
  * C-c .: toggle "electric terminator mode". When this mode is turned on, simply typing a period will send the current command to Rocq (normally you have to type a period and then type C-c C-*n*).

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

– *RocqIDE* is a simpler stand-alone IDE. It is distributed with the Rocq Platform, so it should be available once you have Rocq 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 RocqIDE should consider running it with the "asynchronous" and "error resilience" modes disabled:

  coqide -async-proofs off \

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

1.4.3 Exercises

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

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

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 anyplace where they can be found by search engines.

1.4.4 Downloading the Rocq Files

A tar file containing the full sources for the "release version" of this book (as a collection of Rocq 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.

1.4.5 Chapter Dependencies

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

1.4.6 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 and Arthur Azevedo de Amorim and Chris Casinghino and Marco Gaboardi and Michael Greenberg and Cătălin Hriţcu and Vilhelm Sjöberg and Brent Yorgey}, editor = {Benjamin C. Pierce}, title = "Logical Foundations", series = "Software Foundations", volume = "1", year = "2026", publisher = "Electronic textbook", note = {Version 7.0, \URL{http://softwarefoundations.cis.upenn.edu}} }

1.5 Resources

1.5.1 Sample Exams

A large compendium of exams from many offerings of CIS5000 ("Software Foundations") at the University of Pennsylvania can be found at {https://www.seas.upenn.edu/~cis5000/current/exams/inde 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 The video quality in the 2017 lectures is poor at the beginning but gets better in the later lectures.

1.6 Note for Instructors and Contributors

If you plan to use these materials in your own teaching, or if you are using software foundations for self study and are finding things you'd like to help add or improve, your contributions are welcome! You are warmly invited to join the private SF git repo.

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 Rocq," the Author of Record is Benjamin C. Pierce. For Volume 3, "Verified Functional Algorithms," and volume 5, "Verifiable C," the Author of Record is Andrew W. Appel. For Volume 6, "Separation Logic Foundations," the author of record is Arthur Charguéraud. For components outside of designated volumes (e.g., typesetting and grading tools and other software infrastructure), the Author of Record is Benjamin C. Pierce.

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

We'll set you up with access to the git repository and developers' mailing lists. In the repository you'll find the files *INSTRUCTORS* and *CONTRIBUTING* 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 Rocq

2.2 Introduction

The *functional style* of programming is founded on simple, everyday mathematical intuitions: 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. Rocq offers all of these features.

The first half of this chapter introduces some key elements of Rocq's native functional programming language, *Gallina*. The second half introduces some basic *tactics* that can be used to prove properties of Gallina programs.

2.3 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

ensure that you get full credit for your work!), please be careful to follow these rules:

- Do not change the names of exercises. Otherwise the grading scripts will be unable to find your solution.
- 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 the keyword *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 before 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`. This will make sure you get partial credit if you use that main theorem to solve a later exercise.

You will also notice that each chapter (e.g., *Basics.v*) is accompanied by a *test script* (e.g., *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:

rocq compile -Q . LF Basics.v rocq compile -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.
- If you are handing in multiple files at the same time (i.e., if more than one chapter is assigned in the same week), you should make a *single* submission with all the files at once by using the "Add another file" button just above the comment box.

2.4 Data and Functions

2.4.1 Enumerated Types

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

Naturally, the Rocq distribution also 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, in this course we will explicitly recapitulate almost all the definitions we need, rather than getting them from the standard library.

2.4.2 Days of the Week

To see how the datatype definition mechanism works, let's start with a very simple example. The following declaration tells Rocq 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_working_day (d:day) : day :=
  match d with
  | monday ⇒ tuesday
  | tuesday ⇒ wednesday
  | wednesday ⇒ thursday
  | thursday ⇒ friday
  | friday ⇒ monday
  | saturday ⇒ monday
  | sunday ⇒ monday
  end.
```

Note that the argument and return types of this function are explicitly declared on the first line. Like most functional programming languages, Rocq 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 can check that it works on some examples. There are actually three different ways to do examples in Rocq. First, we can use the command Compute to evaluate a compound expression involving next_working_day.

```
Compute (next_working_day friday).
```

```
Compute (next_working_day (next_working_day saturday)).
```

(We show Rocq's responses in comments; if you have a computer handy, this would be an excellent moment to fire up the Rocq interpreter under your favorite IDE (see the Preface

for installation instructions) and try it for yourself. Load this file, *Basics.v*, from the book's Rocq sources, find the above example, submit it to Rocq, and observe the result.)

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

```
Example test_next_working_day:
  (next_working_day (next_working_day saturday)) = tuesday.
```

This declaration does two things: it makes an assertion (that the second working day 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 Rocq to *verify* it, like this:

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

The details are not important just now, but essentially this little script 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 Rocq to *extract*, from our `Definition`, a program in a more conventional programming language (OCaml, Scheme, or Haskell) with a high-performance compiler. This facility is very useful, 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 Rocq'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 Rocq was developed. We'll come back to this topic in later chapters.

The `Require Export` statement on the next line tells Rocq to use the String module from the standard library. We'll use strings for various things in later chapters, but we need to `Require` it here so that the grading scripts can use it for internal purposes. `From Stdlib Require Export String.`

2.4.3 Booleans

Following the pattern of the days of the week above, 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, Rocq does, of course, provide a default implementation of the booleans, together with a multitude of useful functions and lemmas. Wherever possible, we've named our own definitions and theorems to match the ones in the standard library.)

The last two of these illustrate Rocq'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 more familiar and readable 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 Rocq code within comments; this convention, also used by the *rocq doc* 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 (and without the brackets).

These examples are also an opportunity to introduce one more feature of Rocq's programming language: conditional expressions...

```
Definition negb' (b:bool) : bool :=
  if b then false
  else true.
```

```
Definition andb' (b1:bool) (b2:bool) : bool :=
  if b1 then b2
  else false.

Definition orb' (b1:bool) (b2:bool) : bool :=
  if b1 then true
  else b2.
```

Rocq's conditionals are exactly like those found in any other language, with one small generalization:

Since the **bool** type is not built in, Rocq actually supports conditional expressions over *any* inductively defined type with exactly two clauses in its definition. The guard is considered true if it evaluates to the "constructor" of the first clause of the `Inductive` definition (which happens to be called `true` in this case) and false if it evaluates to the second.

For example we can define the following datatype **bw**, with two constructors representing black (`b`) and white (*w*) and define a function `invert` that inverts values of this type using a conditional.

```
Inductive bw : Type :=
  | bw_black
  | bw_white.

Definition invert (x: bw) : bw :=
  if x then bw_white
  else bw_black.

Compute (invert bw_black).

Compute (invert bw_white).
```

Exercise: 1 star, standard (nandb) The *Admitted* command 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 *Admitted*s with real proofs.

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

Hint: if `simpl` will not simplify the goal in your proof, it's probably because you defined `nandb` without using a `match` expression. Try a different definition of `nandb`, or just skip over `simpl` and go directly to `reflexivity`. We'll explain what's happening later in the chapter.

```
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.
  □
```

2.4.4 Types

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

```
Check true.
```

If the thing after Check is followed by a colon and a type declaration, Rocq will verify that the type of the expression matches the given type and signal 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**."

2.4.5 New Types from Old

The types we have defined so far are examples of simple "enumerated types": their definitions explicitly enumerate a finite set of elements, called *constructors*. Here is a more interesting type definition, **color**, 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.
An `Inductive` definition does two things:

- It introduces a set of new *constructors*. E.g., `red`, primary, true, false, monday, etc. are constructors.
- It groups them into a new named type, like **bool**, **rgb**, or **color**.

Constructor expressions are formed by applying a constructor to zero or more other constructors or constructor expressions, obeying the declared number and types of the constructor arguments. E.g., these are valid constructor expressions...

- `red`
- true
- primary `red`
- etc.

...but these are not:

- `red` primary
- true `red`
- primary (primary `red`)
- etc.

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 ("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 we just did (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.)

2.4.6 Modules

Rocq provides a *module system* to aid in organizing large developments. We won't need most of its features, but one is useful here: 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 foo : rgb := blue.
End PLAYGROUND.

Definition foo : bool := true.

Check Playground.foo : rgb.
Check foo : bool.
```

2.4.7 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 :=
  | B1
  | B0.

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 below, which tests a nybble to see if all its bits are B0.

```
Definition all_zero (nb : nybble) : bool :=
  match nb with
  | (bits B0 B0 B0 B0) ⇒ true
  | (bits _ _ _ _) ⇒ false
  end.
```

(The underscore (_) here is a *wildcard pattern*, which avoids inventing variable names that will not be used.)

```
Compute (all_zero (bits B1 B0 B1 B0)).
Compute (all_zero (bits B0 B0 B0 B0)).

End TUPLEPLAYGROUND.
```

2.4.8 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. You are certainly familiar with decimal notation (base 10), using the digits 0 through 9, for example, to form the number 123. You may very likely also 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 our forebears might have done to count days by making scratches on the walls of their caves. To represent unary numbers with a Rocq datatype, we use two constructors. The capital-letter O constructor represents zero. The S constructor can be applied to the representation of the natural number n, yielding the representation of $n+1$, where S stands for "successor" (or "scratch"). 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 – i.e., if n is a natural number, then S n is too.

Again, let's look at this a bit more closely. 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 that we gave to Rocq. 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 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 otherNat : Type :=
  | stop
  | tick (foo : otherNat).
```

The *interpretation* of these marks arises 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.

```
End NatPlayground.
```

Because natural numbers are such a pervasive kind of data, Rocq does provide 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. Rocq 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.

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 even (n:nat) : bool :=
  match n with
  | O ⇒ true
  | S O ⇒ false
  | S (S n') ⇒ even n'
  end.
```

We could define odd by a similar `Fixpoint` declaration, but here is a simpler way:

```
Definition odd (n:nat) : bool :=
  negb (even n).

Example test_odd1: odd 1 = true.
Proof. simpl. reflexivity. Qed.
Example test_odd2: odd 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 shortly.)

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

```
Module NatPlayground2.

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

Adding three to two gives us five (whew!):

```
Compute (plus 3 2).
```

The steps of simplification that Rocq performs here can be visualized as follows:

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

```
Fixpoint mult (n m : nat) : nat :=
  match n with
  | O ⇒ O
  | S n' ⇒ plus m (mult n' m)
  end.

Example test_mult1: (mult 3 3) = 9.
Proof. simpl. reflexivity. Qed.
```

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

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

End NatPlayground2.

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

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

factorial(0) = 1 factorial(n) = n * factorial(n-1) (if n>0)

Translate this into Rocq.

Hint: Make sure you put a := between the header we've provided and your definition. If you see an error like "The reference factorial was not found in the current environment," it means you've forgotten the :=.

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

Example test_factorial1: (factorial 3) = 6.
  Admitted.
Example test_factorial2: (factorial 5) = (mult 10 12).
  Admitted.
```

□

Again, we can make numerical expressions easier to read and write by introducing notations for addition, subtraction, and multiplication.

```
Notation "x + y" := (plus x y)
                        (at level 50, left associativity)
                        : nat_scope.
Notation "x - y" := (minus x y)
                        (at level 50, left associativity)
                        : nat_scope.
Notation "x * y" := (mult x y)
                        (at level 40, left associativity)
                        : nat_scope.

Check ((0 + 1) + 1) : nat.
```

(The `level`, `associativity`, and *nat_scope* annotations control how these notations are treated by Rocq's parser. The details are not important for present purposes, but interested readers can check out 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 Rocq's parser to accept $x + y$ in place of plus x y and, conversely, to its pretty-printer to display plus x y as $x + y$.

When we say that Rocq comes with almost nothing built-in, we really mean it: even testing equality is a user-defined operation! Here is a function eqb that tests natural numbers for **eq**uality, yielding a boolean. Note the use of nested `match`es – we could also have used a simultaneous match, as in minus.

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

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

```
Fixpoint leb (n m : nat) : bool :=
  match n with
  | O ⇒ true
  | S n' ⇒
      match m with
```

```
      | O ⇒ false
      | S m' ⇒ leb n' m'
      end
  end.

Example test_leb1: leb 2 2 = true.
Proof. simpl. reflexivity. Qed.
Example test_leb2: leb 2 4 = true.
Proof. simpl. reflexivity. Qed.
Example test_leb3: leb 4 2 = false.
Proof. simpl. reflexivity. Qed.
```

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 both look like equality: $=$ and $=?$. We'll have much more to say about their differences and similarities 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 a boolean *expression* whose value (either true or false) we can compute.

Exercise: 1 star, standard (ltb) Fill in the definition of an ltb function that tests natural numbers for *l*ess-*t*han, yielding a boolean.

Hint: 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.
```

□

2.5 Proof by Simplification

Now that we've looked at 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 made 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.

```
Example plus_1_1 : 1 + 1 = 2.
Proof. simpl. reflexivity. Qed.
```

The same sort of "proof by simplification" can also be used to establish more interesting properties. 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 off directly from the definition of plus.

```
Theorem plus_O_n : ∀ n : nat, 0 + n = n.
Proof.
  intros n. simpl. reflexivity. Qed.
```

(You may notice that the above statement looks different if you look at the $.v$ file in your IDE than it does if you view the HTML rendition in your browser. In $.v$ files, we write the universal quantifier ∀ 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 suggested. In the examples we have seen, the calls to `simpl` were actually not required because `reflexivity` will do 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:

```
Theorem plus_O_n' : ∀ n : nat, 0 + n = n.
Proof.
  intros n. reflexivity. Qed.
```

Moreover, it is 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 just 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 Rocq.

Second, we've added the quantifier ∀ n:`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.

Incidentally, 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'' : ∀ n : 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 : ∀ n:nat, 1 + n = S n.
Proof.
  intros n. reflexivity. Qed.

Theorem mult_0_l : ∀ n:nat, 0 × n = 0.
Proof.
  intros n. reflexivity. Qed.
```

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

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 Rocq performs on the terms before checking that they are equal.

2.6 Proof by Rewriting

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

```
Theorem plus_id_example : ∀ n m:nat,
  n = m →
  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 Rocq 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 Rocq 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 Rocq to apply the rewrite from left to right. Indeed, we can omit the arrow, and Rocq will default to rewriting left to right. To rewrite from right to left, use `rewrite` ←. Try making this change in the above proof and see what changes.)

Exercise: 1 star, standard (plus_id_exercise) Remove "*Admitted*." and fill in the proof. (Note that the theorem has *two* hypotheses – $n = m$ and $m = o$ – each to the left of an implication arrow.)

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

□

The *Admitted* command tells Rocq that we want to skip trying to prove this theorem and just accept it as a given. This is often useful for developing longer proofs: 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 Rocq's safe, 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_O.

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, Rocq will try to fill in appropriate values for them by matching the body of the previous theorem statement against the current goal.

```
Theorem mult_n_0_m_0 : ∀ p q : nat,
  (p × 0) + (q × 0) = 0.
Proof.
  intros p q.
  rewrite ← mult_n_O.
```

```
  rewrite ← mult_n_O.
  reflexivity. Qed.
```

Exercise: 1 star, standard (mult_n_1) Use mult_n_Sm and mult_n_O to prove the following theorem. (Recall that 1 is S O.)

```
Theorem mult_n_1 : ∀ p : nat,
  p × 1 = p.
Proof.
  Admitted.
```

□

2.7 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 : ∀ n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros n.
  simpl. Abort.
```

The reason for getting stuck is that the definitions of both eqb and + begin by performing a `match` on their first argument. 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 $n = S\ n'$ 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 Rocq to consider separate cases where $n = O$ and $n = S\ n'$ is called `destruct`.

```
Theorem plus_1_neq_0 : ∀ n : nat,
  (n + 1) =? 0 = 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 Rocq to accept the theorem.

The annotation "as [| n']" is called an *intro pattern*. It tells Rocq 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 doesn't take any arguments. The second component gives a single name, n', since S is a unary constructor.

In each subgoal, Rocq 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 Rocq 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, Rocq simply expects you to prove each subgoal in sequence, one at a time. But it is a good idea to use bullets, and you should make a habit of doing it! For one thing, bullets make the structure of a proof apparent, improving readability. Moreover, bullets instruct Rocq 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 larger developments, where fragile proofs can lead to long debugging sessions!

There are no hard and fast rules for how proofs should be formatted in Rocq – 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 Rocq 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 (or, if you have a wide screen or good eyes, 120) 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 Rocq will fill in variable names automatically. This is generally considered bad style, since Rocq 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 : ∀ b c, andb b c = 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 instead of using bullets:

```
Theorem andb_commutative' : ∀ b c, andb b c = 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 :
  ∀ b c d, andb (andb b c) d = andb (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 1: Thus far, we've only ever used `simpl` to simplify the goal. It is also possible to use it to simplify hypotheses: use `simpl in` H, where H is a hypothesis, to simplify it. You may find this useful in this exercise.

Hint 2: You will eventually need to destruct both booleans, as in the theorems above. It is better to simplify the hypothesis before attempting to destruct the second boolean.

Hint 3: If you reach a contradiction in the hypotheses, focus on how to `rewrite` using that contradiction.

```
Theorem andb_true_elim2 : ∀ b c : bool,
  andb b c = true → c = true.
Proof.
  Admitted.
```

□

Before closing the chapter, we should 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 Rocq 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' : ∀ n : 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'' :
  ∀ 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 : ∀ n : nat,`

```
  0 =? (n + 1) = false.
Proof.
```

Admitted.

□

2.7.1 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 just 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 Rocq, we can specify its *precedence level* and its *associativity*. The precedence level n is specified by writing `at level n`; this helps Rocq 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 $\times$ say that the expression 1+2*3*4 is shorthand for (1+((2*3)*4)). Rocq 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*. Rocq 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 pair type type **bool**×**bool** (which we'll see in a later chapter) it guesses *type_scope*. Occasionally, it is necessary to help it out by writing, for example, (*x*×*y*)%**nat**, and sometimes in what Rocq prints it will use %**nat** to indicate what scope a notation is in.

Notation scopes also apply to numeral notations (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: Rocq's notation mechanism is not especially powerful. Don't expect too much from it.

2.7.2 Fixpoints and Structural Recursion (Optional)

Here is a copy of the definition of addition:

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

When Rocq 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. Rocq demands that some argument of *every* `Fixpoint` definition be "decreasing."

This requirement is a fundamental feature of Rocq's design: In particular, it guarantees that every function that can be defined in Rocq will terminate on all inputs. However, because Rocq'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 Rocq 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 Rocq to reject the whole file!)

2.8 More Exercises

2.8.1 Warmups

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 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 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 :
  ∀ (b c : bool),
  (andb b c = orb b c) →
  b = c.
Proof.
    Admitted.
```

□

2.8.2 Course Late Policies, Formalized

Suppose that a course has a grading policy based on late days, where a student's final letter grade is lowered if they submit too many homework assignments late.

In the next series of problems, we model this situation using the features of Rocq that we have seen so far and prove some simple facts about this grading policy.

```
Module LateDays.
```

First, we inroduce a datatype for modeling the "letter" component of a grade. `Inductive letter : Type :=`

```
  | A | B | C | D | F.
```

Then we define the modifiers – a `Natural A` is just a "plain" grade of `A`. `Inductive modifier : Type :=`

```
  | Plus | Natural | Minus.
```

A full **grade**, then, is just a **letter** and a **modifier**.

We might write, informally, "A-" for the Rocq value Grade A Minus, and similarly "C" for the Rocq value Grade C Natural. `Inductive` **grade** : `Type` :=
  Grade (*l*:**letter**) (*m*:**modifier**).

We will want to be able to say when one grade is "better" than another. In other words, we need a way to compare two grades. As with natural numbers, we could define **bool**-valued functions *grade_eqb*, *grade_ltb*, etc., and that would work fine. However, we can also define a slightly more informative type for comparing two values, as shown below. This datatype has three constructors that can be used to indicate whether two values are "equal", "less than", or "greater than" one another. (This definition also appears in the Rocq standard libary.)

```
Inductive comparison : Type :=
  | Eq
  | Lt
  | Gt.
```

Using pattern matching, it is not difficult to define the comparison operation for two letters *l1* and *l2* (see below). This definition uses two features of `match` patterns: First, recall that we can match against *two* values simultaneously by separating them and the corresponding patterns with comma ,. This is simply a convenient abbreviation for nested pattern matching. For example, the match expression on the left below is just shorthand for the lower-level "expanded version" shown on the right:

match l1, l2 with | A, A => Eq | A => match l2 with | A, _=> Gt | A => Eq end | _=> Gt end end As another shorthand, we can also match one of several possibilites by using | in the pattern. For example the pattern C , (A | B) stands for two cases: C, A and C, B.

```
Definition letter_comparison (l1 l2 : letter) : comparison :=
  match l1, l2 with
  | A, A ⇒ Eq
  | A, _ ⇒ Gt
  | B, A ⇒ Lt
  | B, B ⇒ Eq
  | B, _ ⇒ Gt
  | C, (A | B) ⇒ Lt
  | C, C ⇒ Eq
  | C, _ ⇒ Gt
  | D, (A | B | C) ⇒ Lt
  | D, D ⇒ Eq
  | D, _ ⇒ Gt
  | F, (A | B | C | D) ⇒ Lt
  | F, F ⇒ Eq
  end.
```

We can test the letter_comparison operation by trying it out on a few examples. `Compute` letter_comparison B A.

==> Lt Compute letter_comparison D D.
==> Eq Compute letter_comparison B F.
==> Gt

As a further sanity check, we can prove that the letter_comparison function does indeed give the result Eq when comparing a letter *l* against itself.

Exercise: 1 star, standard (letter_comparison) Theorem letter_comparison_Eq :
∀ *l*, letter_comparison *l l* = Eq.
Proof.
Admitted.
□

We can follow the same strategy to define the comparison operation for two grade modifiers. We consider them to be ordered as Plus > Natural > Minus. Definition modifier_comparison (*m1 m2* : **modifier**) : **comparison** :=
match *m1*, *m2* with
| Plus, Plus ⇒ Eq
| Plus, _ ⇒ Gt
| Natural, Plus ⇒ Lt
| Natural, Natural ⇒ Eq
| Natural, _ ⇒ Gt
| Minus, (Plus | Natural) ⇒ Lt
| Minus, Minus ⇒ Eq
end.

Exercise: 2 stars, standard (grade_comparison) Use pattern matching to complete the following definition.

(This ordering on grades is sometimes called "lexicographic" ordering: we first compare the letters, and we only consider the modifiers in the case that the letters are equal. I.e. all grade variants of A are greater than all grade variants of B.)

Hint: match against *g1* and *g2* simultaneously, but don't try to enumerate all the cases. Instead do case analysis on the result of a suitable call to letter_comparison to end up with just 3 possibilities.

Definition grade_comparison (*g1 g2* : **grade**) : **comparison**
. *Admitted*.

The following "unit tests" of your grade_comparison function should pass once you have defined it correctly.

Example test_grade_comparison1 :
(*grade_comparison* (Grade A Minus) (Grade B Plus)) = Gt.
Admitted.

Example test_grade_comparison2 :
(*grade_comparison* (Grade A Minus) (Grade A Plus)) = Lt.

```
  Admitted.

Example test_grade_comparison3 :
  (grade_comparison (Grade F Plus) (Grade F Plus)) = Eq.
  Admitted.

Example test_grade_comparison4 :
  (grade_comparison (Grade B Minus) (Grade C Plus)) = Gt.
  Admitted.
```

□

Now that we have a definition of grades and how they compare to one another, let us implement a late-penalty fuction.

First, we define what it means to lower the **letter** component of a grade. Since F is already the lowest grade possible, we just leave it alone.

```
Definition lower_letter (l : letter) : letter :=
  match l with
  | A ⇒ B
  | B ⇒ C
  | C ⇒ D
  | D ⇒ F
  | F ⇒ F
  end.
```

Our formalization can already help us understand some corner cases of the grading policy. For example, we might expect that if we use the lower_letter function its result will actually be lower, as claimed in the following theorem. But this theorem is not provable! (Do you see why?)

```
Theorem lower_letter_lowers: ∀ (l : letter),
  letter_comparison (lower_letter l) l = Lt.
Proof.
  intros l.
  destruct l.
  - simpl. reflexivity.
  - simpl. reflexivity.
  - simpl. reflexivity.
  - simpl. reflexivity.
  - simpl. Abort.
```

The problem, of course, has to do with the "edge case" of lowering F, as we can see like this:

```
Theorem lower_letter_F_is_F:
  lower_letter F = F.
Proof.
  simpl. reflexivity.
Qed.
```

With this insight, we can state a better version of the lower letter theorem that actually is provable. In this version, the hypothesis about F says that F is strictly smaller than l,

which rules out the problematic case above. In other words, as long as l is bigger than F, it will be lowered.

Exercise: 2 stars, standard (lower_letter_lowers) Prove the following theorem.

```
Theorem lower_letter_lowers:
  ∀ (l : letter),
    letter_comparison F l = Lt →
    letter_comparison (lower_letter l) l = Lt.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard (lower_grade) We can now use the lower_letter definition as a helper to define what it means to lower a grade by one step. Complete the definition below so that it sends a grade g to one step lower (unless it is already Grade F Minus, which should remain unchanged). Once you have implemented it correctly, the subsequent "unit test" examples should hold trivially.

Hint: To make this a succinct definition that is easy to prove properties about, you will probably want to use nested pattern matching. The outer match should not match on the specific letter component of the grade – it should consider only the modifier. You should definitely *not* try to enumerate all of the cases.

Our solution is under 10 lines of code total.

```
Definition lower_grade (g : grade) : grade
  . Admitted.

Example lower_grade_A_Plus :
  lower_grade (Grade A Plus) = (Grade A Natural).
Proof.
  Admitted.

Example lower_grade_A_Natural :
  lower_grade (Grade A Natural) = (Grade A Minus).
Proof.
  Admitted.

Example lower_grade_A_Minus :
  lower_grade (Grade A Minus) = (Grade B Plus).
Proof.
  Admitted.

Example lower_grade_B_Plus :
  lower_grade (Grade B Plus) = (Grade B Natural).
Proof.
  Admitted.
```

```
Example lower_grade_F_Natural :
  lower_grade (Grade F Natural) = (Grade F Minus).
Proof.
    Admitted.

Example lower_grade_twice :
  lower_grade (lower_grade (Grade B Minus)) = (Grade C Natural).
Proof.
    Admitted.

Example lower_grade_thrice :
  lower_grade (lower_grade (lower_grade (Grade B Minus))) = (Grade C Minus).
Proof.
    Admitted.
```

Rocq makes no distinction between an `Example` and a `Theorem`. We state the following as a `Theorem` only as a hint that we will use it in proofs below.

```
Theorem lower_grade_F_Minus
: lower_grade (Grade F Minus) = (Grade F Minus).
Proof.
    Admitted.
```

Exercise: 3 stars, standard (lower_grade_lowers) Prove the following theorem, which says that, as long as the grade starts out above F-, the lower_grade option does indeed lower the grade. As usual, destructing everything in sight is *not* a good idea. Judicious use of `destruct` along with rewriting is a better strategy.

Hint: If you defined your grade_comparison function as suggested, you will only need to destruct a **letter** in one case. The case for F will probably benefit from lower_grade_F_Minus.

```
Theorem lower_grade_lowers :
  ∀ (g : grade),
    grade_comparison (Grade F Minus) g = Lt →
    grade_comparison (lower_grade g) g = Lt.
Proof.
    Admitted.
```

□

Now that we have implemented and tested a function that lowers a grade by one step, we can implement a specific late-days policy. Given a number of *late_days*, the apply_late_policy function computes the final grade from g, the initial grade.

This function encodes the following policy:

```
Definition apply_late_policy (late_days : nat) (g : grade) : grade :=
  if late_days <? 9 then g
  else if late_days <? 17 then lower_grade g
  else if late_days <? 21 then lower_grade (lower_grade g)
  else lower_grade (lower_grade (lower_grade g)).
```

Sometimes it is useful to be able to "unfold" a definition to be able to make progress on a proof. Soon, we will see how to do this in a much simpler way automatically, but for now, it is easy to prove that a use of any definition like apply_late_policy is equal to its right hand side just by using reflexivity.

This result is useful because it allows us to use `rewrite` to expose the internals of the definition.

```
Theorem apply_late_policy_unfold :
  ∀ (late_days : nat) (g : grade),
    (apply_late_policy late_days g)
    =
    (if late_days <? 9 then g else
       if late_days <? 17 then lower_grade g
       else if late_days <? 21 then lower_grade (lower_grade g)
            else lower_grade (lower_grade (lower_grade g))).
Proof.
  intros. reflexivity.
Qed.
```

Now let's prove some properties about this policy.

The next theorem states that if a student accrues no more than eight late days throughout the semester, their grade is unaffected. It is easy to prove: once you use the apply_late_policy_unfold you can rewrite using the hypothesis.

Exercise: 2 stars, standard (no_penalty_for_mostly_on_time)

```
Theorem no_penalty_for_mostly_o
:
  ∀ (late_days : nat) (g : grade),
    (late_days <? 9 = true) →
    apply_late_policy late_days g = g.
Proof.
    Admitted.
```

□

The following theorem states that, if a student has between 9 and 16 late days, their final grade is lowered by one step.

Exercise: 2 stars, standard (graded_lowered_once)

```
Theorem grade_lowered_once :
  ∀ (late_days : nat) (g : grade),
    (late_days <? 9 = false) →
    (late_days <? 17 = true) →
    (apply_late_policy late_days g) = (lower_grade g).
Proof.
    Admitted.
```

□

```
End LateDays.
```

2.8.3 Binary Numerals

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 S O 2 B0 (B1 Z) S (S O) 3 B1 (B1 Z) S (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.

(Comprehension check: What unary numeral does B0 Z represent?)

```
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.
Example test_bin_incr7 : bin_to_nat (B0 (B0 (B0 (B1 Z)))) = 8.
  Admitted.
```

□

2.9 Optional: 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 instructor 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 Rocq 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 first run *rocq compile -Q . LF Basics.v* and then run *rocq compile -Q . LF BasicsTest.v*; or, if you have make installed, you can run *make BasicsTest.vo*. (Make sure you do this in a directory that also contains a file named *_CoqProject* containing the single line *-Q . LF*.)

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.
- 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 in this case, 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. (In the present file, there are no advanced exercises.)

- Fourth, you will see a list of "Allowed Axioms". These are unproven theorems that your solution is permitted to depend upon, aside from the fundamental axioms of Rocq's logic. 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` command to work, Rocq needs to be able to find a compiled version of the previous chapter (*Basics.v*). This compiled version, called *Basics.vo*, is analogous to the *.class* files compiled from *.java* source files and the *.o* files compiled from .c files.

To compile *Basics.v* and obtain *Basics.vo*, first make sure that the files *Basics.v*, *Induction.v*, and *_CoqProject* are in the current directory.

The *_CoqProject* file should contain just the following line:

- Q . LF

This maps the current directory (".", which contains *Basics.v*, *Induction.v*, etc.) to the prefix (or "logical directory") "*LF*". Proof General, CoqIDE, and VSCoq read *_CoqProject* automatically, to find out to where to look for the file *Basics.vo* corresponding to the library LF.Basics.

Once the files are in place, there are various ways to build *Basics.vo* from an IDE, or you can build it from the command line. From an IDE...

- 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*. This can also be found in the menu: "Coq" > "Auto Compilation" > "Compile Before Require".

- In CoqIDE: One thing you can do on all platforms is open *Basics.v*; then, in the "Compile" menu, click on "Compile Buffer".

- For VSCode users, open the terminal pane at the bottom and then follow the command line instructions below. (If you downloaded the project setup .tgz file, just doing 'make' should build all the code.)

To compile *Basics.v* from the command line...

- First, generate a *Makefile* using the *rocq makefile* utility, which comes installed with Rocq. (If you obtained the whole volume as a single archive, a *Makefile* should already exist and you can skip this step.)

  rocq makefile -f _*CoqProject* *.v -o Makefile

  You should rerun that command whenever you add or remove Rocq files in this 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 Rocq compiler, *rocq compile*. You can also run *rocq compile* directly:

  rocq compile -Q . LF Basics.v

- Since *make* also calculates dependencies between source files to compile them in the right order, *make* should generally be preferred over running *rocq compile* explicitly. But as a last (but not terrible) resort, you can simply compile each file manually as you go. For example, before starting work on the present chapter, you would need to run the following command:

  rocq compile -Q . LF Basics.v

  Then, once you've finished this chapter, you'd do

  rocq compile -Q . LF Induction.v

  to get ready to work on the next one. If you ever remove the .vo files, you'd need to give both commands again (in that order).

Troubleshooting:

- For many of the alternatives above you need to make sure that the *rocq* executable is in your *PATH*.

- If you get complaints about missing identifiers, it may be because the "load path" for Rocq is not set up correctly. The `Print` *LoadPath*. command may be helpful in sorting out such issues.

- When trying to compile a later chapter, if you see a message like

  Compiled library Induction makes inconsistent assumptions over library Basics

  a common reason is that the library Basics was modified and recompiled without also recompiling `Induction` which depends on it. Recompile `Induction`, or everything if too many files are affected (for instance by running *make* and if even this doesn't work then *make clean*; *make*).

- If you get complaints about missing identifiers later in this file it may be because the "load path" for Rocq 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 Rocq on your machine. It may be that commands (like *rocq compile*) that you execute in a terminal window are getting a different version of Rocq than commands executed by Proof General or CoqIDE.

- 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 rocq from the command line*. This typically happens when there are two incompatible versions of Rocq installed on your system (one associated with CoqIDE, and one associated with *rocq* from the terminal). The workaround for this situation is compiling using CoqIDE only (i.e. choosing "make" from the menu), and avoiding using *rocq* 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 add_0_r_firsttry : ∀ n:nat,
  n + 0 = n.
```

... 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 just fine, but in the branch where $n =$ S *n'* for some *n'* we get stuck in exactly the same way.

```
Theorem add_0_r_secondtry : ∀ n:nat,
  n + 0 = n.
Proof.
  intros n. destruct n as [| n'] eqn:E.
  -
    reflexivity.   -
    simpl. Abort.
```

We could use `destruct n'` to get a bit further, but, since n can be arbitrarily large, we'll never get all the way 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 a discrete math course, probably) 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 $P(\mathsf{O})$ holds;
- show that, for any n', if $P(n')$ holds, then so does $P(\mathsf{S}\ n')$;
- conclude that $P(n)$ holds for all n.

In Rocq, the steps are the same, except we typically encounter them in reverse order: we begin with the goal of proving $P(n)$ for all n and apply the `induction` tactic to break it down into two separate subgoals: one where we must show $P(\mathsf{O})$ and another where we must show $P(n') \to P(\mathsf{S}\ n')$. Here's how this works for the theorem at hand...

```
Theorem add_0_r : ∀ n:nat, n + 0 = n.
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 a vertical bar, |. (Strictly speaking, we can omit the `as`... clause and Rocq will choose names for us. In practice, this is a bad practice, as Rocq'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 easily by simplification.

In the second subgoal, n is replaced by $\mathsf{S}\ n'$, 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 $\mathsf{S}\ n' = (\mathsf{S}\ n') + 0$, which simplifies to $\mathsf{S}\ n' = \mathsf{S}\ (n' + 0)$, which in turn follows from IHn'.

```
Theorem minus_n_n : ∀ n,
```

```
  minus n n = 0.
Proof.
  intros n. induction n as [| n' IHn'].
  -
    simpl. reflexivity.
  -
    simpl. rewrite → 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 mul_0_r : ∀ n:nat,
  n × 0 = 0.
Proof.
    Admitted.

Theorem plus_n_Sm : ∀ n m : nat,
  S (n + m) = n + (S m).
Proof.
    Admitted.

Theorem add_comm : ∀ n m : nat,
  n + m = m + n.
Proof.
    Admitted.

Theorem add_assoc : ∀ n m p : 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
  | O ⇒ O
  | S n' ⇒ S (S (double n'))
  end.
```

Use induction to prove this simple fact about double:

```
Lemma double_plus : ∀ n, double n = n + n .
```

```
Proof.
    Admitted.
    □
```

Exercise: 2 stars, standard (eqb_refl) The following theorem relates the computational equality =? on **nat** with the definitional equality = on **bool**.

```
Theorem eqb_refl : ∀ n : nat,
  (n =? n) = true.
Proof.
    Admitted.
    □
```

Exercise: 2 stars, standard, optional (even_S) One inconvenient aspect of our definition of even n is the recursive call on n - 2. This makes proofs about even 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 even (S n) that works better with induction:

```
Theorem even_S : ∀ n : nat,
  even (S n) = negb (even n).
Proof.
    Admitted.
    □
```

3.4 Proofs Within Proofs

In Rocq, 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 involve 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 use the required fact "in place" and then prove it as a separate step. The `replace` tactic allows us to do this.

```
Theorem mult_0_plus' : ∀ n m : nat,
  (n + 0 + 0) × m = n × m.
Proof.
  intros n m.
  replace (n + 0 + 0) with n.
  - reflexivity.
  - rewrite add_comm. simpl. rewrite add_comm. reflexivity.
Qed.
```

The tactic `replace` *e1* `with` *e2* tactic introduces two subgoals.

The first subgoal is the same as the one at the point where we invoke `replace`, except that *e1* is replaced by *e2*. The second subgoal is the equality *e1* = *e2* itself.

As another 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 (add_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` → add_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 add_comm.
Abort.
```

To use add_comm at the point where we need it, we can rewrite $n + m$ to $m + n$ using `replace` and then prove $n + m = m + n$ using add_comm.

```
Theorem plus_rearrange : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  replace (n + m) with (m + n).
  - reflexivity.
  - rewrite add_comm. 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 answer 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 Rocq, 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 probably 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. 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 will 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 reasonably clear standard for judging proofs good or bad.

Because we are using Rocq 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 add_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.
```

Rocq 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 add_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 Rocq you might 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: Rocq 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 usually 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 Rocq proof is completely implicit, whereas the informal proof reminds the reader several times where things stand).

Exercise: 2 stars, advanced, optional (add_comm_informal) Translate your solution for add_comm into an informal proof:
Theorem: Addition is commutative.
Proof:

```
Definition manual_grade_for_add_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 add_assoc as a model. Don't just paraphrase the Rocq tactics into English!
Theorem: (n =? n) = true for any n.
Proof:

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

□

3.6 More Exercises

Exercise: 3 stars, standard, especially useful (mul_comm) Use `replace` to help prove add_shuffle3. You don't need to use induction yet.

```
Theorem add_shuffle3 : ∀ n m p : nat,
  n + (m + p) = m + (n + p).
Proof.
  Admitted.
```

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

```
Theorem mul_comm : ∀ m n : nat,
  m × n = n × 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!)

```
Theorem leb_refl : ∀ n:nat,
  (n <=? n) = true.
Proof.
  Admitted.

Theorem zero_neqb_S : ∀ n:nat,
  0 =? (S n) = false.
Proof.
  Admitted.

Theorem andb_false_r : ∀ b : bool,
  andb b false = false.
Proof.
  Admitted.

Theorem S_neqb_0 : ∀ n:nat,
  (S n) =? 0 = false.
Proof.
  Admitted.

Theorem mult_1_l : ∀ n:nat, 1 × n = n.
Proof.
  Admitted.

Theorem all3_spec : ∀ b c : bool,
  orb
    (andb b c)
    (orb (negb b)
         (negb c))
  = true.
Proof.
```

```
  Admitted.

Theorem mult_plus_distr_r : ∀ n m p : nat,
  (n + m) × p = (n × p) + (m × p).
Proof.
  Admitted.

Theorem mult_assoc : ∀ n m p : nat,
  n × (m × p) = (n × m) × p.
Proof.
  Admitted.
  □
```

3.7 Nat to Bin and Back to Nat

Recall the **bin** type we defined in Basics:

```
Inductive bin : Type :=
  | Z
  | B0 (n : bin)
  | B1 (n : bin)
.
```

Before you start working on the next exercise, replace the stub definitions of incr and bin_to_nat, below, with your solution from Basics. That will make it possible for this file to be graded on its own.

```
Fixpoint incr (m:bin) : bin
  . Admitted.

Fixpoint bin_to_nat (m:bin) : nat
  . Admitted.
```

In Basics, we did some unit testing of bin_to_nat, but we didn't prove its correctness. Now we'll do so.

Exercise: 3 stars, standard, especially useful (binary_commute) Prove that the following diagram commutes:

incr bin ——————-> bin | | bin_to_nat | | bin_to_nat | | v v nat ——————-> nat S

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.

If you want to change your previous definitions of incr or bin_to_nat to make the property easier to prove, feel free to do so!

```
Theorem bin_to_nat_pres_incr : ∀ b : bin,
  bin_to_nat (incr b) = 1 + bin_to_nat b.
Proof.
```

Admitted.

□

Exercise: 3 stars, standard (nat_bin_nat) 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 **bin**, and convert it back, we get the same **nat** which we started with.

Hint: This proof should go through smoothly using the previous exercise about incr as a lemma. If not, revisit your definitions of the functions involved and consider whether they are more complicated than necessary: the shape of a proof by induction will match the recursive structure of the program being verified, so make the recursions as simple as possible.

```
Theorem nat_bin_nat : ∀ n, bin_to_nat (nat_to_bin n) = n.
Proof.
  Admitted.
```

□

3.8 Bin to Nat and Back to Bin (Advanced)

The opposite direction – starting with a **bin**, converting to **nat**, then converting back to **bin** – turns out to be problematic. That is, the following theorem does not hold.

```
Theorem bin_nat_bin_fails : ∀ b, nat_to_bin (bin_to_nat b) = b.
Abort.
```

Let's explore why that theorem fails, and how to prove a modified version of it. We'll start with some lemmas that might seem unrelated, but will turn out to be relevant.

Exercise: 2 stars, advanced (double_bin) Prove this lemma about double, which we defined earlier in the chapter.

```
Lemma double_incr : ∀ n : nat, double (S n) = S (S (double n)).
Proof.
  Admitted.
```

Now define a similar doubling function for **bin**.

```
Definition double_bin (b:bin) : bin
  . Admitted.
```

Check that your function correctly doubles zero.

```
Example double_bin_zero : double_bin Z = Z.
```

Admitted.

Prove this lemma, which corresponds to double_incr.

```
Lemma double_incr_bin : ∀ b,
  double_bin (incr b) = incr (incr (double_bin b)).
Proof.
  Admitted.
```

□

Let's return to our desired theorem:

```
Theorem bin_nat_bin_fails : ∀ b, nat_to_bin (bin_to_nat b) = b.
Abort.
```

The theorem fails because there are some **bin** such that we won't necessarily get back to the *original* **bin**, but instead to an "equivalent" **bin**. (We deliberately leave that notion undefined here for you to think about.)

Explain in a comment, below, why this failure occurs. Your explanation will not be graded, but it's important that you get it clear in your mind before going on to the next part. If you're stuck on this, think about alternative implementations of double_bin that might have failed to satisfy double_bin_zero yet otherwise seem correct.

To solve that problem, we can introduce a *normalization* function that selects the simplest **bin** out of all the equivalent **bin**. Then we can prove that the conversion from **bin** to **nat** and back again produces that normalized, simplest **bin**.

Exercise: 4 stars, advanced (bin_nat_bin) Define normalize. You will need to keep its definition as simple as possible for later proofs to go smoothly. Do not use bin_to_nat or nat_to_bin, but do use double_bin.

Hint: Structure the recursion such that it *always* reaches the end of the **bin** and *only* processes each bit only once. Do not try to "look ahead" at future bits.

```
Fixpoint normalize (b:bin) : bin
  . Admitted.
```

It would be wise to do some Example proofs to check that your definition of normalize works the way you intend before you proceed. They won't be graded, but fill them in below.

Finally, prove the main theorem. The inductive cases could be a bit tricky.

Hint: 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. We have one lemma for the B0 case (which also makes use of double_incr_bin) and another for the B1 case.

```
Theorem bin_nat_bin : ∀ b, nat_to_bin (bin_to_nat b) = normalize b.
Proof.
  Admitted.
```

□

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 the following):

```
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.
```

Functions for extracting the first and second components of a pair can then be defined by pattern matching.

```
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 will be convenient to write them with the standard mathematical notation (x,y) instead of pair x y. We can tell Rocq 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 understanding 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.

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 just `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 : ∀ (p : natprod),
  p = (fst p, snd p).
Proof.
  intros p. destruct p as [n m]. simpl. reflexivity. Qed.
```

Notice that, by contrast with the behavior of `destruct` on **nat**s, where it generates two subgoals, `destruct` generates just one subgoal here. That's because **natprod**s can only be constructed in one way.

Exercise: 1 star, standard (snd_fst_is_swap) `Theorem snd_fst_is_swap : ∀ (p : natprod),`
`  (snd p, fst p) = swap_pair p.`
`Proof.`
`  Admitted.`
□

Exercise: 1 star, standard, optional (fst_swap_is_snd) `Theorem fst_swap_is_snd : ∀ (p : natprod),`
`  fst (swap_pair p) = 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 convenient to write lists in familiar 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 Rocq 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 Rocq 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 "rocq doc" tool that the bracketed part should be displayed as Rocq 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 Rocq's syntax for declaring n-ary notations and translating them to nested sequences of binary constructors.

Again, don't worry if some of these parsing details are puzzling: all the notations you'll need in this course will be defined for you.

Repeat

Next let's look at several functions for constructing and manipulating lists. First is the `repeat` function, which 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
  | O ⇒ 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 ⇒ O
```

```
  | h :: t ⇒ S (length t)
  end.
```

Append

The app function appends (concatenates) 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 more handy functions for working 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.
```

For the next problem, countoddmembers, we're giving you a header that uses the keyword `Definition` instead of `Fixpoint`. The point of stating the question this way is to encourage you to implement the function by using already-defined functions, rather than writing your own recursive definition.

```
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.

Hint: there are natural ways of writing alternate that fail to satisfy Rocq's requirement that all `Fixpoint` definitions be *structurally recursive*, as mentioned in Basics. If you encounter this difficulty, consider pattern matching against both lists at the same time with the "multiple pattern" syntax we've seen before.

```
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 way of representating 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 completed with `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 those 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*.)

We've deliberately given you a header that does not give explicit names to the arguments. Implement sum in terms of an already-defined function, without changing the header.

```
Definition sum : bag → bag → bag
  . Admitted.

Example test_sum1: count 1 (sum [1;2;3] [1;4;1]) = 3.
  Admitted.
```

```
Definition add (v : nat) (s : 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.

Fixpoint member (v : nat) (s : 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 : nat) (s : 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:nat) (s: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 included (s1 : bag) (s2 : bag) : bool
  . Admitted.

Example test_included1: included [1;2] [2;1;4;1] = true.
  Admitted.
Example test_included2: included [1;2;2] [2;1;4;1] = false.
  Admitted.
```
□

Exercise: 2 stars, standard, optional (add_inc_count) Adding a value to a bag should increase the value's count by one. State this as a theorem and prove it in Rocq.

```
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 `reflexivity` is enough for this theorem...

```
Theorem nil_app : ∀ l : 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 : ∀ l:natlist,
  pred (length l) = length (tl l).
Proof.
  intros l. destruct l as [| n l'].
  -
    reflexivity.
  -
    reflexivity. Qed.
```

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

Usually, though, interesting theorems about lists require induction for their proofs. We'll see how to do this next.

(Micro-Sermon: As we get deeper into this material, simply *reading* proof scripts will not help you very much. Rather, it is important to step through the details of each one using Rocq 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 else `S` applied to another number; and a list can be either `nil` or else `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. Thus, 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 : ∀ l1 l2 l3 : 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 we saw with 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 Rocq 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 Rocq 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 Rocq proof. So a natural-language proof – one written for human readers – would include more explicit signposts; in particular, it helps the reader stay oriented to 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. □

Generalizing Statements

In some situations, it is necessary to generalize a statement in order to prove it by induction. Intuitively, the reason is that a more general statement also yields a more general (stronger) inductive hypothesis. If you find yourself stuck in a proof, it may help to step back and see whether you can prove a stronger statement.

```
Theorem repeat_double_firsttry : ∀ c n: nat,
  repeat n c ++ repeat n c = repeat n (c + c).
Proof.
  intros c. induction c as [| c' IHc'].
  -
    intros n. simpl. reflexivity.
  -
    intros n. simpl.
Abort.
```

To get a more general inductive hypothesis, we can generalize the statement as follows:

```
Theorem repeat_plus: ∀ c1 c2 n: nat,
    repeat n c1 ++ repeat n c2 = repeat n (c1 + c2).
Proof.
```

```
  intros c1 c2 n.
  induction c1 as [| c1' IHc1'].
  - simpl. reflexivity.
  - simpl.
    rewrite ← IHc1'.
    reflexivity.
  Qed.
```

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


    simpl.
    rewrite ← IHl'.
Abort.
```

A first attempt to make progress would be to prove exactly the statement that we are missing at this point. But this attempt will fail because the inductive hypothesis is not general enough. Theorem app_rev_length_S_firsttry: ∀ l n,
  length (rev l ++ [n]) = S (length (rev l)).
Proof.
  intros l. induction l as [| m l' IHl'].
  -

```
    intros n. simpl. reflexivity.
  -
    intros n. simpl.
Abort.
```

It turns out that the above lemma is more specific than it needs to be. We can strengthen the lemma to work not only on reversed lists but on general lists.

```
Theorem app_length_S:
∀ l n,
  length (l ++ [n]) = S (length l).
Proof.
  intros l n. induction l as [| m l' IHl'].
  -
    simpl. reflexivity.
  -
    simpl.
    rewrite IHl'.
    reflexivity.
Qed.
```

Now we can complete the original proof.

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

Note that the app_length_S lemma we proved above is pretty narrow, requiring that the second list contains only a single element. We can prove a more general version for any two lists.

```
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.
```

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

Theorem: For all lists *l1* and *l2*, length (*l1* ++ *l2*) = length *l1* + length *l2*.
Proof: By induction on *l1*.

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

  length (□ ++ l2) = length □ + length l2,

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

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

  length (l1' ++ l2) = length l1' + length l2.

  We must show

  length ((n::l1') ++ l2) = length (n::l1') + length l2.

  This follows directly from the definitions of length and ++ together with the induction hypothesis. □

Theorem: For all lists *l*, length (rev *l*) = length *l*.
Proof: By induction on *l*.

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

  length (rev □) = length □,

  which follows directly from the definitions of length and rev.

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

  length (rev l') = length l'.

  We must show

  length (rev (n :: l')) = length (n :: l').

  By the definition of rev, this follows from

  length ((rev l') ++ *n*) = S (length l')

  which, by the previous lemma, is the same as

  length (rev l') + length *n* = S (length l').

  This follows directly from the induction hypothesis and the definition of length. □

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*, length (rev *l*) = length *l*.

Proof: First observe, by a straightforward induction on *l*, that length (*l* ++ [*n*]) = S (length *l*) for any *l*. The main property then follows by another induction on *l*, using the observation together with the induction hypothesis in the case where *l* = *n'*::*l'*. □

Which style is preferable in a given situation depends on the sophistication of the expected audience and 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 because we're trying to be ultra-clear about the details.

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.

Rocq'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 Rocq 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 defined identifier 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 VSRocq, you can open a tab for performing searches with *Command-Control-K*.

4.4.3 List Exercises, Part 1

Exercise: 3 stars, standard (list_exercises) More practice with lists:

```
Theorem app_nil_r : ∀ l : natlist,
  l ++ [] = l.
Proof.
  Admitted.

Theorem rev_app_distr: ∀ l1 l2 : natlist,
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
```

```
  Admitted.
```

An *involution* is a function that is its own inverse. That is, applying the function twice yield the original input.

```
Theorem rev_involutive : ∀ l : natlist,
  rev (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 : ∀ l1 l2 l3 l4 : natlist,
  l1 ++ (l2 ++ (l3 ++ l4)) = ((l1 ++ l2) ++ l3) ++ l4.
Proof.
  Admitted.
```

An exercise about your implementation of nonzeros:

```
Lemma nonzeros_app : ∀ l1 l2 : natlist,
  nonzeros (l1 ++ l2) = (nonzeros l1) ++ (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 :
  (eqblist nil nil = true).
  Admitted.

Example test_eqblist2 :
  eqblist [1;2;3] [1;2;3] = true.
  Admitted.

Example test_eqblist3 :
  eqblist [1;2;3] [1;2;4] = false.
  Admitted.

Theorem eqblist_refl : ∀ l:natlist,
  true = 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
: ∀ (s : bag),
  1 <=? (count 1 (1 :: s)) = true.
Proof.
  Admitted.
```
□

The following lemma about leb might help you in the next exercise (it will also be useful in later chapters).

```
Theorem leb_n_Sn : ∀ n,
  n <=? (S n) = 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
∀ (s : bag),
  (count 0 (remove_one 0 s)) <=? (count 0 s) = 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 Rocq. (You may find that the difficulty of the proof depends on how you defined count!

Hint: If you defined count using =? you may find it useful to know that `destruct` works on arbitrary expressions, not just simple identifiers.)

Exercise: 3 stars, advanced (involution_injective) Prove that every involution is injective.

Involutions were defined above in rev_involutive. An *injective* function is one-to-one: it maps distinct inputs to distinct outputs, without any collisions.

```
Theorem involution_injective : ∀ (f : nat → nat),
```

```
    (∀ n : nat, n = f (f n)) → (∀ n1 n2 : nat, f n1 = f n2 → n1 = n2).
Proof.
  Admitted.
```

□

Exercise: 2 stars, advanced (rev_injective) Prove that rev is injective. Do not prove this by induction – that would be hard. Instead, re-use the same proof technique that you used for involution_injective. (But: Don't try to use that exercise directly as a lemma: the types are not the same!)

```
Theorem rev_injective : ∀ (l1 l2 : natlist),
  rev l1 = rev l2 → l1 = l2.
Proof.
  Admitted.
```

□

4.5 Options

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

```
Fixpoint nth_bad (l:natlist) (n: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 don't know whether that value actually appears in the input or whether we gave bad arguments. A better alternative is to change the return type of nth_bad to include an error value as a possible outcome. We call this new 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 are allowed to be be capitalized.)

```
Fixpoint nth_error (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ match n with
                | O ⇒ 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 the details.)

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 Rocq, 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 flexibility to change representations later if we want to.

We'll also need an equality test for **id**s:

```
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 : ∀ x, eqb_id x x = true.`

```
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 :`

```
  ∀ (d : partial_map) (x : id) (v: nat),
    find x (update d x v) = Some v.
Proof.
  Admitted.
```

□

Exercise: 1 star, standard (update_neq) `Theorem update_neq :`

```
  ∀ (d : partial_map) (x y : id) (o: nat),
    eqb_id x y = false → find x (update d y o) = find x d.
Proof.
  Admitted.
```

□ `End` PartialMap.

Chapter 5

Library LF.Poly

5.1 Poly: Polymorphism and Higher-Order Functions

```
Set Warnings "-notation-overridden".
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

In the last chapter, we worked 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: not only would we have to make up different constructor names for each datatype, but – even worse – we would also need to define new versions of all the list manipulating functions (length, app, rev, etc.) and all their properties (rev_length, app_assoc, etc.) for each new definition.

To avoid all this repetition, Rocq 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. We can now write **list nat** instead of **natlist**.

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 → Type.
```

The 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 provide, as a first argument, the type of the list they are building. For example, nil **nat** is 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 → **list** X does not explain the meaning of X. (X : Type) → **list** X comes closer. Rocq's notation for this situation is ∀ X : Type, **list** X.

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

Similarly, the type of cons from the definition looks like X → **list** X → **list** X, but using this convention to explain the meaning of X results in the type ∀ X, X → **list** X → **list** X.

```
Check cons : ∀ X : Type, X → list X → list X.
```

(A side note on notations: In .v files, the "forall" quantifier is spelled out in letters. In the corresponding HTML files (and in the way some IDEs show .v files, depending on the settings of their display controls), ∀ 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, optional (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.
```
□

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 Rocq 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 Rocq 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`. Rocq 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, Rocq 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 Rocq 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 Rocq 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 Rocq *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 the (leading) argument names to be treated as implicit, each surrounded by curly braces.

```
Arguments nil {X}.
Arguments cons {X}.
Arguments repeat {X}.
```

Now we don't have to supply any type arguments at all in the example:

```
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 Rocq 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 to be implicit is that, once in a while, Rocq does not have enough local information to determine a type argument; in such cases, we need to tell Rocq 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, Rocq prints the corresponding error message, but continues processing the rest of the file.)

Here, Rocq 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 Rocq has more information available when it gets to the "application" of `nil`):

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

Alternatively, we can disable the implicit argument to `nil` by prefixing the function name with `@`. This allows us to apply `@nil` *explicitly* to an appropriate type.

```
Check @nil : ∀ X : Type, list X.

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

(Note that we cannot just write `nil nat` here, without the `@`: the `Implicit Arguments` declaration above means that `nil nat` is now interpreted by Rocq as `@nil ?X nat`, where `?X` is some unknown type that should be filled in by implicit argument synthesis.) `Fail Check (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, Rocq 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 (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 : ∀ (X : Type), ∀ l : list X,
  l ++ [] = l.
Proof.
  Admitted.

Theorem app_assoc : ∀ A (l m n : list A),
  l ++ m ++ n = (l ++ m) ++ n.
Proof.
```

```
  Admitted.
Lemma app_length : ∀ (X : Type) (l1 l2 : list X),
  length (l1 ++ l2) = length l1 + length l2.
Proof.
  Admitted.
  □
```

Exercise: 2 stars, standard (more_poly_exercises) Here are some slightly more interesting ones...

```
Theorem rev_app_distr: ∀ X (l1 l2 : list X),
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
  Admitted.
Theorem rev_involutive : ∀ X : Type, ∀ l : list X,
  rev (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 : Type) : Type :=
| 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* (i.e., the types of pairs):

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

(The annotation : *type_scope* tells Rocq 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×Y confused. Remember that (*x*,*y*) is a *value* built from two other values, while X×Y is a *type* built from two other types. If *x* has type X and *y* has type Y, then (*x*,*y*) has type X×Y.

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

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

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

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

- 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
               | O ⇒ 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 : ∀ X : Type, list X → 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. – Rocq 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 : ∀ X : Type, (X → X) → X → 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 Xs and a *predicate* on X (a function from X to **bool**) and "filtering" the list to yield 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 even and a list of numbers l, it returns a list containing just the even members of l.

```
Example test_filter1: filter even [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 odd 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. Indeed, 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 ⇒ $n \times n$) 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 Rocq 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 Rocq function partition that, given a set X, a predicate of type X → bool and a list X, 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 odd [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 $l = [n1, n2, n3, ...]$ and returns the list $[f\ n1, f\ n2, f\ n3, ...]$, where f has been applied to each element of l in turn. For example:

```
Example test_map1: map (fun x ⇒ 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 odd [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 ⇒ [even n;odd 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. (Hint: You may need to define an auxiliary lemma.)

```
Theorem map_rev : ∀ (X Y : Type) (f : X → Y) (l : list X),
  map f (rev l) = rev (map f l).
Proof.
    Admitted.
    □
```

Exercise: 2 stars, standard, especially useful (flat_map) The function map maps a list X to a list Y using a function of type X → Y. We can define a similar function, flat_map, which maps a list X to a list Y using a function f of type X → list Y. Your definition should work by 'flattening' the results of f, like so:

flat_map (fun n => n;n+1;n+2) 1;5;10 = 1; 2; 3; 5; 6; 7; 10; 11; 12.

```
Fixpoint flat_map {X Y: Type} (f: X → list Y) (l: list X)
                  : list Y
  . 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 Rocq 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))).]]

```
Example fold_example1 :
  fold andb [true;true;false;true] true = false.
Proof. reflexivity. Qed.

Example fold_example2 :
  fold mult [1;2;3;4] 1 = 24.
Proof. reflexivity. Qed.

Example fold_example3 :
  fold app [[1];[];[2;3];[4]] [] = [1;2;3;4].
Proof. reflexivity. Qed.

Example foldexample4 :
  fold (fun l n ⇒ length l + n) [[1];[];[2;3;2];[4]] 0 = 5.
Proof. reflexivity. Qed.
```

Exercise: 1 star, standard, optional (fold_types_different) Observe that the type of `fold` is parameterized by *two* type variables, X and Y, and the parameter f is a binary operator that takes an X and a Y and returns a Y. Example foldexample4 above shows one instance where it is useful for X and Y to be different. Can you think of any others?

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.

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.
```

Similarly, we can write: `Definition fold_plus :=`
`fold plus.`

```
Check fold_plus : list nat → nat → nat.
```

What's happening here is called *partial application*. In Rocq, the type constructor → is right-associative, meaning a function type like A → B → C is parsed like A → (B → C), or "a function from A to a function from B to C."

We can think of `fold` not as a three-argument function, but as a one-argument function that:

1. Takes an argument f of type X → Y → Y 2. Returns a function of type **list** X → Y → Y that "remembers" f]

When we write `fold plus`, we're giving `fold` its first argument, plus, and getting back a specialized function that can sum up the elements of any list of numbers. This new function still expects two more arguments: a list and a starting value.

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: You may end up in a situation where you feel like `simpl` should be able to simplify fold_length, but does not do anything. In such cases, you can use the `unfold` tactic to inline the definition of a function prior to simplification, e.g., `unfold fold_length. simpl.`. This tactic will be discussed further in the following chapter.

```
Theorem fold_length_correct : ∀ X (l : 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* stating that fold_map is correct, and prove it in Rocq. (Hint: again, remember that `unfold`ing before `simpl`ifying may help.)

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

□

Exercise: 2 stars, advanced (currying) The type X → Y → Z can be read as describing functions that take two arguments, one of type X and another of type Y, and return an output of type Z. Recall from our discussion of partial application that this type is written X → (Y → Z) when fully parenthesized. That is, if we have f : X → Y → Z, and we give f an input of type X, it will give us as output a function of type Y → Z. If we then give that

function an input of type Y, it will return an output of type Z. That is, every function in Rocq takes only one input, but some functions return a function as output. This is precisely what enables partial application, as we saw above with plus3.

By contrast, functions of type $X \times Y \to Z$ – which when fully parenthesized is written $(X \times Y) \to Z$ – require their single input to be a pair. Both arguments must be given at once; there is no possibility of partial application.

It is possible to convert a function between these two types. Converting from $X \times Y \to Z$ to $X \to Y \to Z$ is called *currying*, in honor of the logician Haskell Curry. Converting from $X \to Y \to Z$ to $X \times Y \to Z$ is called *uncurrying*.

We can define currying as follows:

```
Definition prod_curry {X Y Z : Type}
  (f : X × Y → Z) (x : X) (y : Y) : Z := f (x, y).
```

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

```
Definition prod_uncurry {X Y Z : Type}
  (f : X → Y → Z) (p : X × 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 : ∀ (X Y Z : Type)
                          (f : X → Y → Z)
                          x y,
  prod_curry (prod_uncurry f) x y = f x y.
Proof.
  Admitted.
```

```
Theorem curry_uncurry : ∀ (X Y Z : Type)
                          (f : (X × Y) → Z) (p : X × Y),
  prod_uncurry (prod_curry f) p = f p.
Proof.
  Admitted.
```

□

Exercise: 2 stars, advanced, optional (nth_error_informal) Recall the definition of the nth_error function:

Fixpoint nth_error {X : Type} (l : list X) (n : nat) : option X := match l with | □ => None | a :: l' => if n =? O then Some a else nth_error l' (pred n) end.

Write a careful informal proof of the following theorem:

forall X l n, length l = n -> @nth_error X l n = None

Make sure to state the induction hypothesis *explicitly*.

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

□

5.4.1 Church Numerals (Advanced)

The following exercises explore an alternative way of defining natural numbers using the *Church numerals*, which are named after their inventor, the 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 := ∀ X : Type, (X → X) → X → 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 `fun X f x ⇒ f (f ... (f x) ...)`, with n occurrences of f. Let's informally notate that as `fun X f x ⇒ f^n x`, with the convention that `f^0 x` is just x. Note how the doit3times function we've defined previously is actually just the Church representation of 3.

```
Definition three : cnat := @doit3times.
```

So n X f x represents "do it n times", where n is a Church numerals and "it" means applying f starting with x.

Another way to think about the Church representation is that function f represents the successor operation on X, and value x represents the zero element of X. We could even rewrite with those names to make it clearer:

```
Definition zero' : cnat :=
  fun (X : Type) (succ : X → X) (zero : X) ⇒ zero.
```

```
Definition one' : cnat :=
  fun (X : Type) (succ : X → X) (zero : X) ⇒ succ zero.
Definition two' : cnat :=
  fun (X : Type) (succ : X → X) (zero : X) ⇒ succ (succ zero).
```

If we passed in S as *succ* and O as zero, we'd even get the Peano naturals as a result:

```
Example zero_church_peano : zero nat S O = 0.
Proof. reflexivity. Qed.

Example one_church_peano : one nat S O = 1.
Proof. reflexivity. Qed.

Example two_church_peano : two nat S O = 2.
Proof. reflexivity. Qed.
```

One very interesting implication of the Church numerals is that we don't strictly need the natural numbers to be built-in to a functional programming language, or even to be definable with an inductive data type. It's possible to represent them purely (if not efficiently) with functions.

Of course, it's not enough just to "represent" numerals; we need to be able to do arithmetic with the representation. Show that we can by completing the definitions of the following functions. Make sure that the corresponding unit tests pass by proving them with `reflexivity`.

Exercise: 2 stars, advanced (church_scc) Define a function that computes the successor of a Church numeral. Given a Church numeral n, its successor scc n should iterate its function argument once more than n. That is, given `fun X f x ⇒ f^n x` as input, scc should produce `fun X f x ⇒ f^(n+1) x` as output. In other words, do it n times, then do it once more.

```
Definition scc (n : cnat) : cnat
  . Admitted.

Example scc_1 : scc zero = one.
Proof. Admitted.

Example scc_2 : scc one = two.
Proof. Admitted.

Example scc_3 : scc two = three.
Proof. Admitted.
```

□

Exercise: 3 stars, advanced (church_plus) Define a function that computes the addition of two Church numerals. Given `fun X f x ⇒ f^n x` and `fun X f x ⇒ f^m x` as input, plus should produce `fun X f x ⇒ f^(n + m) x` as output. In other words, do it n times, then do it m more times.

Hint: the "zero" argument to a Church numeral need not be just x.

```
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: 3 stars, advanced (church_mult) Define a function that computes the multiplication of two Church numerals.

Hint: the "successor" argument to a Church numeral need not be just f.

Warning: Rocq will not let you pass cnat itself as the type X argument to a Church numeral; you will get a "Universe inconsistency" error. That is Rocq's way of preventing a paradox in which a type contains itself. So leave the type argument unchanged.

```
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: 3 stars, advanced (church_exp) Exponentiation:

Define a function that computes the exponentiation of two Church numerals.

Hint: the type argument to a Church numeral need not just be X. But again, you cannot pass cnat itself as the type argument. Finding the right type can be tricky.

```
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".
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 : ∀ (n m : nat),
  n = m →
  n = m.
Proof.
  intros n m eq.
```

Here, we could finish with "`rewrite → eq. reflexivity.`" as we have done several times before. Or we can finish in a single step by using `apply`:

```
  apply eq. 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.

`apply` also works with *conditional* hypotheses:

```
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 Rocq 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 : ∀ p,
  (∀ n, even n = true → even (S n) = false) →
  (∀ n, even n = false → odd n = true) →
  even p = true →
  odd (S p) = true.
Proof.
  Admitted.
```

□

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

```
Theorem silly3 : ∀ (n m : nat),
  n = m →
  m = n.
```

```
Proof.
  intros n m H.
```

Here we cannot use `apply` directly...

```
  Fail apply H.
```

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

```
  symmetry. apply H. Qed.
```

Exercise: 2 stars, standard (apply_exercise1) You can use `apply` with previously defined theorems, not just hypotheses in the context. Use `Search` to find a previously-defined theorem about rev from Lists. Use that theorem as part of your (relatively short) solution to this exercise. You do not need `induction`.

```
Theorem rev_exercise1 : ∀ (l l' : list nat),
  l = rev l' →
  l' = 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 : ∀ (a b c d e f : nat),
     [a;b] = [c;d] →
     [c;d] = [e;f] →
     [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
  rewrite → eq1. apply eq2. 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 : ∀ (X:Type) (x y z : X),
  x = y → y = z → x = z.
Proof.
  intros X x y z 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 Rocq `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], x with [a,b], and z with [e,f]. However, the matching process doesn't determine an instantiation for y: we have to supply one explicitly by adding "`with` (y:=[c,d])" to the invocation of `apply`.

```
  apply trans_eq with (y:=[c;d]).
  apply eq1. apply eq2. Qed.
```

Actually, the name y in the `with` clause is not required, since Rocq is often smart enough to figure out which variable we are instantiating. We could instead simply write `apply trans_eq with` [c;d].

Rocq also has a built-in 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 additional facts:

- The constructor S is *injective* (or *one-to-one*). That is, if S n = S m, it must also 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 every inductively defined type: all constructors are injective, and the values built from distinct constructors are never equal. For lists, the cons constructor is injective and the empty list 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.

We can *prove* the injectivity of S by using the pred function defined in *Basics.v*.

```
Theorem S_injective : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof.
  intros n m H1.
  assert (H2: n = pred (S n)). { reflexivity. }
  rewrite H2. rewrite H1. simpl. reflexivity.
Qed.
```

Rocq's `assert` tactic, used above, adds the given hypothesis to the context, but it first requires you to prove the hypothesis as a new goal.

This technique for injectivity 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 convenient alternative, Rocq 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' : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof.
  intros n m H.
```

By writing `injection H as Hmn` at this point, we are asking Rocq 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 (called 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 : ∀ (n m o : nat),
  [n;m] = [o;o] →
  n = m.
Proof.
  intros n m o H.
  injection H as H1 H2.
  rewrite H1. rewrite H2. reflexivity.
Qed.
```

Exercise: 3 stars, standard (injection_ex3) `Example injection_ex3 : ∀ (X : Type) (x y z : X) (l j : list X),`
`  x :: y :: l = z :: j →`
`  j = z :: l →`
`  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., false = true), and it solves the current goal immediately. Some examples:

```
Theorem discriminate_ex1 : ∀ (n m : nat),
  false = true →
  n = m.
Proof.
  intros n m contra. discriminate contra. Qed.

Theorem discriminate_ex2 : ∀ (n : nat),
  S n = O →
  2 + 2 = 5.
Proof.
  intros n contra. discriminate contra. Qed.
```

These examples are instances of a logical principle known as the *principle of explosion*, which asserts that a contradictory hypothesis entails anything (even manifestly false things!).

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 hold, *then* the nonsensical conclusion would too – 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 :
  ∀ (X : Type) (x y z : X) (l j : list X),
    x :: y :: l = [] →
    x = z.
Proof.
  Admitted.
```

□

For a more useful example, we can use `discriminate` to make a connection between the two different notions of equality (= and =?) that we have seen for natural numbers.

```
Theorem eqb_0_l : ∀ n,
  0 =? n = true → 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') = 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, Rocq confirms that the subgoal we are working on is impossible and removes it from further consideration.

```
    intros H. discriminate H.
Qed.
```

The injectivity of constructors allows us to reason that ∀ (n m : nat), S n = S m → n = m. The converse of this implication is an instance of a more general fact about both constructors and functions, which we will find useful below:

```
Theorem f_equal : ∀ (A B : Type) (f: A → B) (x y: A),
  x = y → f x = f y.
Proof. intros A B f x y eq. rewrite eq. reflexivity. Qed.

Theorem eq_implies_succ_equal : ∀ (n m : nat),
  n = m → S n = S m.
```

```
Proof. intros n m H. apply f_equal. apply H. Qed.
```

Indeed, there is also a tactic named 'f_equal' that can prove such theorems directly. Given a goal of the form $f\ a1\ \ldots\ an = g\ b1\ \ldots\ bn$, 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.

```
Theorem eq_implies_succ_equal' : ∀ (n m : nat),
  n = m → 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 : ∀ (n m : nat) (b : bool),
  ((S n) =? (S m)) = b →
  (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 → 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": given X → 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 → 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 silly4 : ∀ (n m p q : nat),
  (n = m → p = q) →
  m = n →
  q = p.
Proof.
  intros n m p q 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 seen in math or computer science classes tend to use forward reasoning. By contrast, idiomatic use of Rocq generally favors backward reasoning, though in some situations the forward style can be easier to think about.

6.6 Specializing Hypotheses

Another handy tactic for manipulating hypotheses is `specialize`. It is essentially just a combination of `assert` and `apply`, but it often provides a pleasingly smooth way to nail down overly general assumptions. It works like this:

If H is a quantified hypothesis in the current context – i.e., $H : \forall\ (x{:}T),\ P$ – then `specialize H with (x := e)` will change H so that it looks like P with x replaced by `e`.

For example:

```
Theorem specialize_example: ∀ n,
     (∀ m, m×n = 0)
  → n = 0.
Proof.
  intros n H.
  specialize H with (m := 1).
  rewrite mult_1_l in H.
  apply H. Qed.
```

Exercise: 3 stars, standard (nth_error_always_none) Use `specialize` to prove the the following lemma, following the model of specialize_example above. Do not use `induction`.
```
Lemma nth_error_always_none: ∀ (l : list nat),
  (∀ i, nth_error l i = None) →
  l = [].
Proof.
  Admitted.
```
□

Using `specialize` before `apply` gives us yet another way to control where `apply` does its work.
```
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.
  specialize trans_eq with (y:=[c;d]) as H.
  apply H.
```

```
  apply eq1.
  apply eq2. Qed.
```

Things to note:

- We can `specialize` facts in the global context, not just local hypotheses.
- The `as...` clause at the end tells `specialize` how to name the new hypothesis in this case.

6.7 Varying the Induction Hypothesis

Sometimes it is important to control the exact form of the induction hypothesis when carrying out inductive proofs in Rocq. In particular, we may 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.
then all will be well. But if we begin it with introducing *both* variables
intros n m. induction n.
we get stuck in the middle of the inductive case...

```
Theorem double_injective_FAILED : ∀ n m,
  double n = double m →
  n = m.
Proof.
  intros n m. induction n as [| n' IHn'].
  - simpl. intros eq. destruct m as [| m'] eqn:E.
    + reflexivity.
    + discriminate eq.
  - intros eq. destruct m as [| m'] eqn:E.
    + discriminate eq.
    + 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 where we invoke the induction hypothesis, we have already introduced *m* into the context – intuitively, we have told Rocq, "Let's consider some

particular n and m..." and we now have to prove that, if double n = double m for *these particular* n and m, then $n = m$.

The next tactic, `induction` n says to Rocq: We are going to show the goal by induction on n. That is, we are going to prove, for *all* n, that the proposition

- $P\ n$ = "if double n = double m, then $n = m$"

holds, by showing

- P O

  (i.e., "if double O = double m then O = m") and

- $P\ n \to P$ (S n)

  (i.e., "if double n = double m then $n = m$" implies "if double (S n) = double m then S $n = m$").

If we look closely at the second statement, it is saying something rather strange: that, for a *particular* m, if we know

- "if double n = double m then $n = m$"

then we can prove

- "if double (S n) = double m then S $n = m$".

To see why this is strange, let's think of a particular m – say, 5. The statement is then saying that, if we know

- Q = "if double n = 10 then $n = 5$"

then we can prove

- **R** = "if double (S n) = 10 then S $n = 5$".

But knowing Q doesn't give us any help at all with proving **R**! If we tried to prove **R** from Q, we would start with something like "Suppose double (S n) = 10..." but then we'd be stuck: knowing that double (S n) is 10 tells us nothing helpful about whether double n is 10 (indeed, it strongly suggests that 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 *particular* m.

A successful proof of double_injective keeps m universally quantified 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.
  -
```

Notice that both the goal and the induction hypothesis are different this time: the goal asks us to prove something more general (i.e., we must prove the statement for *every* m), but the induction hypothesis *IH'* is correspondingly more flexible, allowing us to choose any m we like when we apply it.

```
    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.
    +
      f_equal.
```

Since we are now 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.
```

The thing to take away from all this is that you need to be careful, when using induction, that you are not trying to prove something too specific: When proving a property quantified over variables n and m by induction on n, it is sometimes crucial to leave m "generic."

The following exercise, which further strengthens the link between =? and =, follows the same pattern.

Exercise: 2 stars, standard (eqb_true) `Theorem eqb_true : ∀ n m,`

```
  n =? m = true → n = m.
Proof.
    Admitted.
```

□

Exercise: 2 stars, advanced, optional (eqb_true_informal) Give a careful informal proof of eqb_true, stating the induction hypothesis explicitly and being as explicit as possible about quantifiers, everywhere.

```
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 : ∀ n m,
  n + n = m + m →
  n = m.
Proof.
  Admitted.
```

□

The strategy of doing fewer `intros` before an `induction` to obtain a more general IH doesn't always work; 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 : ∀ n m,
  double n = double m →
  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.
    + f_equal.
Abort.
```

The problem is that, to do induction on m, we must first introduce n. (If we simply say `induction m` without introducing anything first, Rocq 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 *regeneralize* 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.
    + 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 double n = double m, then $n = m$.

Proof: Let m be a **nat**. We prove by induction on m that, for any n, if double n = double m then $n = m$.

- First, suppose $m = 0$, and suppose n is a number such that double n = double m. We must show that $n = 0$.

  Since $m = 0$, by the definition of double we have double $n = 0$. There are two cases to consider for n. If $n = 0$ we are done, since $m = 0 = n$, as required. Otherwise, if n = S n' for some n', we derive a contradiction: by the definition of double, we can calculate double n = S (S (double n')), but this contradicts the assumption that double $n = 0$.

- Second, suppose m = S m' and that n is again a number such that double n = double m. We must show that n = S m', with the induction hypothesis that for every number s, if double s = double m' then $s = m'$.

  By the fact that m = S m' and the definition of double, we have double n = S (S (double m')). There are two cases to consider for n.

  If $n = 0$, then by definition double $n = 0$, a contradiction.

  Thus, we may assume that n = S n' for some n', and again by the definition of double we have S (S (double n')) = S (S (double m')), which implies by injectivity that double n' = double m'. Instantiating the induction hypothesis with n' thus allows us to conclude that $n' = m'$, and it follows immediately that S n' = S m'. Since S n' = n and S m' = m, this is just what we wanted to show. □

6.8 Rewriting with conditional statements

Suppose that we want to show that plus is the inverse of minus. Since we are working with natural numbers, we need an assumption to prevent minus from truncating its result. With this assumption, the induction hypothesis becomes $\forall\ m,\ n' <=?\ m = \mathsf{true} \to (m - n') + n' = m$. The beginning of the proof uses techniques we have already seen – in particular, notice how we induct on n before introducing m, so that the induction hypothesis becomes sufficiently general.

```
Lemma sub_add_leb : ∀ n m, n <=? m = true → (m - n) + n = m.
Proof.
  intros n.
  induction n as [| n' IHn'].
  -
    intros m H. rewrite add_0_r. destruct m as [| m'].
    +
      reflexivity.
    +
      reflexivity.
  -
    intros m H. destruct m as [| m'].
    +
      discriminate.
    +
      simpl in H. simpl. rewrite ← plus_n_Sm.
```

At this point, we need to show $\mathsf{S}\ ((m' - n') + n') = \mathsf{S}\ m'$ from the assumption $(n' \leq m') = \mathsf{true}$. We could use the `assert` tactic to prove $(m' - n') + n' = m'$ from the induction hypothesis. However, we can also just use `rewrite` directly: if we rewrite with a conditional statement of the form $P \to \mathsf{a} = \mathsf{b}$, then Rocq tries to rewrite with $\mathsf{a} = \mathsf{b}$, and then asks us to prove P in a new subgoal. If the statement has more than one assumption, then we get one subgoal for each assumption.

```
      rewrite IHn'.
      × reflexivity.
      × apply H.
Qed.
```

Exercise: 3 stars, standard, especially useful (gen_dep_practice) Prove this by induction on l.

```
Theorem nth_error_after_last: ∀ (n : nat) (X : Type) (l : list X),
  length l = n →
  nth_error l n = None.
Proof.
```

```
Admitted.
```
□

6.9 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 stands for.

For example, if we define...

```
Definition square n := n × n.
```

...and try to prove a simple fact about square...

```
Lemma square_mult : ∀ n m, square (n × m) = square n × 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 × m × n = n × n × m).
    { rewrite mul_comm. apply mult_assoc. }
  rewrite H. rewrite mult_assoc. reflexivity.
Qed.
```

At this point, a bit deeper 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: 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 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.
```

But 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 Rocq 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 so hard.

```
  destruct m eqn:E.
```

```
  - reflexivity.
  - reflexivity.
Qed.
```

6.10 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 : ∀ X Y (l : list (X × Y)) l1 l2,
  split l = (l1 , l2) →
  combine l1 l2 = l.
Proof.
  Admitted.
```

□

The *eqn*: part of the `destruct` tactic is optional; although we've chosen to include it most of the time, for the sake of documentation, it can often be omitted without harm.

One example where it *cannot* be omitted is when we are `destruct`ing compound expressions; here, the information recorded by the *eqn*: can actually be critical, and, 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 : nat) : bool :=
  if n =? 3 then true
  else if n =? 5 then true
  else false.
```

Now suppose that we want to convince Rocq 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 : ∀ (n : nat),
  sillyfun1 n = true →
  odd 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 we are assuming $n =? 3 = \texttt{true}$ 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 : ∀ (n : nat),
  sillyfun1 n = true →
  odd n = true.
Proof.
```

```
  intros n eq. unfold sillyfun1 in eq.
  destruct (n =? 3) eqn:Heqe3.
```
Now we have the same state as at the point where we got stuck above, except that the context contains an extra equality assumption, which is exactly what we need to make progress.
```
    - apply eqb_true in Heqe3.
      rewrite → Heqe3. reflexivity.
    -
```
When we come to the second equality test in the body of the function we are reasoning about, we can use *eqn*: again in the same way, allowing us to finish the proof.
```
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
:
  ∀ (f : bool → bool) (b : bool),
  f (f (f b)) = f b.
Proof.
  Admitted.
```
□

6.11 Review

We've now talked about many of Rocq'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 Rocq 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
- `specialize (H ...)`: refine a hypothesis by fixing some of its variables

- `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`... `as`...: 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
- `replace` x `with` y: replaces x with y everywhere for the current goal, creating a subgoal that $x = y$.
- `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 $x = y$

6.12 Additional Exercises

Exercise: 3 stars, standard (eqb_sym) `Theorem eqb_sym : ∀ (n m : 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 **nat**s n m, $(n =? m) = (m =? n)$.

Proof:

Exercise: 3 stars, standard, optional (eqb_trans)

```
Theorem eqb_trans : ∀ n m p,
  n =? m = true →
  m =? p = true →
  n =? p = true.
Proof.
  Admitted.
```

□

Exercise: 3 stars, advanced (split_combine) We proved, in an exercise above, that combine is the inverse of `split`. 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.

Hint: Take a look at the definition of combine in Poly. Your property will need to account for the behavior of combine in its base cases, which possibly drop some list elements.

```
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)

```
Theorem filter_exercise : ∀ (X : Type)
                          (test : X → bool)
                          (x : X) (l lf : list X),
  filter test l = x :: lf →
  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 odd 1;3;5;7;9 = true forallb negb false;false = true forallb even 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 odd 1;0;0;0;0;3 = true existsb even □ = 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 odd [1;3;5;7;9] = true.
Proof. Admitted.

Example test_forallb_2 : forallb negb [false;false] = true.
Proof. Admitted.

Example test_forallb_3 : forallb even [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 odd [1;0;0;0;0;3] = true.
Proof. Admitted.

Example test_existsb_4 : existsb even [] = 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),
  existsb test l = existsb' test l.
Proof. Admitted.
```

□

Chapter 7

Library LF.Logic

7.1 Logic: Logic in Rocq

```
Set Warnings "-notation-overridden".
Require Nat.
From LF Require Export Tactics.
```

We have now seen many examples of factual claims (i.e., *propositions*) and ways of presenting evidence of their truth (*proofs*). In particular, we have worked extensively with equality propositions (*e1* = *e2*), implications (*P* → *Q*), and quantified propositions (∀ *x*, *P*). In this chapter, we will see how Rocq can be used to carry out other familiar forms of logical reasoning.

Before diving into details, we should talk a bit about the status of mathematical statements in Rocq. Rocq is a *typed* language, which means that every sensible expression has an associated type. Logical claims are no exception: any statement we might try to prove in Rocq has a type, namely `Prop`, the type of *propositions*. We can see this with the `Check` command:

```
Check (∀ n m : nat, n + m = m + n) : Prop.
```

Note that *all* syntactically well-formed propositions have type `Prop` in Rocq, regardless of whether they are true or not.

Simply *being* a proposition is one thing; being *provable* is a different thing!

```
Check 2 = 2 : Prop.
```

```
Check 3 = 2 : Prop.
```

```
Check ∀ n : nat, n = 2 : Prop.
```

Indeed, propositions don't just have types – they are *first-class* entities that can be manipulated in all the same ways as any of the other things in Rocq's world.

So far, we've seen one primary place where 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 other ways. For example, we can give a name to a proposition using a `Definition`, just as we give 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 Rocq, 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) : Prop :=
  ∀ x y : A, f x = f y → x = y.

Lemma succ_inj : injective S.
Proof.
  intros x y H. injection H as H1. apply H1.
Qed.
```

The familiar equality operator = is a (binary) function that returns a `Prop`.

The expression $n = m$ is syntactic sugar for **eq** n m (defined in Rocq's standard library using the `Notation` mechanism).

Because **eq** can be used with elements of any type, it is also polymorphic:

```
Check @eq : ∀ A : Type, A → A → 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 ∧ B; it represents the claim that both A and B are true.

```
Example and_example : 3 + 4 = 7 ∧ 2 × 2 = 4.
```

To prove a conjunction, start with the `split` tactic. This 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 and B are each true individually, we can conclude that A ∧ B is also true. The Rocq library provides a function `conj` that does this.

```
Check @conj : ∀ A B : Prop, A → B → A ∧ B.
```

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 `conj` to achieve the same effect as `split`.

```
Example and_example' : 3 + 4 = 7 ∧ 2 × 2 = 4.
Proof.
  apply conj.
  - reflexivity.
  - reflexivity.
Qed.
```

Exercise: 2 stars, standard (plus_is_O)
```
Example plus_is_O :
  ∀ n m : nat, n + m = 0 → n = 0 ∧ 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 can use our good old `destruct` tactic.

When the current proof context contains a hypothesis *H* of the form A ∧ B, writing `destruct H as [HA HB]` will remove *H* from the context and replace it with two new hypotheses: *HA*, stating that A is true, and *HB*, stating that B is true.

```
Lemma and_example2 :
```

```
  ∀ n m : nat, n = 0 ∧ m = 0 → 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 at the point where we introduce it, instead of introducing and then destructing it:

```
Lemma and_example2' :
  ∀ n m : nat, n = 0 ∧ m = 0 → 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 also have stated the theorem with two separate premises:

```
Lemma and_example2'' :
  ∀ n m : nat, n = 0 → m = 0 → 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 :
  ∀ n m : nat, n + m = 0 → n × m = 0.
Proof.
  intros n m H.
  apply plus_is_O in H.
  destruct H as [Hn Hm].
  rewrite Hn. reflexivity.
Qed.
```

Another common situation is that we know A ∧ B but in some context we need just A or just B. In such cases we can do a `destruct` (possibly implicitly, as part of an `intros`) and use an underscore pattern `_` to indicate that the unneeded conjunct should just be thrown away.

```
Lemma proj1 : ∀ P Q : Prop,
  P ∧ Q → P.
Proof.
  intros P Q HPQ.
  destruct HPQ as [HP _].
  apply HP. Qed.
```

Exercise: 1 star, standard, optional (proj2) `Lemma proj2 : ∀ P Q : Prop, P ∧ Q → Q. Proof. Admitted.` □

Finally, we sometimes need to rearrange the order of conjunctions and/or the grouping of multi-way conjunctions. We can see this at work in the proofs of the following commutativity and associativity theorems

```
Theorem and_commut : ∀ P Q : Prop,
  P ∧ Q → Q ∧ P.
Proof.
  intros P Q [HP HQ].
  split.
    - apply HQ.
    - apply HP. Qed.
```

Exercise: 1 star, standard (and_assoc) In the following proof of associativity, notice how the *nested* `intros` pattern breaks the hypothesis $H : P \wedge (Q \wedge \mathsf{R})$ down into $HP : P$, $HQ : Q$, and $HR : \mathsf{R}$. Finish the proof.

```
Theorem and_assoc : ∀ P Q R : Prop,
  P ∧ (Q ∧ R) → (P ∧ Q) ∧ R.
Proof.
  intros P Q R [HP [HQ HR]].
  Admitted.
```
□

The infix notation ∧ is actually just syntactic sugar for `and A B`. That is, `and` is a Rocq operator that takes two propositions as arguments and yields a proposition.

```
Check and : Prop → Prop → Prop.
```

7.2.2 Disjunction

Another important connective is the *disjunction*, or *logical or*, of two propositions: `A ∨ B` is true when either `A` or `B` is. This infix notation stands for `or A B`, where `or : Prop → Prop → 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 factor_is_O:
  ∀ n m : nat, n = 0 ∨ m = 0 → n × m = 0.
Proof.
  intros n m [Hn | Hm].
  -
    rewrite Hn. reflexivity.
  -
    rewrite Hm. rewrite ← mult_n_O.
    reflexivity.
Qed.
```

We can see in this example that, when we perform case analysis on a disjunction `A ∨ B`, we must separately discharge two proof obligations, each showing that the conclusion holds under a different assumption – `A` in the first subgoal and `B` in the second.

The case analysis pattern [*Hn* | *Hm*] allows us to name the hypotheses that are generated for the subgoals.

Conversely, to show that a disjunction holds, it suffices to show that one of its sides holds. This can be done via the tactics `left` and `right`. As their names imply, the first one requires proving the left side of the disjunction, while the second requires proving the right side. Here is a trivial use...

```
Lemma or_intro_l : ∀ A B : Prop, A → A ∨ 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 :
  ∀ n : nat, n = 0 ∨ n = S (pred n).
Proof.
  intros [|n'].
  - left. reflexivity.
  - right. reflexivity.
Qed.
```

Exercise: 2 stars, standard (mult_is_O) `Lemma mult_is_O :`

```
  ∀ n m, n × m = 0 → n = 0 ∨ m = 0.
Proof.
    Admitted.
```

□

Exercise: 1 star, standard (or_commut) `Theorem or_commut : ∀ P Q : Prop, P ∨ Q → Q ∨ P.`

```
Proof.
  Admitted.
```

□

7.2.3 Falsehood and Negation

Up to this point, we have mostly been concerned with proving "positive" statements – addition is commutative, appending lists is associative, etc. We are sometimes also interested in negative results, demonstrating that some proposition is *not* true. Such statements are expressed with the logical negation operator ¬.

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 ¬ P ("not P") as ∀ Q, $P \to Q$.

Rocq actually makes an equivalent but slightly different choice, defining ¬ P as $P \to$ **False**, where **False** is a specific un-provable proposition defined in the standard library.

```
Module NotPlayground.

Definition not (P:Prop) := P → False.

Check not : Prop → Prop.

Notation "~ x" := (not x) : type_scope.

End NotPlayground.
```

Since **False** is a contradictory proposition, the principle of explosion also applies to it. If we can get **False** into the context, we can use `destruct` on it to complete any goal:

```
Theorem ex_falso_quodlibet : ∀ (P:Prop),
  False → 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 Rocq's definition of negation implies the intuitive one mentioned above.

Hint: While getting accustomed to Rocq's definition of not, you might find it helpful to `unfold not` near the beginning of proofs.

```
Theorem not_implies_our_not : ∀ (P:Prop),
  ¬ P → (∀ (Q:Prop), P → Q).
```

```
Proof.
  Admitted.
```
□

Inequality is a very common form of negated statement, so there is a special notation for it:

```
Notation "x <> y" := (~(x = y)) : type_scope.
```

For example:

```
Theorem zero_not_one : 0 ≠ 1.
Proof.
```

The proposition $0 \neq 1$ is exactly the same as $\sim(0 = 1)$ – that is, not $(0 = 1)$ – which unfolds to $(0 = 1) \to$ **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 Rocq. Even though *you* may see perfectly well why a claim involving negation holds, it can be a little tricky at first to see how to make Rocq understand it!

Here are proofs of a few familiar facts to help get you warmed up.

```
Theorem not_False :
  ¬ False.
Proof.
  unfold not. intros H. destruct H. Qed.

Theorem contradiction_implies_anything : ∀ P Q : Prop,
  (P ∧ ¬P) → Q.
Proof.
  intros P Q [HP HNP]. unfold not in HNP.
  apply HNP in HP. destruct HP. Qed.

Theorem double_neg : ∀ P : Prop,
  P → ~~P.
Proof.
  intros P H. unfold not. intros G. apply G. apply H. Qed.
```

Exercise: 2 stars, advanced, optional (double_neg_informal) Write an *informal* proof of double_neg:

Theorem: P implies $\sim\sim P$, for any proposition P.

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

Exercise: 1 star, standard, especially useful (contrapositive) `Theorem contrapositive : ∀ (P Q : Prop), (P → Q) → (¬Q → ¬P).`

```
Proof.
  Admitted.
```
□

Exercise: 1 star, standard (not_both_true_and_false) `Theorem not_both_true_and_false : ∀ P : Prop, ¬ (P ∧ ¬P).`

```
Proof.
  Admitted.
```
□

Exercise: 1 star, advanced (not_PNP_informal) Write an informal proof (in English) of the proposition `∀ P : Prop, ~(P ∧ ¬P)`.

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

Exercise: 2 stars, standard (de_morgan_not_or) *De Morgan's Laws*, named for Augustus De Morgan, describe how negation interacts with conjunction and disjunction. The following law says that "the negation of a disjunction is the conjunction of the negations." There is a dual law de_morgan_not_and_not to which we will return at the end of this chapter. `Theorem de_morgan_not_or : ∀ (P Q : Prop), ¬ (P ∨ Q) → ¬P ∧ ¬Q.`

```
Proof.
  Admitted.
```
□

Exercise: 1 star, standard, optional (not_S_inverse_pred) Since we are working with natural numbers, we can disprove that S and pred are inverses of each other: `Lemma not_S_pred_n : ~(∀ n : nat, S (pred n) = n).`

```
Proof.
  Admitted.
```
□

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 : ∀ b : bool,
  b ≠ true → b = 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, Rocq provides a built-in tactic, *exfalso*, for applying it.

```
Theorem not_true_is_false' : ∀ b : bool,
  b ≠ true → b = false.
Proof.
  intros [] H.   -
    unfold not in H.
    exfalso.      apply H. reflexivity.
  - reflexivity.
Qed.
```

7.2.4 Truth

Besides **False**, Rocq's standard library also defines **True**, a proposition that is trivially true. To prove it, we use the constant I : **True**, which is also defined in the standard library:

```
Lemma True_is_true : True.
Proof. apply I. Qed.
```

Unlike **False**, which is used extensively, **True** is used relatively rarely: it is trivial (and therefore uninteresting) to prove as a goal, and it provides no useful information when it appears as a hypothesis.

However, **True** can be quite useful when defining complex Props using conditionals or as a parameter to higher-order Props. We'll come back to this later.

For now, let's take a look at how we can use **True** and **False** to achieve an effect similar to that of the discriminate tactic, without literally using discriminate.

Pattern-matching lets us do different things for different constructors. If the result of applying two different constructors were hypothetically equal, then we could use match to convert an unprovable statement (like **False**) to one that is provable (like **True**).

```
Definition disc_fn (n: nat) : Prop :=
  match n with
  | O ⇒ True
```

```
  | S _ ⇒ False
  end.

Theorem disc_example : ∀ n, ¬ (O = S n).
Proof.
  intros n contra.
  assert (H : disc_fn O). { simpl. apply I. }
  rewrite contra in H. simpl in H. apply H.
Qed.
```

To generalize this to other constructors, we simply have to provide an appropriate variant of disc_fn. To generalize it to other conclusions, we can use *exfalso* to replace them with False.

The built-in `discriminate` tactic takes care of all this for us.

Exercise: 2 stars, advanced, optional (nil_is_not_cons) Use the same technique as above to show that nil ≠ x :: xs. Do not use the `discriminate` tactic.

```
Theorem nil_is_not_cons : ∀ X (x : X) (xs : list X), ¬ (nil = x :: xs).
Proof.
  Admitted.
```

□

7.2.5 Logical Equivalence

```
Print "<->".
```

The handy "if and only if" connective, which asserts that two propositions have the same truth value, is simply the conjunction of two implications.

Print "<->".

```
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.
```

We can also use apply with an ↔ in either direction, without explicitly thinking about the fact that it is really an **and** underneath.

```
Lemma apply_iff_example1:
  ∀ P Q R : Prop, (P ↔ Q) → (Q → R) → (P → R).
Proof.
  intros P Q R Hiff H HP. apply H. apply Hiff. apply HP.
Qed.

Lemma apply_iff_example2:
  ∀ P Q R : Prop, (P ↔ Q) → (P → R) → (Q → R).
Proof.
  intros P Q R Hiff H HQ. apply H. apply Hiff. apply HQ.
Qed.
```

Exercise: 1 star, standard, optional (iff_properties) Using the above proof that ↔ is symmetric (iff_sym) as a guide, prove that it is also reflexive and transitive.

```
Theorem iff_refl : ∀ P : Prop,
  P ↔ P.
Proof.
  Admitted.

Theorem iff_trans : ∀ P Q R : Prop,
  (P ↔ Q) → (Q ↔ R) → (P ↔ R).
Proof.
  Admitted.
```

□

Exercise: 3 stars, standard (or_distributes_over_and) Theorem or_distributes_over_and : ∀ P Q R : Prop,

```
  P ∨ (Q ∧ R) ↔ (P ∨ Q) ∧ (P ∨ R).
Proof.
  Admitted.
```

□

7.2.6 Setoids and Logical Equivalence

Some of Rocq's tactics treat iff statements specially, avoiding 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 Rocq library that supports it: From *Stdlib* 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 by the other.

We've seen this already with the equality relation = in Rocq: 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. (For these proofs we are not using setoids yet.)

```
Lemma mul_eq_0 : ∀ n m, n × m = 0 ↔ n = 0 ∨ m = 0.
Proof.
  split.
  - apply mult_is_O.
  - apply factor_is_O.
Qed.

Theorem or_assoc :
  ∀ P Q R : Prop, P ∨ (Q ∨ R) ↔ (P ∨ Q) ∨ 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 prove a ternary version of the *mult_eq_0* fact above *without* splitting the top-level iff:

```
Lemma mul_eq_0_ternary :
  ∀ n m p, n × m × p = 0 ↔ n = 0 ∨ m = 0 ∨ p = 0.
Proof.
  intros n m p.
  rewrite mul_eq_0. rewrite mul_eq_0. rewrite or_assoc.
  reflexivity.
Qed.
```

7.2.7 Existential Quantification

Another fundamental 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 Rocq 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 for x, known as the *witness* of the existential. This is done in two steps: First, we explicitly tell Rocq 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 := ∃ n : nat, x = double n.
Check Even : nat → Prop.

Lemma four_is_Even : Even 4.
Proof.
  unfold Even. ∃ 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 : ∀ n,
  (∃ m, n = 4 + m) →
  (∃ o, n = 2 + o).
Proof.
  intros n [m Hm].    ∃ (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 : ∀ (X:Type) (P : X → Prop),
  (∀ x, P x) → ¬ (∃ x, ¬ P x).
Proof.
    Admitted.
```

□

Exercise: 2 stars, standard (dist_exists_or) Prove that existential quantification distributes over disjunction.

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

□

Exercise: 3 stars, standard, optional (leb_plus_exists) `Theorem leb_plus_exists : ∀ n m, n <=? m = true → ∃ x, m = n+x.`

```
Proof.
    Admitted.

Theorem plus_exists_leb : ∀ n m, (∃ x, m = n+x) → n <=? m = true.
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 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 In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] ⇒ False
  | x' :: l' ⇒ x' = x ∨ In x l'
  end.
```

When In is applied to a concrete list, it expands into a concrete sequence of nested disjunctions.

```
Example In_example_1 : In 4 [1; 2; 3; 4; 5].
Proof.
  simpl. right. right. right. left. reflexivity.
Qed.

Example In_example_2 :
  ∀ n, In 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 reason about more generic statements involving In.

```
Theorem In_map :
  ∀ (A B : Type) (f : A → B) (l : list A) (x : A),
          In x l →
          In (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.
```

(Note here how In starts out applied to a variable and only gets expanded when we do case analysis on this variable.)

This way of defining propositions recursively is very convenient in some cases, less so in others. In particular, it is subject to Rocq's usual restrictions regarding definitions 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 strengths and limitations.

Exercise: 2 stars, standard (In_map_iff)

```
Theorem In_map_iff :
  ∀ (A B : Type) (f : A → B) (l : list A) (y : B),
          In y (map f l) ↔
          ∃ x, f x = y ∧ In x l.
Proof.
  intros A B f l y. split.
  - induction l as [|x l' IHl'].
    Admitted.
```

□

Exercise: 2 stars, standard (In_app_iff)

```
Theorem In_app_iff : ∀ A l l' (a:A),
  In a (l++l') ↔ In a l ∨ In a l'.
Proof.
  intros A l. induction l as [|a' l' IH].
    Admitted.
```

□

Exercise: 3 stars, standard, especially useful (All) We noted above that functions returning propositions can be seen as *properties* of their arguments. For instance, if P has type **nat** → Prop, then $P\ n$ says that property P holds of n.

Drawing inspiration from In, 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_In lemma below. (Of course, your definition should *not* just restate the left-hand side of All_In.)

```
Fixpoint All {T : Type} (P : T → Prop) (l : list T) : Prop
  . Admitted.

Theorem All_In :
  ∀ T (P : T → Prop) (l : list T),
    (∀ x, In x l → P x) ↔
    All P l.
Proof.
    Admitted.
    □
```

Exercise: 2 stars, standard, optional (combine_odd_even) Complete the definition of combine_odd_even below. It takes as arguments two properties of numbers, *Podd* and *Peven*, and it should return a property P such that $P\ n$ is equivalent to *Podd* n when n is odd and equivalent to *Peven* n otherwise.

```
Definition combine_odd_even (Podd Peven : nat → Prop) : nat → Prop
  . Admitted.
```

To test your definition, prove the following facts:

```
Theorem combine_odd_even_intro :
  ∀ (Podd Peven : nat → Prop) (n : nat),
    (odd n = true → Podd n) →
    (odd n = false → Peven n) →
    combine_odd_even Podd Peven n.
Proof.
    Admitted.

Theorem combine_odd_even_elim_odd :
  ∀ (Podd Peven : nat → Prop) (n : nat),
    combine_odd_even Podd Peven n →
    odd n = true →
    Podd n.
Proof.
    Admitted.

Theorem combine_odd_even_elim_even :
  ∀ (Podd Peven : nat → Prop) (n : nat),
    combine_odd_even Podd Peven n →
```

```
    odd n = false →
    Peven n.
Proof.
  Admitted.
  □
```

7.4 Applying Theorems to Arguments

One feature that distinguishes Rocq 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 order to use Rocq. This section gives just a taste, leaving a deeper exploration for the optional chapters ProofObjects and IndPrinciples.

We have seen that we can use `Check` to ask Rocq to check whether an expression has a given type:

```
Check plus : nat → nat → nat.
Check @rev : ∀ X, list X → list X.
```

We can also use it to check what theorem a particular identifier refers to:

```
Check add_comm : ∀ n m : nat, n + m = m + n.
Check plus_id_example : ∀ n m : nat, n = m → n + n = m + m.
```

Rocq checks the *statements* of the add_comm and plus_id_example theorems in the same way that it checks the *type* of any term (e.g., plus). If we leave off the colon and type, Rocq will print these types for us.

Why?

The reason is that the identifier add_comm actually refers to a *proof object* – a logical derivation establishing the truth of the statement $\forall\ n\ m$: **nat**, $n + m = m + n$. The type of this object is the proposition that it is a proof of.

The type of an ordinary function tells us what we can do with it.

- If we have a term of type **nat** → **nat** → **nat**, we can give it two **nat**s as arguments and get a **nat** back.

Similarly, the statement of a theorem tells us what we can use that theorem for.

- If we have a term of type $\forall\ n\ m$, $n = m \to n + n = m + m$ and we provide it two numbers n and m and a third "argument" of type $n = m$, we get back a proof object of type $n + n = m + m$.

Operationally, this analogy goes even further: by applying a theorem as if it were a function, i.e., applying it to values and 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 add_comm3 :
  ∀ 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 add_comm twice to make the two sides match. The problem is that the second `rewrite` will undo the effect of the first.

```
Proof.
  intros x y z.
  rewrite add_comm.
  rewrite add_comm.
Abort.
```

We encountered similar issues back in `Induction`, and we saw one way to work around them by using `assert` to derive a specialized version of add_comm that can be used to rewrite exactly where we want.

```
Lemma add_comm3_take2 :
  ∀ x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite add_comm.
  assert (H : y + z = z + y).
    { rewrite add_comm. reflexivity. }
  rewrite H.
  reflexivity.
Qed.
```

A more elegant alternative is to apply add_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 add_comm3_take3 :
  ∀ x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite add_comm.
  rewrite (add_comm y z).
  reflexivity.
Qed.
```

If we really wanted, we could in fact do it for both rewrites.

```
Lemma add_comm3_take4 :
  ∀ x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite (add_comm x (y + z)).
```

```
  rewrite (add_comm y z).
  reflexivity.
Qed.
```

Here's another example of using a theorem about lists like a function. Suppose we have proved the following simple fact about lists...

```
Theorem in_not_nil :
  ∀ A (x : A) (l : list A), In x l → l ≠ [].
Proof.
  intros A x l H. unfold not. intro Hl.
  rewrite Hl in H.
  simpl in H.
  apply H.
Qed.
```

(I.e., if a list l contains some element x, then l must be nonempty.)

Note that one quantified variable (x) does not appear in the conclusion ($l \neq []$).

Intuitively, we should be able to use this theorem to prove the special case where x is 42. However, simply invoking the tactic `apply in_not_nil` will fail because it cannot infer the value of x.

```
Lemma in_not_nil_42 :
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  Fail apply in_not_nil.
Abort.
```

There are several ways to work around this:

We can use `apply ... with ...`: `Lemma in_not_nil_42_take2 :`

```
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  apply in_not_nil with (x := 42).
  apply H.
Qed.
```

Or we can use `apply ... in ...`: `Lemma in_not_nil_42_take3 :`

```
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  apply in_not_nil in H.
  apply H.
Qed.
```

Or – this is the new one – we can explicitly apply the lemma to the value 42 for x: `Lemma in_not_nil_42_take4 :`

```
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  apply (in_not_nil nat 42).
  apply H.
Qed.
```

We can also explicitly apply the lemma to a hypothesis, causing the values of the other parameters to be inferred:

```
Lemma in_not_nil_42_take5 :
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  apply (in_not_nil _ _ _ H).
Qed.
```

You can "use a theorem as a function" in this way with almost any tactic that can take a theorem's name as an argument.

Note, also, that theorem application uses the same inference mechanisms as function application; thus, it is possible, for example, to supply wildcards as arguments to be inferred, or to declare some hypotheses to a theorem as implicit by default. These features are illustrated in the proof below. (The details of how this proof works are not critical – the goal here is just to illustrate applying theorems to arguments.)

```
Example lemma_application_ex :
  ∀ {n : nat} {ns : list nat},
    In n (map (fun m ⇒ m × 0) ns) →
    n = 0.
Proof.
  intros n ns H.
  destruct (proj1 _ _ (In_map_iff _ _ _ _ _) H)
           as [m [Hm _]].
  rewrite mul_0_r in Hm. rewrite ← Hm. reflexivity.
Qed.
```

We will see many more examples in later chapters.

7.5 Working with Decidable Properties

We've seen two different ways of expressing logical claims in Rocq: with *booleans* (of type **bool**), and with *propositions* (of type `Prop`).

Here are the key differences between **bool** and `Prop`:

bool Prop ==== ==== decidable? yes no useable with match? yes no works with rewrite tactic? no yes

The crucial difference between the two worlds is *decidability*. Every (closed) Rocq expression of type **bool** can be simplified in a finite number of steps to either true or false –

i.e., there is a terminating mechanical procedure for deciding whether or not it is true.

This means that, for example, the type **nat** → **bool** is inhabited only by functions that, given a **nat**, always yield either true or false in finite time; and this, in turn, means (by a standard computability argument) that there is *no* function in **nat** → **bool** that checks whether a given number is the code of a terminating Turing machine.

By contrast, the type Prop includes both decidable and undecidable mathematical propositions; in particular, the type **nat** → Prop does contain functions representing properties like "the nth Turing machine halts."

The second row in the table follows directly from this essential difference. To evaluate a pattern match (or conditional) on a boolean, we need to know whether the scrutinee evaluates to true or false; this only works for **bool**, not Prop.

The third row highlights an important practical difference: equality functions like *eqb_nat* that return a boolean cannot be used directly to justify rewriting with the rewrite tactic; propositional equality is required for this.

Since Prop includes *both* decidable and undecidable properties, we have two options when we want to formalize a property that happens to be decidable: we can express it either as a boolean computation or as a function into Prop.

```
Example even_42_bool : even 42 = true.
Proof. reflexivity. Qed.
```

... or that there exists some k such that n = double k. `Example even_42_prop : Even 42.`

```
Proof. unfold Even. ∃ 21. reflexivity. Qed.
```

Of course, it would be deeply strange if these two characterizations of evenness did not describe the same set of natural numbers!

Fortunately, they do!

To prove this, we first need two helper lemmas.

```
Lemma even_double : ∀ k, even (double k) = true.
Proof.
  intros k. induction k as [|k' IHk'].
  - reflexivity.
  - simpl. apply IHk'.
Qed.
```

Exercise: 3 stars, standard (even_double_conv) `Lemma even_double_conv : ∀ n, ∃ k,`

```
  n = if even n then double k else S (double k).
Proof.
```

Admitted.

□

Now the main theorem:

```
Theorem even_bool_prop : ∀ n,
```

```
  even n = true ↔ Even n.
Proof.
  intros n. split.
  - intros H. destruct (even_double_conv n) as [k Hk].
    rewrite Hk. rewrite H. ∃ k. reflexivity.
  - intros [k Hk]. rewrite Hk. apply even_double.
Qed.
```

In view of this theorem, we can say that the boolean computation even 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.
```

So what should we do in situations where some claim could be formalized as either a proposition or a boolean computation? Which should we choose?

In general, *both* can be useful.

For example, booleans are more useful for defining functions. There is no effective way to *test* whether or not a Prop is true, so we cannot use Props in conditional expressions. The following definition is rejected:

```
Fail
Definition is_even_prime n :=
  if n = 2 then true
  else false.
```

Rocq complains that $n = 2$ has type Prop, while it expects an element of **bool** (or some other inductive type with two constructors). This has to do with the *computational* nature of Rocq'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 Rocq developments.) As a consequence, Prop in Rocq 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.

Rather, we have to state this definition using a boolean equality test.

```
Definition is_even_prime n :=
```

```
  if n =? 2 then true
  else false.
```

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 Rocq 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 Rocq 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 simpler, because we don't have to invent the witness 500: Rocq's computation mechanism does it for us!

```
Example even_1000' : even 1000 = true.
Proof. reflexivity. Qed.
```

Now, the useful observation 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 line count in this case, larger proofs can often be made considerably simpler by the use of reflection. As an extreme example, a famous Rocq 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 advantage of booleans is that the *negation* of a claim about booleans is straightforward to state and (when true) prove: simply flip the expected boolean result.

```
Example not_even_1001 : even 1001 = false.
Proof.
  reflexivity.
Qed.
```

In contrast, propositional negation can be difficult to work with directly.

For example, suppose we state the non-evenness of 1001 propositionally:

```
Example not_even_1001' : ~(Even 1001).
```

Proving this directly – by assuming that there is some n such that $1001 = \mathit{double}\ n$ and then somehow reasoning to a contradiction – would be rather complicated.

But if we convert it to a claim about the boolean even function, we can let Rocq do the work for us.

```
Proof.
  rewrite ← even_bool_prop.
  unfold not.
  simpl.
  intro H.
  discriminate H.
Qed.
```

Conversely, there are situations where it can be easier to work with propositions rather than booleans.

In particular, knowing that $(n =?\ m) = \mathsf{true}$ is generally of little direct help in the middle of a proof involving n and m. But if we convert the statement to the equivalent form $n = m$, then we can easily `rewrite` with it.

```
Lemma plus_eqb_example : ∀ n m p : nat,
  n =? m = true → n + p =? m + p = 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 for the moment, but it serves as a good example showing the different strengths of booleans and general propositions.

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 : ∀ b1 b2:bool,
  b1 && b2 = true ↔ b1 = true ∧ b2 = true.
Proof.
  Admitted.

Theorem orb_true_iff : ∀ b1 b2,
  b1 || b2 = true ↔ b1 = true ∨ b2 = 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 : ∀ x y : nat,
  x =? y = false ↔ x ≠ 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 : Type} (eqb : A → A → bool)
                  (l1 l2 : list A) : bool
  . Admitted.

Theorem eqb_list_true_iff :
  ∀ A (eqb : A → A → bool),
    (∀ a1 a2, eqb a1 a2 = true ↔ a1 = a2) →
    ∀ l1 l2, eqb_list eqb l1 l2 = true ↔ l1 = l2.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard, especially useful (All_forallb) Prove the theorem below, which relates forallb, from the exercise *forall_exists_challenge* in chapter Tactics, to the `All` property defined above.

Copy the definition of forallb from your Tactics here so that this file can be graded on its own. `Fixpoint forallb {X : Type} (test : X → bool) (l : list X) : bool`
. *Admitted*.

```
Theorem forallb_true_iff : ∀ X test (l : list X),
  forallb test l = true ↔ All (fun x ⇒ test x = true) l.
Proof.
  Admitted.
```

(Ungraded thought question) Are there any important properties of the function forallb which are not captured by this specification?

7.6 The Logic of Rocq

Rocq's logical core, the *Calculus of Inductive Constructions*, differs in some important ways from other formal systems that are used by mathematicians to write down precise and rigorous definitions and proofs – in particular from Zermelo-Fraenkel Set Theory (ZFC), the most popular foundation for paper-and-pencil mathematics.

We conclude this chapter with a brief discussion of some of the most significant differences between these two worlds.

7.6.1 Functional Extensionality

Rocq's logic is quite minimalistic. This means that one occasionally encounters cases where translating standard mathematical reasoning into Rocq is cumbersome – or even impossible – unless we enrich its core logic with additional axioms.

For example, the equality assertions that we have seen so far mostly have concerned elements of inductive types (**nat**, **bool**, etc.). But, since Rocq'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:

In certain cases Rocq can successfully prove equality propositions stating that two *functions* are equal to each other:

```
Example function_equality_ex1 :
  (fun x ⇒ 3 + x) = (fun x ⇒ (pred 4) + x).
Proof. reflexivity. Qed.
```

This works when Rocq can simplify the functions to the same expression, but this doesn't always happen.

These two functions are equal just by simplification, but in general functions can be equal for more interesting reasons.

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 Rocq's built-in logic. This means that some intuitively obvious propositions are not provable.

```
Example function_equality_ex2 :
  (fun x ⇒ plus x 1) = (fun x ⇒ plus 1 x).
Proof.
  Fail reflexivity. Fail rewrite add_comm.
Abort.
```

However, if we like, we can add functional extensionality to Rocq 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 add_comm.
Qed.
```

Naturally, we need to be quite careful when adding new axioms into Rocq's logic, as this can render it *inconsistent* – that is, it may become possible to prove every proposition, including **False**, 2+2=5, etc.!

In general, 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 two-argument definition:

```
Fixpoint rev_append {X} (l1 l2 : list X) : list X :=
  match l1 with
  | [] ⇒ l2
  | x :: l1' ⇒ rev_append l1' (x :: l2)
  end.

Definition tr_rev {X} (l : list X) : list X :=
  rev_append l [].
```

This version 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.
  □
```

7.6.2 Classical vs. Constructive Logic

We have seen that it is not possible to test whether or not a proposition P holds while defining a Rocq function. You may be surprised to learn that a similar restriction applies in *proofs*! In other words, the following intuitive reasoning principle is not derivable in Rocq:

```
Definition excluded_middle := ∀ P : Prop,
  P ∨ ¬ 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.

However, in the special case where we happen to know that P is reflected in some boolean term `b`, knowing whether it holds or not is trivial: we just have to check the value of `b`.

```
Theorem restricted_excluded_middle : ∀ P b,
  (P ↔ b = true) → P ∨ ¬ 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 : ∀ (n m : nat),
  n = m ∨ n ≠ m.
Proof.
  intros n m.
  apply (restricted_excluded_middle (n = m) (n =? m)).
  symmetry.
  apply eqb_eq.
Qed.
```

Sadly, this trick only works for decidable propositions.

It may seem strange that the general excluded middle is not available by default in Rocq, 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 Rocq make stronger claims than the

analogous statements in standard mathematics. Notably, a Rocq proof of ∃ x, P x always includes a particular value of x for which we can prove P x – in other words, every proof of existence is *constructive*.

Logics like Rocq's, which do not assume the excluded middle, are referred to as *constructive logics*.

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. So 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 × *sqrt* 2) = *sqrt* 2 ^ 2 = 2. □

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 this, we finish the proof knowing that such a and b exist, but not being sure of their actual values.

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 Rocq'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 in Software Foundations 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, ¬ ∃ x, P x. From this premise, it is not hard to derive ∀ x, ¬ P x. If we manage to show that this 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 ∃ x, P x using a proof of ¬ ¬ (∃ x, P x). Allowing ourselves to remove double negations from arbitrary statements is equivalent to assuming the excluded middle law, as shown in one of the exercises below. Thus, this line of reasoning cannot be encoded in Rocq without assuming additional axioms.

Exercise: 3 stars, standard (excluded_middle_irrefutable) Proving the consistency of Rocq with the general excluded middle axiom requires complicated reasoning that cannot be carried out within Rocq 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 the negation of such an axiom leads to a contradiction. If $\neg\ (P \vee \neg P)$ were provable, then by de_morgan_not_or as proved above, $P \wedge \neg P$ would be provable, which would be a contradiction. So, it is safe to add $P \vee \neg P$ as an axiom for any particular P.

Succinctly: for any proposition P, *Rocq* `is` *consistent* ==> *Rocq* + $(P \vee \neg P)$ `is` *consistent*.

```
Theorem excluded_middle_irrefutable: ∀ (P : Prop),
  ¬ ¬ (P ∨ ¬ P).
Proof.
  Admitted.
  □
```

Exercise: 3 stars, advanced (not_exists_dist) It is a theorem of classical logic that the following two assertions are equivalent:

~ (exists x, ~ 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 →
  ∀ (X:Type) (P : X → Prop),
    ¬ (∃ x, ¬ P x) → (∀ x, P x).
Proof.
  Admitted.
  □
```

Exercise: 5 stars, standard, optional (classical_axioms) For those who like a challenge, here is an exercise adapted from the Coq'Art book by Bertot and Casteran (p. 123). Each of the following five statements, together with excluded_middle, can be considered as characterizing classical logic. We can't prove any of them in Rocq, but we can consistently add any *one* of them as an axiom if we wish to work in classical logic.

To see this, prove that all six propositions (these five 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 := ∀ P Q: Prop,
  ((P → Q) → P) → P.

Definition double_negation_elimination := ∀ P:Prop,
  ~~P → P.

Definition de_morgan_not_and_not := ∀ P Q:Prop,
```

```
  ~(~P ∧ ¬Q) → P ∨ Q.

Definition implies_to_or := ∀ P Q:Prop,
  (P → Q) → (¬P ∨ Q).

Definition consequentia_mirabilis := ∀ P:Prop,
  (¬P → P) → P.
```

Chapter 8

Library LF.IndProp

8.1 IndProp: Inductively Defined Propositions

```
Set Warnings "-notation-overridden".
From LF Require Export Logic.
```

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: *inductively defined propositions*.

To begin, some examples...

8.2.1 Example: The Collatz Conjecture

The *Collatz Conjecture* is a famous open problem in number theory.

Its statement is quite simple. First, we define a function csf on numbers, as follows (where csf stands for "Collatz step function"):

```
Fixpoint div2 (n : nat) : nat :=
  match n with
    0 ⇒ 0
  | 1 ⇒ 0
  | S (S n) ⇒ S (div2 n)
  end.

Definition csf (n : nat) : nat :=
  if even n then div2 n
  else (3 × n) + 1.
```

Next, we look at what happens when we repeatedly apply csf to some given starting number. For example, csf 12 is 6, and csf 6 is 3, so by repeatedly applying csf we get the sequence 12, 6, 3, 10, 5, 16, 8, 4, 2, 1.

Similarly, if we start with 19, we get the longer sequence 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

Both of these sequences eventually reach 1. The question posed by Collatz was: Is the sequence starting from *any* positive natural number guaranteed to reach 1 eventually?

To formalize this question in Rocq, we might try to define a recursive *function* that calculates the total number of steps that it takes for such a sequence to reach 1.

```
Fail Fixpoint reaches1_in (n : nat) : nat :=
  if n =? 1 then 0
  else 1 + reaches1_in (csf n).
```

You can write this definition in a standard programming language. This definition is, however, rejected by Rocq's termination checker, since the argument to the recursive call, csf n, is not "obviously smaller" than n.

Indeed, this isn't just a pointless limitation: functions in Rocq are required to be total, to ensure logical consistency.

Moreover, we can't fix it by devising a more clever termination checker: deciding whether this particular function is total would be equivalent to settling the Collatz conjecture!

Another idea could be to express the concept of "eventually reaches 1 in the Collatz sequence" as an *recursively defined property* of numbers **Collatz_holds_for** : **nat** → Prop.

```
Fail Fixpoint Collatz_holds_for (n : nat) : Prop :=
  match n with
  | 0 ⇒ False
  | 1 ⇒ True
  | _ ⇒ if even n then Collatz_holds_for (div2 n)
                  else Collatz_holds_for ((3 × n) + 1)
  end.
```

This recursive function is also rejected by the termination checker, since, while we could in principle convince Rocq that div2 n is smaller than n, we certainly can't convince it that $(3 \times n) + 1$ is smaller than n!

Fortunately, there is another way to do it: We can express the concept "reaches 1 eventually in the Collatz sequence" as an *inductively defined property* of numbers. Intuitively, this property is defined by a set of rules:

(Chf_one) Collatz_holds_for 1
  even n = true Collatz_holds_for (div2 n)

(Chf_even) Collatz_holds_for n
  even n = false Collatz_holds_for ((3 * n) + 1)

(Chf_odd) Collatz_holds_for n

So there are three ways to prove that a number n eventually reaches 1 in the Collatz sequence:

- n is 1;
- n is even and div2 n eventually reaches 1;
- n is odd and $(3 \times n) + 1$ eventually reaches 1.

We can prove that a number reaches 1 by constructing a (finite) derivation using these rules. For instance, here is the derivation proving that 12 reaches 1 (where we left out the evenness/oddness premises):

———————————————— (Chf_one) Collatz_holds_for 1 ————————————————
(Chf_even) Collatz_holds_for 2 ———————————————— (Chf_even) Collatz_holds_for 4 ———————————————— (Chf_even) Collatz_holds_for 8 ———————————————— (Chf_even) Collatz_holds_for 16 ————————————————
(Chf_odd) Collatz_holds_for 5 ———————————————— (Chf_even) Collatz_holds_for 10 ———————————————— (Chf_odd) Collatz_holds_for 3 ———————————————— (Chf_even) Collatz_holds_for 6 ————————————————
(Chf_even) Collatz_holds_for 12

Formally in Rocq, the **Collatz_holds_for** property is *inductively defined*:

```
Inductive Collatz_holds_for : nat → Prop :=
  | Chf_one : Collatz_holds_for 1
  | Chf_even (n : nat) : even n = true →
                         Collatz_holds_for (div2 n) →
                         Collatz_holds_for n
  | Chf_odd (n : nat) : even n = false →
                         Collatz_holds_for ((3 × n) + 1) →
                         Collatz_holds_for n.
```

What we've done here is to use Rocq's `Inductive` definition mechanism to characterize the property "Collatz holds for..." by stating three different ways in which it can hold: (1) Collatz holds for 1, (2) if Collatz holds for div2 n and n is even then Collatz holds for n, and (3) if Collatz holds for $(3 \times n) + 1$ and n is odd then Collatz holds for n. This Rocq definition directly corresponds to the three rules we wrote informally above.

For particular numbers, we can now prove that the Collatz sequence reaches 1 (we'll look more closely at how it works a bit later in the chapter):

```
Example Collatz_holds_for_12 : Collatz_holds_for 12.
Proof.
  apply Chf_even. reflexivity. simpl.
  apply Chf_even. reflexivity. simpl.
  apply Chf_odd. reflexivity. simpl.
  apply Chf_even. reflexivity. simpl.
```

```
  apply Chf_odd. reflexivity. simpl.
  apply Chf_even. reflexivity. simpl.
  apply Chf_even. reflexivity. simpl.
  apply Chf_even. reflexivity. simpl.
  apply Chf_even. reflexivity. simpl.
  apply Chf_one.
Qed.
```

The Collatz conjecture then states that the sequence beginning from *any* positive number reaches 1:

```
Conjecture collatz : ∀ n, n ≠ 0 → Collatz_holds_for n.
```

If you succeed in proving this conjecture, you've got a bright future as a number theorist! But don't spend too long on it – it's been open since 1937.

8.2.2 Example: Binary relation for comparing numbers

A binary *relation* on a set X has Rocq type X → X → Prop. This is a family of propositions parameterized by two elements of X – i.e., a proposition about pairs of elements of X.

For example, one familiar binary relation on **nat** is **le** : **nat** → **nat** → Prop, the less-than-or-equal-to relation, which can be inductively defined by the following two rules:

(le_n) le n n

le n m

(le_S) le n (S m) These rules say that there are two ways to show that a number is less than or equal to another: either observe that they are the same number, or, if the second has the form S *m*, give evidence that the first is less than or equal to *m*.

This corresponds to the following inductive definition in Rocq:

```
Module LePlayground.

Inductive le : nat → nat → Prop :=
  | le_n (n : nat) : le n n
  | le_S (n m : nat) : le n m → le n (S m).

Notation "n <= m" := (le n m) (at level 70).
```

This definition is a bit simpler and more elegant than the Boolean function leb we defined in Basics. As usual, **le** and leb are equivalent, and there is an exercise about that later.

```
Example le_3_5 : 3 ≤ 5.
Proof.
  apply le_S. apply le_S. apply le_n. Qed.

End LePlayground.
```

8.2.3 Example: Transitive Closure

Another example: The *reflexive and transitive closure* of a relation **R** is the smallest relation that contains **R** and that is reflexive and transitive. This can be defined by the following three rules (where we added a reflexivity rule to **clos_trans**):

R x y

(t_step) clos_trans R x y
clos_trans R x y clos_trans R y z

(t_trans) clos_trans R x z

In Rocq this looks as follows:

```
Inductive clos_trans {X: Type} (R: X→X→Prop) : X→X→Prop :=
  | t_step (x y : X) :
      R x y →
      clos_trans R x y
  | t_trans (x y z : X) :
      clos_trans R x y →
      clos_trans R y z →
      clos_trans R x z.
```

For example, suppose we define a "parent of" relation on a group of people...

```
Inductive Person : Type := Sage | Cleo | Ridley | Moss.

Inductive parent_of : Person → Person → Prop :=
  po_SC : parent_of Sage Cleo
| po_SR : parent_of Sage Ridley
| po_CM : parent_of Cleo Moss.
```

In this example, Sage is a parent of both Cleo and Ridley; and Cleo is a parent of Moss.

The **parent_of** relation is not transitive, but we can define an "ancestor of" relation as its transitive closure:

```
Definition ancestor_of : Person → Person → Prop :=
  clos_trans parent_of.
```

Here is a derivation showing that Sage is an ancestor of Moss:

———————————————————————(po_SC) ———————————————————————(po_C
parent_of Sage Cleo parent_of Cleo Moss ———————————————————————(t_step)
———————————————————————(t_step) ancestor_of Sage Cleo ancestor_of
Cleo Moss ———
ancestor_of Sage Moss

```
Example ancestor_of_ex : ancestor_of Sage Moss.
Proof.
  unfold ancestor_of. apply t_trans with Cleo.
  - apply t_step. apply po_SC.
```

```
  - apply t_step. apply po_CM. Qed.
```

Computing the transitive closure can be undecidable even for a relation R that is decidable (e.g., the cms relation below), so in general we can't expect to define transitive closure as a boolean function. Fortunately, Rocq allows us to define transitive closure as an inductive relation.

The transitive closure of a binary relation cannot, in general, be expressed in first-order logic. The logic of Rocq is, however, much more powerful, and can easily define such inductive relations.

8.2.4 Example: Reflexive and Transitive Closure

As another example, the *reflexive and transitive closure* of a relation **R** is the smallest relation that contains **R** and that is reflexive and transitive. This can be defined by the following three rules (where we added a reflexivity rule to **clos_trans**):

R x y

(rt_step) clos_refl_trans R x y

(rt_refl) clos_refl_trans R x x

clos_refl_trans R x y clos_refl_trans R y z

(rt_trans) clos_refl_trans R x z

```
Inductive clos_refl_trans {X: Type} (R: X→X→Prop) : X→X→Prop :=
  | rt_step (x y : X) :
      R x y →
      clos_refl_trans R x y
  | rt_refl (x : X) :
      clos_refl_trans R x x
  | rt_trans (x y z : X) :
      clos_refl_trans R x y →
      clos_refl_trans R y z →
      clos_refl_trans R x z.
```

For instance, this enables an equivalent definition of the Collatz conjecture. First we define a binary relation corresponding to the "Collatz step function" csf:

```
Definition cs (n m : nat) : Prop := csf n = m.
```

This Collatz step relation can be used in conjunction with the reflexive and transitive closure operation to define a *Collatz multi-step* (cms) relation, expressing that a number n reaches another number m in zero or more Collatz steps:

```
Definition cms n m := clos_refl_trans cs n m.
Conjecture collatz' : ∀ n, n ≠ 0 → cms n 1.
```

This cms relation defined in terms of **clos_refl_trans** allows for more interesting derivations than the linear ones of the directly-defined **Collatz_holds_for** relation:

csf 16 = 8 csf 8 = 4 csf 4 = 2 csf 2 = 1 ——————(rt_step) ——————(rt_step) ——————(rt_step) ——————(rt_step) cms 16 8 cms 8 4 cms 4 2 cms 2 1 ——————————————————————————————(rt_trans) cms 16 4 cms 4 1 ——————————————————————————————(rt_trans) cms 16 1

Exercise: 1 star, standard, optional (clos_refl_trans_sym) How would you modify the **clos_refl_trans** definition above so as to define the reflexive, symmetric, and transitive closure?

8.2.5 Example: Permutations

The familiar mathematical concept of *permutation* also has an elegant formulation as an inductive relation. For simplicity, let's focus on permutations of lists with exactly three elements.

We can define such permulations by the following rules:

———————————————————————————————

(perm3_swap12) Perm3 a;b;c b;a;c

———————————————————————————————

(perm3_swap23) Perm3 a;b;c a;c;b

———————————————————————————————

(perm3_trans) Perm3 l1 l3
  Perm3 l1 l2 Perm3 l2 l3

———————————————————————————————

For instance we can derive **Perm3** [1;2;3] [3;2;1] as follows:

——————(perm_swap12) ——————————————————————(perm_swap23) Perm3 1;2;3 2;1;3 Perm3 2;1;3 2;3;1 ——————————————————————————————(p ——————(perm_swap12) Perm3 1;2;3 2;3;1 Perm 2;3;1 3;2;1 ——————————————————————————————

Perm3 1;2;3 3;2;1

This definition says:

- If *l2* can be obtained from *l1* by swapping the first and second elements, then *l2* is a permutation of *l1*.
- If *l2* can be obtained from *l1* by swapping the second and third elements, then *l2* is a permutation of *l1*.
- If *l2* is a permutation of *l1* and *l3* is a permutation of *l2*, then *l3* is a permutation of *l1*.

In Rocq **Perm3** is given the following inductive definition:

```
Inductive Perm3 {X : Type} : list X → list X → Prop :=
  | perm3_swap12 (a b c : X) :
```

```
      Perm3 [a;b;c] [b;a;c]
  | perm3_swap23 (a b c : X) :
      Perm3 [a;b;c] [a;c;b]
  | perm3_trans (l1 l2 l3 : list X) :
      Perm3 l1 l2 → Perm3 l2 l3 → Perm3 l1 l3.
```

Exercise: 1 star, standard, optional (perm) According to this definition, is [1;2;3] a permutation of itself?

8.2.6 Example: Evenness (yet again)

We've already seen two ways of stating a proposition that a number n is even: We can say
(1) even n = true (using the recursive boolean function even), or
(2) $\exists\ k$, n = double k (using an existential quantifier).

A third possibility, which we'll use as a simple running example in this chapter, is to say that a number is even if we can *establish* its evenness from the following two rules:

(ev_0) ev 0
ev n

(ev_SS) ev (S (S n))

Intuitively these rules say that:

- The number 0 is even.
- If n is even, then S (S n) is even.

(Defining evenness in this way may seem a bit confusing, since we have already seen two perfectly good ways of doing it. It makes a convenient running example because it is simple and compact, but we will soon return to the more compelling examples above.)

To illustrate how this new definition of evenness works, let's imagine using it to show that 4 is even:

——— (ev_0) ev 0 ——— (ev_SS) ev (S (S 0)) ———
(ev_SS) ev (S (S (S (S 0))))

In words, 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 can translate the informal definition of evenness from above into a formal Inductive declaration, where each "way that a number can be even" corresponds to a separate constructor:

```
Inductive ev : nat → Prop :=
  | ev_0 : ev 0
  | ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

Such definitions are interestingly different from previous uses of `Inductive` for defining inductive datatypes like **nat** or **list**. 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).

or (equivalently but more explicitly):

Inductive list (X:Type) : Type := | nil : list X | cons (x : X) (l : list X) : 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 type (i.e., **list** X). But if we had 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, while in `Inductive ev : nat → Prop := ...`, the unnamed **nat** argument is an index.

We can think of the inductive definition of **ev** as defining a Rocq property `ev : nat → Prop`, together with two "evidence constructors":

```
Check ev_0 : ev 0.
Check ev_SS : ∀ (n : nat), ev n → ev (S (S n)).
```

Indeed, Rocq also accepts the following equivalent definition of **ev**:

```
Module EvPlayground.

Inductive ev : nat → Prop :=
  | ev_0 : ev 0
  | ev_SS : ∀ (n : nat), ev n → ev (S (S n)).

End EvPlayground.
```

These evidence constructors can be thought of as "primitive evidence of evenness", and they can be used later on just like proven theorems. In particular, we can use Rocq's `apply` tactic with the constructor names to obtain evidence for **ev** of 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 to combine several constructors:

```
Theorem ev_4' : ev 4.
```

```
Proof. apply (ev_SS 2 (ev_SS 0 ev_0)). Qed.
```

In this way, 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.
```

□

8.2.7 Constructing Evidence for Permutations

Similarly we can apply the evidence constructors to obtain evidence of **Perm3** [1;2;3] [3;2;1]:

```
Lemma Perm3_rev : Perm3 [1;2;3] [3;2;1].
Proof.
  apply perm3_trans with (l2:=[2;3;1]).
  - apply perm3_trans with (l2:=[2;1;3]).
    + apply perm3_swap12.
    + apply perm3_swap23.
  - apply perm3_swap12.
Qed.
```

And again we can equivalently use function application syntax to combine several constructors. (Note that the Rocq type checker can infer not only types, but also nats and lists, when they are clear from the context.)

```
Lemma Perm3_rev' : Perm3 [1;2;3] [3;2;1].
Proof.
  apply (perm3_trans _ [2;3;1] _
           (perm3_trans _ [2;1;3] _
             (perm3_swap12 _ _ _)
             (perm3_swap23 _ _ _))
           (perm3_swap12 _ _ _)).
Qed.
```

So the informal derivation trees we drew above are not too far from what's happening formally. Formally we're using the evidence constructors to build *evidence trees*, similar to the finite trees we built using the constructors of data types such as nat, list, binary trees, etc.

Exercise: 1 star, standard (Perm3) `Lemma Perm3_ex1 : Perm3 [1;2;3] [2;3;1].`

```
Proof.
    Admitted.
```

```
Lemma Perm3_refl : ∀ (X : Type) (a b c : X),
  Perm3 [a;b;c] [a;b;c].
Proof.
    Admitted.
```

□

8.3 Using Evidence in Proofs

Besides *constructing* evidence that numbers are even, we can also *destruct* such evidence, reasoning about how it could have been built.

Defining **ev** with an `Inductive` declaration tells Rocq 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 proposition **ev** n, then we know that E must be one of two things:

- E = `ev_0` and n = `O`, or
- E = `ev_SS` n' E' and n = `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 either by *case analysis* or by *induction* on such evidence. Let's look at a few examples to see what this means in practice.

8.3.1 Destructing and Inverting 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 = `O` and the case where n = `S` n' for some n'. But for some proofs we may instead want to analyze the evidence for **ev** n *directly*.

As a tool for such proofs, we can formalize the intuitive characterization that we gave above for evidence of **ev** n, using `destruct`.

```
Lemma ev_inversion : ∀ (n : nat),
    ev n →
    (n = 0) ∨ (∃ n', n = S (S n') ∧ ev n').
Proof.
  intros n E. destruct E as [ | n' E'] eqn:EE.
  -
```

```
    left. reflexivity.
  -
    right. ∃ n'. split. reflexivity. apply E'.
Qed.
```

Facts like this are often called "inversion lemmas" because they allow us to "invert" some given information to reason about all the different ways it could have been derived.

Here there are two ways to prove **ev** n, and the inversion lemma makes this explicit.

Exercise: 1 star, standard (le_inversion) Let's prove a similar inversion lemma for **le**.

```
Lemma le_inversion : ∀ (n m : nat),
  le n m →
  (n = m) ∨ (∃ m', m = S m' ∧ le n m').
Proof.
    Admitted.
```

□

We can use the inversion lemma that we proved above to help structure proofs:

```
Theorem evSS_ev : ∀ n, ev (S (S n)) → ev n.
Proof.
  intros n E. apply ev_inversion in E. destruct E as [H0|H1].
  - discriminate H0.
  - destruct H1 as [n' [Hnn' E']]. injection Hnn' as Hnn'.
    rewrite Hnn'. apply E'.
Qed.
```

Note how the inversion lemma produces 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`.

Rocq provides a handy tactic called `inversion` that factors out this common pattern, saving us the trouble of explicitly stating and proving an inversion lemma for every `Inductive` definition we make.

Here, the `inversion` tactic can detect (1) that the first case, where $n = 0$, does not apply and (2) that the n' that appears in the `ev_SS` case must be the same as n. It includes an "`as`" annotation similar to `destruct`, allowing us to assign names rather than have Rocq choose them.

```
Theorem evSS_ev' : ∀ n,
  ev (S (S n)) → ev n.
Proof.
  intros n E. inversion E as [| n' E' Hnn'].
  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. Compare:

```
Theorem one_not_even : ¬ ev 1.
Proof.
  intros H. apply ev_inversion in H. destruct H as [ | [m [Hm _]]].
  - discriminate H.
  - discriminate Hm.
Qed.

Theorem one_not_even' : ¬ ev 1.
Proof. 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 : ∀ n,
  ev (S (S (S (S n)))) → ev n.
Proof.
  Admitted.
```

□

Exercise: 1 star, standard (ev5_nonsense) Prove the following result using `inversion`.

```
Theorem ev5_nonsense :
  ev 5 → 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 `rewrite`s that are typically necessary in the case of `injection`. It can also be applied to analyze evidence for arbitrary inductively defined propositions, not just equality. As examples, we'll use it to re-prove some theorems from chapter Tactics. (Here we are being a bit lazy by omitting the `as` clause from `inversion`, thereby asking Rocq to choose names for the variables and hypotheses that it introduces.)

```
Theorem inversion_ex1 : ∀ (n m o : nat),
  [n; m] = [o; o] → [n] = [m].
Proof.
  intros n m o H. inversion H. reflexivity. Qed.

Theorem inversion_ex2 : ∀ (n : nat),
  S n = O → 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 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 to the proof context all equations that must hold of the arguments given to P – e.g., $n' = n$ in the proof of evSS_ev).

The ev_double exercise above allows us to easily show 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 : ∀ n,
  ev n → Even n.
Proof.
  unfold Even.
```

We could try to proceed by case analysis or induction on n. But since **ev** is mentioned in a premise, this strategy seems unpromising, 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.

```
  intros n E. inversion E as [EQ' | n' E' EQ'].
  - ∃ 0. reflexivity.
  -
    assert (H: (∃ k', n' = double k')
                → (∃ n0, S (S n') = double n0)).
        { intros [k' EQ'']. ∃ (S k'). simpl.
          rewrite ← EQ''. reflexivity. }
    apply H.
```

Unfortunately, now we are stuck. To see this clearly, let's move E' back into the goal from the hypotheses.

```
    generalize dependent E'.
```

Now it is obvious that we are trying to prove another instance of the same theorem we set out to prove – only here we are talking about n' instead of n. `Abort`.

8.3.2 Induction on Evidence

If this story feels familiar, it is no coincidence: We 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 Rocq to generate one subgoal for each constructor that could have been used to build that evidence, while providing an induction hypothesis for each recursive occurrence of the property in question.

To prove that a property of n holds for all even numbers (i.e., those for which **ev** n holds), we can use induction on **ev** n. This requires us to prove two things, corresponding to the two ways in which **ev** n could have been constructed. If it was constructed by `ev_0`, then n=0 and the property must hold of 0. If it was constructed by `ev_SS`, then the evidence of **ev** n is of the form `ev_SS` n' E', where n = `S` (`S` n') and E' is evidence for **ev** n'. In this case, the inductive hypothesis says that the property we are trying to prove holds for n'.

Let's try proving that lemma again:

```
Lemma ev_Even : ∀ n,
  ev n → Even n.
Proof.
  unfold Even. intros n E.
  induction E as [|n' E' IH].
  -
    ∃ 0. reflexivity.
  -
    destruct IH as [k Hk]. rewrite Hk.
    ∃ (S k). simpl. reflexivity.
Qed.
```

Here, we can see that Rocq 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 : ∀ n,
  ev n ↔ 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 – in particular for formalizing the semantics of programming languages.

The following exercises provide simpler examples of this technique, to help you familiarize yourself with it.

Exercise: 2 stars, standard (ev_sum) `Theorem ev_sum : ∀ n m, ev n → ev m → ev (n + m).`

```
Proof.
    Admitted.
    □
```

Exercise: 3 stars, advanced, especially useful (ev_ev__ev) `Theorem ev_ev__ev : ∀ n m,`

```
  ev (n+m) → ev n → 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 : ∀ n m p,
  ev (n+m) → ev (n+p) → ev (m+p).
Proof.
    Admitted.
    □
```

8.3.3 Multiple Induction Hypotheses

Recall the definition of the reflexive, transitive, closure of a relation:

```
Module CLOS_REFL_TRANS_REMAINDER.
Inductive clos_refl_trans {X: Type} (R: X→X→Prop) : X→X→Prop :=
  | rt_step (x y : X) :
        R x y →
        clos_refl_trans R x y
  | rt_refl (x : X) :
        clos_refl_trans R x x
  | rt_trans (x y z : X) :
        clos_refl_trans R x y →
        clos_refl_trans R y z →
        clos_refl_trans R x z.
End CLOS_REFL_TRANS_REMAINDER.
```

Let's say that a relation on a type X is *diagonal* if it refines the identity relation – i.e., if R x y implies $x = y$.

```
Definition isDiagonal {X : Type} (R: X → X → Prop) :=
  ∀ x y, R x y → x = y.
```

Now consider the following lemma about diagonal relations:

```
Lemma closure_of_diagonal_is_diagonal: ∀ X (R: X → X → Prop),
  isDiagonal R →
  isDiagonal (clos_refl_trans R).
Proof.
  intros X R IsDiag x y H.
  induction H as [ x y H | x | x y z H IH H' IH' ].
  - specialize (IsDiag x y H). rewrite → IsDiag. reflexivity.
  - reflexivity.
  -
   rewrite → IH, ← IH'. reflexivity.
Qed.
```

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 → 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 the Logic chapter) of applying theorems to arguments, and note that the same technique works with constructors of inductively defined propositions.

```
Theorem ev'_ev : ∀ n, ev' n ↔ ev n.
Proof.
    Admitted.
```

□

We can do similar inductive proofs on the **Perm3** relation, which we defined earlier as follows:

```
Module Perm3Reminder.

Inductive Perm3 {X : Type} : list X → list X → Prop :=
  | perm3_swap12 (a b c : X) :
        Perm3 [a;b;c] [b;a;c]
  | perm3_swap23 (a b c : X) :
        Perm3 [a;b;c] [a;c;b]
  | perm3_trans (l1 l2 l3 : list X) :
        Perm3 l1 l2 → Perm3 l2 l3 → Perm3 l1 l3.

End Perm3Reminder.

Lemma Perm3_symm : ∀ (X : Type) (l1 l2 : list X),
  Perm3 l1 l2 → Perm3 l2 l1.
```

```
Proof.
  intros X l1 l2 E.
  induction E as [a b c | a b c | l1 l2 l3 E12 IH12 E23 IH23].
  - apply perm3_swap12.
  - apply perm3_swap23.
  - apply (perm3_trans _ l2 _).
    × apply IH23.
    × apply IH12.
Qed.
```

Exercise: 2 stars, standard (Perm3_In) `Lemma Perm3_In : ∀ (X : Type) (x : X) (l1 l2 : list X),`
`  Perm3 l1 l2 → In x l1 → In x l2.`

```
Proof.
  Admitted.
```
□

Exercise: 1 star, standard, optional (Perm3_NotIn) `Lemma Perm3_NotIn : ∀ (X : Type) (x : X) (l1 l2 : list X),`
`  Perm3 l1 l2 → ¬In x l1 → ¬In x l2.`

```
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard, optional (NotPerm3) Proving that something is NOT a permutation is quite tricky. Some of the lemmas above, like Perm3_In can be useful for this. `Example Perm3_example2 : ¬ Perm3 [1;2;3] [1;2;4].`

```
Proof.
  Admitted.
```
□

8.4 Exercising with 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 that we briefly saw above.

`Inductive le : nat → nat → 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).
```

(We've written the definition a bit differently this time, giving explicit names to the arguments to the constructors and moving them to the left of the colons.)

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<=3 or 3<=6), and we can use tactics like `inversion` to extract information from $\leq$ hypotheses in the context (e.g., to prove that $(2 \leq 1) \rightarrow 2+2=5$.)

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 <= 3.
Proof.
  apply le_n. Qed.

Theorem test_le2 :
  3 <= 6.
Proof.
  apply le_S. apply le_S. apply le_S. apply le_n. Qed.

Theorem test_le3 :
  (2 <= 1) -> 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 "n < m" := (lt n m).
```

The $\geq$ operation is defined in terms of $\leq$.

```
Definition ge (m n : nat) : Prop := le n m.
Notation "m >= n" := (ge m n).

End Playground.
```

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'*.

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.

Exercise: 3 stars, standard, especially useful (le_facts) `Lemma le_trans : ∀ m n o, m ≤ n → n ≤ o → m ≤ o.`

```
Proof.
  Admitted.

Theorem O_le_n : ∀ n,
  0 ≤ n.
Proof.
  Admitted.

Theorem n_le_m__Sn_le_Sm : ∀ n m,
  n ≤ m → S n ≤ S m.
Proof.
  Admitted.

Theorem Sn_le_Sm__n_le_m : ∀ n m,
  S n ≤ S m → n ≤ m.
Proof.
  Admitted.

Theorem le_plus_l : ∀ a b,
  a ≤ a + b.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard, especially useful (plus_le_facts1) `Theorem plus_le : ∀ n1 n2 m,`

```
  n1 + n2 ≤ m →
  n1 ≤ m ∧ n2 ≤ m.
Proof.
  Admitted.

Theorem plus_le_cases : ∀ n m p q,
  n + m ≤ p + q → n ≤ p ∨ m ≤ q.
```

Hint: May be easiest to prove by induction on *n*. `Proof.`

```
  Admitted.
```

□

Exercise: 2 stars, standard, especially useful (plus_le_facts2) `Theorem plus_le_compat_l : ∀ n m p, n ≤ m → p + n ≤ p + m. Proof. Admitted.`

`Theorem plus_le_compat_r : ∀ n m p, n ≤ m → n + p ≤ m + p. Proof. Admitted.`

`Theorem le_plus_trans : ∀ n m p, n ≤ m → n ≤ m + p. Proof. Admitted.` □

Exercise: 3 stars, standard, optional (lt_facts) `Theorem lt_ge_cases : ∀ n m, n < m ∨ n ≥ m. Proof. Admitted.`

`Theorem n_lt_m__n_le_m : ∀ n m, n < m → n ≤ m. Proof. Admitted.`

`Theorem plus_lt : ∀ n1 n2 m, n1 + n2 < m → n1 < m ∧ n2 < m. Proof. Admitted.` □

Exercise: 4 stars, standard, optional (leb_le) `Theorem leb_complete : ∀ n m, n <=? m = true → n ≤ m. Proof. Admitted.`

`Theorem leb_correct : ∀ n m, n ≤ m → n <=? m = true.`

```
Proof.
    Admitted.
```

Hint: The next two can easily be proved without using `induction`.

```
Theorem leb_iff : ∀ n m,
  n <=? m = true ↔ n ≤ m.
Proof.
    Admitted.

Theorem leb_true_trans : ∀ n m o,
  n <=? m = true → m <=? o = true → n <=? o = true.
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 : nat → nat → nat → 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 Rocq.

```
Definition fR : nat → nat → nat
  . Admitted.

Theorem R_equiv_fR : ∀ m n o, R m n o ↔ fR m n = o.
Proof.
  Admitted.
  □

End R.
```

Exercise: 4 stars, advanced (subsequence) A list is a *subsequence* of another list if all of the elements in the first list occur in the same order in the second list, possibly with some extra elements in between. For example,

1;2;3
is a subsequence of each of the lists
1;2;3 1;1;1;2;2;3 1;2;7;3 5;6;1;9;9;2;7;3;8
but it is *not* a subsequence of any of the lists
1;2 1;3 5;6;2;1;7;3;8.

- Define an inductive proposition subseq on list nat that captures what it means to be a subsequence. There are a number of correct ways to do this. You should make sure that your definition behaves correctly on all the positive and negative examples above, but you do not need to prove this formally.
- 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*.
- (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*.

```
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 Rocq 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]

Exercise: 2 stars, standard, optional (total_relation) Define an inductive binary relation total_relation that holds between every pair of natural numbers.

```
Inductive total_relation : nat → nat → Prop :=

.

Theorem total_relation_is_total : ∀ n m, total_relation n m.
  Proof.
    Admitted.
    □
```

Exercise: 2 stars, standard, optional (empty_relation) Define an inductive binary relation empty_relation (on numbers) that never holds.

```
Inductive empty_relation : nat → nat → Prop :=

.

Theorem empty_relation_is_empty : ∀ n m, ¬ empty_relation n m.
  Proof.
    Admitted.
    □
```

8.5 Case Study: Regular Expressions

Many of the examples above were simple and – in the case of the **ev** property – even a bit artificial. To give a better sense of the power of inductively defined propositions, we now show how to use them to model a classic concept in computer science: *regular expressions*.

8.5.1 Definitions

Regular expressions are a natural 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 – which in this exercise we represent as *lists* with elements from T.

(Technical aside: 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 present purposes.)

We connect regular expressions and strings by defining when a regular expression *matches* some string.

Informally this looks as follows:

- The regular expression EmptySet does not match any string.
- EmptyStr matches the empty string [].
- 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 $s = s\_1 ++ \ldots ++ s\_k$, 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 intuition into a set of rules, where we write $s =\tilde{} \; re$ to say that re matches s:

(MEmpty) □ =~ EmptyStr

(MChar) x =~ (Char x)
  s1 =~ re1 s2 =~ re2

(MApp) (s1 ++ s2) =~ (App re1 re2)
  s1 =~ re1

(MUnionL) s1 =~ (Union re1 re2)
  s2 =~ re2

(MUnionR) s2 =~ (Union re1 re2)

(MStar0) □ =~ (Star re)
  s1 =~ re s2 =~ (Star re)

(MStarApp) (s1 ++ s2) =~ (Star re)

This directly corresponds to the following `Inductive` definition. We use the notation $s =\tilde{} \; re$ 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 =~ 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 s2 re1 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).
```

Notice that these rules are not *quite* the same as the intuition that we gave at the beginning of the section. First, we don't need to include a rule explicitly stating that no string is matched by EmptySet; indeed, the syntax of inductive definitions doesn't even *allow* us to give such a "negative rule." We just don't happen to include any rule that would have the effect of EmptySet matching some string.

Second, the intuition we gave for Union and Star correspond to two constructors each: MUnionL / MUnionR, and MStar0 / MStarApp. The result is logically equivalent to the original intuition but more convenient to use in Rocq, 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 intuition is indeed equivalent.)

Let's illustrate these rules with a few examples.

8.5.2 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], Rocq 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 string 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 matched by re is also matched by Star re.

```
Lemma MStar1 :
  ∀ T s (re : reg_exp T) ,
    s =~ re →
    s =~ Star re.
Proof.
  intros T s re H.
  rewrite ← (app_nil_r _ s).
  apply MStarApp.
  - apply H.
  - apply MStar0.
Qed.
```

(Note the use of app_nil_r to change the goal of the theorem to exactly the shape expected by MStarApp.)

Exercise: 3 stars, standard (exp_match_ex1) The following lemmas show that the intuition about matching given at the beginning of the chapter can be obtained from the formal inductive definition.

```
Lemma EmptySet_is_empty : ∀ T (s : list T),
```

```
  ¬ (s =~ EmptySet).
Proof.
    Admitted.

Lemma MUnion' : ∀ T (s : list T) (re1 re2 : reg_exp T),
  s =~ re1 ∨ s =~ re2 →
  s =~ Union re1 re2.
Proof.
    Admitted.
```

The next lemma is stated in terms of the `fold` function from the Poly chapter: If *ss* : **list** (**list** *T*) represents a sequence of strings *s1*, ..., *sn*, then `fold app ss []` is the result of concatenating them all together.

```
Lemma MStar' : ∀ T (ss : list (list T)) (re : reg_exp T),
  (∀ s, In s ss → s =~ re) →
  fold app ss [] =~ Star re.
Proof.
    Admitted.
    □
```

Exercise: 2 stars, standard, optional (EmptyStr_not_needed) It turns out that the EmptyStr constructor is actually not needed, since the regular expression matching the empty string can also be defined from Star and EmptySet: `Definition EmptyStr' {T:Type} := @Star T (EmptySet)`.

State and prove that this EmptyStr' definition matches exactly the same strings as the EmptyStr constructor.

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 we want to prove the following intuitive fact: If a string *s* is matched by a regular expression *re*, then all elements of *s* must occur as character literals somewhere in *re*.

To state this as a 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.
```

Now, the main theorem:

```
Theorem in_re_match : ∀ T (s : list T) (re : reg_exp T) (x : T),
  s =~ re →
  In x s →
  In x (re_chars re).
Proof.
  intros T s re x Hmatch Hin.
  induction Hmatch
    as [| x'
         | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
         | s1 re1 re2 Hmatch IH | s2 re1 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 is matched by *re1*), and a second one that applies when x occurs in *s2* (matched by *re2*).

```
    rewrite In_app_iff in *.
    destruct Hin as [Hin | Hin].
    +
      left. apply (IH1 Hin).
    +
      right. apply (IH2 Hin).
  -
    simpl. rewrite In_app_iff.
    left. apply (IH Hin).
  -
    simpl. rewrite In_app_iff.
    right. apply (IH Hin).
  -
    destruct Hin.
  -
    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 In x $s2$.

```
    rewrite In_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 : ∀ T (re : reg_exp T),
  (∃ s, s =~ re) ↔ re_not_empty re = true.
Proof.
  Admitted.
  □
```

8.5.3 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: ∀ 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.
```

Now, 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.

```
  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 MEmpty...

```
- 
  simpl. intros H. apply H.
```

... but most cases get stuck. For MChar, for instance, we must show
s2 =~ Char x' -> x'::s2 =~ Char x'
which is clearly impossible.

```
- intros H. simpl. Abort.
```

The problem here is that `induction` over a Prop hypothesis only works properly with hypotheses that are "fully general," i.e., ones in which all the arguments are just variables, as opposed to more specific expressions like Star *re*.

(In this respect, `induction` on evidence behaves more like `destruct`-without-*eqn*: than like `inversion`.)

A possible, but 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'* = Star *re* equality in the context.

This works, but it makes the statement of the lemma a bit ugly. Fortunately, there is a better way... `Abort`.

The tactic *remember* `e as` *x eqn*:Eq causes Rocq to (1) replace all occurrences of the expression `e` by the variable *x*, and (2) add an equation Eq : *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' eqn:Eq.
```

We now have Eq : *re'* = Star *re*.

```
  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 Eq is contradictory in most cases, allowing us to conclude immediately.

```
- discriminate.
- discriminate.
- discriminate.
- discriminate.
- discriminate.
```

The interesting cases are those that correspond to Star.

```
-
  intros H. apply H.
-
  intros H1. rewrite ← app_assoc.
  apply MStarApp.
  + apply Hmatch1.
  + apply IH2.
    × apply Eq.
    × apply H1.
```

Note that the induction hypothesis *IH2* on the MStarApp case mentions an additional premise Star *re''* = Star *re*, which results from the equality generated by *remember*. 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'' : ∀ T (s : list T) (re : reg_exp T),
  s =~ Star re →
  ∃ ss : list (list T),
    s = fold app ss []
    ∧ ∀ s', In s' ss → s' =~ re.
Proof.
  Admitted.
  □
```

8.5.4 The "Weak" Pumping Lemma

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, this exercise considers a slightly weaker theorem than is usually stated in courses on automata theory – hence the name weak_pumping. The stronger one can be found below.

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*, a 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. }
  rewrite H in Hp1. inversion 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: ∀ T (n m : nat) (l : list T),
  napp (n + m) l = napp n l ++ napp m l.
Proof.
  intros T n m l.
  induction n as [|n IHn].
  - reflexivity.
  - simpl. rewrite IHn, app_assoc. reflexivity.
Qed.

Lemma napp_star :
  ∀ T m s1 s2 (re : reg_exp T),
    s1 =~ re → s2 =~ Star re →
    napp m s1 ++ s2 =~ 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 $s =\tilde{} \; re$ and if the length of s is at least the pumping constant of re, then s can be split into three substrings $s1 \mathbin{++} s2 \mathbin{++} s3$ 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.

This proof is quite long, so to make it more tractable we've broken it up into a number of sub-proofs, which we then assemble to prove the main lemma.

Your job is to complete the proofs of the helper lemmas; the main lemma relies on these. Several of the lemmas about **le** that were in an optional exercise earlier in this chapter may be useful here in particular, lt_ge_cases and plus_le.

Exercise: 2 stars, standard (weak_pumping_char)

```
Lemma weak_pumping_char : ∀ (T : Type) (x : T),
  pumping_constant (Char x) ≤ length [x] →
  ∃ s1 s2 s3 : list T,
    [x] = s1 ++ s2 ++ s3 ∧
    s2 ≠ [ ] ∧
    (∀ m : nat, s1 ++ napp m s2 ++ s3 =~ Char x).
Proof.
  Admitted.
```

□

Exercise: 3 stars, standard (weak_pumping_app)

```
Lemma weak_pumping_app : ∀ (T : Type)
                           (s1 s2 : list T) (re1 re2 : reg_exp T),
  s1 =~ re1 →
  s2 =~ re2 →
  (pumping_constant re1 ≤ length s1 →
  ∃ s2 s3 s4 : list T,
    s1 = s2 ++ s3 ++ s4 ∧
    s3 ≠ [ ] ∧
    (∀ m : nat, s2 ++ napp m s3 ++ s4 =~ re1)) →
  (pumping_constant re2 ≤ length s2 →
    ∃ s1 s3 s4 : list T,
      s2 = s1 ++ s3 ++ s4 ∧
      s3 ≠ [ ] ∧
      (∀ m : nat, s1 ++ napp m s3 ++ s4 =~ re2)) →
  pumping_constant (App re1 re2) ≤ length (s1 ++ s2) →
  ∃ s0 s3 s4 : list T,
    s1 ++ s2 = s0 ++ s3 ++ s4 ∧
    s3 ≠ [ ] ∧
    (∀ m : nat, s0 ++ napp m s3 ++ s4 =~ App re1 re2).
Proof.
  simpl. intros T s1 s2 re1 re2 Hmatch1 Hmatch2 IH1 IH2 Hlen.
  assert (H : pumping_constant re1 ≤ length s1 ∨
              pumping_constant re2 ≤ length s2).
```

```
  {
    admit.
  }
  Admitted.
```
□

Exercise: 3 stars, standard (weak_pumping_union_l) `Lemma weak_pumping_union_l`

```
: ∀ T (s1 : list T) (re1 re2 : reg_exp T),
  s1 =~ re1 →
  (pumping_constant re1 ≤ length s1 →
    ∃ s2 s3 s4 : list T,
      s1 = s2 ++ s3 ++ s4 ∧
      s3 ≠ [ ] ∧
      (∀ m : nat, s2 ++ napp m s3 ++ s4 =~ re1)) →
  pumping_constant (Union re1 re2) ≤ length s1 →
  ∃ s0 s2 s3 : list T,
    s1 = s0 ++ s2 ++ s3 ∧
    s2 ≠ [ ] ∧
    (∀ m : nat, s0 ++ napp m s2 ++ s3 =~ Union re1 re2).
Proof.
  simpl. intros T s1 re1 re2 Hmatch IH Hlen.
  assert (H : pumping_constant re1 ≤ length s1).
  {
    admit.
  }
  Admitted.
```
□

```
Lemma weak_pumping_union_r : ∀ T (s2 : list T) (re1 re2 : reg_exp T),
  s2 =~ re2 →
  (pumping_constant re2 ≤ length s2 →
    ∃ s1 s3 s4 : list T,
      s2 = s1 ++ s3 ++ s4 ∧
      s3 ≠ [ ] ∧
      (∀ m : nat, s1 ++ napp m s3 ++ s4 =~ re2)) →
  pumping_constant (Union re1 re2) ≤ length s2 →
  ∃ s1 s0 s3 : list T,
    s2 = s1 ++ s0 ++ s3 ∧
    s0 ≠ [ ] ∧
    (∀ m : nat, s1 ++ napp m s0 ++ s3 =~ Union re1 re2).
Proof.
   Admitted.
```

Exercise: 2 stars, standard, optional (weak_pumping_star_zero) Lemma weak_pumping_star_zero
: ∀ T (re : reg_exp T),
  pumping_constant (Star re) ≤ @length T [] →
  ∃ s1 s2 s3 : list T,
    [ ] = s1 ++ s2 ++ s3 ∧
    s2 ≠ [ ] ∧
    (∀ m : nat, s1 ++ napp m s2 ++ s3 =~ Star re).
Proof.
  Admitted.
□

Exercise: 4 stars, standard, optional (weak_pumping_star_app) (You may also want the plus_le_cases lemma here.)

```
Lemma weak_pumping_star_app : ∀ T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ re →
  s2 =~ Star re →
  (pumping_constant re ≤ length s1 →
    ∃ s2 s3 s4 : list T,
      s1 = s2 ++ s3 ++ s4
      ∧ s3 ≠ [ ] ∧
      (∀ m : nat, s2 ++ napp m s3 ++ s4 =~ re)) →
  (pumping_constant (Star re) ≤ length s2 →
    ∃ s1 s3 s4 : list T,
      s2 = s1 ++ s3 ++ s4 ∧
      s3 ≠ [ ] ∧
      (∀ m : nat, s1 ++ napp m s3 ++ s4 =~ Star re)) →
  pumping_constant (Star re) ≤ length (s1 ++ s2) →
  ∃ s0 s3 s4 : list T,
    s1 ++ s2 = s0 ++ s3 ++ s4 ∧
    s3 ≠ [ ] ∧
    (∀ m : nat, s0 ++ napp m s3 ++ s4 =~ Star re).
Proof.
  simpl. intros T s1 s2 re Hmatch1 Hmatch2 IH1 IH2 Hlen.
  rewrite app_length in *.
  assert (Hs1re1 : length s1 = 0
                ∨ (length s1 ≠ 0 ∧ length s1 < pumping_constant re)
                ∨ pumping_constant re ≤ length s1).
  {
    induction s1 as [| h s1' IHs1].
    - admit.
    - admit.
  }
```

```
  Admitted.
```
□

```
Lemma weak_pumping : ∀ T (re : reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
  ∃ s1 s2 s3,
    s = s1 ++ s2 ++ s3 ∧
    s2 ≠ [] ∧
    ∀ m, s1 ++ napp m s2 ++ s3 =~ re.
Proof.
  intros T re s Hmatch.
  induction Hmatch
    as [ | x | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
       | s1 re1 re2 Hmatch IH | s2 re1 re2 Hmatch IH
       | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2 ].
  -
    simpl. intros contra. inversion contra.
  - apply weak_pumping_char.
  - apply weak_pumping_app; assumption.
  - apply weak_pumping_union_l; assumption.
  - apply weak_pumping_union_r; assumption.
  - apply weak_pumping_star_zero.
  - apply weak_pumping_star_app; assumption.
Qed.
```

8.5.5 The (Strong) Pumping Lemma

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 strengthens the result to include the claim that length $s1$ + length $s2 \leq$ pumping_constant re.

```
Lemma pumping : ∀ T (re : reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
  ∃ s1 s2 s3,
    s = s1 ++ s2 ++ s3 ∧
    s2 ≠ [] ∧
    length s1 + length s2 ≤ pumping_constant re ∧
    ∀ m, s1 ++ napp m s2 ++ s3 =~ 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 | s2 re1 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 sometimes need to relate boolean computations to statements in `Prop`. But performing this conversion as we did there can result in tedious proof scripts. Consider the proof of the following theorem:

```
Theorem filter_not_empty_In : ∀ n l,
  filter (fun x ⇒ n =? x) l ≠ [] → 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 $n =? m = \mathsf{true}$ into the assumption $n = m$; then we had to `rewrite` using this assumption to complete the case.

We can streamline this sort of reasoning by defining an inductive proposition that yields a better case-analysis principle for $n =? m$. Instead of generating the assumption $(n =? m) = \mathsf{true}$, which usually requires some massaging before we can use it, this principle gives us right away the assumption we really need: $n = m$.

Following the terminology introduced in Logic, we call this the "reflection principle for equality on numbers," and we say that the boolean $n =? m$ is *reflected in* the proposition $n = m$.

```
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. It states that the property P *reflects* (intuitively, is equivalent to) the boolean b: that is, P holds if and only if b = 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 we can extract evidence for P from a proof of reflect P true.

Similarly, the only way to show reflect P false is by tagging evidence for $\neg P$ with the ReflectF constructor.

To put this observation to work, we first prove that the statements $P \leftrightarrow$ b = 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 eqn:Eb.
  - 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 : ∀ P b, reflect P b → (P ↔ b = true).`

```
Proof.
```

Admitted.

□

We can think of reflect as a variant of the usual "if and only if" connective; the advantage of reflect 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).

Let's use reflect to produce a smoother proof of filter_not_empty_In.

We begin by recasting the eqb_eq lemma in terms of reflect:

```
Lemma eqbP : ∀ n m, reflect (n = m) (n =? m).
Proof.
  intros n m. apply iff_reflect. rewrite eqb_eq. reflexivity.
Qed.
```

The proof of filter_not_empty_In now goes as follows. Notice how the calls to `destruct` and `rewrite` in the earlier proof of this theorem are combined here into a single call to `destruct`.

(To see this clearly, execute the two proofs of filter_not_empty_In with Rocq and observe the differences in proof state at the beginning of the first case of the `destruct`.)

```
Theorem filter_not_empty_In' : ∀ n l,
  filter (fun x ⇒ n =? x) l ≠ [] →
  In n l.
```

```
Proof.
  intros n l. induction l as [|m l' IHl'].
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (eqbP n m) as [EQnm | NEQnm].
    +
      intros _. rewrite EQnm. 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
  | [] ⇒ 0
  | m :: l' ⇒ (if n =? m then 1 else 0) + count n l'
  end.

Theorem eqbP_practice : ∀ n l,
  count n l = 0 → ~(In n l).
Proof.
  intros n l Hcount. induction l as [| m l' IHl'].
   Admitted.
```

□

This small example shows reflection giving 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*.

This way of using reflect was popularized by *SSReflect*, a Rocq library that has been used to formalize important results in mathematics, including 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 streamline small proof steps by turning them into 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.

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] has two occurrences of 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→**bool**, and a list *l* of type **list** X. Suppose further that *l* is an in-order merge of two lists, *l1* and *l2*, such that every item in *l1* satisfies *test* and no item in *l2* satisfies test. Then filter *test l* = *l1*.

First define what it means for one list to be a merge of two others. Do this with an inductive relation, not a `Fixpoint`.

```
Inductive merge {X:Type} : list X → list X → list X → Prop :=
```

.

```
Theorem merge_filter : ∀ (X : Set) (test: X→bool) (l l1 l2 : list X),
  merge l1 l2 l →
  All (fun n ⇒ test n = true) l1 →
  All (fun n ⇒ test n = false) l2 →
  filter test l = l1.
Proof.
    Admitted.
```

□

Exercise: 5 stars, advanced, optional (filter_challenge_2) A different way to characterize the behavior of filter goes like this: Among all subsequences of l with the property that *test* evaluates to true on all their members, filter *test* l is the longest. Formalize this claim and prove it.

Exercise: 4 stars, standard, optional (palindromes) A palindrome is a sequence that reads the same backwards as forwards.

- Define an inductive proposition **pal** on **list** X that captures what it means to be a palindrome. (Hint: You'll need three cases.
- Prove (pal_app_rev) that

  forall l, pal (l ++ rev l).
- Prove (pal_rev that)

  forall l, pal l -> l = rev l.

For extra credit, try proving the same theorems with an alternate definition with a *single* constructor of this type:

forall l, l = rev l -> pal l

```
Inductive pal {X:Type} : list X → Prop :=

.

Theorem pal_app_rev : ∀ (X:Type) (l : list X),
  pal (l ++ (rev l)).
Proof.
    Admitted.

Theorem pal_rev : ∀ (X:Type) (l: list X) , pal l → l = rev l.
Proof.
    Admitted.
```

□

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.

```
Theorem palindrome_converse: ∀ {X: Type} (l: list X),
  l = rev l → pal l.
Proof.
  Admitted.
  □
```

Exercise: 4 stars, advanced, optional (NoDup) Recall the definition of the In property from the Logic chapter, which asserts that a value *x* appears at least once in a list *l*:

```
Module RecallIn.
  Fixpoint In (A : Type) (x : A) (l : list A) : Prop :=
    match l with
    | [] ⇒ False
    | x' :: l' ⇒ x' = x ∨ In A x l'
    end.
End RecallIn.
```

Your first task is to use In to define a proposition *disjoint* X *l1* *l2*, which should be provable exactly when *l1* and *l2* are lists (with elements of type X) that have no elements in common.

Next, use In 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: 5 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 and useful lemma.

```
Lemma in_split : ∀ (X:Type) (x:X) (l:list X),
  In x l →
```

```
  ∃ l1 l2 , l = l1 ++ x :: l2.
Proof.
  Admitted.
```

Now define a property **repeats** such that **repeats** X *l* asserts that *l* contains at least one repeated element (of type X).

```
Inductive repeats {X:Type} : list X → Prop :=


.

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 In is decidable, i.e., ∀ *x l*, (In *x l*) ∨ ¬ (In *x l*). However, it is also possible to make the proof go through *without* assuming that In is decidable; if you manage to do this, you will not need the excluded_middle hypothesis.

```
Theorem pigeonhole_principle: excluded_middle →
  ∀ (X:Type) (l1 l2:list X),
  (∀ x, In x l1 → In x l2) →
  length l2 < length l1 →
  repeats l1.
Proof.
  intros EM X l1. induction l1 as [|x l1' 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 Rocq. As a result, Rocq 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: `From Stdlib Require Import Strings.Ascii.`

```
Definition string := list ascii.
```

The Rocq standard library contains a distinct inductive definition of strings of ASCII characters. However, we will use the above definition of strings as lists as 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 : ∀ (P : Prop), P → (P ↔ True).`

```
Proof.
  intros.
  split.
  - intros. constructor.
  - intros _. apply H.
Qed.
```

Each `Prop` whose negation is provable is equivalent to **False**. `Lemma not_equiv_false : ∀ (P : Prop), ¬P → (P ↔ False).`

```
Proof.
  intros.
  split.
  - apply H.
  - intros. destruct H0.
Qed.
```

EmptySet matches no string. `Lemma null_matches_none : ∀ (s : string), (s =~ EmptySet) ↔ False.`

```
Proof.
  intros.
  apply not_equiv_false.
  unfold not. intros. inversion H.
Qed.
```

EmptyStr only matches the empty string. `Lemma empty_matches_eps : ∀ (s : string), s =~ EmptyStr ↔ s = [ ].`

```
Proof.
  split.
  - intros. inversion H. reflexivity.
  - intros. rewrite H. apply MEmpty.
Qed.
```

EmptyStr matches no non-empty string. `Lemma empty_nomatch_ne : ∀ (a : ascii) s, (a :: s =~ EmptyStr) ↔ 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 :`

```
  ∀ (a b : ascii) s, b ≠ a → (b :: s =~ Char a ↔ 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 : ∀ (a : ascii) s, a :: s =~ Char a ↔ 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 : ∀ (s : string) re0 re1,`

```
  s =~ App re0 re1 ↔
  ∃ s0 s1, s = s0 ++ s1 ∧ s0 =~ re0 ∧ s1 =~ re1.
Proof.
  intros.
  split.
  - intros. inversion H. ∃ s1, s2. split.
    × reflexivity.
    × 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 : ∀ (a : ascii) s re0 re1,
  a :: s =~ (App re0 re1) ↔
  ([ ] =~ re0 ∧ a :: s =~ re1) ∨
  ∃ s0 s1, s = s0 ++ s1 ∧ a :: s0 =~ re0 ∧ s1 =~ re1.
Proof.
  Admitted.
```

□

s is matched by Union *re0 re1* iff *s* matched by *re0* or *s* matched by *re1*.

```
Lemma union_disj : ∀ (s : string) re0 re1,
  s =~ Union re0 re1 ↔ s =~ re0 ∨ s =~ 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* is matched by Star *re* iff *s* = *s0* ++ *s1*, where a::*s0* is matched by *re* and *s1* is matched by 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* =~ 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* =~ Star *re* to be a Prop over general variables, using the *remember* tactic.

```
Lemma star_ne : ∀ (a : ascii) s re,
  a :: s =~ Star re ↔
  ∃ s0 s1, s = s0 ++ s1 ∧ a :: s0 =~ re ∧ s1 =~ 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 :=`
`∀ re : reg_exp ascii, reflect ([ ] =~ 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: reg_exp 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' :=
  ∀ s, a :: s =~ re ↔ s =~ 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 := ∀ 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" =~ EmptySet: `Example test_der0 : match_eps (derive c (EmptySet)) = false.`
`Proof.`
`Admitted.`

"c" =~ Char c: `Example test_der1 : match_eps (derive c (Char c)) = true.`
`Proof.`
`Admitted.`

"c" =~ Char d: `Example test_der2 : match_eps (derive c (Char d)) = false.`
`Proof.`
`Admitted.`

"c" =~ 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* =~ Union *re0 re1*) to a Boolean combination of `Prop`'s over simple regex's (e.g., *s* =~ *re0* ∨ *s* =~ *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 *re* matches *s*. I.e., *m* holds the following property: Definition matches_regex *m* : Prop :=
  ∀ (*s* : string) *re*, **reflect** (*s* =~ *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_match_correct) 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* =~ *re* given *s* =~ derive *x re*, and vice versa. Theorem regex_match_correct : 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 in ordinary programming and in the theory of programming languages; 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 instead return an **option** to indicate success or failure. Partial maps are defined in terms of total maps, using None as the default element.

9.2 The 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 (or, 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 Rocq's standard library. 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 Stdlib Require Import Arith.
From Stdlib Require Import Bool.
From Stdlib Require Export Strings.String.
From Stdlib Require Import FunctionalExtensionality.
From Stdlib Require Import List.
Import ListNotations.
```

Documentation for the standard library can be found at https://rocq-prover.org/doc/V9.0.0/stdlib/inde

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 of the + notation, but only one for naturals.)

```
Print Init.Nat.add.
```

We'll see some more uses of `Locate` in the Imp chapter.

9.3 Identifiers

To define maps, we first need a type for the keys that we will 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 Rocq's standard library.

To compare strings, we use the function eqb_refl from the String module in the standard library.

```
Check String.eqb_refl :
  ∀ x : string, (x =? x)%string = true.
```

We will often use a few basic properties of string equality... `Check String.eqb_eq :`

```
  ∀ n m : string, (n =? m)%string = true ↔ n = m.
Check String.eqb_neq :
  ∀ n m : string, (n =? m)%string = false ↔ n ≠ m.
Check String.eqb_spec :
  ∀ x y : string, reflect (x = y) (String.eqb x y).
```

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: two maps that respond to queries in the same way will be represented as exactly the same function, rather than just as "equivalent" list structures. This simplifies proofs that use maps.

We build up to 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) := string → A.
```

Intuitively, a total map over an element type A is just a function that can be used to look up strings, yielding As.

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 :=
  (fun _ ⇒ v).
```

More interesting is the map-updating function, which (as always) 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. The novelty here is that we achieve this effect by wrapping a new function around the old one.

```
Definition t_update {A : Type} (m : total_map A)
                    (x : string) (v : A) :=
  fun x' ⇒ if String.eqb 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 notations to facilitate working with maps.

First, we use the following notation to represent an empty total map with a default value.

```
Notation "'__' '!->' v" := (t_empty v)
  (at level 100, right associativity).

Example example_empty := ( false).
```

We next introduce a symbolic notation for extending an existing map with a new binding.

```
Notation "x '!->' v ';' m" := (t_update m x v)
                                (at level 100, v constr at level 100, 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 on this representation of maps 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.
```

When we 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 was 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 : ∀ (A : Type) (x : string) (v : A),
  (_ !-> v) 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 : ∀ (A : Type) (m : total_map A) x v,
  (x !-> v ; m) 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 : ∀ (A : Type) (m : total_map A) x1 x2 v,
  x1 ≠ x2 →
  (x1 !-> v ; m) x2 = m 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 : ∀ (A : Type) (m : total_map A) x v1 v2,
  (x !-> v2 ; x !-> v1 ; m) = (x !-> v2 ; m).
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard (t_update_same) Given strings $x1$ and $x2$, we can use the tactic `destruct (eqb_spec x1 x2)` to simultaneously perform case analysis on the result of *String.eqb x1 x2* and generate hypotheses about the equality (in the sense of $=$) of $x1$ and $x2$. With the example in chapter IndProp as a template, use *String.eqb_spec* 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 : ∀ (A : Type) (m : total_map A) x,
  (x !-> m x ; m) = m.
Proof.
  Admitted.
```
□

Exercise: 3 stars, standard, especially useful (t_update_permute) Similarly, use *String.eqb_spec* 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 : ∀ (A : Type) (m : total_map A)
                             v1 v2 x1 x2,
  x2 ≠ x1 →
  (x1 !-> v1 ; x2 !-> v2 ; m)
  =
  (x2 !-> v2 ; x1 !-> v1 ; m).
Proof.
  Admitted.
```
□

9.5 Partial maps

Lastly, 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 0, x constr, v at level 200, right associativity).`

We can also hide the last case when it is empty. `Notation "x '|->' v" := (update empty x v)`
`  (at level 0, x constr, v at level 200).`

```
Definition examplepmap :=
  ("Church" |-> true ; "Turing" |-> false).
```

We now straightforwardly lift all of the basic lemmas about total maps to partial maps.

```
Lemma apply_empty : ∀ (A : Type) (x : string),
  @empty A x = None.
Proof.
  intros. unfold empty. rewrite t_apply_empty.
  reflexivity.
Qed.

Lemma update_eq : ∀ (A : Type) (m : partial_map A) x v,
  (x |-> v ; m) x = Some v.
Proof.
  intros. unfold update. rewrite t_update_eq.
  reflexivity.
Qed.
```

The update_eq lemma is used very often in proofs. Adding it to Rocq's global "hint database" allows proof-automation tactics such as `auto` to find it. `#[global] Hint Resolve update_eq : core.`

```
Theorem update_neq : ∀ (A : Type) (m : partial_map A) x1 x2 v,
  x2 ≠ x1 →
  (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 : ∀ (A : Type) (m : 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 : ∀ (A : Type) (m : partial_map A) x v,
  m x = Some v →
  (x |-> v ; m) = m.
Proof.
  intros A m x v H. unfold update. rewrite ← H.
  apply t_update_same.
Qed.

Theorem update_permute : ∀ (A : Type) (m : partial_map A)
                                    x1 x2 v1 v2,
  x2 ≠ x1 →
  (x1 |-> v1 ; x2 |-> v2 ; m) = (x2 |-> v2 ; x1 |-> v1 ; m).
Proof.
  intros A m x1 x2 v1 v2. unfold update.
  apply t_update_permute.
Qed.
```

One last thing: For partial maps, it's convenient to introduce a notion of map inclusion, stating that all the entries in one map are also present in another:

```
Definition includedin {A : Type} (m m' : partial_map A) :=
  ∀ x v, m x = Some v → m' x = Some v.
```

We can then show that map update preserves map inclusion, that is:

```
Lemma includedin_update : ∀ (A : Type) (m m' : partial_map A)
                                    (x : string) (vx : A),
  includedin m m' →
  includedin (x |-> vx ; m) (x |-> vx ; m').
Proof.
  unfold includedin.
  intros A m m' x vx H.
  intros y vy.
  destruct (eqb_spec x y) as [Hxy | 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 quite useful for reasoning about languages with variable binding – e.g., the Simply Typed Lambda Calculus, which we will see in *Programming Language Foundations*, where maps are used to keep track of which program variables are defined in a given scope.

Chapter 10

Library LF.ProofObjects

10.1 ProofObjects: The Curry-Howard Correspondence

```
Set Warnings "-notation-overridden,-notation-incompatible-prefix".
From LF Require Export IndProp.
```

"Algorithms are the computational content of proofs." (Robert Harper)

We have seen that Rocq 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 Rocq's programming and proving facilities are closely related. For example, the keyword `Inductive` is used to declare both data types and propositions, and → 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 Rocq are almost the same thing. In this chapter we will study this connection in more detail.

We have already seen the fundamental idea: provability in Rocq is always witnessed by *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 concrete data structure.

If the proposition is an implication like A → B, then its proof is 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.

```
Inductive ev : nat → Prop :=
  | ev_0 : ev 0
  | ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

We can pronounce the ":" here as either "has type" or "is a proof of." For example, the

second line in the definition of **ev** declares that ev_0 : **ev** 0. Instead of "ev_0 has type **ev** 0," we can say that "ev_0 is a proof of **ev** 0."

This pun between types and propositions – between : as "has type" and : as "is a proof of" or "is evidence for" – is called the *Curry-Howard correspondence*. It proposes a deep connection between the world of logic and the world of computation:

propositions ~ types proofs ~ programs

See *Wadler* 2015 (in Bib.v) for a brief history and modern 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 an earlier proof involving **ev**.

```
Theorem ev_4 : ev 4.
Proof.
  apply ev_SS. apply ev_SS. apply ev_0. Qed.
```

Just 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, with no need for a proof script at all:

```
Check (ev_SS 2 (ev_SS 0 ev_0))
  : ev 4.
```

The expression ev_SS 2 (ev_SS 0 ev_0) instantiates 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 ∀ X, **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 Rocq operates. When Rocq 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, Rocq 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 convenient, but they are not essential in Rocq: in principle, we can always just construct the required evidence by hand. Then we can use `Definition` (rather than `Theorem`) to introduce a global name for 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 Rocq's computational universe (where data structures and programs live), there are two sorts of values that have arrows in their types: *constructors* introduced by `Inductive`ly defined data types, and *functions*.

Similarly, in Rocq's logical universe (where we carry out proofs), there are two ways of giving evidence for an implication: constructors introduced by `Inductive`ly defined propositions, and... functions!

For example, consider this statement:

```
Theorem ev_plus4 : ∀ n, ev n → 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 ∀ n, ev n → 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' : ∀ n, ev n → 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 Rocq 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'' : ∀ n : nat, ev n → 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 most mainstream programming languages, they can be quite useful in programming too, as the flurry of activity in the functional programming community over the past couple of decades 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 :=
  ∀ n, ∀ (E : ev n), 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 :=
  ∀ n, ∀ (_ : ev n), ev (n + 2).
```

Or, equivalently, we can write it in a more familiar way:

```
Definition ev_plus2'' : Prop :=
  ∀ n, ev n → 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 wonder whether we can build *programs* using tactics rather than by writing down explicit terms.

Naturally, the answer is yes!

```
Definition add2 : nat → nat.
intros n.
Show Proof.
apply S.
Show Proof.
apply S.
Show Proof.
apply n. Defined.

Print add2.

Compute add2 2.
```

Notice that we terminated the `Definition` with a . rather than with := followed by a term. This tells Rocq to enter *proof scripting mode* to build an object of type **nat** → **nat**. Also, we terminate the proof with `Defined` rather than `Qed`; this makes the definition *transparent* so that it can be used in computation like a normally-defined function. (`Qed`-defined objects are opaque during computation.)

This feature is mainly useful for writing functions with dependent types, which we won't explore much further in this book. But it does illustrate the uniformity and orthogonality of the basic ideas in Rocq.

10.5 Logical Connectives as Inductive Types

Inductive definitions are powerful enough to express most of the logical connectives we have seen so far. Indeed, only universal quantification (with implication as a special case) is built into Rocq; all the others are defined inductively.

Let's see how.

```
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 $P \wedge Q$ to consist 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 proj1'' P Q (HPQ : P ∧ Q) : P :=
  match HPQ with
  | conj HP HQ ⇒ HP
  end.

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 disjunction:

```
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 : ∀ (P Q : Prop), P → P ∨ Q :=
  fun P Q HP ⇒ or_introl HP.

Theorem inj_l' : ∀ (P Q : Prop), P → P ∨ Q.
Proof.
  intros P Q HP. left. apply HP.
  Show Proof.
Qed.

Definition or_elim : ∀ (P Q R : Prop), (P ∨ Q) → (P → R) → (Q → R) → R :=
  fun P Q R HPQ HPR HQR ⇒
    match HPQ with
    | or_introl HP ⇒ HPR HP
    | or_intror HQ ⇒ HQR HQ
    end.

Theorem or_elim' : ∀ (P Q R : Prop), (P ∨ Q) → (P → R) → (Q → R) → 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' : ∀ P Q, P ∨ Q → Q ∨ 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 → Prop) : Prop :=
  | ex_intro : ∀ x : A, P x → ex P.

Notation "'exists' x , p" :=
  (ex (fun x ⇒ p))
    (at level 200, right associativity) : type_scope.

End Ex.
```

This probably needs 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 extension of the above, enabling syntactic forms such as $\exists\ x\ y,\ P\ x\ y$.

The more familiar form $\exists\ x,$ **ev** 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 : ∃ n, 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.
```

□

To destruct existentials in a proof term we simply use match: `Definition dist_exists_or_term (X:Type) (P Q : X → Prop) : (∃ x, P x ∨ Q x) → (∃ x, P x) ∨ (∃ x, Q x) := fun H ⇒ match H with | ex_intro _ x Hx ⇒ match Hx with | or_introl HPx ⇒ or_introl (ex_intro _ x HPx) | or_intror HQx ⇒ or_intror (ex_intro _ x HQx) end end.`

Exercise: 2 stars, standard (ex_match) Construct a proof object for the following proposition: `Definition ex_match : ∀ (A : Type) (P Q : A → Prop), (∀ x, P x → Q x) → (∃ x, P x) → (∃ x, Q x) . 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 : ∀ P, P → 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.

```
Fail
  Definition contra : False :=
  42.
```

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 $0 = 1$. Fortunately, 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 Rocq's equality relation is not built in. We can define it ourselves:

```
Module EqualityPlayground.

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 Rocq 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 `match`es.

```
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].
```

We can also pattern-match on an equality proof: `Definition eq_add : ∀ (n1 n2 : nat), n1 == n2 → (S n1) == (S n2) :=`

```
  fun n1 n2 Heq ⇒
    match Heq with
    | eq_refl n ⇒ eq_refl (S n)
    end.
```

By pattern-matching against $n1 == n2$, we obtain a term n that replaces $n1$ and $n2$ in the type we have to produce, so instead of $(S\ n1) == (S\ n2)$, we now have to produce

something of type (S n) == (S n), which we establish by eq_refl (S n).

A tactic-based proof runs into some difficulties if we try to use our usual repertoire of tactics, such as rewrite and reflexivity. Those work with *setoid* relations that Rocq knows about, such as =, but not our ==. We could prove to Rocq that == is a setoid, but a simpler way is to use destruct and apply instead.

```
Theorem eq_add' : ∀ (n1 n2 : nat), n1 == n2 → (S n1) == (S n2).
Proof.
  intros n1 n2 Heq.
  Fail rewrite Heq.   destruct Heq as [n].   Fail reflexivity.   apply eq_refl.
Qed.
```

Exercise: 2 stars, standard (eq_cons) Construct the proof object for the following theorem. Use pattern matching on the equality hypotheses.

```
Definition eq_cons : ∀ (X : Type) (h1 h2 : X) (t1 t2 : list X),
    h1 == h2 → t1 == t2 → h1 :: t1 == h2 :: t2
  . Admitted.
```

□

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. Prove that.

```
Lemma equality__leibniz_equality : ∀ (X : Type) (x y: X),
  x == y → ∀ (P : X → Prop), P x → P y.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard (equality__leibniz_equality_term) Construct the proof object for the previous exercise. All it requires is anonymous functions and pattern-matching; the large proof term constructed by tactics in the previous exercise is needessly complicated. Hint: pattern-match as soon as possible. Definition equality__leibniz_equality_term : ∀ (X : Type) (x y: X),

```
    x == y → ∀ P : (X → Prop), P x → P y
  . 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 EqualityPlayground.
```

10.6.1 Inversion, Again

We've seen `inversion` used with both equality hypotheses and hypotheses about inductively defined propositions. Now that we've seen that these are actually the same thing, we're in a position to take a closer look at how `inversion` behaves.

In general, the `inversion` tactic...

- takes a hypothesis H whose type P is inductively defined, and
- for each constructor C in P's definition,
  - generates a new subgoal in which we assume H was built with C,
  - adds the arguments (premises) of C to the context of the subgoal as extra hypotheses,
  - matches the conclusion (result type) of C against the current goal and calculates a set of equalities that must hold in order for C to be applicable,
  - adds these equalities to the context (and, for convenience, rewrites them in the goal), and
  - if the equalities are not satisfiable (e.g., they involve things like S n = O), immediately solves the subgoal.

Example: If we invert a hypothesis built with **or**, there are two constructors, so two subgoals get generated. The conclusion (result type) of the constructor ($P \vee Q$) doesn't place any restrictions on the form of P or Q, so we don't get any extra equalities in the context of the subgoal.

Example: If we invert a hypothesis built with **and**, there is only one constructor, so only one subgoal gets generated. Again, the conclusion (result type) of the constructor ($P \wedge Q$) doesn't place any restrictions on the form of P or Q, so we don't get any extra equalities in the context of the subgoal. The constructor does have two arguments, though, and these can be seen in the context in the subgoal.

Example: If we invert a hypothesis built with **eq**, there is again only one constructor, so only one subgoal gets generated. Now, though, the form of the 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 Rocq's Trusted Computing Base

One question that arises with any automated proof assistant is "why should we trust it?" – i.e., 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 Rocq 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 Rocq – 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 Rocq 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 : ∀ P Q, P ∨ Q → P :=
  fun (P Q : Prop) (A : P ∨ Q) ⇒
    match A with
    | or_introl H ⇒ H
    end.
```

All the types here match correctly, but the `match` only considers one of the possible constructors for **or**. Rocq'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 Rocq that it recurses on argument n, not X.) Were Rocq to allow infinite_loop, then falso would be definable, thus giving evidence for **False**. So Rocq rejects infinite_loop.

Note that the soundness of Rocq depends only on the correctness of this typechecking engine, not on the tactic machinery. If there is a bug in a tactic implementation (which does happen occasionally), that tactic might construct an invalid proof term. But when you type `Qed`, Rocq checks the term for validity from scratch. Only theorems whose proofs pass the

type-checker can be used in further proof developments.

10.8 More Exercises

Most of the following theorems were already proved with tactics in Logic. Now construct the proof objects for them directly.

Exercise: 2 stars, standard (and_assoc) `Definition and_assoc : ∀ P Q R : Prop,`
`P ∧ (Q ∧ R) → (P ∧ Q) ∧ R`
`. Admitted.`
□

Exercise: 3 stars, standard (or_distributes_over_and) `Definition or_distributes_over_and : ∀ P Q R : Prop,`
`P ∨ (Q ∧ R) ↔ (P ∨ Q) ∧ (P ∨ R)`
`. Admitted.`
□

Exercise: 3 stars, standard (negations) `Definition double_neg : ∀ P : Prop,`
`P → ~~P`
`. Admitted.`

`Definition contradiction_implies_anything : ∀ P Q : Prop,`
`(P ∧ ¬P) → Q`
`. Admitted.`

`Definition de_morgan_not_or : ∀ P Q : Prop,`
`¬ (P ∨ Q) → ¬P ∧ ¬Q`
`. Admitted.`
□

Exercise: 2 stars, standard (currying) `Definition curry : ∀ P Q R : Prop,`
`((P ∧ Q) → R) → (P → (Q → R))`
`. Admitted.`

`Definition uncurry : ∀ P Q R : Prop,`
`(P → (Q → R)) → ((P ∧ Q) → R)`
`. Admitted.`
□

10.9 Proof Irrelevance (Advanced)

In Logic we saw that functional extensionality could be added to Rocq. A similar notion about propositions can also be defined (and added as an axiom, if desired):

```
Definition propositional_extensionality : Prop :=
  ∀ (P Q : Prop), (P ↔ Q) → P = Q.
```

Propositional extensionality asserts that if two propositions are equivalent – i.e., each implies the other – then they are in fact equal. The *proof objects* for the propositions might be syntactically different terms. But propositional extensionality overlooks that, just as functional extensionality overlooks the syntactic differences between functions.

Exercise: 1 star, advanced (pe_implies_or_eq) Prove the following consequence of propositional extensionality.

```
Theorem pe_implies_or_eq :
  propositional_extensionality →
  ∀ (P Q : Prop), (P ∨ Q) = (Q ∨ P).
Proof.
  Admitted.
```

□

Exercise: 1 star, advanced (pe_implies_true_eq) Prove that if a proposition P is provable, then it is equal to **True** – as a consequence of propositional extensionality.

```
Lemma pe_implies_true_eq :
  propositional_extensionality →
  ∀ (P : Prop), P → True = P.
Proof. Admitted.
```

□

Exercise: 3 stars, advanced (pe_implies_pi) (Acknowledgment: this theorem and its proof technique are inspired by Gert Smolka's manuscript Modeling and Proving in Computational Type Theory Using the Coq Proof Assistant, 2021.

Another, perhaps surprising, consequence of propositional extensionality is that it implies *proof irrelevance*, which asserts that all proof objects for a proposition are equal.

```
Definition proof_irrelevance : Prop :=
  ∀ (P : Prop) (pf1 pf2 : P), pf1 = pf2.
```

Prove that fact. Use pe_implies_true_eq to establish that the proposition P in proof_irrelevance is equal to **True**. Leverage that equality to establish that both proof objects *pf1* and *pf2* must be just I.

```
Theorem pe_implies_pi :
  propositional_extensionality → proof_irrelevance.
```

Proof. *Admitted*.
□

Chapter 11

Library LF.IndPrinciples

11.1 IndPrinciples: Induction Principles

Every time we declare a new `Inductive` datatype, Rocq 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.

```
Set Warnings "-notation-overridden".
From LF Require Export ProofObjects.
```

11.2 Basics

Here is the induction principle 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 mul_0_r' : ∀ n:nat,
```

```
  n × 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' : ∀ n:nat,
  n + 1 = S n.
Proof.
    Admitted.
    □
```

Rocq 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, Rocq 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 Rocq will generate for the following datatype. Write down your answer on paper or type it into a comment, and then compare it with what Rocq 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 $x1$:$a1$ ... xn:an, that case is:

  "For all x1:a1 ... xn:an, if P holds of each of the arguments of type t, then P holds of c $x1$ ... xn"

  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 nsnoc 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: 2 stars, standard (booltree_ind) Here is a type for trees that contain a boolean value at each leaf and branch.

```
Inductive booltree : Type :=
  | bt_empty
  | bt_leaf (b : bool)
  | bt_branch (b : bool) (t1 t2 : booltree).

Definition booltree_property_type : Type := booltree → Prop.

Definition base_case (P : booltree_property_type) : Prop
  . Admitted.

Definition leaf_case (P : booltree_property_type) : Prop
  . Admitted.

Definition branch_case (P : booltree_property_type) : Prop
  . Admitted.

Definition booltree_ind_type :=
  ∀ (P : booltree_property_type),
    base_case P →
    leaf_case P →
    branch_case P →
    ∀ (b : booltree), P b.
```

Now check the correctness of your answers by proving the following theorem. If you have them right, you can complete the proof with just one tactic: exact booltree_ind. That will work because the automatically generated induction principle booltree_ind has the same type as what you just defined.

```
Theorem booltree_ind_type_correct : booltree_ind_type.
Proof. Admitted.
```

□

Exercise: 2 stars, 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 Rocq generates is that given above:

```
Inductive Toy : Type :=
```

.

Show that your definition is correct by proving the following theorem. You should be able to instantiate f and g with your two constructors, then immediately finish the proof with `exact Toy_ind`. As in the previous exercise, that will work because the automatically generated induction principle Toy_ind will have the same type.

```
Theorem Toy_correct : ∃ f g,
  ∀ P : Toy → Prop,
    (∀ b : bool, P (f b)) →
    (∀ (n : nat) (t : Toy), P t → P (g n t)) →
    ∀ t : Toy, P t.
Proof. Admitted.
```

□

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 □ -> (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 Rocq will generate for the following datatype. Compare your answer with what Rocq prints.

```
Inductive tree (X:Type) : Type :=
  | leaf (x : X)
  | node (t1 t2 : tree X).
```

```
Check tree_ind.
```
□

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 □

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 □

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 Rocq generate for foo'? (Fill in the blanks, then check your answer with Rocq.)

foo'_ind : forall (X : Type) (P : foo' X -> Prop), (forall (l : list X) (f : foo' X), ---------- -> ----------) -> ---------- -> forall f : foo' X, ----------

□

11.4 Induction Hypotheses

Where does the phrase "induction hypothesis" fit into this story?

The induction principle for numbers

forall P : nat -> Prop, P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n

is a generic statement that holds for all propositions P (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 **nat** → Prop.

We can make proofs by induction more explicit by giving this expression a name. For example, instead of stating the theorem mul_0_r as "∀ n, $n \times 0 = 0$," we can write it as "∀ n, P_m0r n", where P_m0r is defined as...

```
Definition P_m0r (n:nat) : Prop :=
  n × 0 = 0.
```

... or equivalently:

```
Definition P_m0r' : nat → Prop :=
  fun n ⇒ n × 0 = 0.
```

Now it is easier to see where P_m0r appears in the proof.

```
Theorem mul_0_r'' : ∀ n:nat,
  P_m0r n.
Proof.
  apply nat_ind.
  - reflexivity.
  -

    intros n IHn.
    unfold P_m0r in IHn. unfold P_m0r. simpl. apply IHn. Qed.
```

This extra naming step isn't something that we do in normal proofs, but it is useful to do it explicitly for an example or two, because it allows us to see exactly what the induction hypothesis is. If we prove ∀ *n*, P_m0r *n* by induction on *n* (using either `induction` or `apply nat_ind`), we see that the first subgoal requires us to prove P_m0r 0 ("*P* holds for zero"), while the second subgoal requires us to prove ∀ *n'*, P_m0r *n'* → P_m0r (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* O holds
  - show that, if *P n'* holds, then so does *P* (S *n'*)
  - conclude that *P n* holds for all n.

So, when we begin a proof with `intros n` and then `induction n`, we are first telling Rocq 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 Rocq actually does in this situation, internally, is it "re-generalizes" the variable we perform induction on. For example, in our original proof that plus is associative...

```
Theorem add_assoc' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p.
  induction n as [| n'].
  - reflexivity.
  -
    simpl. rewrite → IHn'. reflexivity. Qed.
```

It also works to apply `induction` to a variable that is quantified in the goal.

```
Theorem add_comm' : ∀ n m : nat,
  n + m = m + n.
Proof.
  induction n as [| n'].
  - intros m. rewrite → add_0_r. 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 add_comm'' : ∀ n m : nat,
  n + m = m + n.
Proof.
  induction m as [| m'].    - simpl. rewrite → add_0_r. reflexivity.
  - simpl. rewrite ← IHm'.
    rewrite ← plus_n_Sm. reflexivity. Qed.
```

Exercise: 1 star, standard, optional (plus_explicit_prop) Rewrite both add_assoc' and add_comm' and their proofs in the same style as mul_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 Rocq to generate induction priniciples. For example, recall our proposition **ev** from IndProp:

```
Print ev.
```

```
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 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 Rocq 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 Rocq 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 Rocq 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 everything they would need to understand to be able to produce the former.

Question: How much detail does that require?

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* ... *ak*, where <...and here we state the IH for each of the a's that has type S, if any>. We must show <...and here we restate P(c *a1* ... *ak*)>.

    <go on and prove $P(n)$ to finish the case...>
  - <other cases similarly...> □

Example:

- *Theorem*: For all sets X, lists l : **list** X, and numbers n, if length $l = n$ then *index* (S n) l = None.

  Proof: By induction on l.

  - Suppose l = []. We must show, for all numbers n, that, if length [] = n, then *index* (S n) [] = None.

    This follows immediately from the definition of *index*.
  - Suppose $l = x :: l'$ for some x and l', where length $l' = n'$ implies *index* (S n') l' = None, for any number n'. We must show, for all n, that, if length (x::l') = n then *index* (S n) (x::l') = None.

    Let n be a number with length $l = n$. Since

    length l = length (x::l') = S (length l'),

    it suffices to show that

    index (S (length l')) l' = None.

    But this follows directly from the induction hypothesis, picking n' to be length l'. □

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 "$Q \to P$," where Q is some inductively defined proposition (more generally, "For all x y z, Q x y $z \to P$ x y z")>

  Proof: By induction on a derivation of Q. <Or, more generally, "Suppose we are given x, y, and z. We show that Q x y z implies P x y z, by induction on a derivation of Q x y z"...>

  <one case for each constructor c of Q...>

  - Suppose the final rule used to show Q is c. Then <...and here we state the types of all of the a's together with any equalities that follow from the definition of the constructor and the IH for each of the a's that has type Q, if there are any>. We must show <...and here we restate P>.

    <go on and prove P to finish the case...>
  - <other cases similarly...> □

Example

- *Theorem*: The $\leq$ relation is transitive – i.e., for all numbers n, m, and o, if $n \leq m$ and $m \leq o$, then $n \leq o$.

  Proof: By induction on a derivation of $m \leq o$.

  - Suppose the final rule used to show $m \leq o$ is le_n. Then $m = o$ and we must show that $n \leq m$, which is immediate by hypothesis.
  - Suppose the final rule used to show $m \leq o$ is le_S. Then o = S o' for some o' with $m \leq o'$. We must show that $n \leq$ S o'. By induction hypothesis, $n \leq o'$. But then, by le_S, $n \leq$ S o'. □

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 Rocq to do something slightly non-standard.

Recall again the induction principle on naturals that Rocq 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 Rocq lemma that requires a proof. Rocq 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:**nat**, $P\ n \to P\ (S\ n)$. 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 Rocq 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 even function defined in Basics.

```
Lemma even_ev : ∀ n: nat, even 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 even_ev' : ∀ n : nat, (even n = true → ev n) ∧ (even (S n) = true → 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 :
  ∀ (P : nat → Prop),
  P 0 →
  P 1 →
  (∀ n : nat, P n → P (S(S n))) →
  ∀ n : nat , P n :=
    fun P ⇒ fun P0 ⇒ fun P1 ⇒ fun PSS ⇒
      fix f (n:nat) := match n with
                         0 ⇒ P0
                       | 1 ⇒ P1
                       | S (S n') ⇒ PSS n' (f n')
                       end.
```

Once you get the hang of it, it is entirely straightforward to give an explicit proof term for induction principles like this. Proving this as a lemma using tactics is much less intuitive.

The `induction ... using` tactic variant gives a convenient way to utilize a non-standard induction principle like this.

```
Lemma even_ev : ∀ n, even n = true → 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.
```

Exercise: 4 stars, standard, optional (t_tree) What if we wanted to define binary trees as follows, using a constructor that bundles the children and value at a node into a tuple?

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

Inductive t_tree (X : Type) : Type :=
| t_leaf
| t_branch : (t_tree X × X × t_tree X) → t_tree X.

Arguments t_leaf {X}.
Arguments t_branch {X}.
```

Unfortunately, the automatically-generated induction principle is not as strong as we need. It doesn't introduce induction hypotheses for the subtrees.

```
Check t_tree_ind.
```

That will get us in trouble if we want to prove something by induction, such as that reflect is an involution.

```
Fixpoint reflect {X : Type} (t : t_tree X) : t_tree X :=
  match t with
  | t_leaf ⇒ t_leaf
  | t_branch (l, v, r) ⇒ t_branch (reflect r, v, reflect l)
  end.

Theorem reflect_involution : ∀ (X : Type) (t : t_tree X),
    reflect (reflect t) = t.
Proof.
  intros X t. induction t.
  - reflexivity.
  - destruct p as [[l v] r]. simpl. Abort.
```

We get stuck, because we have no inductive hypothesis for l or r. So, we need to define our own custom induction principle, and use it to complete the proof.

First, define the type of the induction principle that you want to use. There are many possible answers. Recall that you can use `match` as part of the definition.

```
Definition better_t_tree_ind_type : Prop
  . Admitted.
```

Second, define the induction principle by giving a term of that type. Use the examples about **nat**, above, as models.

```
Definition better_t_tree_ind : better_t_tree_ind_type
  . Admitted.
```

Finally, prove the theorem. If `induction...using` gives you an error about "Cannot recognize an induction scheme", don't worry about it. The `induction` tactic is picky about the shape of the theorem you pass to it, but it doesn't give you much information to debug what is wrong about that shape. You can use `apply` instead, as we saw at the beginning of this file.

```
Theorem reflect_involution : ∀ (X : Type) (t : t_tree X),
    reflect (reflect t) = t.
Proof. Admitted.
```

□

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 Rocq. 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 Rocq's basic reasoning facilities, so it may be useful to look at this material just after the IndProp chapter.

```
Set Warnings "-notation-overridden".
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.
```

Somewhat confusingly, the Rocq 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 Rocq identifier relation will always refer to a binary relation *on* some set (between the set and itself), whereas in ordinary mathematical English the word "relation" can refer either to this specific 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 Rocq 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: Type} (R: relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.
```

For example, the `next_nat` relation is a partial function.

```
Inductive next_nat : nat → nat → Prop :=
  | nn n : next_nat n (S n).

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 :
  ¬ (partial_function le).
Proof.
  unfold not. unfold partial_function. intros Hc.
  assert (0 = 1) as Nonsense. {
    apply Hc with (x := 0).
    - apply le_n.
    - apply le_S. apply le_n. }
  discriminate Nonsense. Qed.
```

Exercise: 2 stars, standard, optional (total_relation_not_partial_function) Show that the **total_relation** defined in (an exercise in) IndProp is not a partial function.

Copy the definition of **total_relation** from your IndProp here so that this file can be graded on its own. `Inductive` **total_relation** : **nat** → **nat** → `Prop` :=

.

```
Theorem total_relation_not_partial_function :
  ¬ (partial_function total_relation).
Proof.
  Admitted.
```
□

Exercise: 2 stars, standard, optional (empty_relation_partial_function) Show that the **empty_relation** defined in (an exercise in) IndProp is a partial function.

Copy the definition of **empty_relation** from your IndProp here so that this file can be graded on its own. `Inductive` **empty_relation** : **nat** → **nat** → `Prop` :=

.

```
Theorem empty_relation_partial_function :
  partial_function empty_relation.
Proof.
  Admitted.
```
□

Reflexive Relations

A *reflexive* relation on a set X is one for which every element of X is related to itself.

```
Definition reflexive {X: Type} (R: relation X) :=
  ∀ a : X, R a a.

Theorem le_reflexive :
  reflexive le.
Proof.
  unfold reflexive. intros n. apply le_n. Qed.
```

Transitive Relations

A relation R is *transitive* if R a c holds whenever R a b and R b c do.

```
Definition transitive {X: Type} (R: relation X) :=
  ∀ a b c : X, (R a b) → (R b c) → (R a c).

Theorem le_trans :
  transitive le.
```

```
Proof.
  intros n m o Hnm Hmo.
  induction Hmo.
  - apply Hnm.
  - apply le_S. apply IHHmo. Qed.

Theorem lt_trans:
  transitive lt.
Proof.
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  apply le_S in Hnm.
  apply le_trans with (a := (S n)) (b := (S m)) (c := o).
  apply Hnm.
  apply Hmo. Qed.
```

Exercise: 2 stars, 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 : ∀ n m, S n ≤ m → n ≤ m.
Proof.
  intros n m H. apply le_trans with (S n).
```

```
  - apply le_S. apply le_n.
  - apply H.
Qed.
```

Exercise: 1 star, standard, optional (le_S_n)

```
Theorem le_S_n : ∀ n m,
  (S n ≤ S m) → (n ≤ m).
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard, optional (le_Sn_n_inf) Provide an informal proof of the following theorem:

Theorem: For every n, $\neg$ (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 : ∀ n,
  ¬ (S n ≤ 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 Rocq, 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) :=
  ∀ a b : X, (R a b) → (R b a).
```

Exercise: 2 stars, standard, optional (le_not_symmetric)

```
Theorem le_not_symmetric
:
  ¬ (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) :=
  ∀ a b : X, (R a b) → (R b a) → a = b.
```

Exercise: 2 stars, standard, optional (le_antisymmetric) `Theorem le_antisymmetric :`

```
  antisymmetric le.
Proof.
  Admitted.
```

□

Exercise: 2 stars, standard, optional (le_step) `Theorem le_step : ∀ n m p,`

```
  n < m →
  m ≤ S p →
  n ≤ p.
Proof.
  Admitted.
```

□

Equivalence Relations

A relation is an *equivalence* if it's reflexive, symmetric, and transitive.

```
Definition equivalence {X:Type} (R: relation X) :=
  (reflexive R) ∧ (symmetric R) ∧ (transitive R).
```

Partial Orders and Preorders

A relation is a *partial order* when it's reflexive, *anti*-symmetric, and transitive. In the Rocq standard library it's called just "order" for short.

```
Definition order {X:Type} (R: relation X) :=
  (reflexive R) ∧ (antisymmetric R) ∧ (transitive R).
```

A preorder is almost like a partial order, but doesn't have to be antisymmetric.

```
Definition preorder {X:Type} (R: relation X) :=
  (reflexive R) ∧ (transitive R).

Theorem le_order :
  order le.
Proof.
  unfold order. split.
    - 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 Rocq standard library:

```
Inductive clos_refl_trans {A: Type} (R: relation A) : relation A :=
  | rt_step x y (H : R x y) : clos_refl_trans R x y
  | rt_refl x : clos_refl_trans R x x
  | rt_trans x y z
        (Hxy : clos_refl_trans R x y)
        (Hyz : clos_refl_trans R y z) :
        clos_refl_trans R x z.
```

For example, the reflexive and transitive closure of the next_nat relation coincides with the le relation.

```
Theorem next_nat_closure_is_le : ∀ n m,
  (n ≤ m) ↔ ((clos_refl_trans next_nat) n m).
Proof.
  intros n m. split.
  -
    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 :=
  | rt1n_refl : clos_refl_trans_1n R x x
```

```
| rt1n_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 : ∀ (X:Type) (R:relation X) (x y : X),
  R x y → clos_refl_trans_1n R x y.
Proof.
  intros X R x y H.
  apply rt1n_trans with y. apply H. apply rt1n_refl. Qed.
```

Exercise: 2 stars, standard, optional (rsc_trans) `Lemma rsc_trans :`

```
  ∀ (X:Type) (R: relation X) (x y z : X),
      clos_refl_trans_1n R x y →
      clos_refl_trans_1n R y z →
      clos_refl_trans_1n R x z.
Proof.
    Admitted.
```

□

Then we use these facts to prove that the two definitions of reflexive, transitive closure do indeed define the same relation.

Exercise: 3 stars, standard, optional (rtc_rsc_coincide) `Theorem rtc_rsc_coincide :`

```
  ∀ (X:Type) (R: relation X) (x y : X),
    clos_refl_trans R x y ↔ 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 Rocq as a tool 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, in *Programming Language Foundations* (*Software Foundations*, volume 2), we develop a theory of *program equivalence* and introduce *Hoare Logic*, a popular logic for reasoning about imperative programs.

```
Set Warnings "-notation-overridden".
From Stdlib Require Import Bool.
From Stdlib Require Import Init.Nat.
From Stdlib Require Import Arith.
From Stdlib Require Import EqNat. Import Nat.
From Stdlib Require Import Lia.
From Stdlib Require Import List. Import ListNotations.
From Stdlib Require Import Strings.String.
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 with *variables*, and finally a language of *commands* including assignment, conditionals, 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)
  | BNeq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BGt (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 course on compilers) 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 | a <= a | a > a | ~ b | b && b

Compared to the Rocq 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 Rocq 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 nailing down every detail 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)
  | BNeq a1 a2 ⇒ negb ((aeval a1) =? (aeval a2))
  | BLe a1 a2 ⇒ (aeval a1) <=? (aeval a2)
  | BGt a1 a2 ⇒ negb ((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 gain confidence that 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 certain the optimization is correct – that evaluating an optimized expression *always* gives the same result as the original – we should prove it!

```
Theorem optimize_0plus_sound: ∀ a,
  aeval (optimize_0plus a) = 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 Rocq 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 Rocq'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 – Rocq'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 Rocq'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 : ∀ (P : `Prop`), P → P.

```
Proof.
  intros P HP.
  try reflexivity.    apply HP. Qed.

Theorem silly2 : ∀ ae, aeval ae = aeval ae.
Proof.
    try reflexivity. Qed.
```

There is not much 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 : ∀ n, 0 <=? n = true.
Proof.
  intros.
  destruct n.
    - simpl. reflexivity.
    - simpl. reflexivity.
Qed.
```

We can simplify this proof using the ; tactical:

```
Lemma foo' : ∀ n, 0 <=? n = true.
Proof.
  intros.
  destruct n;

  simpl;

  reflexivity.
Qed.
```

Using `try` and ; together, we can get rid of the repetition in the proof that was bothering us a little while ago.

```
Theorem optimize_0plus_sound': ∀ 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.
```

Rocq experts often use this "...; `try`... " idiom after a tactic like `induction` to take care of many similar cases all at once. Indeed, 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* = 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* = S *n'* for some *n'*, then again optimize_0plus simply calls itself recursively, and the result follows from the IH. □

However, this proof can still be improved: the first case (for a = 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'': ∀ 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*, ..., *Tn* are tactics, then

T; *T1* | *T2* | ... | *Tn*

is a tactic that first performs *T* and then performs *T1* on the first subgoal generated by *T*, performs *T2* on the second subgoal, etc.

So *T*;*T'* is just special notation for the case when all of the *Ti*'s are the same tactic; i.e., *T*;*T'* is shorthand for:

T; *T'* | *T'* | ... | *T'*

The `repeat` Tactical

The `repeat` tactical takes another tactic and keeps applying this tactic until it fails or until it succeeds but doesn't make any progress.

Here is an example proving that 10 is in a long list using `repeat`.

```
Theorem In10 : In 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 *succeeds* without changing the goal at all (i.e., it repeats zero times).

```
Theorem In10' : In 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 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 : ∀ (m n : nat),
  m + n = n + m.
Proof.
  intros m n.
Admitted.
```

Wait – did we just write an infinite loop in Rocq?!?!

Sort of.

While evaluation in Rocq's term language, Gallina, is guaranteed to terminate, *tactic* evaluation is not. This does not affect Rocq's logical consistency, however, since the job of `repeat` and other tactics is to guide Rocq 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 at all, not that we have constructed a bad proof.

Exercise: 3 stars, standard (optimize_0plus_b_sound) Since the optimize_0plus transformation doesn't change the value of **aexp**s, we should be able to apply it to all the **aexp**s that appear in a **bexp** without changing the **bexp**'s value. Write a function that performs this transformation on **bexp**s and prove it is sound. Use the tacticals we've just seen to make the proof as short and elegant as possible.

```
Fixpoint optimize_0plus_b (b : bexp) : bexp
  . Admitted.

Example optimize_0plus_b_test1:
```

```
  optimize_0plus_b (BNot (BGt (APlus (ANum 0) (ANum 4)) (ANum 8))) =
                   (BNot (BGt (ANum 4) (ANum 8))).
Proof. Admitted.

Example optimize_0plus_b_test2:
  optimize_0plus_b (BAnd (BLe (APlus (ANum 0) (ANum 4)) (ANum 5)) BTrue) =
                   (BAnd (BLe (ANum 4) (ANum 5)) BTrue).
Proof. Admitted.

Theorem optimize_0plus_b_sound : ∀ b,
  beval (optimize_0plus_b b) = 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 incrementally to something more interesting.)

13.3.2 Defining New Tactics

Rocq also provides facilities for "programming" in tactic scripts.

The `Ltac` idiom illustrated below gives a handy way to define "shorthand tactics" that bundle several tactics into a single command.

Ltac also includes syntactic pattern-matching on the goal and context, as well as general programming facilities.

It is useful for proof automation and there are several idioms for programming with Ltac. Because it is a language style you might not have seen before, a good reference is the textbook "Certified Programming with dependent types" *CPDT*, which is more advanced that what we will need in this course, but is considered by many the best reference for Ltac programming.

Just for future reference: Rocq provides two other ways of defining new tactics. There is a `Tactic Notation` command that allows defining new tactics with custom control over their concrete syntax. And there is also a low-level API that can be used to build tactics that directly manipulate Rocq's internal structures. We will not need either of these for present purposes.

Here's an example `Ltac` script called invert.

```
Ltac invert H :=
  inversion H; subst; clear H.
```

This defines a new tactic called invert that takes a hypothesis H as an argument and performs the sequence of commands `inversion H; subst; clear H`. This gives us quick way to do inversion on evidence and constructors, rewrite with the generated equations, and remove the redundant hypothesis at the end.

```
Lemma invert_example1: ∀ {a b c: nat}, [a ;b] = [a;c] → b = c.
  intros.
  invert H.
  reflexivity.
Qed.
```

13.3.3 The *lia* Tactic

The *lia* tactic implements a decision procedure for integer linear arithmetic, a subset of propositional logic and arithmetic.

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 ≠) and ordering (≤ and >), and
- the logical connectives ∧, ∨, ¬, and →,

then invoking *lia* will either solve the goal or fail, meaning that the goal is actually false. (If the goal is *not* of this form, *lia* will fail.)

```
Example silly_presburger_example : ∀ m n o p,
  m + n ≤ n + o ∧ o + 3 = p + 3 →
  m ≤ p.
Proof.
  intros. lia.
Qed.

Example add_comm__lia : ∀ m n,
    m + n = n + m.
Proof.
  intros. lia.
Qed.

Example add_assoc__lia : ∀ m n p,
    m + (n + p) = m + n + p.
Proof.
  intros. lia.
Qed.
```

(Note the `From Stdlib Require Import Lia.` at the top of this file, which makes *lia* available.)

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*: Given 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 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 `Fixpoint`s. Another way to think about evaluation – one that is often more flexible – is as a *relation* between expressions and their values. This perspective leads to `Inductive` definitions like the following...

```
Module AEVALR_FIRST_TRY.

Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      aevalR (ANum n) n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
```

```
      aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (AMult e1 e2) (n1 × n2).
```

```
Module HypothesisNames.
```

A small notational aside. We could also write the definition of **aevalR** as follow, with explicit names for the hypotheses in each case:

```
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      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 Rocq 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 our case study of regular expressions in chapter IndProp, Rocq 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.

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)

...can be written like this as an inference rule:

e1 ==> n1 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 →.

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 sometimes indicated by saying something like "Let **aevalR** be the smallest relation closed under the following rules...".

For example, we could define ==> as the smallest relation closed under these rules:

(E_ANum) ANum n ==> n

e1 ==> n1 e2 ==> n2

(E_APlus) APlus e1 e2 ==> n1+n2

e1 ==> n1 e2 ==> n2

(E_AMinus) AMinus e1 e2 ==> n1-n2
e1 ==> n1 e2 ==> n2

(E_AMult) AMult e1 e2 ==> n1*n2

Exercise: 1 star, standard, optional (beval_rules) Here, again, is the Rocq definition of the beval function:

Fixpoint beval (e : bexp) : bool := match e with | BTrue => true | BFalse => false | BEq a1 a2 => (aeval a1) =? (aeval a2) | BNeq a1 a2 => negb ((aeval a1) =? (aeval a2)) | BLe a1 a2 => (aeval a1) <=? (aeval a2) | BGt 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 aevalR_iff_aeval : ∀ a n,
  (a ==> n) ↔ 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.
```

Again, we can make the proof quite a bit shorter using some tacticals.

```
Theorem aevalR_iff_aeval' : ∀ a n,
  (a ==> n) ↔ 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' b" (at level 90, left associativity).
Inductive bevalR: bexp → bool → Prop :=

where "e '==>b' b" := (bevalR e b) : type_scope
.

Lemma bevalR_iff_beval : ∀ b bv,
  b ==>b bv ↔ 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 fine.

However, there are many situations 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) →
      (mult n2 n3 = n1) → (ADiv a1 a2) ==> n3

where "a '==>' n" := (aevalR a n) : type_scope.
```

Notice that this evaluation relation corresponds to a *partial* function: There are some inputs for which it 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 of these styles 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 Rocq automatically generates nice inversion and induction principles from `Inductive` definitions.

On the other hand, functional definitions can often be more convenient:

- Functions are automatically deterministic and total; for a relational definition, we have to *prove* these properties explicitly if we need them.
- With functions we can also take advantage of Rocq'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 Rocq 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

Let's return to defining Imp, where 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 use total maps from the Maps chapter.

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 able to mention a finite number of them. Because each variable stores a natural number, we can represent the state as a total map from strings (variable names) to **nat**, and will use 0 as default value in the store.

```
Definition state := total_map nat.
```

13.5.2 Syntax

We can add variables to the arithmetic expressions we had before simply by including one more constructor:

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | AId (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 **bexp**s is unchanged (except that it now refers to the new **aexp**s):

```
Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BNeq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BGt (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 AId allows us to use plain strings when an **aexp** is expected; the string will implicitly be wrapped with AId.

- *Declare Custom Entry* **com** tells Rocq to create a new "custom grammar" for parsing Imp expressions and programs. The first notation declaration after this tells Rocq that anything between <{ and }> 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 +, -, ×, =, ≤, etc., when they occur between <{ and }>.

```
Coercion AId : string >-> aexp.
Coercion ANum : nat >-> aexp.

Declare Custom Entry com.
Declare Scope com_scope.

Notation "<{ e }>" := e
  (e custom com, format "'[hv' <{ '/ ' '[v' e ']' '/' }> ']'") : com_scope.

Notation "( x )" := x (in custom com, x at level 99).
Notation "x" := x (in custom com at level 0, x constr at level 0).
```

```
Notation "f x .. y" := (.. (f x) .. y)
                    (in custom com at level 0, only parsing,
                    f constr at level 0, x constr at level 1,
                        y constr at level 1).
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" := (BGt 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" := (BNeq 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 && ~$(X \leq 4)$ instead of BAnd true (BNot (BLe X 4)).

```
Definition example_aexp : aexp := <{ 3 + (X × 2) }>.
Definition example_bexp : bexp := <{ true && ¬(X ≤ 4) }>.
```

13.5.4 Evaluation

The arith and boolean evaluators must now be extended to handle variables in the obvious way, taking a state *st* as an extra argument:

```
Fixpoint aeval (st : state)
               (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | AId 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}> ⇒ negb ((aeval st a1) =? (aeval st a2))
  | <{a1 ≤ a2}> ⇒ (aeval st a1) <=? (aeval st a2)
  | <{a1 > a2}> ⇒ negb ((aeval st a1) <=? (aeval st a2))
  | <{¬ b1}> ⇒ negb (beval st b1)
  | <{b1 && b2}> ⇒ andb (beval st b1) (beval st b2)
  end.
```

We can use our notation for total maps in the specific case of states – i.e., we write the empty state as (__ !-> 0).

```
Definition empty_st := (__ !-> 0).
```

Also, we can add a notation for a "singleton state" with just one variable bound to a value.

```
Notation "x '!->' v" := (x !-> v ; empty_st) (at level 100, right associativity).

Example aexp1 :
    aeval (X !-> 5) <{ 3 + (X × 2) }>
  = 13.
Proof. reflexivity. Qed.
Example aexp2 :
    aeval (X !-> 5 ; Y !-> 4) <{ Z + (X × Y) }>
  = 20.
Proof. reflexivity. Qed.

Example bexp1 :
    beval (X !-> 5) <{ true && ¬(X ≤ 4) }>
  = true.
Proof. reflexivity. Qed.
```

13.6 Commands

Now we are ready to define the syntax and behavior of Imp *commands* (or *statements*).

13.6.1 Syntax

Informally, commands c are described by the following BNF grammar.

c := skip | x := a | c ; c | if b then c else c end | while b do c end

Here is the formal definition of the abstract syntax of commands:

```
Inductive com : Type :=
  | CSkip
  | CAsgn (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).
```

As we did 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" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
    format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
    format "'[v' 'if' x 'then' '/  ' y '/' 'else' '/  ' z '/' 'end' ']'") : 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,
    format "'[v' 'while' x 'do' '/  ' y '/' 'end' ']'") : com_scope.
```

For example, here is the factorial function again, written as a formal Rocq definition. When this command terminates, the variable Y will contain the factorial of the initial value of X.

```
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

Rocq 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 it hard to understand what the expressions we enter actually mean. In such situations 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.

Print example_bexp.

Set Printing Coercions.
Print example_bexp.

Print fact_in_coq.
Unset Printing Coercions.
```

13.6.3 `Locate` Again

Finding identifiers

When used with an identifier, the `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`.

Finding notations

When faced with an unknown notation, you can use `Locate` with a string containing one of its symbols to see its possible interpretations. `Locate "&&".`

```
Locate ";".

Locate "while".
```

13.6.4 More Examples

Assignment:

```
Definition plus2 : com :=
  <{ X := X + 2 }>.

Definition XtimesYinZ : com :=
  <{ Z := X × Y }>.
```

Loops

```
Definition subtract_slowly_body : com :=
  <{ Z := Z - 1 ;
     X := X - 1 }>.

Definition subtract_slowly : com :=
```

```
  <{ while 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 (with a bogus *while* case).

```
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 more conventional 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.

Rocq 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 Rocq aims to be not just a functional programming language but also a consistent logic, any potentially non-terminating function needs to be rejected.

Here is an example showing what would go wrong if Rocq 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 Rocq's logical consistency.

Thus, because it doesn't terminate on all inputs, *ceval_fun* cannot be written in Rocq – at least not without additional tricks and workarounds (see chapter ImpCEvalFun if you're curious about those).

13.7.2 Evaluation as a Relation

Here's a better way: define **ceval** as a *relation* rather than a *function* – i.e., make its result a `Prop` rather than a state, similar to what we did for **aevalR** above.

This is an important change. Besides freeing us from awkward workarounds, it gives us a ton 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_Asgn) 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 = `if b then c1 else c2 end` => st'
  beval st b = false st = `c2` => st'

(E_IfFalse) st = `if b then c1 else c2 end` => st'
  beval st b = false

(E_WhileFalse) st = *while* `b do c end` => st
  beval st b = true st = `c` => st' st' = *while* `b do c end` => st''

(E_WhileTrue) st = *while* `b do c end` => st''

Here is the formal definition. Make sure you understand how it corresponds to the inference rules.

```
Reserved Notation
         "st0 '=[' c ']=>' st1"
         (at level 40, c custom com at level 99,
          st0 constr, st1 constr at next level,
          format "'[hv' st0 =[ '/ ' '[' c ']' '/' ]=> st1 ']'").

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      st =[ skip ]=> st
  | E_Asgn : ∀ st a n x,
      aeval st a = n →
      st =[ x := a ]=> (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''

  where "st0 =[ c ]=> st1" := (ceval c st0 st1).
```

The cost of defining evaluation as a relation instead of a function is that we now need to construct a *proof* that some program evaluates to some result state, rather than just letting Rocq'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_Asgn. reflexivity.
  -
    apply E_IfFalse.
    + reflexivity.
    + apply E_Asgn. 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 + ... + 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: ∀ 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.
  - 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 into more systematic and powerful techniques for reasoning about Imp programs in *Programming Language Foundations*, but we can already do a few things (albeit in a somewhat low-level way) just by working with the bare definitions. This section explores some examples.

```
Theorem plus2_spec : ∀ st n st',
  st X = n →
  st =[ plus2 ]=> st' →
  st' X = n + 2.
Proof.
  intros st n st' HX Heval.
```

Inverting *Heval* essentially forces Rocq 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×string) := None.
```

□

Exercise: 3 stars, standard, especially useful (loop_never_stops)

```
Theorem loop_never_stops
: ∀ st st',
  ~(st =[ loop ]=> 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:
  ∀ c, no_whiles c = true ↔ 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 **aexp**s 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 : ∀ 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 : ∀ 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 : ∀ (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, standard, optional (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
  | CAsgn (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).

Notation "'break'" := CBreak (in custom com at level 0) : com_scope.
Notation "'skip'" := CSkip
  (in custom com at level 0) : com_scope.
Notation "x := y" := (CAsgn x y)
  (in custom com at level 0, x constr at level 0, y at level 85, no associativity,
    format "x := y") : com_scope.
Notation "x ; y" := (CSeq x y)
  (in custom com at level 90,
    right associativity,
    format "'[v' x ; '/' y ']'") : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" := (CIf x y z)
  (in custom com at level 89, x at level 99, y at level 99, z at level 99,
```

```
    format "'[v' 'if' x 'then' '/ ' y '/' 'else' '/ ' z '/' 'end' ']'") : 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,
    format "'[v' 'while' x 'do' '/ ' y '/' 'end' ']'") : 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
         "st0 '=[' c ']=>' st1 '/' s"
         (at level 40, c custom com at level 99,
          st0 constr, st1 constr at next level,
          format "'[hv' st0 =[ '/ ' '[' c ']' '/' ]=> st1 / s ']'").
```

Intuitively, *st* =[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* = SBreak) or that execution should continue normally (*s* = SContinue).

The definition of the "*st* =[c]=> *st'* / *s*" relation is very similar to the one we gave above for the regular evaluation relation (*st* =[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.

Theorem seq_continue : ∀ c1 c2 st st' st'',
  st =[ c1 ]=> st' / SContinue →
  st' =[ c2 ]=> st'' / SContinue →
  st =[ c1 ; c2 ]=> st'' / SContinue.
```

```
Proof.
    Admitted.

Theorem seq_stops_on_break : ∀ c1 c2 st st',
  st =[ c1 ]=> st' / SBreak →
  st =[ c1 ; c2 ]=> st' / SBreak.
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: ∀ (c:com) st st1 st2 s1 s2,`

```
      st =[ c ]=> st1 / s1 →
      st =[ c ]=> st2 / s2 →
      st1 = st2 ∧ 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 Rocq.

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 Rocq

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 Rocq'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,-notation-incompatible-prefix".
From Stdlib Require Import Strings.String.
From Stdlib Require Import Strings.Ascii.
From Stdlib Require Import Arith.
From Stdlib Require Import Init.Nat.
From Stdlib Require Import EqNat.
From Stdlib Require Import List. Import ListNotations.
From LF Require Import Maps Imp.
Local Open Scope com_scope.
```

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' e1 'OR' e2"
  := (
   let result := e1 in
   match result with
      | SomeE _ ⇒ result
      | NoneE _ ⇒ e2
      end)
  (right associativity,
   at level 60, e1 at next level, e2 at next level).
```

Generic Combinators for Building Parsers

```
Open Scope string_scope.
Definition parser (T : Type) :=
  list token → optionE (T × list token).
Fixpoint many_helper {T} (p : parser T) acc steps xs :=
  match steps, p xs with
  | 0, _ ⇒
      NoneE "Too many recursive calls"
  | _, NoneE _ ⇒
      SomeE ((rev acc), xs)
  | S steps', SomeE (t, xs') ⇒
      many_helper p (t :: acc) steps' xs'
  end.
```

A (step-indexed) parser that expects zero or more ps:

```
Definition many {T} (p : parser T) (steps : nat) : parser (list T) :=
  many_helper p [] steps.
```

A parser that expects a given token, followed by p:

```
Definition firstExpect {T} (t : token) (p : parser T)
                       : parser T :=
  fun xs ⇒ match xs with
            | x::xs' ⇒
              if string_dec x t
              then p xs'
              else NoneE ("expected '" ++ t ++ "'.")
            | [] ⇒
              NoneE ("expected '" ++ t ++ "'.")
```

```
end.
```

A parser that expects a particular token:

```
Definition expect (t : token) : parser unit :=
  firstExpect t (fun xs ⇒ SomeE (tt, xs)).
```

A Recursive-Descent Parser for Imp

Identifiers:

```
Definition parseIdentifier (xs : list token)
                           : optionE (string × list token) :=
match xs with
| [] ⇒ NoneE "Expected identifier"
| x::xs' ⇒
    if forallb isLowerAlpha (list_of_string x) then
      SomeE (x, xs')
    else
      NoneE ("Illegal identifier:'" ++ x ++ "'")
end.
```

Numbers:

```
Definition parseNumber (xs : list token)
                       : optionE (nat × list token) :=
match xs with
| [] ⇒ NoneE "Expected number"
| x::xs' ⇒
    if forallb isDigit (list_of_string x) then
      SomeE (fold_left
               (fun n d ⇒
                  10 × n + (nat_of_ascii d -
                            nat_of_ascii "0"%char))
               (list_of_string x)
               0,
             xs')
    else
      NoneE "Expected number"
end.
```

Parse arithmetic expressions

```
Fixpoint parsePrimaryExp (steps:nat)
                         (xs : list token)
                       : optionE (aexp × list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
```

```
  | S steps' ⇒
      TRY ' (i, rest) ← parseIdentifier xs ;;
          SomeE (AId 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 e0 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

```
Set Warnings "-notation-overridden,-notation-incompatible-prefix".
From Stdlib Require Import Lia.
From Stdlib Require Import Arith.
From Stdlib Require Import PeanoNat.
Import Nat.
From Stdlib Require Import EqNat.
From LF Require Import Imp Maps.
Local Open Scope com_scope.
```

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

Rocq 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 Rocq 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(!) Rocq program showing what would go wrong if Rocq 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 Rocq's logical consistency.

Thus, because it doesn't terminate on all inputs, the full version of ceval_step1 cannot be written in Rocq – 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
  | O ⇒ 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
  | O ⇒ 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.
```

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.

Example example_test_ceval :
     test_ceval empty_st

     <{ X := 2;
        if (X ≤ 1)
        then Y := 3
        else Z := 4
        end }>

     = Some (2, 0, 4).
Proof. reflexivity. Qed.
```

Exercise: 1 star, standard, optional (pup_to_n) Write an Imp program that sums the numbers from 1 to X (inclusive – i.e., 1 + 2 + ... + X) in the variable Y. Make sure your solution satisfies the test that follows.

```
Definition pup_to_n : com
  . Admitted.

Example pup_to_n_1 :
  test_ceval (X !-> 5) pup_to_n
  = Some (0, 15, 0).
   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: ∀ c st st',
      (∃ i, ceval_step st c i = Some st') →
      st =[ c ]=> 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_Asgn. reflexivity.
    +
      destruct (ceval_step st c1 i') eqn:Heqr1.
      ×
        apply E_Seq with s.
          apply IHi'. rewrite Heqr1. reflexivity.
          apply IHi'. assumption.
      ×
        discriminate H1.
    +
      destruct (beval st b) eqn:Heqr.
      ×
        apply E_IfTrue. rewrite Heqr. reflexivity.
        apply IHi'. assumption.
      ×
        apply E_IfFalse. rewrite Heqr. reflexivity.
        apply IHi'. assumption.
    + destruct (beval st b) eqn :Heqr.
      ×
       destruct (ceval_step st c i') eqn:Heqr1.
       {
         apply E_WhileTrue with s. rewrite Heqr.
         reflexivity.
         apply IHi'. rewrite Heqr1. reflexivity.
         apply IHi'. assumption. }
       { discriminate H1. }
      ×
        injection H1 as H2. rewrite ← H2.
        apply E_WhileFalse. apply Heqr. Qed.
```

Exercise: 4 stars, advanced (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: ∀ i1 i2 st st' c,
  i1 ≤ i2 →
  ceval_step st c i1 = Some st' →
  ceval_step st c i2 = Some st'.
Proof.
induction i1 as [|i1']; intros i2 st st' c Hle Hceval.
  -
    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:Heqst1'o.
      ×
        apply (IHi1' i2') in Heqst1'o; try assumption.
        rewrite Heqst1'o. simpl. simpl in Hceval.
        apply (IHi1' i2') in Hceval; try assumption.
      ×
        discriminate Hceval.
    +
      simpl in Hceval. simpl.
      destruct (beval st b); apply (IHi1' i2') in Hceval;
        assumption.
    +
      simpl in Hceval. simpl.
      destruct (beval st b); try assumption.
```

```
    destruct (ceval_step st c i1') eqn: Heqst1'o.
    ×
      apply (IHi1' i2') in Heqst1'o; try assumption.
      rewrite → Heqst1'o. simpl. simpl in Hceval.
      apply (IHi1' 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: ∀ c st st',
      st =[ c ]=> st' →
      ∃ i, ceval_step st c i = Some st'.
Proof.
  intros c st st' Hce.
  induction Hce.
   Admitted.
   □

Theorem ceval_and_ceval_step_coincide: ∀ c st st',
      st =[ c ]=> st'
  ↔ ∃ i, ceval_step st c i = 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' : ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 He1 He2.
  apply ceval__ceval_step in He1.
  apply ceval__ceval_step in He2.
  inversion He1 as [i1 E1].
  inversion He2 as [i2 E2].
```

```
apply ceval_step_more with (i2 := i1 + i2) in E1.
apply ceval_step_more with (i2 := i1 + i2) in E2.
rewrite E1 in E2. inversion E2. reflexivity.
lia. lia. Qed.
```

Chapter 16

Library LF.Extraction

16.1 Extraction: Extracting OCaml from Rocq

16.2 Basic Extraction

In its simplest form, extracting an efficient program from one written in Rocq 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).

```
From Stdlib Require Extraction.
Set Extraction Output Directory ".".
Extraction Language OCaml.
```

Now we load up the Rocq environment with some definitions, either directly or by importing them from other modules.

```
Set Warnings "-notation-overridden,-notation-incompatible-prefix".
From Stdlib Require Import Arith.
From Stdlib Require Import Init.Nat.
From Stdlib Require Import EqNat.
From LF Require Import ImpCEvalFun.
```

Finally, we tell Rocq the name of a definition to extract and the name of a file to put the extracted code into.

```
Extraction "imp1.ml" ceval_step.
```

When Rocq processes this command, it generates a file *imp1.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 *imp1.ml* now.

16.3 Controlling Extraction of Specific Types

We can tell Rocq to extract certain `Inductive` definitions to specific OCaml types. For each one, we must say

- how the Rocq 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 Rocq module. To do this, we first need to set up the right correspondence between Rocq strings and lists of OCaml characters.

```
From Stdlib Require Import ExtrOcamlBasic.
From Stdlib 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,-notation-incompatible-prefix".
From Stdlib Require Import Lia.
From Stdlib Require Import Strings.String.
From LF Require Import Maps.
From LF Require Import Imp.
```

Up to now, we've used the manual part of Rocq's tactic facilities. In this chapter, we'll learn more about some of Rocq'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 some of 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 one other 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 the following proof, repeated with just a few small changes from the Imp chapter. We will simplify this proof in several stages.

```
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.
  - 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 only one at a time.

```
Example auto_example_1 : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
  intros P Q R H1 H2 H3.
  apply H2. apply H1. assumption.
Qed.
```

The `auto` tactic tries to free us from this drudgery by *searching* for a sequence of applications that will prove the goal:

```
Example auto_example_1' : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
  auto.
Qed.
```

The `auto` tactic solves goals that are solvable by any combination of `intros` and `apply`.

Using auto is always "safe" in the sense that it will never fail and will never change the proof state: either it completely solves the current goal, or it does nothing.

Here is a larger example showing auto's power:

```
Example auto_example_2 : ∀ P Q R S T U : Prop,
  (P → Q) →
  (P → R) →
  (T → R) →
  (S → T → U) →
  ((P → Q) → (P → S)) →
  T →
  P →
  U.
Proof. auto. Qed.
```

Proof search could, in principle, take an arbitrarily long time, so there is a limit to how deep auto will search by default.

If auto is not solving our goal as expected we can use debug auto to see a trace.

```
Example auto_example_3 : ∀ (P Q R S T U: Prop),
  (P → Q) →
  (Q → R) →
  (R → S) →
  (S → T) →
  (T → U) →
  P →
  U.
Proof.
  auto.

  debug 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 : ∀ P Q R : 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.

Example auto_example_5' : ∀ (P Q R S T U W: Prop),
  (U → T) →
  (W → U) →
  (R → S) →
  (S → T) →
  (P → R) →
  (U → T) →
  P →
  T.
Proof.
  intros.
  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. lia. Qed.

Example auto_example_6 : ∀ n m p q : nat,
  (p = q → (n ≤ m ∧ m ≤ n)) →
  p = q →
  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 Rocq 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 unfold uses of d, thus enabling further possibilities for applying lemmas that it knows about.

It is also possible to define specialized hint databases (besides *core*) that can be activated

only when needed; indeed, it is good style to create your own hint databases instead of polluting *core*.

See the Rocq reference manual for details.

```
Hint Resolve le_antisym : core.

Example auto_example_6' : ∀ n m p q : nat,
  (p = q → (n ≤ m ∧ m ≤ n)) →
  p = q →
  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' : ∀ x,
  (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
Proof.
  auto. Qed.
```

Note that the `Hint Unfold is_fortytwo` command above the example is needed because, unlike the normal `apply` tactic, the `simple apply` steps that are performed by `auto` do not do any automatic unfolding.

Let's take a first pass over ceval_deterministic, using `auto` to simplify the proof script.

```
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;
    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: ∀ c st st1 st2,
  st =[ c ]=> st1 →
  st =[ c ]=> st2 →
  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 degree 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 compli-

cated: 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, Rocq has a built-in tactic `congruence` that will do the job in this case. We'll ignore 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'': ∀ 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; 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's a bit better, but we really want Rocq 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 rewriting a 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: ∀ x, ?P x → ?L = ?R,
    H2: ?P ?X
    ⊢ _
    ⇒ rewrite (H1 X H2) in *
  end.
```

The pattern ∀ x, ?P x → ?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'''': ∀ 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.
Qed.
```

The big payoff in this approach is that the new proof script is more robust in the face of changes to our language. To test this, let's try adding a *REPEAT* command to the language.

```
Module Repeat.

Inductive com : Type :=
  | CSkip
  | CAsgn (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (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" :=
         (CAsgn 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'" :=
         (CIf 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_Asgn : ∀ 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: ∀ 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': ∀ 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 Rocq. 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 The `eapply` and `eauto` tactics

To close the chapter, let's look at one more convenience feature of Rocq: 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_Asgn. reflexivity.
  - apply E_IfFalse. reflexivity. apply E_Asgn. reflexivity.
Qed.
```

In the first step of the proof, we had to explicitly provide a longish expression to help Rocq 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 Rocq 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_Asgn. If Rocq could just wait until we get to this step, there would be no need for us to give the value explicitly. This is exactly what the `eapply` tactic allows:

```
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_Asgn.
   reflexivity.
  - apply E_IfFalse. reflexivity. apply E_Asgn. 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 skips checking 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`. Rocq 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 Rocq will (rightly) refuse to accept the proof. Remember that the Rocq 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. Unshelve. exact 0.
```

Rocq 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` would yield an error. (Try it!) `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 Rocq'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, Rocq 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: A Streamlined Treatment of Automation

So far, we've been doing all our proofs using just a small handful of Rocq's tactics and completely ignoring its powerful facilities for constructing parts of proofs automatically. Getting used to them will take some work – Rocq'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.

In this chapter, we'll learn about

- *tacticals*, which allow tactics to be combined;
- new tactics that make dealing with hypothesis names less fussy and more maintainable;
- *automatic solvers* that can prove limited classes of theorems without any human assistance;
- *proof search* with the `auto` tactic; and
- the *Ltac* language for writing tactics.

These features enable startlingly short proofs. Used properly, they can also make proofs more maintainable and robust to changes in underlying definitions.

This chapter is an alternative to the combination of Imp and Auto, which cover roughly the same material about automation, but in the context of programming language metatheory. A deeper treatment of `auto` can be found in the *UseAuto* chapter in *Programming Language Foundations*.

```
Set Warnings "-notation-overridden,-notation-incompatible-prefix".
From Stdlib Require Import Arith List.
From LF Require Import IndProp.
```

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 | s2 re1 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.
```

The amount of repetition in that proof is annoying. And if we wanted to extend the optimization function to handle other, similar, rewriting opportunities, it would start to be a real problem. We can streamline the proof with *tacticals*, which we turn to, next.

18.2 Tacticals

Tacticals are tactics that take other tactics as arguments – "higher-order tactics," if you will.

18.2.1 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 : ∀ n, 1 + n = S n.
Proof. try reflexivity. Qed.

Theorem silly2 : ∀ (P : Prop), P → P.
Proof.
  intros P HP.
  Fail reflexivity.
  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.

18.2.2 The Sequence Tactical ; (Simple Form)

In its most common form, the sequence tactical, written with semicolon ;, 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 simple_semi : ∀ n, (n + 1 =? 0) = false.
Proof.
  intros n.
  destruct n eqn:E.
```

```
    - simpl. reflexivity.
    - simpl. reflexivity.
Qed.
```

We can simplify this proof using the ; tactical:

```
Lemma simple_semi' : ∀ n, (n + 1 =? 0) = false.
Proof.
  intros n.
  destruct n;

  simpl;

  reflexivity.
Qed.
```

Or even more tersely, `destruct` can do the `intro`, and `simpl` can be omitted:

```
Lemma simple_semi'' : ∀ n, (n + 1 =? 0) = false.
Proof.
  destruct n; reflexivity.
Qed.
```

Exercise: 3 stars, standard (try_sequence) Prove the following theorems using `try` and ;. Like simple_semi'' above, each proof script should be a sequence *t1*; ...; *tn*. of tactics, and there should be only one period in between `Proof.` and `Qed.`. Let's call that a "one shot" proof.

```
Theorem andb_eq_orb :
  ∀ (b c : bool),
  (andb b c = orb b c) →
  b = c.
Proof. Admitted.

Theorem add_assoc : ∀ n m p : nat,
    n + (m + p) = (n + m) + p.
Proof. Admitted.

Fixpoint nonzeros (lst : list nat) :=
  match lst with
  | [] ⇒ []
  | 0 :: t ⇒ nonzeros t
  | h :: t ⇒ h :: nonzeros t
  end.

Lemma nonzeros_app : ∀ lst1 lst2 : list nat,
  nonzeros (lst1 ++ lst2) = (nonzeros lst1) ++ (nonzeros lst2).
Proof. Admitted.
```

□

Using `try` and ; together, we can improve the proof about regular expression optimization.

```
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 | s2 re1 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.
```

18.2.3 The Sequence Tactical ; (Local Form)

The sequence tactical ; also has a more general form than the simple *T*; *T'* we saw above. If *T*, *T1*, ..., *Tn* are tactics, then

[*T*; [*T1* | *T2* | ... | *Tn*]]

is a tactic that first performs *T* and then locally performs *T1* on the first subgoal generated by *T*, locally performs *T2* on the second subgoal, etc.

So *T*; *T'* is just special notation for the case when all of the *Ti*'s are the same tactic; i.e., *T*; *T'* is shorthand for:

T; *T'* | *T'* | ... | *T'*

For example, the following proof makes it clear which tactics are used to solve the base case vs. the inductive case.

```
Theorem app_length : ∀ (X : Type) (lst1 lst2 : list X),
    length (lst1 ++ lst2) = (length lst1) + (length lst2).
Proof.
```

```
  intros; induction lst1;
    [reflexivity | simpl; rewrite IHlst1; reflexivity].
Qed.
```

The identity tactic `idtac` always succeeds without changing the proof state. We can use it to factor out `reflexivity` in the previous proof.

```
Theorem app_length' : ∀ (X : Type) (lst1 lst2 : list X),
    length (lst1 ++ lst2) = (length lst1) + (length lst2).
Proof.
  intros; induction lst1;
    [idtac | simpl; rewrite IHlst1];
    reflexivity.
Qed.
```

Exercise: 1 star, standard (notry_sequence) Prove the following theorem with a one-shot proof, but this time, do not use `try`.

```
Theorem add_assoc' : ∀ n m p : nat,
    n + (m + p) = (n + m) + p.
Proof. Admitted.
```

□

We can use the local form of the sequence tactical to give a slightly neater version of our optimization proof. Two lines change, as shown below with <===.

```
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 | s2 re1 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.

18.2.4 The repeat Tactical

The repeat tactical takes another tactic and keeps applying this tactic until it fails or stops making progress. Here is an example showing that 10 is in a long list:

```
Theorem In10 : In 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 In10' : In 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). Evaluation in Rocq's term language, Gallina, is guaranteed to terminate, but tactic evaluation is not. This does not affect Rocq's logical consistency, however, since the job of repeat and other tactics is to guide Rocq in constructing proofs. If the construction process diverges, it simply means that we have failed to construct a proof, not that we have constructed an incorrect proof.

Exercise: 1 star, standard (ev100) Prove that 100 is even. Your proof script should be quite short.

```
Theorem ev100: ev 100.
Proof. Admitted.
```

□

18.2.5 An Optimization Exercise

Exercise: 4 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 _ 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 ⇒ EmptyStr
  | Star re ⇒ Star (re_opt re)
  | EmptySet ⇒ EmptySet
  | EmptyStr ⇒ EmptyStr
  | Char x ⇒ Char x
  end.

Lemma re_opt_match : ∀ T (re: reg_exp T) s,
  s =~ re → s =~ 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 | s2 re1 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.
      × apply IH2.
      × apply IH2.
      × apply IH2.
      × apply IH2.
      × apply IH2.
      × 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.
+ 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' : ∀ T (re: reg_exp T) s,
  s =~ re → s =~ re_opt re.
Proof.
   Admitted.
Definition manual_grade_for_re_opt : option (nat×string) := None.
   □
```

18.3 Tactics that Make Mentioning Names Unnecessary

So far we have been dependent on knowing the names of hypotheses. For example, to prove the following simple theorem, we hardcode the name *HP*:

```
Theorem hyp_name : ∀ (P : Prop), P → P.
Proof.
  intros P HP. apply HP.
Qed.
```

We took the trouble to invent a name for *HP*, then we had to remember that name. If we later change the name in one place, we have to change it everywhere. Likewise, if we were to add new arguments to the theorem, we would have to adjust the `intros` list. That makes it challenging to maintain large proofs. So, Rocq provides several tactics that make it possible to write proof scripts that do not hardcode names.

18.3.1 The `assumption` tactic

The `assumption` tactic is useful to streamline the proof above. It looks through the hypotheses and, if it finds the goal as one them, it uses that to finish the proof.

```
Theorem no_hyp_name : ∀ (P : Prop), P → P.
Proof.
  intros. assumption.
Qed.
```

Some might argue to the contrary that hypothesis names improve self-documention of proof scripts. Maybe they do, sometimes. But in the case of the two proofs above, the first mentions unnecessary detail, whereas the second could be paraphrased simply as "the conclusion follows from the assumptions."

Anyway, unlike informal (good) mathematical proofs, Rocq proof scripts are generally not that illuminating to readers. Worries about rich, self-documenting names for hypotheses might be misplaced.

18.3.2 The *contradiction* tactic

The *contradiction* tactic handles some ad hoc situations where a hypothesis contains **False**, or two hypotheses derive **False**.

```
Theorem false_assumed : False → 0 = 1.
Proof.
  intros H. destruct H.
Qed.

Theorem false_assumed' : False → 0 = 1.
Proof.
  intros. contradiction.
```

```
Qed.

Theorem contras : ∀ (P : Prop), P → ¬P → 0 = 1.
Proof.
  intros P HP HNP. exfalso. apply HNP. apply HP.
Qed.

Theorem contras' : ∀ (P : Prop), P → ¬P → 0 = 1.
Proof.
  intros. contradiction.
Qed.
```

18.3.3 The `subst` tactic

The `subst` tactic substitutes away an identifier, replacing it everywhere and eliminating it from the context. That helps us to avoid naming hypotheses in `rewrite`s.

```
Theorem many_eq : ∀ (n m o p : nat),
  n = m →
  o = p →
  [n; o] = [m; p].
Proof.
  intros n m o p Hnm Hop. rewrite Hnm. rewrite Hop. reflexivity.
Qed.

Theorem many_eq' : ∀ (n m o p : nat),
  n = m →
  o = p →
  [n; o] = [m; p].
Proof.
  intros. subst. reflexivity.
Qed.
```

Actually there are two forms of this tactic.

- `subst x` finds an assumption $x = \texttt{e}$ or $\texttt{e} = x$ in the context, replaces x with `e` throughout the context and current goal, and removes the assumption from the context.
- `subst` substitutes away *all* assumptions of the form $x = \texttt{e}$ or $\texttt{e} = x$.

18.3.4 The `constructor` tactic

The `constructor` tactic tries to find a constructor `c` (from the appropriate `Inductive` definition in the current environment) that can be applied to solve the current goal.

```
Check ev_0 : ev 0.
Check ev_SS : ∀ n : nat, ev n → ev (S (S n)).
```

```
Example constructor_example: ∀ (n:nat),
    ev (n + n).
Proof.
  induction n; simpl.
  - constructor.   - rewrite add_comm. simpl. constructor.     assumption.
Qed.
```

Warning: if more than one constructor can apply, `constructor` picks the first one, in the order in which they were defined in the `Inductive` definition. That might not be the one you want.

18.4 Automatic Solvers

Rocq has several special-purpose tactics that can solve certain kinds of goals in a completely automated way. These tactics are based on sophisticated algorithms developed for verification in specific mathematical or logical domains.

Some automatic solvers are *decision procedures*, which are algorithms that always terminate, and always give a correct answer. Here, that means that they always find a correct proof, or correctly determine that the goal is invalid. Other automatic solvers are *incomplete*: they might fail to find a proof of a valid goal.

18.4.1 Linear Integer Arithmetic: The *lia* Tactic

The *lia* tactic implements a decision procedure for integer linear arithmetic, a subset of propositional logic and arithmetic. As input it accepts goals constructed as follows:

- variables and constants of type **nat**, Z, and other integer types;
- arithmetic operators +, -, ×, and ^;
- equality = and ordering <, >, ≤, ≥; and
- the logical connectives ∧, ∨, ¬, →, and ↔; and constants **True** and **False**.

Linear goals involve (in)equalities over expressions of the form `c1` × *x1* + ... + *cn* × *xn*, where *ci* are constants and *xi* are variables.

- For linear goals, *lia* will either solve the goal or fail, meaning that the goal is actually invalid.
- For non-linear goals, *lia* will also either solve the goal or fail. But in this case, the failure does not necessarily mean that the goal is invalid – it might just be beyond *lia*'s reach to prove because of non-linearity.

Also, *lia* will do `intros` as necessary.

```
From Stdlib Require Import Lia.

Theorem lia_succeed1 : ∀ (n : nat),
  n > 0 → n × 2 > n.
Proof. lia. Qed.

Theorem lia_succeed2 : ∀ (n m : nat),
    n × m = m × n.
Proof.
  lia. Qed.

Theorem lia_fail1 : 0 = 1.
Proof.
  Fail lia. Abort.

Theorem lia_fail2 : ∀ (n : nat),
    n ≥ 1 → 2 ^ n = 2 × 2 ^ (n - 1).
Proof.
  Fail lia. Abort.
```

There are other tactics that can solve arithmetic goals. The `ring` and `field` tactics, for example, can solve equations over the algebraic structures of *rings* and *fields*, from which the tactics get their names. These tactics do not do `intros`.

```
From Stdlib Require Import Ring.

Theorem mult_comm : ∀ (n m : nat),
    n × m = m × n.
Proof.
  intros n m. ring.
Qed.
```

18.4.2 Equalities: The `congruence` Tactic

The *lia* tactic makes use of facts about addition and multiplication to prove equalities. A more basic way of treating such formulas is to regard every function appearing in them as a black box: nothing is known about the function's behavior. Based on the properties of equality itself, it is still possible to prove some formulas. For example, $y = f\ x \to g\ y = g\ (f\ x)$, even if we know nothing about f or g:

```
Theorem eq_example1 :
  ∀ (A B C : Type) (f : A → B) (g : B → C) (x : A) (y : B),
    y = f x → g y = g (f x).
Proof.
  intros. rewrite H. reflexivity.
Qed.
```

The essential properties of equality are that it is:

- reflexive
- symmetric
- transitive
- a *congruence*: it respects function and predicate application.

It is that congruence property that we're using when we `rewrite` in the proof above: if `a = b` then `f a = f b`. (The ProofObjects chapter explores this idea further under the name "Leibniz equality".)

The `congruence` tactic is a decision procedure for equality with uninterpreted functions and other symbols.

```
Theorem eq_example1' :
  ∀ (A B C : Type) (f : A → B) (g : B → C) (x : A) (y : B),
    y = f x → g y = g (f x).
Proof.
  congruence.
Qed.
```

The `congruence` tactic is able to work with constructors, even taking advantage of their injectivity and distinctness.

```
Theorem eq_example2 : ∀ (n m o p : nat),
    n = m →
    o = p →
    (n, o) = (m, p).
Proof.
  congruence.
Qed.

Theorem eq_example3 : ∀ (X : Type) (h : X) (t : list X),
    [] ≠ h :: t.
Proof.
  congruence.
Qed.
```

18.4.3 Propositions: The `intuition` Tactic

A *tautology* is a logical formula that is always provable. A formula is *propositional* if it does not use quantifiers – or at least, if quantifiers do not have to be instantiated to carry out the proof. The `intuition` tactic implements a decision procedure for propositional tautologies in Rocq's constructive (that is, intuitionistic) logic. Even if a goal is not a propositional tautology, `intuition` will still attempt to reduce it to simpler subgoals.

```
Theorem intuition_succeed1 : ∀ (P : Prop),
```

```
    P → P.
Proof. intuition. Qed.

Theorem intuition_succeed2 : ∀ (P Q : Prop),
    ¬ (P ∨ Q) → ¬P ∧ ¬Q.
Proof. intuition. Qed.

Theorem intuition_simplify1 : ∀ (P : Prop),
    ~~P → P.
Proof.
  intuition. Abort.

Theorem intuition_simplify2 : ∀ (x y : nat) (P Q : nat → Prop),
  x = y ∧ (P x → Q x) ∧ P x → Q y.
Proof.
  Fail congruence.  intuition.  congruence.
Qed.
```

In the previous example, neither `congruence` nor `intuition` alone can solve the goal. But after `intuition` simplifies the propositions involved in the goal, `congruence` can succeed. For situations like this, `intuition` takes an optional argument, which is a tactic to apply to all the unsolved goals that `intuition` generated. Using that we can offer a shorter proof:

```
Theorem intuition_simplify2' : ∀ (x y : nat) (P Q : nat → Prop),
  x = y ∧ (P x → Q x) ∧ P x → Q y.
Proof.
  intuition congruence.
Qed.
```

18.4.4 Exercises with Automatic Solvers

Exercise: 2 stars, standard (automatic_solvers) The exercises below are gleaned from previous chapters, where they were proved with (relatively) long proof scripts. Each should now be provable with just a single invocation of an automatic solver.

```
Theorem plus_id_exercise_from_basics : ∀ n m o : nat,
  n = m → m = o → n + m = m + o.
Proof. Admitted.

Theorem add_assoc_from_induction : ∀ n m p : nat,
    n + (m + p) = (n + m) + p.
Proof. Admitted.

Theorem S_injective_from_tactics : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof. Admitted.

Theorem or_distributes_over_and_from_logic : ∀ P Q R : Prop,
```

```
    P ∨ (Q ∧ R) ↔ (P ∨ Q) ∧ (P ∨ R).
Proof. Admitted.
```

□

18.5 Search Tactics

The automated solvers we just discussed are capable of finding proofs in specific domains. Some of them might pay attention to local hypotheses, but overall they don't make use of any custom lemmas we've proved, or that are provided by libraries that we load.

Another kind of automation that Rocq provides does just that: the `auto` tactic and its variants search for proofs that can be assembled out of hypotheses and lemmas.

18.5.1 The `auto` Tactic

Until this chapter, our proof scripts mostly applied relevant hypotheses or lemmas by name, and one at a time.

```
Example auto_example_1 : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
  intros P Q R H1 H2 H3.
  apply H2. apply H1. apply H3.
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' : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → 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 : ∀ P Q R S T U : Prop,
  (P → Q) →
  (P → R) →
```

```
  (T → R) →
  (S → T → U) →
  ((P → Q) → (P → S)) →
  T →
  P →
  U.
Proof. auto. Qed.
```

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 : ∀ (P Q R S T U: Prop),
  (P → Q) →
  (Q → R) →
  (R → S) →
  (S → T) →
  (T → U) →
  P →
  U.
Proof.
  auto.
  auto 6.
Qed.
```

The `auto` tactic considers the hypotheses in the current context together with a *hint database* of other lemmas and constructors. Some common facts about equality and logical operators are installed in the hint database by default.

```
Example auto_example_4 : ∀ P Q R : 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 with theorem t just for the purposes of one application of `auto` by writing `auto using` t.

```
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 a hint database named *db* by writing

Create HintDb db.

to create the database, then

Hint Resolve T : db.

to add *T* to the database, where *T* is a top-level theorem or a constructor of an inductively defined proposition (i.e., anything whose type is an implication). We tell `auto` to use that database by writing `auto with db`. Technically creation of the database is optional; Rocq will create it automatically the first time we use `Hint`.

```
Create HintDb le_db.
Hint Resolve le_antisym : le_db.

Example auto_example_6' : ∀ n m p : nat,
  (n ≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
Proof.
  auto with le_db.
Qed.
```

As a shorthand, we can write

Hint Constructors c : db.

to tell Rocq 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 : db.

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.

```
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 : le_db.

Example auto_example_7' : ∀ x,
  (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
```

```
Proof. info_auto with le_db. Qed.
```

The "global" database that `auto` always uses is named *core*. You can add your own hints to it, but the Rocq manual discourages that, preferring instead to have specialized databases for specific developments. Many of the important libraries have their own hint databases that you can tag in: *arith*, **bool**, *datatypes* (including lists), etc.

```
Example auto_example_8 : ∀ (n m : nat),
    n + m = m + n.
Proof.
  auto.   info_auto with arith. Qed.
```

Exercise: 3 stars, standard (re_opt_match_auto) Use `auto` to shorten your proof of re_opt_match even more. Eliminate all uses of `apply`, thus removing the need to name specific constructors and lemmas about regular expressions. The number of lines of proof script won't decrease that much, because `auto` won't be able to find `induction`, `destruct`, or `inversion` opportunities by itself.

Hint: again, use a bottom-up approach. Always keep the proof compiling. You might find it easier to return to the original, very long proof, and shorten it, rather than starting with re_opt_match'; but, either way can work.

```
Lemma re_opt_match'' : ∀ T (re: reg_exp T) s,
  s =~ re → s =~ re_opt re.
Proof.
  Admitted.
Definition manual_grade_for_re_opt_match'' : option (nat×string) := None.
```
□

Exercise: 3 stars, advanced, optional (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 IndProp.

```
Import PUMPING.

Lemma weak_pumping : ∀ T (re : reg_exp T) s,
    s =~ re →
    pumping_constant re ≤ length s →
    ∃ s1 s2 s3 ,
      s = s1 ++ s2 ++ s3 ∧
        s2 ≠ [] ∧
        ∀ m, s1 ++ napp m s2 ++ s3 =~ 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 IndProp.

```
Lemma pumping : ∀ T (re : reg_exp T) s,
    s =~ re →
    pumping_constant re ≤ length s →
    ∃ s1 s2 s3,
      s = s1 ++ s2 ++ s3 ∧
        s2 ≠ [] ∧
        length s1 + length s2 ≤ pumping_constant re ∧
        ∀ m, s1 ++ napp m s2 ++ s3 =~ re.

Proof.
  Admitted.
Definition manual_grade_for_pumping_redux_strong : option (nat×string) := None.
```

□

18.5.2 The `eauto` variant

There is a variant of `auto` (and other tactics, such as `apply`) that makes it possible to delay instantiation of quantifiers. To motivate this feature, consider again this simple example:

```
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.
  apply Nat.le_trans with (b + b × c).
  - apply H1.
  - simpl in H2. rewrite mul_comm. apply H2.
Qed.
```

In the first step of the proof, we had to explicitly provide a longish expression to help Rocq 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 Rocq 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 Nat.le_trans.    - apply H1.    - simpl in H2. rewrite mul_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 ∃, `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 Nat.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`. Rocq 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.

18.6 Ltac: The Tactic Language

Most of the tactics we have been using are implemented in OCaml, where they are able to use an API to access Rocq's internal structures at a low level. But this is seldom worth the trouble for ordinary Rocq users.

Rocq has a high-level language called Ltac for programming new tactics in Rocq itself, without having to escape to OCaml. Actually we've been using Ltac all along – anytime we are in proof mode, we've been writing Ltac programs. At their most basic, those programs are just invocations of built-in tactics. The tactical constructs we learned at the beginning of this chapter are also part of Ltac.

What we turn to, next, is ways to use Ltac to reduce the amount of proof script we have to write ourselves.

18.6.1 Ltac Functions

Here is a simple `Ltac` example:

```
Ltac simpl_and_try tac := simpl; try tac.
```

This defines a new tactic called *simpl_and_try* that takes one tactic *tac* as an argument and is defined to be equivalent to `simpl; try tac`. Now writing "*simpl_and_try* `reflexivity`." in a proof will be the same as writing "`simpl; try reflexivity`."

```
Example sat_ex1 : 1 + 1 = 2.
Proof. simpl_and_try reflexivity. Qed.

Example sat_ex2 : ∀ (n : nat), 1 - 1 + n + 1 = 1 + n.
Proof. simpl_and_try reflexivity. lia. Qed.
```

Of course, that little tactic is not so useful. But it demonstrates that we can parameterize Ltac-defined tactics, and that their bodies are themselves tactics that will be run in the context of a proof. So Ltac can be used to create functions on tactics.

For a more useful tactic, consider these three proofs from Basics, and how structurally similar they all are:

```
Theorem plus_1_neq_0 : ∀ n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros n. destruct n.
  - reflexivity.
  - reflexivity.
Qed.

Theorem negb_involutive : ∀ b : bool,
  negb (negb b) = b.
Proof.
  intros b. destruct b.
  - reflexivity.
  - reflexivity.
Qed.

Theorem andb_commutative : ∀ b c, andb b c = andb c b.
Proof.
  intros b c. destruct b.
  - destruct c.
    + reflexivity.
    + reflexivity.
  - destruct c.
```

```
    + reflexivity.
    + reflexivity.
Qed.
```

We can factor out the common structure:

- Do a destruct.
- For each branch, finish with `reflexivity` – if possible.

```
Ltac destructpf x :=
  destruct x; try reflexivity.

Theorem plus_1_neq_0' : ∀ n : nat,
    (n + 1) =? 0 = false.
Proof. intros n; destructpf n. Qed.

Theorem negb_involutive' : ∀ b : bool,
  negb (negb b) = b.
Proof. intros b; destructpf b. Qed.

Theorem andb_commutative' : ∀ b c, andb b c = andb c b.
Proof.
  intros b c; destructpf b; destructpf c.
Qed.
```

Exercise: 1 star, standard (andb3_exchange) Re-prove the following theorem from Basics, using only `intros` and *destructpf*. You should have a one-shot proof.

```
Theorem andb3_exchange :
  ∀ b c d, andb (andb b c) d = andb (andb b d) c.
Proof. Admitted.
```

□

Exercise: 2 stars, standard (andb_true_elim2) The following theorem from Basics can't be proved with *destructpf*.

```
Theorem andb_true_elim2 : ∀ b c : bool,
  andb b c = true → c = true.
Proof.
  intros b c. destruct b eqn:Eb.
  - simpl. intros H. rewrite H. reflexivity.
  - simpl. intros H. destruct c eqn:Ec.
    + reflexivity.
    + rewrite H. reflexivity.
Qed.
```

Uncomment the definition of *destructpf'*, below, and define your own, improved version of *destructpf*. Use it to prove the theorem.

Your one-shot proof should need only `intros` and *destructpf'*.

```
Theorem andb_true_elim2' : ∀ b c : bool,
    andb b c = true → c = true.
Proof. Admitted.
```

Double-check that `intros` and your new *destructpf'* still suffice to prove this earlier theorem – i.e., that your improved tactic is general enough to still prove it in one shot:

```
Theorem andb3_exchange' :
  ∀ b c d, andb (andb b c) d = andb (andb b d) c.
Proof. Admitted.
```

□

18.6.2 Ltac Pattern Matching

Here is another common proof pattern that we have seen in many simple proofs by induction:

```
Theorem app_nil_r : ∀ (X : Type) (lst : list X),
    lst ++ [] = lst.
Proof.
  intros X lst. induction lst as [ | h t IHt].
  - reflexivity.
  - simpl. rewrite IHt. reflexivity.
Qed.
```

At the point we `rewrite`, we can't substitute away t: it is present on both sides of the equality in the inductive hypothesis $IHt : t ++ [] = t$. How can we pick out which hypothesis to rewrite in an Ltac tactic?

To solve this and other problems, Ltac contains a pattern-matching tactic `match goal`. It allows us to match against the *proof state* rather than against a program.

```
Theorem match_ex1 : True.
Proof.
  match goal with
  | [ ⊢ True ] ⇒ apply I
  end.
Qed.
```

The syntax is similar to a `match` in Gallina (Rocq's term language), but has some new features:

- The word `goal` here is a keyword, rather than an expression being matched. It means to match against the proof state, rather than a program term.

- The square brackets around the pattern can often be omitted, but they do make it easier to visually distinguish which part of the code is the pattern.
- The turnstile ⊢ separates the hypothesis patterns (if any) from the conclusion pattern. It represents the big horizontal line shown by your IDE in the proof state: the hypotheses are to the left of it, the conclusion is to the right.
- The hypotheses in the pattern need not completely describe all the hypotheses present in the proof state. It is fine for there to be additional hypotheses in the proof state that do not match any of the patterns. The point is for `match goal` to pick out particular hypotheses of interest, rather than fully specify the proof state.
- The right-hand side of a branch is a tactic to run, rather than a program term.

The single branch above therefore specifies to match a goal whose conclusion is the term **True** and whose hypotheses may be anything. If such a match occurs, it will run `apply I`.

There may be multiple branches, which are tried in order.

```
Theorem match_ex2 : True ∧ True.
Proof.
  match goal with
  | [ ⊢ True ] ⇒ apply I
  | [ ⊢ True ∧ True ] ⇒ split; apply I
  end.
Qed.
```

To see what branches are being tried, it can help to insert calls to the identity tactic `idtac`. It optionally accepts an argument to print out as debugging information.

```
Theorem match_ex2' : True ∧ True.
Proof.
  match goal with
  | [ ⊢ True ] ⇒ idtac "branch 1"; apply I
  | [ ⊢ True ∧ True ] ⇒ idtac "branch 2"; split; apply I
  end.
Qed.
```

Only the second branch was tried. The first one did not match the goal.

The semantics of the tactic `match goal` have a big difference with the semantics of the term `match`. With the latter, the first matching pattern is chosen, and later branches are never considered. In fact, an error is produced if later branches are known to be redundant.

```
Fail Definition redundant_match (n : nat) : nat :=
  match n with
  | x ⇒ x
  | 0 ⇒ 1
  end.
```

But with `match goal`, if the tactic for the branch fails, pattern matching continues with the next branch, until a branch succeeds, or all branches have failed.

```
Theorem match_ex2'' : True ∧ True.
Proof.
  match goal with
  | [ ⊢ _ ] ⇒ idtac "branch 1"; apply I
  | [ ⊢ True ∧ True ] ⇒ idtac "branch 2"; split; apply I
  end.
Qed.
```

The first branch was tried but failed, then the second branch was tried and succeeded. If all the branches fail, the `match goal` fails.

```
Theorem match_ex2''' : True ∧ True.
Proof.
  Fail match goal with
  | [ ⊢ _ ] ⇒ idtac "branch 1"; apply I
  | [ ⊢ _ ] ⇒ idtac "branch 2"; apply I
  end.
Abort.
```

Next, let's try matching against hypotheses. We can bind a hypothesis name, as with H below, and use that name on the right-hand side of the branch.

```
Theorem match_ex3 : ∀ (P : Prop), P → P.
Proof.
  intros P HP.
  match goal with
  | [ H : _ ⊢ _ ] ⇒ apply H
  end.
Qed.
```

The actual name of the hypothesis is of course HP, but the pattern binds it as H. Using `idtac`, we can even observe the actual name: stepping through the following proof causes "HP" to be printed.

```
Theorem match_ex3' : ∀ (P : Prop), P → P.
Proof.
  intros P HP.
  match goal with
  | [ H : _ ⊢ _ ] ⇒ idtac H; apply H
  end.
Qed.
```

We'll keep using `idtac` for awhile to observe the behavior of `match goal`, but, note that it isn't necessary for the successful proof of any of the following examples.

If there are multiple hypotheses that match, which one does Ltac choose? Here is a big difference with regular `match` against terms: Ltac will try all possible matches until one succeeds (or all have failed).

```
Theorem match_ex4 : ∀ (P Q : Prop), P → Q → P.
Proof.
  intros P Q HP HQ.
  match goal with
  | [ H : _ ⊢ _ ] ⇒ idtac H; apply H
  end.
Qed.
```

That example prints "HQ" followed by "HP". Ltac first matched against the most recently introduced hypothesis *HQ* and tried applying it. That did not solve the goal. So Ltac *backtracks* and tries the next most-recent matching hypothesis, which is *HP*. Applying that does succeed.

But if there were no successful hypotheses, the entire match would fail:

```
Theorem match_ex5 : ∀ (P Q R : Prop), P → Q → R.
Proof.
  intros P Q R HP HQ.
  Fail match goal with
  | [ H : _ ⊢ _ ] ⇒ idtac H; apply H
  end.
Abort.
```

So far we haven't been very demanding in how to match hypotheses. The *wildcard* (aka *joker*) pattern we've used matches everything. We could be more specific by using *metavariables*:

```
Theorem match_ex5 : ∀ (P Q : Prop), P → Q → P.
Proof.
  intros P Q HP HQ.
  match goal with
  | [ H : ?X ⊢ ?X ] ⇒ idtac H; apply H
  end.
Qed.
```

Note that this time, the only hypothesis printed by `idtac` is *HP*. The *HQ* hypothesis is ruled out, because it does not have the form ?X ⊢ ?X.

The occurrences of ?X in that pattern are as *metavariables* that stand for the same term appearing both as the type of hypothesis *H* and as the conclusion of the goal.

(The syntax of `match goal` requires that ? to distinguish metavariables from other identifiers that might be in scope. However, the ? is used only in the pattern. On the right-hand side of the branch, it's actually required to drop the ?.)

Now we have seen yet another difference between `match goal` and regular `match` against terms: `match goal` allows a metavariable to be used multiple times in a pattern, each time

standing for the same term. The regular `match` does not allow that:

```
Fail Definition dup_first_two_elts (lst : list nat) :=
  match lst with
  | x :: x :: _ ⇒ true
  | _ ⇒ false
  end.
```

The technical term for this is *linearity*: regular `match` requires pattern variables to be *linear*, meaning that they are used only once. Tactic `match goal` permits *non-linear* metavariables, meaning that they can be used multiple times in a pattern and must bind the same term each time.

Now that we've learned a bit about `match goal`, let's return to the proof pattern of some simple inductions:

```
Theorem app_nil_r' : ∀ (X : Type) (lst : list X),
    lst ++ [] = lst.
Proof.
  intros X lst. induction lst as [ | h t IHt].
  - reflexivity.
  - simpl. rewrite IHt. reflexivity.
Qed.
```

With `match goal`, we can automate that proof pattern:

```
Ltac simple_induction t :=
  induction t; simpl;
  try match goal with
      | [H : _ = _ ⊢ _] ⇒ rewrite H
      end;
  reflexivity.

Theorem app_nil_r'' : ∀ (X : Type) (lst : list X),
    lst ++ [] = lst.
Proof.
  intros X lst. simple_induction lst.
Qed.
```

That works great! Here are two other proofs that follow the same pattern.

```
Theorem add_assoc'' : ∀ n m p : nat,
    n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n.
  - reflexivity.
  - simpl. rewrite IHn. reflexivity.
Qed.

Theorem add_assoc''' : ∀ n m p : nat,
```

```
    n + (m + p) = (n + m) + p.
Proof.
  intros n m p. simple_induction n.
Qed.

Theorem plus_n_Sm : ∀ n m : nat,
    S (n + m) = n + (S m).
Proof.
  intros n m. induction n.
  - reflexivity.
  - simpl. rewrite IHn. reflexivity.
Qed.

Theorem plus_n_Sm' : ∀ n m : nat,
    S (n + m) = n + (S m).
Proof.
  intros n m. simple_induction n.
Qed.
```

18.6.3 Using `match goal` to Prove Tautologies

The Ltac source code of `intuition` can be found in the GitHub repo for Rocq in *theories/Init/Tauto.v*. At heart, it is a big loop that runs `match goal` to find propositions it can `apply` and `destruct`.

Let's build our own simplified "knock off" of `intuition`. Here's a start on implication:

```
Ltac imp_intuition :=
  repeat match goal with
         | [ H : ?P ⊢ ?P ] ⇒ apply H
         | [ ⊢ ∀ _, _ ] ⇒ intro
         | [ H1 : ?P → ?Q, H2 : ?P ⊢ _ ] ⇒ apply H1 in H2
         end.
```

That tactic repeatedly matches against the goal until the match fails to make progress. At each step, the `match goal` does one of three things:

- Finds that the conclusion is already in the hypotheses, in which case the goal is finished.
- Finds that the conclusion is a quantification, in which case it is introduced. Since implication $P \to Q$ is itself a quantification $\forall$ (_ : P), Q, this case handles introduction of implications, too.
- Finds that two formulas of the form $?P \to ?Q$ and $?P$ are in the hypotheses. This is the first time we've seen an example of matching against two hypotheses simultaneously. Note that the metavariable $?P$ is once more non-linear: the same formula must occur in two different hypotheses. In this case, the tactic uses forward reasoning to change hypothesis *H2* into $?Q$.

Already we can prove many theorems with this tactic:

```
Example imp1 : ∀ (P : Prop), P → P.
Proof. imp_intuition. Qed.

Example imp2 : ∀ (P Q : Prop), P → (P → Q) → Q.
Proof. imp_intuition. Qed.

Example imp3 : ∀ (P Q R : Prop), (P → Q → R) → (Q → P → R).
Proof. imp_intuition. Qed.
```

Suppose we were to add a new logical connective: **nor**, the "not or" connective.

```
Inductive nor (P Q : Prop) :=
| stroke : ¬P → ¬Q → nor P Q.
```

Classically, **nor** P Q would be equivalent to $\tilde{}(P \vee Q)$. But constructively, only one direction of that is provable.

```
Theorem nor_not_or : ∀ (P Q : Prop),
    nor P Q → ¬ (P ∨ Q).
Proof.
  intros. destruct H. unfold not. intros. destruct H. auto. auto.
Qed.
```

Some other usual theorems about **nor** are still provable, though.

```
Theorem nor_comm : ∀ (P Q : Prop),
    nor P Q ↔ nor Q P.
Proof.
  intros P Q. split.
  - intros H. destruct H. apply stroke; assumption.
  - intros H. destruct H. apply stroke; assumption.
Qed.

Theorem nor_not : ∀ (P : Prop),
    nor P P ↔ ¬P.
Proof.
  intros P. split.
  - intros H. destruct H. assumption.
  - intros H. apply stroke; assumption.
Qed.
```

Exercise: 4 stars, advanced (nor_intuition) Create your own tactic *nor_intuition*. It should be able to prove the three theorems above – *nor_not_and*, nor_comm, and nor_not – fully automatically. You may not use `intuition` or any other automated solvers in your solution.

Begin by copying the code from *imp_intuition*. You will then need to expand it to handle conjunctions, negations, bi-implications, and **nor**.

Each of the three theorems below, and many others involving these logical connectives, should be provable with just `Proof. nor_intuition. Qed.`

```
Theorem nor_comm' : ∀ (P Q : Prop),
    nor P Q ↔ nor Q P.
Proof. Admitted.

Theorem nor_not' : ∀ (P : Prop),
    nor P P ↔ ¬P.
Proof. Admitted.

Theorem nor_not_and' : ∀ (P Q : Prop),
    nor P Q → ¬ (P ∧ Q).
Proof. Admitted.
Definition manual_grade_for_nor_intuition : option (nat×string) := None.
```

□

18.7 Review

We've learned a lot of new features and tactics in this chapter:

- `try`, ;, `repeat`
- `assumption`, *contradiction*, `subst`, `constructor`
- *lia*, `congruence`, `intuition`
- `auto`, `eauto`, `eapply`
- Ltac functions and `match goal`

Chapter 19

Library LF.Postscript

19.1 Postscript

Congratulations: We've made it to the end of *Logical Foundations*!

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
- *Rocq*, 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 several choices for further reading in later volumes of the *Software Foundations* series. Some of these are intended to be accessible to readers immediately after finishing *Logical Foundations*; others require a few chapters from Volume 2, *Programming Language Foundations*. The Preface chapter in each volume gives details about prerequisites.

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 Rocq, the #*coq* area of Stack Overflow ({https://stackoverflow.com/questions/tag 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 Rocq:
  - 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 Rocq 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 Stdlib 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 "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 Stdlib 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 ((@eq bool (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 ((@eq bool (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 ((@eq nat (factorial 5) (Nat.mul 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 ((@eq bool (ltb 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 (
(∀ (n m o : nat) (_ : @eq nat n m) (_ : @eq nat m o),
 @eq nat (Nat.add n m) (Nat.add 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 ((∀ p : nat, @eq nat (Nat.mul p 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 (
(∀ (b c : bool) (_ : @eq bool (andb b c) true), @eq bool c 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 ((∀ n : nat, @eq bool (eqb 0 (Nat.add n 1)) 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, @eq bool (f x) x)
   (b : bool),
 @eq 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 "————————- letter_comparison ————————".
idtac " ".

idtac "#> LateDays.letter_comparison_Eq".
idtac "Possible points: 1".
check_type @LateDays.letter_comparison_Eq (
(∀ l : LateDays.letter,
 @eq LateDays.comparison (LateDays.letter_comparison l l) LateDays.Eq)).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.letter_comparison_Eq.
Goal True.
idtac " ".

idtac "————————- grade_comparison ————————".
idtac " ".

idtac "#> LateDays.test_grade_comparison1".
idtac "Possible points: 0.5".
check_type @LateDays.test_grade_comparison1 (
(@eq LateDays.comparison
   (LateDays.grade_comparison (LateDays.Grade LateDays.A LateDays.Minus)
      (LateDays.Grade LateDays.B LateDays.Plus))
```

```
    LateDays.Gt)).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.test_grade_comparison1.
Goal True.
idtac " ".

idtac "#> LateDays.test_grade_comparison2".
idtac "Possible points: 0.5".
check_type @LateDays.test_grade_comparison2 (
(@eq LateDays.comparison
    (LateDays.grade_comparison (LateDays.Grade LateDays.A LateDays.Minus)
        (LateDays.Grade LateDays.A LateDays.Plus))
    LateDays.Lt)).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.test_grade_comparison2.
Goal True.
idtac " ".

idtac "#> LateDays.test_grade_comparison3".
idtac "Possible points: 0.5".
check_type @LateDays.test_grade_comparison3 (
(@eq LateDays.comparison
    (LateDays.grade_comparison (LateDays.Grade LateDays.F LateDays.Plus)
        (LateDays.Grade LateDays.F LateDays.Plus))
    LateDays.Eq)).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.test_grade_comparison3.
Goal True.
idtac " ".

idtac "#> LateDays.test_grade_comparison4".
idtac "Possible points: 0.5".
check_type @LateDays.test_grade_comparison4 (
(@eq LateDays.comparison
    (LateDays.grade_comparison (LateDays.Grade LateDays.B LateDays.Minus)
        (LateDays.Grade LateDays.C LateDays.Plus))
    LateDays.Gt)).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.test_grade_comparison4.
Goal True.
idtac " ".
```

```
idtac "————————————- lower_letter_lowers ————————————".
idtac " ".

idtac "#> LateDays.lower_letter_lowers".
idtac "Possible points: 2".
check_type @LateDays.lower_letter_lowers (
(∀ (l : LateDays.letter)
    (_ : @eq LateDays.comparison (LateDays.letter_comparison LateDays.F l)
            LateDays.Lt),
 @eq LateDays.comparison
    (LateDays.letter_comparison (LateDays.lower_letter l) l) LateDays.Lt)).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_letter_lowers.
Goal True.
idtac " ".

idtac "————————————- lower_grade ————————————".
idtac " ".

idtac "#> LateDays.lower_grade_A_Plus".
idtac "Possible points: 0.25".
check_type @LateDays.lower_grade_A_Plus (
(@eq LateDays.grade
    (LateDays.lower_grade (LateDays.Grade LateDays.A LateDays.Plus))
    (LateDays.Grade LateDays.A LateDays.Natural))).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_grade_A_Plus.
Goal True.
idtac " ".

idtac "#> LateDays.lower_grade_A_Natural".
idtac "Possible points: 0.25".
check_type @LateDays.lower_grade_A_Natural (
(@eq LateDays.grade
    (LateDays.lower_grade (LateDays.Grade LateDays.A LateDays.Natural))
    (LateDays.Grade LateDays.A LateDays.Minus))).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_grade_A_Natural.
Goal True.
idtac " ".

idtac "#> LateDays.lower_grade_A_Minus".
idtac "Possible points: 0.25".
```

```
check_type @LateDays.lower_grade_A_Minus (
(@eq LateDays.grade
   (LateDays.lower_grade (LateDays.Grade LateDays.A LateDays.Minus))
   (LateDays.Grade LateDays.B LateDays.Plus))).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_grade_A_Minus.
Goal True.
idtac " ".

idtac "#> LateDays.lower_grade_B_Plus".
idtac "Possible points: 0.25".
check_type @LateDays.lower_grade_B_Plus (
(@eq LateDays.grade
   (LateDays.lower_grade (LateDays.Grade LateDays.B LateDays.Plus))
   (LateDays.Grade LateDays.B LateDays.Natural))).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_grade_B_Plus.
Goal True.
idtac " ".

idtac "#> LateDays.lower_grade_F_Natural".
idtac "Possible points: 0.25".
check_type @LateDays.lower_grade_F_Natural (
(@eq LateDays.grade
   (LateDays.lower_grade (LateDays.Grade LateDays.F LateDays.Natural))
   (LateDays.Grade LateDays.F LateDays.Minus))).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_grade_F_Natural.
Goal True.
idtac " ".

idtac "#> LateDays.lower_grade_twice".
idtac "Possible points: 0.25".
check_type @LateDays.lower_grade_twice (
(@eq LateDays.grade
   (LateDays.lower_grade
      (LateDays.lower_grade (LateDays.Grade LateDays.B LateDays.Minus)))
   (LateDays.Grade LateDays.C LateDays.Natural))).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_grade_twice.
Goal True.
```

```
idtac " ".

idtac "#> LateDays.lower_grade_thrice".
idtac "Possible points: 0.25".
check_type @LateDays.lower_grade_thrice (
(@eq LateDays.grade
    (LateDays.lower_grade
        (LateDays.lower_grade
            (LateDays.lower_grade (LateDays.Grade LateDays.B LateDays.Minus))))
    (LateDays.Grade LateDays.C LateDays.Minus))).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_grade_thrice.
Goal True.
idtac " ".

idtac "#> LateDays.lower_grade_F_Minus".
idtac "Possible points: 0.25".
check_type @LateDays.lower_grade_F_Minus (
(@eq LateDays.grade
    (LateDays.lower_grade (LateDays.Grade LateDays.F LateDays.Minus))
    (LateDays.Grade LateDays.F LateDays.Minus))).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_grade_F_Minus.
Goal True.
idtac " ".

idtac "————————- lower_grade_lowers ————————".
idtac " ".

idtac "#> LateDays.lower_grade_lowers".
idtac "Possible points: 3".
check_type @LateDays.lower_grade_lowers (
(∀ (g : LateDays.grade)
    (_ : @eq LateDays.comparison
            (LateDays.grade_comparison
                (LateDays.Grade LateDays.F LateDays.Minus) g)
            LateDays.Lt),
 @eq LateDays.comparison
    (LateDays.grade_comparison (LateDays.lower_grade g) g) LateDays.Lt)).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.lower_grade_lowers.
Goal True.
```

```
idtac " ".
idtac "———————- no_penalty_for_mostly_on_time ———————".
idtac " ".
idtac "#> LateDays.no_penalty_for_mostly_on_time".
idtac "Possible points: 2".
check_type @LateDays.no_penalty_for_mostly_on_time (
(∀ (late_days : nat) (g : LateDays.grade)
    (_ : @eq bool (ltb late_days 9) true),
 @eq LateDays.grade (LateDays.apply_late_policy late_days g) g)).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.no_penalty_for_mostly_on_time.
Goal True.
idtac " ".
idtac "———————- graded_lowered_once ———————".
idtac " ".
idtac "#> LateDays.grade_lowered_once".
idtac "Possible points: 2".
check_type @LateDays.grade_lowered_once (
(∀ (late_days : nat) (g : LateDays.grade)
    (_ : @eq bool (ltb late_days 9) false)
    (_ : @eq bool (ltb late_days 17) true),
 @eq LateDays.grade (LateDays.apply_late_policy late_days g)
    (LateDays.lower_grade g))).
idtac "Assumptions:".
Abort.
Print Assumptions LateDays.grade_lowered_once.
Goal True.
idtac " ".
idtac "———————- binary ———————".
idtac " ".
idtac "#> test_bin_incr1".
idtac "Possible points: 0.5".
check_type @test_bin_incr1 ((@eq bin (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 ((@eq bin (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 ((@eq bin (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 ((@eq nat (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 (
(@eq nat (bin_to_nat (incr (B1 Z))) (Nat.add 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 (
(@eq nat (bin_to_nat (incr (incr (B1 Z)))) (Nat.add 2 (bin_to_nat (B1 Z))))).
idtac "Assumptions:".
Abort.
Print Assumptions test_bin_incr6.
Goal True.
idtac " ".

idtac " ".
```

```
idtac "Max points - standard: 28".
idtac "Max points - advanced: 28".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 "———— LateDays.letter_comparison_Eq ————".
Print Assumptions LateDays.letter_comparison_Eq.
idtac "———— LateDays.test_grade_comparison1 ————".
Print Assumptions LateDays.test_grade_comparison1.
idtac "———— LateDays.test_grade_comparison2 ————".
Print Assumptions LateDays.test_grade_comparison2.
idtac "———— LateDays.test_grade_comparison3 ————".
Print Assumptions LateDays.test_grade_comparison3.
idtac "———— LateDays.test_grade_comparison4 ————".
Print Assumptions LateDays.test_grade_comparison4.
idtac "———— LateDays.lower_letter_lowers ————".
Print Assumptions LateDays.lower_letter_lowers.
idtac "———— LateDays.lower_grade_A_Plus ————".
Print Assumptions LateDays.lower_grade_A_Plus.
idtac "———— LateDays.lower_grade_A_Natural ————".
Print Assumptions LateDays.lower_grade_A_Natural.
idtac "———— LateDays.lower_grade_A_Minus ————".
Print Assumptions LateDays.lower_grade_A_Minus.
idtac "———— LateDays.lower_grade_B_Plus ————".
Print Assumptions LateDays.lower_grade_B_Plus.
idtac "———— LateDays.lower_grade_F_Natural ————".
Print Assumptions LateDays.lower_grade_F_Natural.
idtac "———— LateDays.lower_grade_twice ————".
Print Assumptions LateDays.lower_grade_twice.
idtac "———— LateDays.lower_grade_thrice ————".
Print Assumptions LateDays.lower_grade_thrice.
idtac "———— LateDays.lower_grade_F_Minus ————".
Print Assumptions LateDays.lower_grade_F_Minus.
idtac "———— LateDays.lower_grade_lowers ————".
Print Assumptions LateDays.lower_grade_lowers.
idtac "———— LateDays.no_penalty_for_mostly_on_time ————".
Print Assumptions LateDays.no_penalty_for_mostly_on_time.
idtac "———— LateDays.grade_lowered_once ————".
Print Assumptions LateDays.grade_lowered_once.
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 Stdlib 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 "#> mul_0_r".
idtac "Possible points: 0.5".
check_type @mul_0_r ((∀ n : nat, @eq nat (Nat.mul n 0) 0)).
idtac "Assumptions:".
Abort.
Print Assumptions mul_0_r.
Goal True.
idtac " ".

idtac "#> plus_n_Sm".
idtac "Possible points: 0.5".
check_type @plus_n_Sm ((∀ n m : nat, @eq nat (S (Nat.add n m)) (Nat.add n (S m)))).
idtac "Assumptions:".
Abort.
Print Assumptions plus_n_Sm.
Goal True.
idtac " ".

idtac "#> add_comm".
idtac "Possible points: 0.5".
check_type @add_comm ((∀ n m : nat, @eq nat (Nat.add n m) (Nat.add m n))).
idtac "Assumptions:".
Abort.
Print Assumptions add_comm.
Goal True.
idtac " ".

idtac "#> add_assoc".
idtac "Possible points: 0.5".
check_type @add_assoc (
(∀ n m p : nat,
 @eq nat (Nat.add n (Nat.add m p)) (Nat.add (Nat.add n m) p))).
idtac "Assumptions:".
Abort.
Print Assumptions add_assoc.
Goal True.
idtac " ".

idtac "———————- double_plus ———————".
idtac " ".

idtac "#> double_plus".
idtac "Possible points: 2".
check_type @double_plus ((∀ n : nat, @eq nat (double n) (Nat.add n n))).
idtac "Assumptions:".
```

```
Abort.
Print Assumptions double_plus.
Goal True.
idtac " ".
idtac "———————- eqb_refl ———————".
idtac " ".
idtac "#> eqb_refl".
idtac "Possible points: 2".
check_type @eqb_refl ((∀ n : nat, @eq bool (eqb n n) true)).
idtac "Assumptions:".
Abort.
Print Assumptions eqb_refl.
Goal True.
idtac " ".
idtac "———————- mul_comm ———————".
idtac " ".
idtac "#> add_shuffle3".
idtac "Possible points: 1".
check_type @add_shuffle3 (
(∀ n m p : nat,
 @eq nat (Nat.add n (Nat.add m p)) (Nat.add m (Nat.add n p)))).
idtac "Assumptions:".
Abort.
Print Assumptions add_shuffle3.
Goal True.
idtac " ".
idtac "#> mul_comm".
idtac "Possible points: 2".
check_type @mul_comm ((∀ m n : nat, @eq nat (Nat.mul m n) (Nat.mul n m))).
idtac "Assumptions:".
Abort.
Print Assumptions mul_comm.
Goal True.
idtac " ".
idtac "———————- binary_commute ———————".
idtac " ".
idtac "#> bin_to_nat_pres_incr".
idtac "Possible points: 3".
check_type @bin_to_nat_pres_incr (
(∀ b : bin, @eq nat (bin_to_nat (incr b)) (Nat.add 1 (bin_to_nat b)))).
idtac "Assumptions:".
```

```
Abort.
Print Assumptions bin_to_nat_pres_incr.
Goal True.
idtac " ".

idtac "————————- nat_bin_nat ————————".
idtac " ".

idtac "#> nat_bin_nat".
idtac "Possible points: 3".
check_type @nat_bin_nat ((∀ n : nat, @eq nat (bin_to_nat (nat_to_bin n)) n)).
idtac "Assumptions:".
Abort.
Print Assumptions nat_bin_nat.
Goal True.
idtac " ".

idtac "————————- double_bin ————————".
idtac " ".

idtac "#> double_incr".
idtac "Advanced".
idtac "Possible points: 0.5".
check_type @double_incr ((∀ n : nat, @eq nat (double (S n)) (S (S (double n))))).
idtac "Assumptions:".
Abort.
Print Assumptions double_incr.
Goal True.
idtac " ".

idtac "#> double_bin_zero".
idtac "Advanced".
idtac "Possible points: 0.5".
check_type @double_bin_zero ((@eq bin (double_bin Z) Z)).
idtac "Assumptions:".
Abort.
Print Assumptions double_bin_zero.
Goal True.
idtac " ".

idtac "#> double_incr_bin".
idtac "Advanced".
idtac "Possible points: 1".
check_type @double_incr_bin (
(∀ b : bin, @eq bin (double_bin (incr b)) (incr (incr (double_bin b))))).
idtac "Assumptions:".
Abort.
```

```
Print Assumptions double_incr_bin.
Goal True.
idtac " ".
idtac "———————- bin_nat_bin ———————".
idtac " ".
idtac "#> bin_nat_bin".
idtac "Advanced".
idtac "Possible points: 6".
check_type @bin_nat_bin (
(∀ b : bin, @eq bin (nat_to_bin (bin_to_nat b)) (normalize b))).
idtac "Assumptions:".
Abort.
Print Assumptions bin_nat_bin.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 15".
idtac "Max points - advanced: 23".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 "———— mul_0_r ————".
Print Assumptions mul_0_r.
idtac "———— plus_n_Sm ————".
Print Assumptions plus_n_Sm.
idtac "———— add_comm ————".
Print Assumptions add_comm.
idtac "———— add_assoc ————".
Print Assumptions add_assoc.
idtac "———— double_plus ————".
Print Assumptions double_plus.
idtac "———— eqb_refl ————".
Print Assumptions eqb_refl.
idtac "———— add_shuffle3 ————".
Print Assumptions add_shuffle3.
idtac "———— mul_comm ————".
Print Assumptions mul_comm.
idtac "———— bin_to_nat_pres_incr ————".
Print Assumptions bin_to_nat_pres_incr.
idtac "———— nat_bin_nat ————".
Print Assumptions nat_bin_nat.
idtac "".
idtac "********** Advanced **********".
idtac "———— double_incr ————".
Print Assumptions double_incr.
idtac "———— double_bin_zero ————".
Print Assumptions double_bin_zero.
idtac "———— double_incr_bin ————".
Print Assumptions double_incr_bin.
idtac "———— bin_nat_bin ————".
Print Assumptions bin_nat_bin.
Abort.
```

Chapter 24

Library LF.ListsTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib 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 "————————- snd_fst_is_swap ————————".
```

```
idtac " ".

idtac "#> NatList.snd_fst_is_swap".
idtac "Possible points: 1".
check_type @NatList.snd_fst_is_swap (
(∀ p : NatList.natprod,
 @eq 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 (
(@eq NatList.natlist
   (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 (
(@eq NatList.natlist
   (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 (
(@eq nat
    (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 (
(@eq nat (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 (
(@eq NatList.natlist
    (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 (
(@eq NatList.natlist
    (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 (
(@eq NatList.natlist
    (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 (
(@eq nat
    (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 (
(@eq nat
    (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 (
(@eq nat
    (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 (
(@eq nat
    (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 (
(@eq bool
   (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 (
(@eq bool
   (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 "———————- list_exercises ———————".
idtac " ".

idtac "#> NatList.app_nil_r".
idtac "Possible points: 0.5".
check_type @NatList.app_nil_r (
(∀ l : NatList.natlist,
 @eq 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,
 @eq 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,
 @eq 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,
 @eq 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,
 @eq 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, @eq bool 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,
 @eq bool (leb 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,
 @eq bool
    (leb (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 "————————————- involution_injective ————————————".
idtac " ".

idtac "#> NatList.involution_injective".
idtac "Advanced".
idtac "Possible points: 3".
check_type @NatList.involution_injective (
(∀ (f : ∀ _ : nat, nat) (_ : ∀ n : nat, @eq nat n (f (f n)))
    (n1 n2 : nat) (_ : @eq nat (f n1) (f n2)),
 @eq nat n1 n2)).
idtac "Assumptions:".
Abort.
Print Assumptions NatList.involution_injective.
Goal True.
idtac " ".

idtac "————————————- rev_injective ————————————".
idtac " ".

idtac "#> NatList.rev_injective".
idtac "Advanced".
idtac "Possible points: 2".
check_type @NatList.rev_injective (
(∀ (l1 l2 : NatList.natlist)
    (_ : @eq NatList.natlist (NatList.rev l1) (NatList.rev l2)),
 @eq NatList.natlist 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 (
(@eq NatList.natoption (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 (
(@eq NatList.natoption (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 ((∀ x : id, @eq bool (eqb_id x x) true)).
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 (
(∀ (d : PartialMap.partial_map) (x : id) (v : nat),
 @eq NatList.natoption (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 (
(∀ (d : PartialMap.partial_map) (x y : id) (o : nat)
    (_ : @eq bool (eqb_id x y) false),
 @eq NatList.natoption (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: 17".
idtac "Max points - advanced: 28".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 "------- 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.involution_injective ————".
Print Assumptions NatList.involution_injective.
idtac "———— NatList.rev_injective ————".
Print Assumptions NatList.rev_injective.
Abort.
```

Chapter 25

Library LF.PolyTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib 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 "————————- poly_exercises ————————".
```

```
idtac " ".

idtac "#> app_nil_r".
idtac "Possible points: 0.5".
check_type @app_nil_r (
(∀ (X : Type) (l : list X), @eq (list X) (@app X l (@nil X)) l)).
idtac "Assumptions:".
Abort.
Print Assumptions app_nil_r.
Goal True.
idtac " ".

idtac "#> app_assoc".
idtac "Possible points: 1".
check_type @app_assoc (
(∀ (A : Type) (l m n : list A),
 @eq (list A) (@app A l (@app A m n)) (@app A (@app A l m) n))).
idtac "Assumptions:".
Abort.
Print Assumptions app_assoc.
Goal True.
idtac " ".

idtac "#> app_length".
idtac "Possible points: 0.5".
check_type @app_length (
(∀ (X : Type) (l1 l2 : list X),
 @eq nat (@length X (@app X l1 l2)) (Nat.add (@length X l1) (@length X l2)))).
idtac "Assumptions:".
Abort.
Print Assumptions app_length.
Goal True.
idtac " ".

idtac "———————- more_poly_exercises ———————".
idtac " ".

idtac "#> rev_app_distr".
idtac "Possible points: 1".
check_type @rev_app_distr (
(∀ (X : Type) (l1 l2 : list X),
 @eq (list X) (@rev X (@app X l1 l2)) (@app X (@rev X l2) (@rev X l1)))).
idtac "Assumptions:".
Abort.
Print Assumptions rev_app_distr.
Goal True.
```

```
idtac " ".

idtac "#> rev_involutive".
idtac "Possible points: 1".
check_type @rev_involutive (
(∀ (X : Type) (l : list X), @eq (list X) (@rev X (@rev X l)) l)).
idtac "Assumptions:".
Abort.
Print Assumptions rev_involutive.
Goal True.
idtac " ".

idtac "———————- split ———————".
idtac " ".

idtac "#> split".
idtac "Possible points: 1".
check_type @split ((∀ (X Y : Type) (_ : list (prod X Y)), prod (list X) (list Y))).
idtac "Assumptions:".
Abort.
Print Assumptions split.
Goal True.
idtac " ".

idtac "#> test_split".
idtac "Possible points: 1".
check_type @test_split (
(@eq (prod (list nat) (list bool))
    (@split nat bool
        (@cons (prod nat bool) (@pair nat bool 1 false)
            (@cons (prod nat bool) (@pair nat bool 2 false)
                (@nil (prod nat bool)))))
    (@pair (list nat) (list bool) (@cons nat 1 (@cons nat 2 (@nil nat)))
        (@cons bool false (@cons bool false (@nil bool)))))).
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 (
(@eq (list nat)
```

```
    (filter_even_gt7
        (@cons nat 1
            (@cons nat 2
                (@cons nat 6
                    (@cons nat 9
                        (@cons nat 10
                            (@cons nat 3 (@cons nat 12 (@cons nat 8 (@nil nat))))))))))
    (@cons nat 10 (@cons nat 12 (@cons nat 8 (@nil nat))))).
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 (
(@eq (list nat)
    (filter_even_gt7
        (@cons nat 5
            (@cons nat 2 (@cons nat 6 (@cons nat 19 (@cons nat 129 (@nil nat)))))))
    (@nil nat))).
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 (
(∀ (X : Type) (_ : ∀ _ : X, bool) (_ : list X),
 prod (list X) (list X))).
idtac "Assumptions:".
Abort.
Print Assumptions partition.
Goal True.
idtac " ".

idtac "#> test_partition1".
idtac "Possible points: 1".
check_type @test_partition1 (
```

```
(@eq (prod (list nat) (list nat))
    (@partition nat odd
        (@cons nat 1
            (@cons nat 2 (@cons nat 3 (@cons nat 4 (@cons nat 5 (@nil nat)))))))
    (@pair (list nat) (list nat)
        (@cons nat 1 (@cons nat 3 (@cons nat 5 (@nil nat))))
        (@cons nat 2 (@cons nat 4 (@nil nat))))).
idtac "Assumptions:".
Abort.
Print Assumptions test_partition1.
Goal True.
idtac " ".

idtac "#> test_partition2".
idtac "Possible points: 1".
check_type @test_partition2 (
(@eq (prod (list nat) (list nat))
    (@partition nat (fun _ : nat ⇒ false)
        (@cons nat 5 (@cons nat 9 (@cons nat 0 (@nil nat)))))
    (@pair (list nat) (list nat) (@nil nat)
        (@cons nat 5 (@cons nat 9 (@cons nat 0 (@nil nat)))))).
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 (
(∀ (X Y : Type) (f : ∀ _ : X, Y) (l : list X),
 @eq (list Y) (@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 (
(∀ (X Y : Type) (_ : ∀ _ : X, list Y) (_ : list X), 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 (
(@eq (list nat)
    (@flat_map nat nat
        (fun n : nat ⇒ @cons nat n (@cons nat n (@cons nat n (@nil nat))))
        (@cons nat 1 (@cons nat 5 (@cons nat 4 (@nil nat)))))
    (@cons nat 1
        (@cons nat 1
            (@cons nat 1
                (@cons nat 5
                    (@cons nat 5
                        (@cons nat 5
                            (@cons nat 4 (@cons nat 4 (@cons nat 4 (@nil nat))))))))))).
idtac "Assumptions:".
Abort.
Print Assumptions test_flat_map1.
Goal True.
idtac " ".

idtac "———————- fold_length ———————".
idtac " ".

idtac "#> Exercises.fold_length_correct".
idtac "Possible points: 2".
check_type @Exercises.fold_length_correct (
(∀ (X : Type) (l : list X),
 @eq nat (@Exercises.fold_length X l) (@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),
 @eq Z (@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 : ∀ _ : prod X Y, Z) (p : prod X Y),
 @eq Z (@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 "———————- church_scc ———————".
idtac " ".

idtac "#> Exercises.Church.scc_2".
idtac "Advanced".
idtac "Possible points: 1".
check_type @Exercises.Church.scc_2 (
(@eq Exercises.Church.cnat (Exercises.Church.scc Exercises.Church.one)
    Exercises.Church.two)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.scc_2.
Goal True.
```

```
idtac " ".

idtac "#> Exercises.Church.scc_3".
idtac "Advanced".
idtac "Possible points: 1".
check_type @Exercises.Church.scc_3 (
(@eq Exercises.Church.cnat (Exercises.Church.scc Exercises.Church.two)
    Exercises.Church.three)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.scc_3.
Goal True.
idtac " ".

idtac "———————- church_plus ———————".
idtac " ".

idtac "#> Exercises.Church.plus_1".
idtac "Advanced".
idtac "Possible points: 1".
check_type @Exercises.Church.plus_1 (
(@eq Exercises.Church.cnat
    (Exercises.Church.plus Exercises.Church.zero Exercises.Church.one)
    Exercises.Church.one)).
idtac "Assumptions:".
Abort.
Print Assumptions Exercises.Church.plus_1.
Goal True.
idtac " ".

idtac "#> Exercises.Church.plus_2".
idtac "Advanced".
idtac "Possible points: 1".
check_type @Exercises.Church.plus_2 (
(@eq Exercises.Church.cnat
    (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: 1".
```

```
check_type @Exercises.Church.plus_3 (
(@eq Exercises.Church.cnat
    (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: 1".
check_type @Exercises.Church.mult_1 (
(@eq Exercises.Church.cnat
    (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: 1".
check_type @Exercises.Church.mult_2 (
(@eq Exercises.Church.cnat
    (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 (
(@eq Exercises.Church.cnat
    (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: 1".
check_type @Exercises.Church.exp_1 (
(@eq Exercises.Church.cnat
    (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: 1".
check_type @Exercises.Church.exp_2 (
(@eq Exercises.Church.cnat
    (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 (
(@eq Exercises.Church.cnat
```

```
    (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: 21".
idtac "Max points - advanced: 34".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 "---------- app_nil_r ----------".
Print Assumptions app_nil_r.
idtac "---------- app_assoc ----------".
Print Assumptions app_assoc.
idtac "---------- app_length ----------".
Print Assumptions app_length.
```

```
idtac "------- rev_app_distr -------".
Print Assumptions rev_app_distr.
idtac "------- rev_involutive -------".
Print Assumptions rev_involutive.
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 "------- Exercises.uncurry_curry -------".
Print Assumptions Exercises.uncurry_curry.
idtac "------- Exercises.curry_uncurry -------".
Print Assumptions Exercises.curry_uncurry.
idtac "------- Exercises.Church.scc_2 -------".
Print Assumptions Exercises.Church.scc_2.
idtac "------- Exercises.Church.scc_3 -------".
Print Assumptions Exercises.Church.scc_3.
idtac "------- Exercises.Church.plus_1 -------".
Print Assumptions Exercises.Church.plus_1.
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 Stdlib 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: 2".
check_type @rev_exercise1 (
(∀ (l l' : list nat) (_ : @eq (list nat) l (@rev nat l')),
 @eq (list nat) l' (@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 (
(∀ (X : Type) (x y z : X) (l j : list X)
    (_ : @eq (list X) (@cons X x (@cons X y l)) (@cons X z j))
    (_ : @eq (list X) j (@cons X z l)),
 @eq X 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 (
(∀ (X : Type) (x y z : X) (l _ : list X)
    (_ : @eq (list X) (@cons X x (@cons X y l)) (@nil X)),
 @eq X x z)).
idtac "Assumptions:".
Abort.
Print Assumptions discriminate_ex3.
Goal True.
idtac " ".
idtac "———————- nth_error_always_none ———————".
idtac " ".
idtac "#> nth_error_always_none".
```

```
idtac "Possible points: 3".
check_type @nth_error_always_none (
(∀ (l : list nat)
    (_ : ∀ i : nat, @eq (option nat) (@nth_error nat l i) (@None nat)),
 @eq (list nat) l (@nil nat))).
idtac "Assumptions:".
Abort.
Print Assumptions nth_error_always_none.
Goal True.
idtac " ".

idtac "———————- eqb_true ———————".
idtac " ".

idtac "#> eqb_true".
idtac "Possible points: 2".
check_type @eqb_true ((∀ (n m : nat) (_ : @eq bool (eqb n m) true), @eq nat n m)).
idtac "Assumptions:".
Abort.
Print Assumptions eqb_true.
Goal True.
idtac " ".

idtac "———————- plus_n_n_injective ———————".
idtac " ".

idtac "#> plus_n_n_injective".
idtac "Possible points: 3".
check_type @plus_n_n_injective (
(∀ (n m : nat) (_ : @eq nat (Nat.add n n) (Nat.add m m)), @eq nat 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 (
(∀ (n : nat) (X : Type) (l : list X) (_ : @eq nat (@length X l) n),
 @eq (option X) (@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 (
(∀ (X Y : Type) (l : list (prod X Y)) (l1 : list X)
    (l2 : list Y)
    (_ : @eq (prod (list X) (list Y)) (@split X Y l)
            (@pair (list X) (list Y) l1 l2)),
 @eq (list (prod X Y)) (@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), @eq 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, @eq bool (eqb n m) (eqb 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) (_ : @eq (list X) (@filter X test l) (@cons X x lf)),
 @eq bool (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' (
(∀ (X : Type) (test : ∀ _ : X, bool) (l : list X),
 @eq bool (@existsb X test l) (@existsb' X test l))).
idtac "Assumptions:".
Abort.
Print Assumptions existsb_existsb'.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 25".
idtac "Max points - advanced: 37".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
```

```
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 "---------- nth_error_always_none ----------".
Print Assumptions nth_error_always_none.
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 "---------- 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 Stdlib 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 "————————- plus_is_O ————————".
```

```
idtac " ".
idtac "#> plus_is_O".
idtac "Possible points: 2".
check_type @plus_is_O (
(∀ (n m : nat) (_ : @eq nat (Nat.add n m) 0),
 and (@eq nat n 0) (@eq nat m 0))).
idtac "Assumptions:".
Abort.
Print Assumptions plus_is_O.
Goal True.
idtac " ".
idtac "———————- and_assoc ———————".
idtac " ".
idtac "#> and_assoc".
idtac "Possible points: 1".
check_type @and_assoc ((∀ (P Q R : Prop) (_ : and P (and Q R)), and (and P Q) R)).
idtac "Assumptions:".
Abort.
Print Assumptions and_assoc.
Goal True.
idtac " ".
idtac "———————- mult_is_O ———————".
idtac " ".
idtac "#> mult_is_O".
idtac "Possible points: 2".
check_type @mult_is_O (
(∀ (n m : nat) (_ : @eq nat (Nat.mul n m) 0),
 or (@eq nat n 0) (@eq nat m 0))).
idtac "Assumptions:".
Abort.
Print Assumptions mult_is_O.
Goal True.
idtac " ".
idtac "———————- or_commut ———————".
idtac " ".
idtac "#> or_commut".
idtac "Possible points: 1".
check_type @or_commut ((∀ (P Q : Prop) (_ : or P Q), or Q P)).
idtac "Assumptions:".
Abort.
Print Assumptions or_commut.
```

```
Goal True.
idtac " ".
idtac "———————- contrapositive ———————".
idtac " ".
idtac "#> contrapositive".
idtac "Possible points: 1".
check_type @contrapositive (
(∀ (P Q : Prop) (_ : ∀ _ : P, Q) (_ : not Q), not 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 ((∀ P : Prop, not (and P (not P)))).
idtac "Assumptions:".
Abort.
Print Assumptions not_both_true_and_false.
Goal True.
idtac " ".
idtac "———————- not_PNP_informal ———————".
idtac " ".
idtac "#> Manually graded: not_PNP_informal".
idtac "Advanced".
idtac "Possible points: 1".
print_manual_grade manual_grade_for_not_PNP_informal.
idtac " ".
idtac "———————- de_morgan_not_or ———————".
idtac " ".
idtac "#> de_morgan_not_or".
idtac "Possible points: 2".
check_type @de_morgan_not_or (
(∀ (P Q : Prop) (_ : not (or P Q)), and (not P) (not Q))).
idtac "Assumptions:".
Abort.
Print Assumptions de_morgan_not_or.
Goal True.
```

```
idtac " ".
idtac "———————————- or_distributes_over_and ———————————".
idtac " ".
idtac "#> or_distributes_over_and".
idtac "Possible points: 3".
check_type @or_distributes_over_and (
(∀ P Q R : Prop, iff (or P (and Q R)) (and (or P Q) (or P 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),
 not (@ex X (fun x : X ⇒ not (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),
 iff (@ex X (fun x : X ⇒ or (P x) (Q x)))
    (or (@ex X (fun x : X ⇒ P x)) (@ex X (fun 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: 2".
check_type @In_map_iff (
(∀ (A B : Type) (f : ∀ _ : A, B) (l : list A) (y : B),
 iff (@In B y (@map A B f l))
    (@ex A (fun x : A ⇒ and (@eq B (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),
 iff (@In A a (@app A l l')) (or (@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),
 iff (∀ (x : T) (_ : @In T x l), P x) (@All T P l))).
idtac "Assumptions:".
Abort.
Print Assumptions All_In.
Goal True.
idtac " ".

idtac "———————- even_double_conv ———————".
idtac " ".

idtac "#> even_double_conv".
idtac "Possible points: 3".
check_type @even_double_conv (
(∀ n : nat,
 @ex nat
```

```
    (fun k : nat ⇒ @eq nat n (if even n then double k else S (double k))))).
idtac "Assumptions:".
Abort.
Print Assumptions even_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,
 iff (@eq bool (andb b1 b2) true) (and (@eq bool b1 true) (@eq bool 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,
 iff (@eq bool (orb b1 b2) true) (or (@eq bool b1 true) (@eq bool 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, iff (@eq bool (eqb x y) false) (not (@eq nat 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, iff (@eq bool (eqb a1 a2) true) (@eq A a1 a2))
    (l1 l2 : list A),
 iff (@eq bool (@eqb_list A eqb l1 l2) true) (@eq (list A) 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 (
(∀ (X : Type) (test : ∀ _ : X, bool) (l : list X),
 iff (@eq bool (@forallb X test l) true)
    (@All X (fun x : X ⇒ @eq bool (test x) true) l))).
idtac "Assumptions:".
Abort.
Print Assumptions forallb_true_iff.
Goal True.
idtac " ".

idtac "———————- tr_rev_correct ———————".
idtac " ".

idtac "#> tr_rev_correct".
idtac "Possible points: 6".
check_type @tr_rev_correct (
(∀ X : Type, @eq (∀ _ : list X, list X) (@tr_rev X) (@rev X))).
idtac "Assumptions:".
Abort.
Print Assumptions tr_rev_correct.
Goal True.
idtac " ".

idtac "———————- excluded_middle_irrefutable ———————".
idtac " ".

idtac "#> excluded_middle_irrefutable".
idtac "Possible points: 3".
check_type @excluded_middle_irrefutable ((∀ P : Prop, not (not (or P (not 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) (X : Type) (P : ∀ _ : X, Prop)
    (_ : not (@ex X (fun x : X ⇒ not (P x)))) (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: 47".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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_is_O -------".
Print Assumptions plus_is_O.
idtac "------- and_assoc -------".
Print Assumptions and_assoc.
idtac "------- mult_is_O -------".
Print Assumptions mult_is_O.
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 "------- de_morgan_not_or -------".
Print Assumptions de_morgan_not_or.
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 In_app_iff.
idtac "------- All_In -------".
Print Assumptions All_In.
idtac "------- even_double_conv -------".
Print Assumptions even_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 "———— tr_rev_correct ————".
Print Assumptions tr_rev_correct.
idtac "———— excluded_middle_irrefutable ————".
Print Assumptions excluded_middle_irrefutable.
idtac "".
idtac "********** Advanced **********".
idtac "———— not_PNP_informal ————".
idtac "MANUAL".
idtac "———— not_exists_dist ————".
Print Assumptions not_exists_dist.
Abort.
```

Chapter 28

Library LF.IndPropTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib 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 ((∀ n : nat, ev (double n))).
idtac "Assumptions:".
Abort.
Print Assumptions ev_double.
Goal True.
idtac " ".

idtac "————————- Perm3 ————————".
idtac " ".

idtac "#> Perm3_ex1".
idtac "Possible points: 0.5".
check_type @Perm3_ex1 (
(@Perm3 nat (@cons nat 1 (@cons nat 2 (@cons nat 3 (@nil nat))))
    (@cons nat 2 (@cons nat 3 (@cons nat 1 (@nil nat))))))).
idtac "Assumptions:".
Abort.
Print Assumptions Perm3_ex1.
Goal True.
idtac " ".

idtac "#> Perm3_refl".
idtac "Possible points: 0.5".
check_type @Perm3_refl (
(∀ (X : Type) (a b c : X),
 @Perm3 X (@cons X a (@cons X b (@cons X c (@nil X))))
    (@cons X a (@cons X b (@cons X c (@nil X)))))).
idtac "Assumptions:".
Abort.
Print Assumptions Perm3_refl.
Goal True.
idtac " ".

idtac "————————- le_inversion ————————".
idtac " ".

idtac "#> le_inversion".
idtac "Possible points: 1".
check_type @le_inversion (
(∀ (n m : nat) (_ : le n m),
 or (@eq nat n m)
    (@ex nat (fun m' : nat ⇒ and (@eq nat m (S m')) (le n m'))))).
idtac "Assumptions:".
```

```
Abort.
Print Assumptions le_inversion.
Goal True.
idtac " ".

idtac "———————- inversion_practice ———————".
idtac " ".

idtac "#> SSSSev__even".
idtac "Possible points: 1".
check_type @SSSSev__even ((∀ (n : nat) (_ : ev (S (S (S (S n))))), 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 ((∀ _ : ev 5, @eq nat (Nat.add 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 ((∀ (n m : nat) (_ : ev n) (_ : ev m), ev (Nat.add 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 ((∀ (n m : nat) (_ : ev (Nat.add n m)) (_ : ev n), ev m)).
```

```
idtac "Assumptions:".
Abort.
Print Assumptions ev_ev__ev.
Goal True.
idtac " ".

idtac "———————- Perm3_In ———————".
idtac " ".

idtac "#> Perm3_In".
idtac "Possible points: 2".
check_type @Perm3_In (
(∀ (X : Type) (x : X) (l1 l2 : list X) (_ : @Perm3 X l1 l2)
    (_ : @In X x l1),
 @In X x l2)).
idtac "Assumptions:".
Abort.
Print Assumptions Perm3_In.
Goal True.
idtac " ".

idtac "———————- le_facts ———————".
idtac " ".

idtac "#> le_trans".
idtac "Possible points: 0.5".
check_type @le_trans ((∀ (m n o : nat) (_ : le m n) (_ : le n o), le m o)).
idtac "Assumptions:".
Abort.
Print Assumptions le_trans.
Goal True.
idtac " ".

idtac "#> O_le_n".
idtac "Possible points: 0.5".
check_type @O_le_n ((∀ n : nat, le 0 n)).
idtac "Assumptions:".
Abort.
Print Assumptions O_le_n.
Goal True.
idtac " ".

idtac "#> n_le_m__Sn_le_Sm".
idtac "Possible points: 0.5".
check_type @n_le_m__Sn_le_Sm ((∀ (n m : nat) (_ : le n m), le (S n) (S m))).
idtac "Assumptions:".
Abort.
```

```
Print Assumptions n_le_m__Sn_le_Sm.
Goal True.
idtac " ".

idtac "#> Sn_le_Sm__n_le_m".
idtac "Possible points: 1".
check_type @Sn_le_Sm__n_le_m ((∀ (n m : nat) (_ : le (S n) (S m)), le n m)).
idtac "Assumptions:".
Abort.
Print Assumptions Sn_le_Sm__n_le_m.
Goal True.
idtac " ".

idtac "#> le_plus_l".
idtac "Possible points: 0.5".
check_type @le_plus_l ((∀ a b : nat, le a (Nat.add a b))).
idtac "Assumptions:".
Abort.
Print Assumptions le_plus_l.
Goal True.
idtac " ".

idtac "————————- plus_le_facts1 ————————".
idtac " ".

idtac "#> plus_le".
idtac "Possible points: 1".
check_type @plus_le (
(∀ (n1 n2 m : nat) (_ : le (Nat.add n1 n2) m), and (le n1 m) (le n2 m))).
idtac "Assumptions:".
Abort.
Print Assumptions plus_le.
Goal True.
idtac " ".

idtac "#> plus_le_cases".
idtac "Possible points: 1".
check_type @plus_le_cases (
(∀ (n m p q : nat) (_ : le (Nat.add n m) (Nat.add p q)),
 or (le n p) (le m q))).
idtac "Assumptions:".
Abort.
Print Assumptions plus_le_cases.
Goal True.
idtac " ".

idtac "————————- plus_le_facts2 ————————".
```

```
idtac " ".
idtac "#> plus_le_compat_l".
idtac "Possible points: 0.5".
check_type @plus_le_compat_l (
(∀ (n m p : nat) (_ : le n m), le (Nat.add p n) (Nat.add p m))).
idtac "Assumptions:".
Abort.
Print Assumptions plus_le_compat_l.
Goal True.
idtac " ".
idtac "#> plus_le_compat_r".
idtac "Possible points: 0.5".
check_type @plus_le_compat_r (
(∀ (n m p : nat) (_ : le n m), le (Nat.add n p) (Nat.add m p))).
idtac "Assumptions:".
Abort.
Print Assumptions plus_le_compat_r.
Goal True.
idtac " ".
idtac "#> le_plus_trans".
idtac "Possible points: 1".
check_type @le_plus_trans ((∀ (n m p : nat) (_ : le n m), le n (Nat.add m p))).
idtac "Assumptions:".
Abort.
Print Assumptions le_plus_trans.
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 ((∀ l : list nat, subseq l l)).
idtac "Assumptions:".
Abort.
```

```
Print Assumptions subseq_refl.
Goal True.
idtac " ".

idtac "#> subseq_app".
idtac "Advanced".
idtac "Possible points: 2".
check_type @subseq_app (
(∀ (l1 l2 l3 : list nat) (_ : subseq l1 l2), subseq l1 (@app nat l2 l3))).
idtac "Assumptions:".
Abort.
Print Assumptions subseq_app.
Goal True.
idtac " ".

idtac "#> subseq_trans".
idtac "Advanced".
idtac "Possible points: 3".
check_type @subseq_trans (
(∀ (l1 l2 l3 : list nat) (_ : subseq l1 l2) (_ : subseq l2 l3),
 subseq l1 l3)).
idtac "Assumptions:".
Abort.
Print Assumptions subseq_trans.
Goal True.
idtac " ".

idtac "———————- exp_match_ex1 ———————".
idtac " ".

idtac "#> EmptySet_is_empty".
idtac "Possible points: 0.5".
check_type @EmptySet_is_empty (
(∀ (T : Type) (s : list T), not (@exp_match T s (@EmptySet T)))).
idtac "Assumptions:".
Abort.
Print Assumptions EmptySet_is_empty.
Goal True.
idtac " ".

idtac "#> MUnion'".
idtac "Possible points: 0.5".
check_type @MUnion' (
(∀ (T : Type) (s : list T) (re1 re2 : reg_exp T)
    (_ : or (@exp_match T s re1) (@exp_match T s re2)),
 @exp_match T s (@Union T re1 re2))).
```

```
idtac "Assumptions:".
Abort.
Print Assumptions MUnion'.
Goal True.
idtac " ".

idtac "#> MStar'".
idtac "Possible points: 2".
check_type @MStar' (
(∀ (T : Type) (ss : list (list T)) (re : reg_exp T)
    (_ : ∀ (s : list T) (_ : @In (list T) s ss), @exp_match T s re),
 @exp_match T (@fold (list T) (list T) (@app T) ss (@nil T)) (@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 ((∀ (T : Type) (_ : reg_exp T), 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),
 iff (@ex (list T) (fun s : list T ⇒ @exp_match T s re))
    (@eq bool (@re_not_empty T re) true))).
idtac "Assumptions:".
Abort.
Print Assumptions re_not_empty_correct.
Goal True.
idtac " ".

idtac "———————- weak_pumping_char ———————".
idtac " ".

idtac "#> Pumping.weak_pumping_char".
idtac "Possible points: 2".
```

```
check_type @Pumping.weak_pumping_char (
(∀ (T : Type) (x : T)
   (_ : le (@Pumping.pumping_constant T (@Char T x))
           (@length T (@cons T x (@nil T)))),
 @ex (list T)
   (fun s1 : list T ⇒
    @ex (list T)
      (fun s2 : list T ⇒
       @ex (list T)
         (fun s3 : list T ⇒
          and (@eq (list T) (@cons T x (@nil T)) (@app T s1 (@app T s2 s3)))
            (and (not (@eq (list T) s2 (@nil T)))
                 (∀ m : nat,
                  @exp_match T (@app T s1 (@app T (@Pumping.napp T m s2) s3))
                    (@Char T x))))))))).
idtac "Assumptions:".
Abort.
Print Assumptions Pumping.weak_pumping_char.
Goal True.
idtac " ".

idtac "———————- weak_pumping_app ———————".
idtac " ".

idtac "#> Pumping.weak_pumping_app".
idtac "Possible points: 3".
check_type @Pumping.weak_pumping_app (
(∀ (T : Type) (s1 s2 : list T) (re1 re2 : reg_exp T)
   (_ : @exp_match T s1 re1) (_ : @exp_match T s2 re2)
   (_ : ∀ _ : le (@Pumping.pumping_constant T re1) (@length T s1),
        @ex (list T)
          (fun s3 : list T ⇒
           @ex (list T)
             (fun s4 : list T ⇒
              @ex (list T)
                (fun s5 : list T ⇒
                 and (@eq (list T) s1 (@app T s3 (@app T s4 s5)))
                   (and (not (@eq (list T) s4 (@nil T)))
                        (∀ m : nat,
                         @exp_match T
                           (@app T s3 (@app T (@Pumping.napp T m s4) s5)) re1))))))
   (_ : ∀ _ : le (@Pumping.pumping_constant T re2) (@length T s2),
        @ex (list T)
          (fun s3 : list T ⇒
```

```
              @ex (list T)
                (fun s4 : list T ⇒
                 @ex (list T)
                   (fun s5 : list T ⇒
                    and (@eq (list T) s2 (@app T s3 (@app T s4 s5)))
                      (and (not (@eq (list T) s4 (@nil T)))
                         (∀ m : nat,
                          @exp_match T
                            (@app T s3 (@app T (@Pumping.napp T m s4) s5)) re2))))))
    (_ : le (@Pumping.pumping_constant T (@App T re1 re2))
            (@length T (@app T s1 s2))),
  @ex (list T)
    (fun s0 : list T ⇒
     @ex (list T)
       (fun s3 : list T ⇒
        @ex (list T)
          (fun s4 : list T ⇒
           and (@eq (list T) (@app T s1 s2) (@app T s0 (@app T s3 s4)))
             (and (not (@eq (list T) s3 (@nil T)))
                (∀ m : nat,
                 @exp_match T (@app T s0 (@app T (@Pumping.napp T m s3) s4))
                   (@App T re1 re2)))))))).
idtac "Assumptions:".
Abort.
Print Assumptions Pumping.weak_pumping_app.
Goal True.
idtac " ".

idtac "———————- weak_pumping_union_l ———————".
idtac " ".

idtac "#> Pumping.weak_pumping_union_l".
idtac "Possible points: 3".
check_type @Pumping.weak_pumping_union_l (
(∀ (T : Type) (s1 : list T) (re1 re2 : reg_exp T)
    (_ : @exp_match T s1 re1)
    (_ : ∀ _ : le (@Pumping.pumping_constant T re1) (@length T s1),
         @ex (list T)
           (fun s2 : list T ⇒
            @ex (list T)
              (fun s3 : list T ⇒
               @ex (list T)
                 (fun s4 : list T ⇒
                  and (@eq (list T) s1 (@app T s2 (@app T s3 s4)))
```

```
                        (and (not (@eq (list T) s3 (@nil T)))
                           (∀ m : nat,
                            @exp_match T
                              (@app T s2 (@app T (@Pumping.napp T m s3) s4)) re1))))))
    (_ : le (@Pumping.pumping_constant T (@Union T re1 re2)) (@length T s1)),
  @ex (list T)
    (fun s0 : list T ⇒
     @ex (list T)
       (fun s2 : list T ⇒
        @ex (list T)
          (fun s3 : list T ⇒
           and (@eq (list T) s1 (@app T s0 (@app T s2 s3)))
             (and (not (@eq (list T) s2 (@nil T)))
                (∀ m : nat,
                 @exp_match T (@app T s0 (@app T (@Pumping.napp T m s2) s3))
                   (@Union T re1 re2))))))).
idtac "Assumptions:".
Abort.
Print Assumptions Pumping.weak_pumping_union_l.
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), iff P (@eq bool 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) (_ : @eq nat (count n l) 0),
 not (@In 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 "#> merge_filter".
idtac "Advanced".
idtac "Possible points: 6".
check_type @merge_filter (
(∀ (X : Set) (test : ∀ _ : X, bool) (l l1 l2 : list X)
    (_ : @merge X l1 l2 l)
    (_ : @All X (fun n : X ⇒ @eq bool (test n) true) l1)
    (_ : @All X (fun n : X ⇒ @eq bool (test n) false) l2),
 @eq (list X) (@filter X test l) l1)).
idtac "Assumptions:".
Abort.
Print Assumptions merge_filter.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 44".
idtac "Max points - advanced: 59".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 "---------- Perm3_ex1 ----------".
Print Assumptions Perm3_ex1.
idtac "---------- Perm3_refl ----------".
Print Assumptions Perm3_refl.
idtac "---------- le_inversion ----------".
Print Assumptions le_inversion.
idtac "---------- SSSSev__even ----------".
Print Assumptions SSSSev__even.
idtac "---------- ev5_nonsense ----------".
Print Assumptions ev5_nonsense.
idtac "---------- ev_sum ----------".
Print Assumptions ev_sum.
idtac "---------- Perm3_In ----------".
Print Assumptions Perm3_In.
idtac "---------- le_trans ----------".
Print Assumptions le_trans.
idtac "---------- O_le_n ----------".
Print Assumptions O_le_n.
idtac "---------- n_le_m__Sn_le_Sm ----------".
Print Assumptions n_le_m__Sn_le_Sm.
idtac "---------- Sn_le_Sm__n_le_m ----------".
Print Assumptions Sn_le_Sm__n_le_m.
idtac "---------- le_plus_l ----------".
Print Assumptions le_plus_l.
idtac "---------- plus_le ----------".
Print Assumptions plus_le.
idtac "---------- plus_le_cases ----------".
Print Assumptions plus_le_cases.
idtac "---------- plus_le_compat_l ----------".
Print Assumptions plus_le_compat_l.
```

```
idtac "------- plus_le_compat_r -------".
Print Assumptions plus_le_compat_r.
idtac "------- le_plus_trans -------".
Print Assumptions le_plus_trans.
idtac "------- R_provability -------".
idtac "MANUAL".
idtac "------- EmptySet_is_empty -------".
Print Assumptions EmptySet_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 "------- Pumping.weak_pumping_char -------".
Print Assumptions Pumping.weak_pumping_char.
idtac "------- Pumping.weak_pumping_app -------".
Print Assumptions Pumping.weak_pumping_app.
idtac "------- Pumping.weak_pumping_union_l -------".
Print Assumptions Pumping.weak_pumping_union_l.
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 "------- subseq_trans -------".
Print Assumptions subseq_trans.
idtac "------- merge_filter -------".
Print Assumptions merge_filter.
Abort.
```

Chapter 29

Library LF.MapsTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib 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),
 @eq (∀ _ : string, A) (@t_update A m x (m x)) 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)
    (_ : not (@eq string x2 x1)),
 @eq (∀ _ : string, A) (@t_update A (@t_update A m x2 v2) x1 v1)
    (@t_update A (@t_update A m x1 v1) x2 v2))).
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 "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 Stdlib 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 (
(∀ (P Q R : Prop) (_ : and P Q) (_ : and Q R), and P 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' ((∀ (P Q : Prop) (_ : or P Q), or Q 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 ((@ex nat (fun n : nat ⇒ ev (S n)))).
idtac "Assumptions:".
Abort.
Print Assumptions Props.ex_ev_Sn.
Goal True.
idtac " ".

idtac "———————- ex_match ———————".
idtac " ".

idtac "#> Props.ex_match".
idtac "Possible points: 2".
check_type @Props.ex_match (
(∀ (A : Type) (P Q : ∀ _ : A, Prop)
    (_ : ∀ (x : A) (_ : P x), Q x) (_ : @ex A (fun x : A ⇒ P x)),
 @ex A (fun x : A ⇒ Q x))).
idtac "Assumptions:".
Abort.
Print Assumptions Props.ex_match.
Goal True.
idtac " ".

idtac "———————- p_implies_true ———————".
idtac " ".

idtac "#> Props.p_implies_true".
idtac "Possible points: 1".
check_type @Props.p_implies_true ((∀ (P : Type) (_ : P), 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' ((∀ (P : Type) (_ : Props.False), P)).
idtac "Assumptions:".
Abort.
Print Assumptions Props.ex_falso_quodlibet'.
Goal True.
idtac " ".

idtac "———————- eq_cons ———————".
```

```
idtac " ".

idtac "#> EqualityPlayground.eq_cons".
idtac "Possible points: 2".
check_type @EqualityPlayground.eq_cons (
(∀ (X : Type) (h1 h2 : X) (t1 t2 : list X)
   (_ : @EqualityPlayground.eq X h1 h2)
   (_ : @EqualityPlayground.eq (list X) t1 t2),
 @EqualityPlayground.eq (list X) (@cons X h1 t1) (@cons X h2 t2))).
idtac "Assumptions:".
Abort.
Print Assumptions EqualityPlayground.eq_cons.
Goal True.
idtac " ".

idtac "———————- equality__leibniz_equality ———————".
idtac " ".

idtac "#> EqualityPlayground.equality__leibniz_equality".
idtac "Possible points: 2".
check_type @EqualityPlayground.equality__leibniz_equality (
(∀ (X : Type) (x y : X) (_ : @EqualityPlayground.eq X x y)
   (P : ∀ _ : X, Prop) (_ : P x),
 P y)).
idtac "Assumptions:".
Abort.
Print Assumptions EqualityPlayground.equality__leibniz_equality.
Goal True.
idtac " ".

idtac "———————- equality__leibniz_equality_term ———————".
idtac " ".

idtac "#> EqualityPlayground.equality__leibniz_equality_term".
idtac "Possible points: 2".
check_type @EqualityPlayground.equality__leibniz_equality_term (
(∀ (X : Type) (x y : X) (_ : @EqualityPlayground.eq X x y)
   (P : ∀ _ : X, Prop) (_ : P x),
 P y)).
idtac "Assumptions:".
Abort.
Print Assumptions EqualityPlayground.equality__leibniz_equality_term.
Goal True.
idtac " ".

idtac "———————- and_assoc ———————".
idtac " ".
```

```
idtac "#> and_assoc".
idtac "Possible points: 2".
check_type @and_assoc ((∀ (P Q R : Prop) (_ : and P (and Q R)), and (and P Q) R)).
idtac "Assumptions:".
Abort.
Print Assumptions and_assoc.
Goal True.
idtac " ".

idtac "———————- or_distributes_over_and ———————".
idtac " ".

idtac "#> or_distributes_over_and".
idtac "Possible points: 3".
check_type @or_distributes_over_and (
(∀ P Q R : Prop, iff (or P (and Q R)) (and (or P Q) (or P R)))).
idtac "Assumptions:".
Abort.
Print Assumptions or_distributes_over_and.
Goal True.
idtac " ".

idtac "———————- negations ———————".
idtac " ".

idtac "#> double_neg".
idtac "Possible points: 1".
check_type @double_neg ((∀ (P : Prop) (_ : P), not (not P))).
idtac "Assumptions:".
Abort.
Print Assumptions double_neg.
Goal True.
idtac " ".

idtac "#> contradiction_implies_anything".
idtac "Possible points: 1".
check_type @contradiction_implies_anything ((∀ (P Q : Prop) (_ : and P (not P)), Q)).
idtac "Assumptions:".
Abort.
Print Assumptions contradiction_implies_anything.
Goal True.
idtac " ".

idtac "#> de_morgan_not_or".
idtac "Possible points: 1".
check_type @de_morgan_not_or (
(∀ (P Q : Prop) (_ : not (or P Q)), and (not P) (not Q))).
```

```
idtac "Assumptions:".
Abort.
Print Assumptions de_morgan_not_or.
Goal True.
idtac " ".
idtac "———————- currying ———————".
idtac " ".
idtac "#> curry".
idtac "Possible points: 1".
check_type @curry (
(∀ (P Q R : Prop) (_ : ∀ _ : and P Q, R) (_ : P) (_ : Q), R)).
idtac "Assumptions:".
Abort.
Print Assumptions curry.
Goal True.
idtac " ".
idtac "#> uncurry".
idtac "Possible points: 1".
check_type @uncurry (
(∀ (P Q R : Prop) (_ : ∀ (_ : P) (_ : Q), R) (_ : and P Q), R)).
idtac "Assumptions:".
Abort.
Print Assumptions uncurry.
Goal True.
idtac " ".
idtac "———————- pe_implies_or_eq ———————".
idtac " ".
idtac "#> pe_implies_or_eq".
idtac "Advanced".
idtac "Possible points: 1".
check_type @pe_implies_or_eq (
(∀ (_ : propositional_extensionality) (P Q : Prop),
 @eq Prop (or P Q) (or Q P))).
idtac "Assumptions:".
Abort.
Print Assumptions pe_implies_or_eq.
Goal True.
idtac " ".
idtac "———————- pe_implies_true_eq ———————".
idtac " ".
idtac "#> pe_implies_true_eq".
```

```
idtac "Advanced".
idtac "Possible points: 1".
check_type @pe_implies_true_eq (
(∀ (_ : propositional_extensionality) (P : Prop) (_ : P),
 @eq Prop True P)).
idtac "Assumptions:".
Abort.
Print Assumptions pe_implies_true_eq.
Goal True.
idtac " ".

idtac "———————- pe_implies_pi ———————".
idtac " ".

idtac "#> pe_implies_pi".
idtac "Advanced".
idtac "Possible points: 3".
check_type @pe_implies_pi ((∀ _ : propositional_extensionality, proof_irrelevance)).
idtac "Assumptions:".
Abort.
Print Assumptions pe_implies_pi.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 28".
idtac "Max points - advanced: 33".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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.ex_match -------".
Print Assumptions Props.ex_match.
idtac "------- Props.p_implies_true -------".
Print Assumptions Props.p_implies_true.
idtac "------- Props.ex_falso_quodlibet' -------".
Print Assumptions Props.ex_falso_quodlibet'.
idtac "------- EqualityPlayground.eq_cons -------".
Print Assumptions EqualityPlayground.eq_cons.
idtac "------- EqualityPlayground.equality__leibniz_equality -------".
Print Assumptions EqualityPlayground.equality__leibniz_equality.
idtac "------- EqualityPlayground.equality__leibniz_equality_term -------".
Print Assumptions EqualityPlayground.equality__leibniz_equality_term.
idtac "------- and_assoc -------".
Print Assumptions and_assoc.
idtac "------- or_distributes_over_and -------".
Print Assumptions or_distributes_over_and.
idtac "------- double_neg -------".
Print Assumptions double_neg.
idtac "------- contradiction_implies_anything -------".
Print Assumptions contradiction_implies_anything.
idtac "------- de_morgan_not_or -------".
Print Assumptions de_morgan_not_or.
idtac "------- curry -------".
Print Assumptions curry.
idtac "------- uncurry -------".
Print Assumptions uncurry.
```

```
idtac "".
idtac "********** Advanced **********".
idtac "-------- pe_implies_or_eq --------".
Print Assumptions pe_implies_or_eq.
idtac "-------- pe_implies_true_eq --------".
Print Assumptions pe_implies_true_eq.
idtac "-------- pe_implies_pi --------".
Print Assumptions pe_implies_pi.
Abort.
```

Chapter 31

Library LF.IndPrinciplesTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib 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' ((∀ n : nat, @eq nat (Nat.add n 1) (S n))).
idtac "Assumptions:".
Abort.
Print Assumptions plus_one_r'.
Goal True.
idtac " ".
idtac "———————- booltree_ind ———————".
idtac " ".
idtac "#> booltree_ind_type_correct".
idtac "Possible points: 2".
check_type @booltree_ind_type_correct (booltree_ind_type).
idtac "Assumptions:".
Abort.
Print Assumptions booltree_ind_type_correct.
Goal True.
idtac " ".
idtac "———————- toy_ind ———————".
idtac " ".
idtac "#> Toy_correct".
idtac "Possible points: 2".
check_type @Toy_correct (
(@ex (∀ _ : bool, Toy)
   (fun f : ∀ _ : bool, Toy ⇒
    @ex (∀ (_ : nat) (_ : Toy), Toy)
      (fun g : ∀ (_ : nat) (_ : Toy), Toy ⇒
       ∀ (P : ∀ _ : Toy, Prop) (_ : ∀ b : bool, P (f b))
         (_ : ∀ (n : nat) (t : Toy) (_ : P t), P (g n t))
         (t : Toy),
        P t)))).
idtac "Assumptions:".
Abort.
Print Assumptions Toy_correct.
Goal True.
idtac " ".
idtac " ".
idtac "Max points - standard: 6".
idtac "Max points - advanced: 6".
idtac "".
```

```
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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_type_correct ----------".
Print Assumptions booltree_ind_type_correct.
idtac "---------- Toy_correct ----------".
Print Assumptions Toy_correct.
idtac "".
idtac "********** Advanced **********".
Abort.
```

Chapter 32

Library LF.RelTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib 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 "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 Stdlib 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_test1".
idtac "Possible points: 0.5".
check_type @AExp.optimize_0plus_b_test1 (
(@eq AExp.bexp
   (AExp.optimize_0plus_b
      (AExp.BNot
         (AExp.BGt (AExp.APlus (AExp.ANum 0) (AExp.ANum 4)) (AExp.ANum 8))))
   (AExp.BNot (AExp.BGt (AExp.ANum 4) (AExp.ANum 8))))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.optimize_0plus_b_test1.
Goal True.
idtac " ".

idtac "#> AExp.optimize_0plus_b_test2".
idtac "Possible points: 0.5".
check_type @AExp.optimize_0plus_b_test2 (
(@eq AExp.bexp
   (AExp.optimize_0plus_b
      (AExp.BAnd
         (AExp.BLe (AExp.APlus (AExp.ANum 0) (AExp.ANum 4)) (AExp.ANum 5))
         AExp.BTrue))
   (AExp.BAnd (AExp.BLe (AExp.ANum 4) (AExp.ANum 5)) AExp.BTrue))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.optimize_0plus_b_test2.
Goal True.
idtac " ".

idtac "#> AExp.optimize_0plus_b_sound".
idtac "Possible points: 2".
check_type @AExp.optimize_0plus_b_sound (
(∀ b : AExp.bexp,
 @eq bool (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.bevalR_iff_beval".
```

```
idtac "Possible points: 3".
check_type @AExp.bevalR_iff_beval (
(∀ (b : AExp.bexp) (bv : bool),
 iff (AExp.bevalR b bv) (@eq bool (AExp.beval b) bv))).
idtac "Assumptions:".
Abort.
Print Assumptions AExp.bevalR_iff_beval.
Goal True.
idtac " ".

idtac "———————- ceval_example2 ———————".
idtac " ".

idtac "#> ceval_example2".
idtac "Possible points: 2".
check_type @ceval_example2 (
(ceval (CSeq (CAsgn X (ANum 0)) (CSeq (CAsgn Y (ANum 1)) (CAsgn Z (ANum 2))))
    empty_st
    (@Maps.t_update nat
        (@Maps.t_update nat (@Maps.t_update nat empty_st 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, not (ceval loop st 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 (
(∀ c : com, iff (@eq bool (no_whiles c) true) (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 (
(@eq (list nat)
    (s_execute empty_st (@nil nat)
        (@cons sinstr (SPush 5)
            (@cons sinstr (SPush 3)
                (@cons sinstr (SPush 1) (@cons sinstr SMinus (@nil sinstr))))))
    (@cons nat 2 (@cons nat 5 (@nil nat))))).
idtac "Assumptions:".
Abort.
Print Assumptions s_execute1.
Goal True.
idtac " ".

idtac "#> s_execute2".
idtac "Possible points: 0.5".
check_type @s_execute2 (
(@eq (list nat)
    (s_execute (@Maps.t_update nat empty_st X 3)
        (@cons nat 3 (@cons nat 4 (@nil nat)))
        (@cons sinstr (SPush 4)
            (@cons sinstr (SLoad X)
                (@cons sinstr SMult (@cons sinstr SPlus (@nil sinstr))))))
    (@cons nat 15 (@cons nat 4 (@nil nat))))).
idtac "Assumptions:".
Abort.
Print Assumptions s_execute2.
Goal True.
idtac " ".

idtac "#> s_compile1".
```

```
idtac "Possible points: 1.5".
check_type @s_compile1 (
(@eq (list sinstr) (s_compile (AMinus (AId X) (AMult (ANum 2) (AId Y))))
    (@cons sinstr (SLoad X)
        (@cons sinstr (SPush 2)
            (@cons sinstr (SLoad Y)
                (@cons sinstr SMult (@cons sinstr SMinus (@nil sinstr))))))))).
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),
 @eq (list nat) (s_execute st stack (@app sinstr 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),
 @eq (list nat) (s_execute st stack (s_compile e))
    (@cons nat (aeval st e) stack))).
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),
  @eq (list nat) (s_execute st (@nil nat) (s_compile e))
     (@cons nat (aeval st e) (@nil nat)))).
idtac "Assumptions:".
Abort.
Print Assumptions s_compile_correct.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 29".
idtac "Max points - advanced: 29".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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_test1 ----------".
Print Assumptions AExp.optimize_0plus_b_test1.
idtac "---------- AExp.optimize_0plus_b_test2 ----------".
Print Assumptions AExp.optimize_0plus_b_test2.
idtac "---------- AExp.optimize_0plus_b_sound ----------".
Print Assumptions AExp.optimize_0plus_b_sound.
idtac "---------- AExp.bevalR_iff_beval ----------".
Print Assumptions AExp.bevalR_iff_beval.
```

```
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 **********".
Abort.
```

Chapter 34

Library LF.ImpParserTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib 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 "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 Stdlib 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 "————————- ceval_step__ceval_inf ————————".
```

```
idtac " ".

idtac "#> Manually graded: ceval_step__ceval_inf".
idtac "Advanced".
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 (
(∀ (c : Imp.com) (st st' : Imp.state) (_ : Imp.ceval c st st'),
 @ex nat
    (fun i : nat ⇒
     @eq (option Imp.state) (ceval_step st c i) (@Some Imp.state st')))).
idtac "Assumptions:".
Abort.
Print Assumptions ceval__ceval_step.
Goal True.
idtac " ".

idtac " ".

idtac "Max points - standard: 3".
idtac "Max points - advanced: 9".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 "---------- ceval__ceval_step ----------".
Print Assumptions ceval__ceval_step.
idtac "".
idtac "********** Advanced **********".
idtac "---------- ceval_step__ceval_inf ----------".
idtac "MANUAL".
Abort.
```

Chapter 36

Library LF.ExtractionTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib 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 "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 Stdlib 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 "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 Stdlib 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 "————————- try_sequence ————————".
```

```
idtac " ".

idtac "#> andb_eq_orb".
idtac "Possible points: 1".
check_type @andb_eq_orb (
(∀ (b c : Basics.bool)
    (_ : @eq Basics.bool (Basics.andb b c) (Basics.orb b c)),
 @eq Basics.bool b c)).
idtac "Assumptions:".
Abort.
Print Assumptions andb_eq_orb.
Goal True.
idtac " ".

idtac "#> add_assoc".
idtac "Possible points: 1".
check_type @add_assoc (
(∀ n m p : nat,
 @eq nat (Nat.add n (Nat.add m p)) (Nat.add (Nat.add n m) p))).
idtac "Assumptions:".
Abort.
Print Assumptions add_assoc.
Goal True.
idtac " ".

idtac "#> nonzeros_app".
idtac "Possible points: 1".
check_type @nonzeros_app (
(∀ lst1 lst2 : Poly.list nat,
 @eq (Poly.list nat) (nonzeros (@Poly.app nat lst1 lst2))
    (@Poly.app nat (nonzeros lst1) (nonzeros lst2)))).
idtac "Assumptions:".
Abort.
Print Assumptions nonzeros_app.
Goal True.
idtac " ".

idtac "————————- notry_sequence ————————".
idtac " ".

idtac "#> add_assoc'".
idtac "Possible points: 1".
check_type @add_assoc' (
(∀ n m p : nat,
 @eq nat (Nat.add n (Nat.add m p)) (Nat.add (Nat.add n m) p))).
idtac "Assumptions:".
```

```
Abort.
Print Assumptions add_assoc'.
Goal True.
idtac " ".
idtac "———————- ev100 ———————".
idtac " ".
idtac "#> ev100".
idtac "Possible points: 1".
check_type @ev100 ((IndProp.ev 100)).
idtac "Assumptions:".
Abort.
Print Assumptions ev100.
Goal True.
idtac " ".
idtac "———————- re_opt ———————".
idtac " ".
idtac "#> Manually graded: re_opt".
idtac "Possible points: 6".
print_manual_grade manual_grade_for_re_opt.
idtac " ".
idtac "———————- automatic_solvers ———————".
idtac " ".
idtac "#> plus_id_exercise_from_basics".
idtac "Possible points: 0.5".
check_type @plus_id_exercise_from_basics (
(∀ (n m o : nat) (_ : @eq nat n m) (_ : @eq nat m o),
 @eq nat (Nat.add n m) (Nat.add m o))).
idtac "Assumptions:".
Abort.
Print Assumptions plus_id_exercise_from_basics.
Goal True.
idtac " ".
idtac "#> add_assoc_from_induction".
idtac "Possible points: 0.5".
check_type @add_assoc_from_induction (
(∀ n m p : nat,
 @eq nat (Nat.add n (Nat.add m p)) (Nat.add (Nat.add n m) p))).
idtac "Assumptions:".
Abort.
Print Assumptions add_assoc_from_induction.
Goal True.
```

```
idtac " ".
idtac "#> S_injective_from_tactics".
idtac "Possible points: 0.5".
check_type @S_injective_from_tactics (
(∀ (n m : nat) (_ : @eq nat (S n) (S m)), @eq nat n m)).
idtac "Assumptions:".
Abort.
Print Assumptions S_injective_from_tactics.
Goal True.
idtac " ".
idtac "#> or_distributes_over_and_from_logic".
idtac "Possible points: 0.5".
check_type @or_distributes_over_and_from_logic (
(∀ P Q R : Prop, iff (or P (and Q R)) (and (or P Q) (or P R)))).
idtac "Assumptions:".
Abort.
Print Assumptions or_distributes_over_and_from_logic.
Goal True.
idtac " ".
idtac "———————- re_opt_match_auto ———————".
idtac " ".
idtac "#> Manually graded: re_opt_match''".
idtac "Possible points: 3".
print_manual_grade manual_grade_for_re_opt_match''.
idtac " ".
idtac "———————- andb3_exchange ———————".
idtac " ".
idtac "#> andb3_exchange".
idtac "Possible points: 1".
check_type @andb3_exchange (
(∀ b c d : Basics.bool,
 @eq Basics.bool (Basics.andb (Basics.andb b c) d)
   (Basics.andb (Basics.andb b d) c))).
idtac "Assumptions:".
Abort.
Print Assumptions andb3_exchange.
Goal True.
idtac " ".
idtac "———————- andb_true_elim2 ———————".
idtac " ".
idtac "#> andb_true_elim2'".
```

```
idtac "Possible points: 1.5".
check_type @andb_true_elim2' (
(∀ (b c : Basics.bool)
    (_ : @eq Basics.bool (Basics.andb b c) Basics.true),
 @eq Basics.bool c Basics.true)).
idtac "Assumptions:".
Abort.
Print Assumptions andb_true_elim2'.
Goal True.
idtac " ".

idtac "#> andb3_exchange'".
idtac "Possible points: 0.5".
check_type @andb3_exchange' (
(∀ b c d : Basics.bool,
 @eq Basics.bool (Basics.andb (Basics.andb b c) d)
    (Basics.andb (Basics.andb b d) c))).
idtac "Assumptions:".
Abort.
Print Assumptions andb3_exchange'.
Goal True.
idtac " ".

idtac "———————- nor_intuition ———————".
idtac " ".

idtac "#> Manually graded: nor_intuition".
idtac "Advanced".
idtac "Possible points: 6".
print_manual_grade manual_grade_for_nor_intuition.
idtac " ".

idtac " ".

idtac "Max points - standard: 19".
idtac "Max points - advanced: 25".
idtac "".
idtac "Allowed Axioms:".
idtac "functional_extensionality".
idtac "FunctionalExtensionality.functional_extensionality_dep".
idtac "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
```

```
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 "---------- andb_eq_orb ----------".
Print Assumptions andb_eq_orb.
idtac "---------- add_assoc ----------".
Print Assumptions add_assoc.
idtac "---------- nonzeros_app ----------".
Print Assumptions nonzeros_app.
idtac "---------- add_assoc' ----------".
Print Assumptions add_assoc'.
idtac "---------- ev100 ----------".
Print Assumptions ev100.
idtac "---------- re_opt ----------".
idtac "MANUAL".
idtac "---------- plus_id_exercise_from_basics ----------".
Print Assumptions plus_id_exercise_from_basics.
idtac "---------- add_assoc_from_induction ----------".
Print Assumptions add_assoc_from_induction.
idtac "---------- S_injective_from_tactics ----------".
Print Assumptions S_injective_from_tactics.
idtac "---------- or_distributes_over_and_from_logic ----------".
Print Assumptions or_distributes_over_and_from_logic.
idtac "---------- re_opt_match" ----------".
idtac "MANUAL".
idtac "---------- andb3_exchange ----------".
Print Assumptions andb3_exchange.
idtac "---------- andb_true_elim2' ----------".
Print Assumptions andb_true_elim2'.
idtac "---------- andb3_exchange' ----------".
Print Assumptions andb3_exchange'.
idtac "".
```

```
idtac "********** Advanced **********".
idtac "———— nor_intuition ————".
idtac "MANUAL".
Abort.
```

Chapter 39

Library LF.PostscriptTest

```
Set Warnings "-notation-overridden,-parsing".
From Stdlib 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 "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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 Stdlib 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 "plus_le".
idtac "le_trans".
idtac "le_plus_l".
idtac "add_le_cases".
idtac "Sn_le_Sm__n_le_m".
idtac "O_le_n".
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.
```