Software Foundations

Benjamin C. Pierce

Arthur Azevedo de Amorim

Chris Casinghino

Marco Gaboardi

Michael Greenberg

Cătălin Hriţcu

Vilhelm Sjöberg

Brent Yorgey

with Loris D'Antoni, Andrew W. Appel, Arthur Charguéraud, Anthony Cowley, Jeffrey Foster, Dmitri Garbuzov, Michael Hicks, Ranjit Jhala, Greg Morrisett, Jennifer Paykin, Mukund Raghothaman, Chung-chieh Shan, Leonid Spesivtsev, Andrew Tolmach, Stephanie Weirich and Steve Zdancewic.

Idris translation by Eric Bailey, Alex Gryzlov and Erlend Hamberg.

Contents

CHAPTER 1

Preface

1. Welcome

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

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

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

2. Overview

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

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

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

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

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

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

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

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

- *Automated theorem provers* provide "push-button" operation: you give them a proposition and they return either *true*, *false*, or *ran out of time*.

Although their capabilities are limited to fairly specific sorts of reasoning, they have matured tremendously in recent years and are used now in a huge variety of settings. Examples of such tools include SAT solvers, SMT solvers, and model checkers.

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

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

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

- As a *platform for modeling programming languages*, it has become a standard tool for researchers who need to describe and reason about complex language definitions. It has been used, for example, to check the security of the JavaCard platform, obtaining the highest level of common criteria certification, and for formal specifications of the x86 and LLVM instruction sets and programming languages such as C.

- As an *environment for developing formally certified software*, Coq has been used, for example, to build CompCert, a fully-verified optimizing compiler for C, for proving the correctness of subtle algorithms involving floating point numbers, and as the basis for CertiCrypt, an environment for reasoning about the security of cryptographic algorithms.

- As a *realistic environment for functional programming with dependent types*, it has inspired numerous innovations. For example, the Ynot project at Harvard embedded "relational Hoare reasoning" (an extension of the *Hoare Logic* we will see later in this course) in Coq.

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

to a Coq formalization of the Feit-Thompson Theorem – the first major step in the classification of finite simple groups.

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

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

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

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

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

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

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

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

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

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

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

Second, we develop a methodology for proving that particular Imp programs satisfy formal specifications of their behavior. We introduce the notion of *Hoare triples* – Imp programs annotated with pre- and post-conditions describing what should be true about the memory in which they are started and what they promise to make true about the memory in which they terminate – and the reasoning principles of *Hoare Logic*, a "domain-specific logic" specialized for convenient compositional reasoning about imperative programs, with concepts like "loop invariant" built in.

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

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

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

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

2.6. Further Reading. This text is intended to be self contained, but readers looking for a deeper treatment of a particular topic will find suggestions for further reading in the [Postscript] chapter.

3. Practicalities

3.1. Chapter Dependencies. A diagram of the dependencies between chapters and some suggested paths through the material can be found in the file [deps.html].

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

- A current installation of Coq, available from the Coq home page. Everything should work with version 8.4. (Version 8.5 will *not* work, due to a few incompatible changes in Coq between 8.4 and 8.5.)
- An IDE for interacting with Coq. Currently, there are two choices:
  - Proof General is an Emacs-based IDE. It tends to be preferred by users who are already comfortable with Emacs. It requires a separate installation (google "Proof General").
  - CoqIDE is a simpler stand-alone IDE. It is distributed with Coq, so it should "just work" once you have Coq installed. It can also be compiled from scratch, but on some platforms this may involve installing additional packages for GUI libraries and such.

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

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

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

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

3.4. Downloading the Coq Files. A tar file containing the full sources for the "release version" of these notes (as a collection of Coq scripts and HTML files) is available here:

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

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

4. Translations

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

CHAPTER 2

Basics

REMINDER:

```
##################################################
###  PLEASE DO NOT DISTRIBUTE SOLUTIONS PUBLICLY  ###
##################################################

          (See the Preface for why.)
```

```
||| Basics: Functional Programming in Idris
module Basics

%access public export
```

`postulate` is Idris's "escape hatch" that says accept this definition without proof. Instead of using it to mark the holes, similar to Coq's `Admitted`, we use Idris's holes directly. In practice, holes (and `postulate`) are useful when you're incrementally developing large proofs.

1. Introduction

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

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

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

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

2. Enumerated Types

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

Naturally, the Idris distribution comes with extensive standard libraries providing definitions of booleans, numbers, and many common data structures like lists and hash tables (see the `prelude` and `contrib` packages), as well as the means to write type-safe effectful code (see the `effects` package) and `pruvlioj`, a toolkit for proof automation and program construction. But there is nothing magic or primitive about these library definitions. To illustrate this, we will explicitly recapitulate all the definitions we need in this course, rather than just getting them implicitly from the library.

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

2.1. Days of the Week. The following declaration tells Idris that we are defining a new set of data values – a *type*.

```
namespace Days

  ||| Days of the week.
  data Day = ||| `Monday` is a `Day`.
             Monday
           | ||| `Tuesday` is a `Day`.
             Tuesday
           | ||| `Wednesday` is a `Day`.
             Wednesday
           | ||| `Thursday` is a `Day`.
             Thursday
           | ||| `Friday` is a `Day`.
             Friday
           | ||| `Saturday` is a `Day`.
             Saturday
           | ||| `Sunday` is a `Day`.
             Sunday
```

The type is called `Day`, and its members are `Monday`, `Tuesday`, etc. The right hand side of the definition can be read "`Monday` is a `Day`, `Tuesday` is a `Day`, etc."

Using the \idr{%name} directive, we can tell Idris how to choose default variable names for a particular *type*.

```
%name Day day, day1, day2
```

Now, if Idris needs to choose a name for a variable of type *Day*, it will choose day by default, followed by day1 and day2 if any of its predecessors are already in scope.

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

Type the following:

```
nextWeekday : Day → Day
```

Then with the point on nextWeekday, call *idris-add-clause*.

```
nextWeekday : Day → Day
nextWeekday day = ?nextWeekday_rhs
```

With the point on day, call *idris-case-split*.

```
nextWeekday : Day → Day
nextWeekday Monday = ?nextWeekday_rhs_1
nextWeekday Tuesday = ?nextWeekday_rhs_2
nextWeekday Wednesday = ?nextWeekday_rhs_3
nextWeekday Thursday = ?nextWeekday_rhs_4
nextWeekday Friday = ?nextWeekday_rhs_5
nextWeekday Saturday = ?nextWeekday_rhs_6
nextWeekday Sunday = ?nextWeekday_rhs_7
```

Fill in the proper *Day* constructors and align whitespace as you like.

```
||| Determine the next weekday after a day.
nextWeekday : Day → Day
nextWeekday Monday    = Tuesday
nextWeekday Tuesday   = Wednesday
nextWeekday Wednesday = Thursday
nextWeekday Thursday  = Friday
nextWeekday Friday    = Monday
nextWeekday Saturday  = Monday
nextWeekday Sunday    = Monday
```

Call *idris-load-file* to load the *Basics* module with the finished nextWeekday definition.

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

First, we can evaluate an expression involving nextWeekday in a REPL.

```
λΠ> nextWeekday Friday
Monday : Day

λΠ> nextWeekday (nextWeekday Saturday)
Tuesday : Day
```

Mention other editors? Discuss idris-mode?

We show Idris's responses in comments, but, if you have a computer handy, this would be an excellent moment to fire up the Idris interpreter under your favorite Idris-friendly text editor – such as Emacs or Vim – and try this for and try this for yourself. Load this file, `Basics.lidr` from the book's accompanying Idris sources, find the above example, submit it to the Idris REPL, and observe the result.

Second, we can record what we *expect* the result to be in the form of a proof.

```
||| The second weekday after `Saturday` is `Tuesday`.
testNextWeekday :
  (nextWeekday (nextWeekday Saturday)) = Tuesday
```

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

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

```
testNextWeekday = Refl
```

Edit this

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

(For simple proofs like this, you can call `idris-add-clause` with the point on the name (`testNextWeekday`) in the type signature and then call `idris-proof-search` with the point on the resultant hole to have Idris solve the proof for you.)

Verify the "main uses" claim.

Third, we can ask Idris to *generate*, from our definition, a program in some other, more conventional, programming (C, JavaScript and Node are bundled with Idris) with a high-performance compiler. This facility is very interesting, since it gives us a way to construct *fully certified* programs in mainstream languages. Indeed, this is one of the main uses for which Idris was developed. We'll come back to this topic in later chapters.

3. Booleans

```
namespace Booleans
```

In a similar way, we can define the standard type `Bool` of booleans, with members `False` and `True`.

```
||| Boolean Data Type
data Bool = True | False
```

This definition is written in the simplified style, similar to `Day`. It can also be written in the verbose style:

```
data Bool : Type where
     True : Bool
    False : Bool
```

The verbose style is more powerful because it allows us to assign precise types to individual constructors. This will become very useful later on.

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

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

```
not : (b : Bool) -> Bool
not True  = False
not False = True

andb : (b1 : Bool) -> (b2 : Bool) -> Bool
andb True  b2 = b2
andb False b2 = False

orb : (b1 : Bool) -> (b2 : Bool) -> Bool
orb True  b2 = True
orb False b2 = b2
```

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

```
testOrb1 : (orb True  False) = True
testOrb1 = Refl

testOrb2 : (orb False False) = False
testOrb2 = Refl

testOrb3 : (orb False True)  = True
testOrb3 = Refl

testOrb4 : (orb True  True)  = True
testOrb4 = Refl
```

Edit this.

We can also introduce some familiar syntax for the boolean operations we have just defined. The `syntax` command defines a new symbolic notation for an existing definition, and `infixl` specifies left-associative fixity.

```
infixl 4 &&, ||

(&&) : Bool → Bool → Bool
(&&) = andb

(||) : Bool → Bool → Bool
(||) = orb

testOrb5 : False || False || True = True
testOrb5 = Refl
```

3.0.1. *Exercise: 1 star (nandb).* Fill in the hole `?nandb_rhs` and complete the following function; then make sure that the assertions below can each be verified by Idris. (Fill in each of the holes, following the model of the `orb` tests above.) The function should return `True` if either or both of its inputs are `False`.

```
nandb : (b1 : Bool) → (b2 : Bool) → Bool
nandb b1 b2 = ?nandb_rhs

test_nandb1 : (nandb True  False) = True
test_nandb1 = ?test_nandb1_rhs

test_nandb2 : (nandb False False) = True
test_nandb2 = ?test_nandb2_rhs

test_nandb3 : (nandb False True)  = True
test_nandb3 = ?test_nandb3_rhs

test_nandb4 : (nandb True  True)  = False
test_nandb4 = ?test_nandb4_rhs
```

□

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

```
andb3 : (b1 : Bool) → (b2 : Bool) → (b3 : Bool) → Bool
andb3 b1 b2 b3 = ?andb3_rhs

test_andb31 : (andb3 True  True  True)  = True
test_andb31 = ?test_andb31_rhs

test_andb32 : (andb3 False True  True)  = False
test_andb32 = ?test_andb32_rhs

test_andb33 : (andb3 True  False True)  = False
test_andb33 = ?test_andb33_rhs

test_andb34 : (andb3 True  True  False) = False
test_andb34 = ?test_andb34_rhs
```

□

4. Function Types

Every expression in Idris has a type, describing what sort of thing it computes. The `:type` (or `:t`) REPL command asks Idris to print the type of an expression.

For example, the type of `not True` is `Bool`.

```
λΠ> :type True
True : Bool
λΠ> :t not True
not True : Bool
```

Confirm the "function types" wording.

Functions like `not` itself are also data values, just like `True` and `False`. Their types are called *function types*, and they are written with arrows.

```
λΠ> :t not
not : Bool → Bool
```

The type of `not`, written `Bool → Bool` and pronounced "`Bool` arrow `Bool`," can be read, "Given an input of type `Bool`, this function produces an output of type `Bool`." Similarly, the type of `andb`, written `Bool → Bool → Bool`, can be read, "Given two inputs, both of type `Bool`, this function produces an output of type `Bool`."

5. Modules

Flesh this out and discuss namespaces

Idris provides a *module system*, to aid in organizing large developments.

6. Numbers

```
namespace Numbers
```

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

```
data Nat : Type where
       Z : Nat
       S : Nat → Nat
```

The clauses of this definition can be read:

- `Z` is a natural number.
- `S` is a "constructor" that takes a natural number and yields another one – that is, if `n` is a natural number, then `S n` is too.

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

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

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

The same rules apply for our definitions of `Day` and `Bool`. The annotations we used for their constructors are analogous to the one for the `Z` constructor, indicating that they don't take any arguments.

These three conditions are the precise force of inductive declarations. They imply that the expression `Z`, the expression `S Z`, the expression `S (S Z)`, the expression `S (S (S Z))` and so on all belong to the set `Nat`, while other expressions like `True`, `andb True False`, and `S (S False)` do not.

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

```
pred : (n : Nat) → Nat
pred  Z     = Z
pred (S k) = k
```

The second branch can be read: "if `n` has the form `S k` for some `k`, then return `k`."

```
minusTwo : (n : Nat) → Nat
minusTwo  Z         = Z
minusTwo (S  Z)     = Z
minusTwo (S (S k)) = k
```

Because natural numbers are such a pervasive form of data, Idris provides a tiny bit of built-in magic for parsing and printing them: ordinary Arabic numerals can be used as an alternative to the "unary" notation defined by the constructors `S` and `Z`. Idris prints numbers in Arabic form by default:

```
λΠ> S (S (S (S Z)))
4 : Nat
λΠ> minusTwo 4
2 : Nat
```

The constructor `S` has the type `Nat → Nat`, just like the functions `minusTwo` and `pred`:

```
λΠ> :t S
λΠ> :t pred
λΠ> :t minusTwo
```

These are all things that can be applied to a number to yield a number. However, there is a fundamental difference between the first one and the other two: functions like `pred` and `minusTwo` come with *computation rules* – e.g., the definition of `pred` says

that `pred 2` can be simplified to `1` – while the definition of `S` has no such behavior attached. Although it is like a function in the sense that it can be applied to an argument, it does not *do* anything at all!

For most function definitions over numbers, just pattern matching is not enough: we also need recursion. For example, to check that a number `n` is even, we may need to recursively check whether `n-2` is even.

```
||| Determine whether a number is even.
||| @n a number
evenb : (n : Nat) -> Bool
evenb  Z        = True
evenb (S  Z)    = False
evenb (S (S k)) = evenb k
```

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

```
||| Determine whether a number is odd.
||| @n a number
oddb : (n : Nat) -> Bool
oddb n = not (evenb n)

testOddb1 : oddb 1 = True
testOddb1 = Refl

testOddb2 : oddb 4 = False
testOddb2 = Refl
```

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

```
namespace Playground2

  plus : (n : Nat) -> (m : Nat) -> Nat
  plus  Z    m = m
  plus (S k) m = S (Playground2.plus k m)
```

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

```
ΛΠ> plus 3 2
```

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

```
plus (S (S (S Z))) (S (S Z))
```

↪ `S (plus (S (S Z)) (S (S Z)))` by the second clause of `plus`

↪ `S (S (plus (S Z) (S (S Z))))` by the second clause of `plus`

↪ `S (S (S (plus Z (S (S Z)))))` by the second clause of `plus`

↪ `S (S (S (S (S Z))))` by the first clause of `plus`

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

```
mult : (n, m : Nat) → Nat
mult  Z    = Z
mult (S k) = plus m (mult k m)

testMult1 : (mult 3 3) = 9
testMult1 = Refl
```

You can match two expressions at once:

```
minus : (n, m : Nat) → Nat
minus  Z     _    = Z
minus  n     Z    = n
minus (S k) (S j) = minus k j
```

Verify this.

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

```
exp : (base, power : Nat) → Nat
exp base  Z    = S Z
exp base (S p) = mult base (exp base p)
```

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

$$factorial(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times factorial(n-1), & \text{otherwise} \end{cases}$$

Translate this into Idris.

```
factorial : (n : Nat) → Nat
factorial n = ?factorial_rhs

testFactorial1 : factorial 3 = 6
testFactorial1 = ?testFactorial1_rhs

testFactorial2 : factorial 5 = mult 10 12
testFactorial2 = ?testFactorial2_rhs
```

□

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

```
syntax [x] "+" [y] = plus  x y
syntax [x] "-" [y] = minus x y
syntax [x] "*" [y] = mult  x y
```

```
λΠ> :t (0 + 1) + 1
```

(The details are not important, but interested readers can refer to the optional "More on Syntax" section at the end of this chapter.)

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

Mention interfaces here? Say this is infix

The `(==)` function tests *Nat*ural numbers for equality, yielding a *Bool*ean.

```
||| Test natural numbers for equality.
(==) : (n, m : Nat) → Bool
(==)  Z     Z    = True
(==)  Z    (S j) = False
(==) (S k)  Z    = False
(==) (S k) (S j) = (==) k j
```

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

```
||| Test whether a number is less than or equal to another.
lte : (n, m : Nat) → Bool
lte  Z     m    = True
lte  n     Z    = False
lte (S k) (S j) = lte k j
```

```
testLte1 : lte 2 2 = True
testLte1 = Refl

testLte2 : lte 2 4 = True
testLte2 = Refl

testLte3 : lte 4 2 = False
testLte3 = Refl
```

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

```
blt_nat : (n, m : Nat) → Bool
blt_nat n m = ?blt_nat_rhs

test_blt_nat_1 : blt_nat 2 2 = False
test_blt_nat_1 = ?test_blt_nat_1_rhs
```

```
test_blt_nat_2 : blt_nat 2 4 = True
test_blt_nat_2 = ?test_blt_nat_2_rhs

test_blt_nat_3 : blt_nat 4 2 = False
test_blt_nat_3 = ?test_blt_nat_3_rhs
```

□

7. Proof by Simplification

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

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

```
plus_Z_n : (n : Nat) -> 0 + n = n
plus_Z_n n = Refl
```

It will be useful later to know that [reflexivity] does some simplification – for example, it tries "unfolding" defined terms, replacing them with their right-hand sides. The reason for this is that, if reflexivity succeeds, the whole goal is finished and we don't need to look at whatever expanded expressions *Refl* has created by all this simplification and unfolding.

Other similar theorems can be proved with the same pattern.

```
plus_1_l : (n : Nat) -> 1 + n = S n
plus_1_l n = Refl

mult_0_l : (n : Nat) -> 0 * n = 0
mult_0_l n = Refl
```

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

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

```
plus_n_Z : (n : Nat) -> n = n + 0
plus_n_Z n = Refl
```

```
When checking right hand side of plus_n_Z with expected type
             n = n + 0

     Type mismatch between
```

```
        plus n 0 = plus n 0 (Type of Refl)
and
        n = plus n 0 (Expected type)

Specifically:
        Type mismatch between
                plus n 0
        and
                n
```

(Can you explain why this happens?)

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

8. Proof by Rewriting

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

```
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 the existence of some numbers `n` and `m`. We also need to assume the hypothesis `n = m`.

Edit, mention the "generate initial pattern match" editor command

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 Idris to perform this replacement is called `rewrite`.

```
plus_id_example n m prf = rewrite prf in Refl
```

The first two variables on the left side move the universally quantified variables `n` and `m` into the context. The third moves the hypothesis `n = m` into the context and gives it the name `prf`. The right side tells Idris to rewrite the current goal (`n + n = m + m`) by replacing the left side of the equality hypothesis `prf` with the right side.

8.0.1. *Exercise: 1 star (plus_id_exercise).* Fill in the proof.

```
plus_id_exercise : (n, m, o : Nat) → (n = m) → (m = o) → n + m = m + o
plus_id_exercise n m o prf prf1 = ?plus_id_exercise_rhs
```

□

The prefix `?` on the right-hand side of an equation tells Idris that we want to skip trying to prove this theorem and just leave a hole. This can be useful for developing longer proofs, since we can state subsidiary lemmas that we believe will be useful for making some larger argument, use holes to delay defining them 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.

Decide whether to discuss `postulate`.

```
-- Be careful, though: every time you say `postulate` you are leaving a door
-- open for total nonsense to enter Idris's nice, rigorous, formally checked
-- world!
```

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

```
mult_0_plus : (n, m : Nat) → (0 + n) * m = n * (0 + m)
mult_0_plus n m = Refl
```

Unlike in Coq, we don't need to perform such a rewrite for `mult_0_plus` in Idris and can just use `Refl` instead.

8.0.2. *Exercise: 2 starts (mult_S_1).*

```
mult_S_1 : (n, m : Nat) → (m = S n) → m * (1 + n) = m * m
mult_S_1 n m prf = ?mult_S_1_rhs
```

□

9. 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 `Refl` tactic as above, we get stuck.

```
plus_1_neq_0_firsttry : (n : Nat) → (n + 1) == 0 = False
plus_1_neq_0_firsttry n = Refl -- does nothing!
```

The reason for this is that the definitions of both `(==)` and `+` begin by performing a `match` on their first argument. But here, the first argument to `+` is the unknown number `n` and the argument to `(==)` 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 `Z`, then we can calculate the final result of `(n + 1) == 0` and check that it is, indeed, `False`. And if `n = S k` for some `k`, then, although we don't know exactly what number

`n + 1` yields, we can calculate that, at least, it will begin with one `S`, and this is enough to calculate that, again, `(n + 1) == 0` will yield `False`.

To tell Idris to consider, separately, the cases where `n = Z` and where `n = S k`, simply case split on `n`.

Mention case splitting interactively in Emacs, Atom, etc.

```
plus_1_neq_0 : (n : Nat) -> (n + 1) == 0 = False
plus_1_neq_0  Z    = Refl
plus_1_neq_0 (S k) = Refl
```

Case splitting on `n` generates *two* holes, which we must then prove, separately, in order to get Idris to accept the theorem.

In this example, each of the holes is easily filled by a single use of `Refl`, which itself performs some simplification – e.g., the first one simplifies `(S k + 1) == 0` to `False` by first rewriting `(S k + 1)` to `S (k + 1)`, then unfolding `(==)`, simplifying its pattern matching.

There are no hard and fast rules for how proofs should be formatted in Idris. However, if the places where multiple holes are generated are lifted to lemmas, 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 Idris users sometimes tend to the extremes, either writing each tactic on its own line or writing entire proofs on one line. Good style lies somewhere in the middle. One reasonable convention is to limit yourself to 80-character lines.

The case splitting strategy 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.

```
||| A proof that boolean negation is involutive.
not_involutive : (b : Bool) -> not (not b) = b
not_involutive True  = Refl
not_involutive False = Refl
```

Note that the case splitting here doesn't introduce any variables because none of the subcases of the patterns need to bind any, so there is no need to specify any names.

It is sometimes useful to case split on more than one parameter, generating yet more proof obligations. For example:

```
andb_commutative : (b, c : Bool) -> b && c = c && b
andb_commutative True  True  = Refl
andb_commutative True  False = Refl
andb_commutative False True  = Refl
andb_commutative False False = Refl
```

In more complex proofs, it is often better to lift subgoals to lemmas:

```
andb_commutative'_rhs_1 : (c : Bool) → c = c && True
andb_commutative'_rhs_1 True  = Refl
andb_commutative'_rhs_1 False = Refl

andb_commutative'_rhs_2 : (c : Bool) → False = c && False
andb_commutative'_rhs_2 True  = Refl
andb_commutative'_rhs_2 False = Refl

andb_commutative' : (b, c : Bool) → b && c = c && b
andb_commutative' True  = andb_commutative'_rhs_1
andb_commutative' False = andb_commutative'_rhs_2
```

9.0.1. *Exercise: 2 stars (andb_true_elim2).* Prove the following claim, lift cases (and subcases) to lemmas when case split.

```
andb_true_elim_2 : (b, c : Bool) → (b && c = True) → c = True
andb_true_elim_2 b c prf = ?andb_true_elim_2_rhs
```

□

9.0.2. *Exercise: 1 star (zero_nbeq_plus_1).*

```
zero_nbeq_plus_1 : (n : Nat) → 0 = (n + 1) = False
zero_nbeq_plus_1 n = ?zero_nbeq_plus_1_rhs
```

□

Discuss associativity.

10. Structural Recursion (Optional)

Here is a copy of the definition of addition:

```
plus' : Nat → Nat → Nat
plus'  Z       right = right
plus' (S left) right = S (plus' left right)
```

When Idris 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 `left` – i.e., that we make recursive calls only on strictly smaller values of `left`. This implies that all calls to `plus'` will eventually terminate. Idris demands that some argument of *every* recursive definition is "decreasing."

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

Verify the previous claims.

Add decreasing exercise.

11. More Exercises

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

```
identity_fn_applied_twice : (f : Bool → Bool) →
                            ((x : Bool) → f x = x) →
                            (b : Bool) → f (f b) = b
identity_fn_applied_twice f g b = ?identity_fn_applied_twice_rhs
```

Now state and prove a theorem `negation_fn_applied_twice` similar to the previous one but where the second hypothesis says that the function `f` has the property that `f x = not x`.

```
-- FILL IN HERE
```

□

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

```
andb_eq_orb : (b, c : Bool) → (b && c = b || c) → b = c
andb_eq_orb b c prf = ?andb_eq_orb_rhs
```

□

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

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

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

(Hint: Recall that the definition of `Nat` from class,

```
data Nat : Type where
       Z : Nat
       S : Nat → Nat
```

says nothing about what `Z` and `S` "mean." It just says "`Z` is in the set called `Nat`, and if `n` is in the set then so is `S n`." The interpretation of `Z` as zero and `S` as successor/plus one comes from the way that we *use* `Nat` values, by writing functions to do things with them, proving things about them, and so on. Your definition of `Bin` should be correspondingly simple; it is the functions you will write next that will give it mathematical meaning.)

(b) Next, write an increment function `incr` for binary numbers, and a function `bin_to_nat` to convert binary numbers to unary numbers.

(c) Write five unit tests `test_bin_incr_1`, `test_bin_incr_2`, etc. for your increment and binary-to-unary functions. Notice that incrementing a binary number and then converting it to unary should yield the same result as first converting it to unary and then incrementing.

```
-- FILL IN HERE
```

□

CHAPTER 3

Induction : Proof by Induction

```
module Induction
```

First, we import all of our definitions from the previous chapter.

```
import Basics
```

Next, we import the following *Prelude* modules, since we'll be dealing with natural numbers.

```
import Prelude.Interfaces
import Prelude.Nat
```

For `import Basics` to work, you first need to use `idris` to compile `Basics.lidr` into `Basics.ibc`. This is like making a .class file from a .java file, or a .o file from a .c file. There are at least two ways to do it:

- In your editor with an Idris plugin, e.g. Emacs:

  Open `Basics.lidr`. Evaluate `idris-load-file`.

  There exists similar support for Vim, Sublime Text and Visual Studio Code as well.

- From the command line:

  Run `idris --check --total --noprelude src/Basics.lidr`.

  Refer to the Idris man page (or `idris --help` for descriptions of the flags.

```
%access public export

%default total
```

1. Proof by Induction

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

```
Theorem plus_n_O_firsttry : forall n:nat,
  n = n + 0.
```

... cannot be proved in the same simple way in Coq, but as we saw in `Basics`, Idris's *Refl* just works.

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

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

- show that `P Z` holds;
- show that, for any `k`, if `P k` holds, then so does `P (S k)`;
- conclude that `P n` holds for all `n`.

In Idris, the steps are the same and can often be written as function clauses by case splitting. Here's how this works for the theorem at hand.

```
plus_n_Z : (n : Nat) -> n = n + 0
plus_n_Z  Z    = Refl
plus_n_Z (S k) =
  let inductiveHypothesis = plus_n_Z k in
    rewrite inductiveHypothesis in Refl
```

In the first clause, `n` is replaced by `Z` and the goal becomes `0 = 0`, which follows by `Refl`exivity. In the second, `n` is replaced by `S k` and the goal becomes `S k = S (plus k 0)`. Then we define the inductive hypothesis, `k = k + 0`, which can be written as `plus_n_Z k`, and the goal follows from it.

```
minus_diag : (n : Nat) -> minus n n = 0
minus_diag  Z    = Refl
minus_diag (S k) = minus_diag k
```

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

```
mult_0_r : (n : Nat) -> n * 0 = 0
mult_0_r n = ?mult_0_r_rhs

plus_n_Sm : (n, m : Nat) -> S (n + m) = n + (S m)
plus_n_Sm n m = ?plus_n_Sm_rhs

plus_comm : (n, m : Nat) -> n + m = m + n
plus_comm n m = ?plus_comm_rhs

plus_assoc : (n, m, p : Nat) -> n + (m + p) = (n + m) + p
plus_assoc n m p = ?plus_assoc_rhs
```

□

1.0.2. *Exercise: 2 stars (double_plus).* Consider the following function, which doubles its argument:

```
double : (n : Nat) -> Nat
double  Z    = Z
double (S k) = S (S (double k))
```

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

```
double_plus : (n : Nat) → double n = n + n
double_plus n = ?double_plus_rhs
```

□

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

```
evenb_S : (n : Nat) → evenb (S n) = not (evenb n)
evenb_S n = ?evenb_S_rhs
```

□

2. Proofs Within Proofs

Edit the section

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

```
mult_0_plus' : (n, m : Nat) → (0 + n) * m = n * m
mult_0_plus' n m = Refl
```

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

The `assert` tactic is handy in many sorts of situations. For example, suppose we want to prove that `(n + m) + (p + q) = (m + n) + (p + q)`. The only difference between the two sides of the `=` is that the arguments `m` and `n` to the first inner `+` are swapped, so it seems we should be able to use the commutativity of addition (`plus_comm`) to rewrite one into the other. However, the `rewrite` tactic is a little

stupid about *where* it applies the rewrite. There are three uses of + here, and it turns out that doing rewrite → plus_comm will affect only the *outer* one…

```
plus_rearrange_firsttry : (n, m, p, q : Nat) →
                          (n + m) + (p + q) = (m + n) + (p + q)
plus_rearrange_firsttry n m p q = rewrite plus_comm in Refl
```

```
When checking right hand side of plus_rearrange_firsttry with expected type
        n + m + (p + q) = m + n + (p + q)

_ does not have an equality type ((n1 : Nat) →
(n1 : Nat) → plus n1 m1 = plus m1 n1)
```

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

```
plus_rearrange : (n, m, p, q : Nat) →
                 (n + m) + (p + q) = (m + n) + (p + q)
plus_rearrange n m p q = rewrite plus_rearrange_lemma n m in Refl
  where
  plus_rearrange_lemma : (n, m : Nat) → n + m = m + n
  plus_rearrange_lemma = plus_comm
```

3. More Exercises

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

```
plus_swap : (n, m, p : Nat) → n + (m + p) = m + (n + p)
plus_swap n m p = ?plus_swap_rhs
```

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

```
mult_comm : (m, n : Nat) → m * n = n * m
mult_comm m n = ?mult_comm_rhs
```

□

3.0.2. *Exercise: 3 stars, optional (more_exercises).*

Edit

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!)

```
lte_refl : (n : Nat) -> True = lte n n
lte_refl n = ?lte_refl_rhs

zero_nbeq_S : (n : Nat) -> 0 == (S n) = False
zero_nbeq_S n = ?zero_nbeq_S_rhs

andb_false_r : (b : Bool) -> b && False = False
andb_false_r b = ?andb_false_r_rhs

plus_ble_compat_l : (n, m, p : Nat) ->
                    lte n m = True -> lte (p + n) (p + m) = True
plus_ble_compat_l n m p prf = ?plus_ble_compat_l_rhs

S_nbeq_0 : (n : Nat) -> (S n) == 0 = False
S_nbeq_0 n = ?S_nbeq_0_rhs

mult_1_l : (n : Nat) -> 1 * n = n
mult_1_l n = ?mult_1_l_rhs

all3_spec : (b, c : Bool) ->
            (b && c) || ((not b) || (not c)) = True
all3_spec b c = ?all3_spec_rhs

mult_plus_distr_r : (n, m, p : Nat) -> (n + m) * p = (n * p) + (m * p)
mult_plus_distr_r n m p = ?mult_plus_distr_r_rhs

mult_assoc : (n, m, p : Nat) -> n * (m * p) = (n * m) * p
mult_assoc n m p = ?mult_assoc_rhs
```

□

3.0.3. *Exercise: 2 stars, optional (beq_nat_refl).*

Edit

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

```
beq_nat_refl : (n : Nat) -> True = n == n
beq_nat_refl n = ?beq_nat_refl_rhs
```

□

3.0.4. *Exercise: 2 stars, optional (plus_swap').*

Edit

The `replace` tactic allows you to specify a particular subterm to rewrite and what you want it rewritten to: `replace (t) with (u)` replaces (all copies of) expression `t` in the goal by expression `u`, and generates `t = u` as an additional subgoal. This is often useful when a plain `rewrite` acts on the wrong part of the goal.

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

```
plus_swap' : (n, m, p : Nat) → n + (m + p) = m + (n + p)
plus_swap' n m p = ?plus_swap'_rhs
```

□

3.0.5. *Exercise: 3 stars, recommended (binary_commute).* Recall the `incr` and `bin_to_nat` functions that you wrote for the `binary` exercise in the `Basics` chapter. Prove that the following diagram commutes:

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

That is, incrementing a binary number and then converting it to a (unary) natural number yields the same result as first converting it to a natural number and then incrementing. Name your theorem `bin_to_nat_pres_incr` ("pres" for "preserves").

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

□

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

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

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

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

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

□

CHAPTER 4

Lists : Working with Structured Data

```
module Lists

import Basics

%hide Prelude.Basics.fst
%hide Prelude.Basics.snd
%hide Prelude.Nat.pred
%hide Prelude.List.(++)

%access public export
%default total
```

1. Pairs of Numbers

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

```
data NatProd : Type where
  Pair : Nat → Nat → NatProd
```

This declaration can be read: "There is just one way to construct a pair of numbers: by applying the constructor *Pair* to two arguments of type *Nat*."

```
λΠ> :t Pair 3 5
```

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

```
fst : (p : NatProd) → Nat
fst (Pair x y) = x

snd : (p : NatProd) → Nat
snd (Pair x y) = y

λΠ> fst (Pair 3 5)
3 : Nat
```

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

```
syntax "(" [x] "," [y] ")" = Pair x y
```

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

```
ΛΠ> fst (3,5)
3 : Nat
```

```
fst' : (p : NatProd) -> Nat
fst' (x,y) = x

snd' : (p : NatProd) -> Nat
snd' (x,y) = y

swap_pair : (p : NatProd) -> NatProd
swap_pair (x,y) = (y,x)
```

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

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

```
surjective_pairing' : (n,m : Nat) -> (n,m) = (fst (n,m), snd (n,m))
surjective_pairing' n m = Refl
```

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

```
surjective_pairing_stuck : (p : NatProd) -> p = (fst p, snd p)
surjective_pairing_stuck p = Refl
```

```
When checking right hand side of
  surjective_pairing_stuck with expected type p = Pair (fst p) (snd p)
...
Type mismatch between p and Pair (fst p) (snd p)
```

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

```
surjective_pairing : (p : NatProd) -> p = (fst p, snd p)
surjective_pairing p = case p of (n,m) => Refl
```

Notice that `case` matches just one pattern here. That's because `NatProd`s can only be constructed in one way.

1.1. Exercise: 1 star (snd_fst_is_swap).

```
snd_fst_is_swap : (p : NatProd) -> (snd p, fst p) = swap_pair p
snd_fst_is_swap p = ?snd_fst_is_swap_rhs
```

□

1.2. Exercise: 1 star, optional (fst_swap_is_snd).

```
fst_swap_is_snd : (p : NatProd) -> fst (swap_pair p) = snd p
fst_swap_is_snd p = ?fst_swap_is_snd_rhs
```

□

2. 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."

```
data NatList : Type where
  Nil : NatList
  (::) : Nat → NatList → NatList
```

For example, here is a three-element list:

```
mylist : NatList
mylist = (::) 1 ((::) 2 ((::) 3 Nil))
```

Edit the section - Idris's list sugar automatically works for anything with constructors `Nil` and `(::)`

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

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

```
mylist1 : NatList
mylist1 = 1 :: (2 :: (3 :: Nil))

mylist2 : NatList
mylist2 = 1::2::3::[]

mylist3 : NatList
mylist3 = [1,2,3]
```

The at level 60 part tells Coq how to parenthesize expressions that involve both :: and some other infix operator. For example, since we defined + as infix notation for the plus function at level 50,

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

the `+` operator will bind tighter than `::` , so `1 + 2 :: [3]` will be parsed, as we'd expect, as `(1 + 2) :: [3]` rather than `1 + (2 :: [3])`.

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

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

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

```
repeat : (n, count : Nat) → NatList
repeat n Z = []
repeat n (S k) = n :: repeat n k
```

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

```
length : (l : NatList) → Nat
length [] = Z
length (h :: t) = S (length t)
```

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

```
app : (l1, l2 : NatList) → NatList
app [] l2 = l2
app (h :: t) l2 = h :: app t l2
```

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

```
infixr 7 ++

(++) : (x, y : NatList) → NatList
(++) = app

test_app1 : [1,2,3] ++ [4,5,6] = [1,2,3,4,5,6]
test_app1 = Refl

test_app2 : [] ++ [4,5] = [4,5]
test_app2 = Refl

test_app3 : [1,2,3] ++ [] = [1,2,3]
test_app3 = Refl
```

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

```
hd : (default : Nat) → (l : NatList) → Nat
hd default [] = default
hd default (h :: t) = h
```

```
tl : (l : NatList) → NatList
tl [] = []
tl (h :: t) = t

test_hd1 : hd 0 [1,2,3] = 1
test_hd1 = Refl

test_hd2 : hd 0 [] = 0
test_hd2 = Refl

test_tl : tl [1,2,3] = [2,3]
test_tl = Refl
```

2.5. Exercises.

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

```
nonzeros : (l : NatList) → NatList
nonzeros l = ?nonzeros_rhs

test_nonzeros : nonzeros [0,1,0,2,3,0,0] = [1,2,3]
test_nonzeros = ?test_nonzeros_rhs

oddmembers : (l : NatList) → NatList
oddmembers l = ?oddmembers_rhs

test_oddmembers : oddmembers [0,1,0,2,3,0,0] = [1,3]
test_oddmembers = ?test_oddmembers_rhs

countoddmembers : (l : NatList) → Nat
countoddmembers l = ?countoddmembers_rhs

test_countoddmembers1 : countoddmembers [1,0,3,1,4,5] = 4
test_countoddmembers1 = ?test_countoddmembers1_rhs
```

□

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

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

```
alternate : (l1, l2 : NatList) → NatList
alternate l1 l2 = ?alternate_rhs
```

```
test_alternate1 : alternate [1,2,3] [4,5,6] =
                            [1,4,2,5,3,6]
test_alternate1 = ?test_alternate1_rhs

test_alternate2 : alternate [1] [4,5,6] = [1,4,5,6]
test_alternate2 = ?test_alternate2_rhs

test_alternate3 : alternate [1,2,3] [4] = [1,4,2,3]
test_alternate3 = ?test_alternate3_rhs

test_alternate4 : alternate [] [20,30] = [20,30]
test_alternate4 = ?test_alternate4_rhs
```

□

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

```
Bag : Type
Bag = NatList
```

2.6.1. *Exercise: 3 stars, recommended (bag_functions).* Complete the following definitions for the functions `count`, `sum`, `add`, and `member` for bags.

```
count : (v : Nat) → (s : Bag) → Nat
count v s = ?count_rhs
```

All these proofs can be done just by *Refl*.

```
test_count1 : count 1 [1,2,3,1,4,1] = 3
test_count1 = ?test_count1_rhs

test_count2 : count 6 [1,2,3,1,4,1] = 0
test_count2 = ?test_count2_rhs
```

Multiset `sum` is similar to set `union`: `sum a b` contains all the elements of `a` and of `b`. (Mathematicians usually define union on multisets a little bit differently, which is why we don't use that name for this operation.)

How to forbid recursion here? Edit

For `sum` we're giving you a header that does not give explicit names to the arguments. Moreover, it uses the keyword Definition instead of Fixpoint, so even if you had names for the arguments, you wouldn't be able to process them recursively. The point of stating the question this way is to encourage you to think about whether sum can be implemented in another way – perhaps by using functions that have already been defined.

```
sum : Bag → Bag → Bag
sum x y = ?sum_rhs

test_sum1 : count 1 (sum [1,2,3] [1,4,1]) = 3
test_sum1 = ?test_sum1_rhs
```

```
add : (v : Nat) → (s : Bag) → Bag
add v s = ?add_rhs

test_add1 : count 1 (add 1 [1,4,1]) = 3
test_add1 = ?test_add1_rhs

test_add2 : count 5 (add 1 [1,4,1]) = 0
test_add2 = ?test_add2_rhs

member : (v : Nat) → (s : Bag) → Bool
member v s = ?member_rhs

test_member1 : member 1 [1,4,1] = True
test_member1 = ?test_member1_rhs

test_member2 : member 2 [1,4,1] = False
test_member2 = ?test_member2_rhs
```

□

2.6.2. *Exercise: 3 stars, 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.

```
remove_one : (v : Nat) → (s : Bag) → Bag
remove_one v s = ?remove_one_rhs

test_remove_one1 : count 5 (remove_one 5 [2,1,5,4,1]) = 0
test_remove_one1 = ?test_remove_one1_rhs

test_remove_one2 : count 5 (remove_one 5 [2,1,4,1]) = 0
test_remove_one2 = ?test_remove_one2_rhs

test_remove_one3 : count 4 (remove_one 5 [2,1,5,4,1,4]) = 2
test_remove_one3 = ?test_remove_one3_rhs

test_remove_one4 : count 5 (remove_one 5 [2,1,5,4,5,1,4]) = 1
test_remove_one4 = ?test_remove_one4_rhs

remove_all : (v : Nat) → (s : Bag) → Bag
remove_all v s = ?remove_all_rhs

test_remove_all1 : count 5 (remove_all 5 [2,1,5,4,1]) = 0
test_remove_all1 = ?test_remove_all1_rhs

test_remove_all2 : count 5 (remove_all 5 [2,1,4,1]) = 0
test_remove_all2 = ?test_remove_all2_rhs

test_remove_all3 : count 4 (remove_all 5 [2,1,5,4,1,4]) = 2
test_remove_all3 = ?test_remove_all3_rhs

test_remove_all4 : count 5
                   (remove_all 5 [2,1,5,4,5,1,4,5,1,4]) = 0
test_remove_all4 = ?test_remove_all4_rhs
```

```
subset : (s1 : Bag) → (s2 : Bag) → Bool
subset s1 s2 = ?subset_rhs

test_subset1 : subset [1,2] [2,1,4,1] = True
test_subset1 = ?test_subset1_rhs

test_subset2 : subset [1,2,2] [2,1,4,1] = False
test_subset2 = ?test_subset2_rhs
```

□

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

```
bag_theorem : ?bag_theorem
```

□

3. Reasoning About Lists

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

```
nil_app : (l : NatList) → ([] ++ l) = l
nil_app l = Refl
```

… because the `[]` is substituted into the "scrutinee" (the value 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.

```
tl_length_pred : (l : NatList) → pred (length l) = length (tl l)
tl_length_pred [] = Refl
tl_length_pred (n::l') = Refl
```

Here, the *Nil* case works because we've chosen to define `tl Nil = Nil`. Notice that the case for *Cons* 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.

3.0.1. *Micro-Sermon.* Simply reading example proof scripts will not get you very far! It is important to work through the details of each one, using Idris and thinking about what each step achieves. Otherwise it is more or less guaranteed that the exercises will make no sense when you get to them. 'Nuff said.

3.1. 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 `data` declaration defines a set of data values that can be built up using the declared constructors: a boolean can be either `True` or `False`; a number can be either `Z` or `S` applied to another number; a list can be either `Nil` or `Cons` applied to a number and a list.

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

- First, show that `P` is true of `l` when `l` is `Nil`.
- Then show that `P` is true of `l` when `l` is `Cons n l'` for some number `n` and some smaller list `l'`, assuming that `P` is true for `l'`.

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

```
app_assoc : (l1, l2, l3 : NatList) → ((l1 ++ l2) ++ l3) = (l1 ++ (l2 ++ l3))
app_assoc [] l2 l3 = Refl
app_assoc (n::l1') l2 l3 =
  let inductiveHypothesis = app_assoc l1' l2 l3 in
    rewrite inductiveHypothesis in Refl
```

Edit

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

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

Theorem: For all lists `l1`, `l2`, and `l3`,

```
\idr{(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) ++ l 3) = n :: (l1' ++ (l2 ++ l3))`,

  which is immediate from the induction hypothesis. □

3.1.1. *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`:

```
rev : (l : NatList) → NatList
rev Nil = Nil
rev (h :: t) = (rev t) ++ [h]

test_rev1 : rev [1,2,3] = [3,2,1]
test_rev1 = Refl

test_rev2 : rev Nil = Nil
test_rev2 = Refl
```

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

```
rev_length_firsttry : (l : NatList) → length (rev l) = length l
rev_length_firsttry Nil = Refl
rev_length_firsttry (n :: l') =
-- Now we seem to be stuck: the goal is an equality involving `++`, but we don't
-- have any useful equations in either the immediate context or in the global
-- environment! We can make a little progress by using the IH to rewrite the
-- goal...
  let inductiveHypothesis = rev_length_firsttry l' in
    rewrite inductiveHypothesis in
-- ... but now we can't go any further.
      Refl
```

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

```
app_length : (l1, l2 : NatList) →
             length (l1 ++ l2) = (length l1) + (length l2)
app_length Nil l2 = Refl
app_length (n :: l1') l2 =
```

```
  let inductiveHypothesis = app_length l1' l2 in
    rewrite inductiveHypothesis in
      Refl
```

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

Now we can complete the original proof.

```
rev_length : (l : NatList) → length (rev l) = length l
rev_length Nil = Refl
rev_length (n :: l') =
  rewrite app_length (rev l') [n] in
-- Prelude's version of `Induction.plus_comm`
    rewrite plusCommutative (length (rev l')) 1 in
      let inductiveHypothesis = rev_length l' in
        rewrite inductiveHypothesis in Refl
```

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

Theorem: For all lists `l1` and `l2`,

```
\idr{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 `++`.

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

Theorem: For all lists `l`, `length (rev l) = length l`.

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

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

3.2. Search.

Edit, mention `:s` and `:apropos`?

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

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

```
(* Search rev. *)
```

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!

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

3.3. List Exercises, Part 1.

3.3.1. *Exercise: 3 stars (list_exercises).* More practice with lists:

```
app_nil_r : (l : NatList) → (l ++ []) = l
app_nil_r l = ?app_nil_r_rhs

rev_app_distr : (l1, l2 : NatList) → rev (l1 ++ l2) = (rev l2) ++ (rev l1)
rev_app_distr l1 l2 = ?rev_app_distr_rhs

rev_involutive : (l : NatList) → rev (rev l) = l
rev_involutive l = ?rev_involutive_rhs
```

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.

```
app_assoc4 : (l1, l2, l3, l4 : NatList) →
             (l1 ++ (l2 ++ (l3 ++ l4))) = ((l1 ++ l2) ++ l3) ++ l4
app_assoc4 l1 l2 l3 l4 = ?app_assoc4_rhs
```

An exercise about your implementation of `nonzeros`:

```
nonzeros_app : (l1, l2 : NatList) →
               nonzeros (l1 ++ l2) = (nonzeros l1) ++ (nonzeros l2)
nonzeros_app l1 l2 = ?nonzeros_app_rhs
```

□

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

```
beq_NatList : (l1, l2 : NatList) → Bool
beq_NatList l1 l2 = ?beq_NatList_rhs

test_beq_NatList1 : beq_NatList Nil Nil = True
test_beq_NatList1 = ?test_beq_NatList1_rhs

test_beq_NatList2 : beq_NatList [1,2,3] [1,2,3] = True
test_beq_NatList2 = ?test_beq_NatList2_rhs

test_beq_NatList3 : beq_NatList [1,2,3] [1,2,4] = False
test_beq_NatList3 = ?test_beq_NatList3_rhs

beq_NatList_refl : (l : NatList) → True = beq_NatList l l
beq_NatList_refl l = ?beq_NatList_refl_rhs
```

□

3.4. List Exercises, Part 2.

3.4.1. *Exercise: 3 stars, advanced (bag_proofs).* Here are a couple of little theorems to prove about your definitions about bags above.

```
count_member_nonzero : (s : Bag) → lte 1 (count 1 (1 :: s)) = True
count_member_nonzero s = ?count_member_nonzero_rhs
```

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

```
ble_n_Sn : (n : Nat) -> lte n (S n) = True
ble_n_Sn Z = Refl
ble_n_Sn (S k) =
  let inductiveHypothesis = ble_n_Sn k in
    rewrite inductiveHypothesis in Refl

remove_decreases_count : (s : Bag) ->
                         lte (count 0 (remove_one 0 s)) (count 0 s) = True
remove_decreases_count s = ?remove_decreases_count_rhs
```

□

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

```
bag_count_sum : ?bag_count_sum
```

□

3.4.3. *Exercise: 4 stars, advanced (rev_injective).* Prove that the `rev` function is injective – that is,

```
rev_injective : (l1, l2 : NatList) -> rev l1 = rev l2 -> l1 = l2
rev_injective l1 l2 prf = ?rev_injective_rhs
```

(There is a hard way and an easy way to do this.)

□

4. Options

Suppose we want to write a function that returns the `n`th 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…

```
nth_bad : (l : NatList) -> (n : Nat) -> Nat
nth_bad Nil n = 42 -- arbitrary!
nth_bad (a :: l') n = case n == 0 of
                        True => a
                        False => nth_bad l' (pred n)
```

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

```
data NatOption : Type where
  Some : Nat -> NatOption
  None : NatOption
```

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

```
nth_error : (l : NatList) -> (n : Nat) -> NatOption
nth_error Nil n = None
nth_error (a :: l') n = case n == 0 of
                          True => Some a
                          False => nth_error l' (pred n)

test_nth_error1 : nth_error [4,5,6,7] 0 = Some 4
test_nth_error1 = Refl

test_nth_error2 : nth_error [4,5,6,7] 3 = Some 7
test_nth_error2 = Refl

test_nth_error3 : nth_error [4,5,6,7] 9 = None
test_nth_error3 = Refl
```

This example is also an opportunity to introduce one more small feature of Idris programming language: conditional expressions…

```
nth_error' : (l : NatList) -> (n : Nat) -> NatOption
nth_error' Nil n = None
nth_error' (a :: l') n = if n == 0
                          then Some a
                          else nth_error' l' (pred n)
```

Edit or remove this paragraph, doesn't seem to hold in Idris

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

The function below pulls the *Nat* out of a *NatOption*, returning a supplied default in the *None* case.

```
option_elim : (d : Nat) -> (o : NatOption) -> Nat
option_elim d (Some k) = k
option_elim d None = d
```

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

```
hd_error : (l : NatList) -> NatOption
hd_error l = ?hd_error_rhs

test_hd_error1 : hd_error [] = None
test_hd_error1 = ?test_hd_error1_rhs
```

```
test_hd_error2 : hd_error [1] = Some 1
test_hd_error2 = ?test_hd_error2_rhs

test_hd_error3 : hd_error [5,6] = Some 5
test_hd_error3 = ?test_hd_error3_rhs
```

□

4.0.2. *Exercise: 1 star, optional (option_elim_hd).* This exercise relates your new `hd_error` to the old `hd`.

```
option_elim_hd : (l : NatList) → (default : Nat) →
                 hd default l = option_elim default (hd_error l)
option_elim_hd l default = ?option_elim_hd_rhs
```

□

5. Partial Maps

As a final illustration of how data structures can be defined in Idris, 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.

```
data Id : Type where
  MkId : Nat → Id
```

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

We'll also need an equality test for `Id`s:

```
beq_id : (x1, x2 : Id) → Bool
beq_id (MkId n1) (MkId n2) = n1 == n2
```

5.0.1. *Exercise: 1 star (beq_id_refl).*

```
beq_id_refl : (x : Id) → True = beq_id x x
beq_id_refl x = ?beq_id_refl_rhs
```

□

Now we define the type of partial maps:

```
namespace PartialMap

  data PartialMap : Type where
    Empty : PartialMap
    Record : Id → Nat → PartialMap → PartialMap
```

This declaration can be read: "There are two ways to construct a `PartialMap`: either using the constructor `Empty` to represent an empty partial map, or by applying

the constructor *Record* to a key, a value, and an existing *PartialMap* to construct a *PartialMap* with an additional key-to-value mapping."

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

```
update : (d : PartialMap) → (x : Id) → (value : Nat) → PartialMap
update d x value = Record x value d
```

Last, the find function searches a *PartialMap* 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.

```
find : (x : Id) → (d : PartialMap) → NatOption
find x Empty = None
find x (Record y v d') = if beq_id x y
                         then Some v
                         else find x d'
```

5.0.2. *Exercise: 1 star (update_eq).*

```
update_eq : (d : PartialMap) → (x : Id) → (v : Nat) →
            find x (update d x v) = Some v
update_eq d x v = ?update_eq_rhs
```

□

5.0.3. *Exercise: 1 star (update_neq).*

```
update_neq : (d : PartialMap) → (x, y : Id) → (o : Nat) →
             beq_id x y = False →
             find x (update d y o) = find x d
update_neq d x y o prf = ?update_neq_rhs
```

□

5.0.4. *Exercise: 2 stars (baz_num_elts).* Consider the following inductive definition:

```
data Baz : Type where
  Baz1 : Baz → Baz
  Baz2 : Baz → Bool → Baz
```

How *many* elements does the type *Baz* have? (Answer in English or the natural language of your choice.)

□

CHAPTER 5

Poly : Polymorphism and Higher-Order Functions

```
module Poly

import Basics

%hide Prelude.List.length
%hide Prelude.List.filter
%hide Prelude.List.partition
%hide Prelude.Functor.map
%hide Prelude.Nat.pred
%hide Basics.Playground2.plus

%access public export

%default total
```

1. 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.

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

```
data BoolList : Type where
  BoolNil : BoolList
  BoolCons : Bool → BoolList → BoolList
```

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

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

```
data List : (x : Type) -> Type where
  Nil : List x
  Cons : x -> List x -> List x
```

(This type is already defined in Idris' standard library, but the `Cons` constructor is named `(::)`).

This is exactly like the definition of `NatList` from the previous chapter, except that the `Nat` argument to the `Cons` constructor has been replaced by an arbitrary type `x`, a binding for `x` has been added to the header, and the occurrences of `NatList` in the types of the constructors have been replaced by `List x`. (We can re-use the constructor names `Nil` and `Cons` because the earlier definition of `NatList` was inside of a `namespace` definition that is now out of scope.)

What sort of thing is `List` itself? One good way to think about it is that `List` is a *function* from `Type`s to inductive definitions; or, to put it another way, `List` is a function from `Type`s to `Type`s. For any particular type `x`, the type `List x` is an inductively defined set of lists whose elements are of type `x`.

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

```
λΠ> :t Nil
Prelude.List.Nil : List elem
λΠ> :t (::)
Prelude.List.(::) : elem -> List elem -> List elem
```

How to edit these 3 paragraphs? Implicits are defined later in this chapter, and Idris doesn't require type parameters to constructors

(Side note on notation: In .v files, the "forall" quantifier is spelled out in letters. In the generated HTML files and in the way various IDEs show .v files (with certain settings of their display controls), ∀ is usually typeset as the usual mathematical "upside down A," but 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.)

The "∀ X" in these types can be read as an additional argument to the constructors that determines the expected types of the arguments that follow. When `Nil` and `Cons` are used, these arguments are supplied in the same way as the others. For example, the list containing `2` and `1` is written like this:

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

(We've written `Nil` and `Cons` explicitly here because we haven't yet defined the `[]` and `::` notations for the new version of lists. We'll do that in a bit.)

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

```
repeat : (x_ty : Type) → (x : x_ty) → (count : Nat) → List x_ty
repeat x_ty x Z = Nil
repeat x_ty x (S count') = x :: repeat x_ty x count'
```

As with `Nil` and `Cons`, we can use `repeat` by applying it first to a type and then to its list argument:

```
test_repeat1 : repeat Nat 4 2 = 4 :: (4 :: Nil)
test_repeat1 = Refl
```

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

```
test_repeat2 : repeat Bool False 1 = False :: Nil
test_repeat2 = Refl
```

1.1.1. *Exercise: 2 stars (mumble_grumble).*

Explain implicits and `{x=foo}` syntax first? Move after the "Supplying Type Arguments Explicitly" section?

```
namespace MumbleGrumble
```

Consider the following two inductively defined types.

```
  data Mumble : Type where
    A : Mumble
    B : Mumble → Nat → Mumble
    C : Mumble

  data Grumble : (x : Type) → Type where
    D : Mumble → Grumble x
    E : x → Grumble x
```

Which of the following are well-typed elements of `Grumble x` for some type `x`?

- `D (B A 5)`
- `D (B A 5) {x=Mumble}`
- `D (B A 5) {x=Bool}`
- `E True {x=Bool}`
- `E (B C 0) {x=Mumble}`
- `E (B C 0) {x=Bool}`
- `C`

```
-- FILL IN HERE
```

□

Merge 3 following sections into one about Idris implicits? Mention the lowercase/uppercase distinction.

1.1.2. *Type Annotation Inference.*

This has already happened earlier at repeat, delete most of

Let's write the definition of repeat again, but this time we won't specify the types of any of the arguments. Will Idris 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 Idris has assigned to repeat':

Check repeat'. (* ===> forall X : Type, X -> nat -> list X) *Check repeat. (* ===> forall X : Type, X -> nat -> list X *)

It has exactly the same type type as repeat. Idris 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 Z and S means it must be a Nat; and so on.

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

1.1.3. *Type Argument Synthesis.*

We should mention the _ parameters but it won't work like

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_ty. But since the second argument to repeat is an element of x_ty, it seems entirely obvious that the first argument can only be x_ty — why should we have to write it explicitly?

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

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

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

we can also replace the types with _

```
repeat' (X : _) (x : _) (count : _) : list X :=
```

to tell Idris to attempt to infer the missing information.

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

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

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

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

…we can use argument synthesis to write this:

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

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

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

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

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

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

```
repeat' : {x_ty : Type} → (x : x_ty) → (count : Nat) → List x_ty
repeat' x Z = Nil
repeat' x (S count') = x :: repeat' x count'
```

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

We will use the latter style whenever possible, but we will continue to use explicit declarations in data types. 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:

```
data List' : {x : Type} → Type where
  Nil' : List'
  Cons' : x → List' → List'
```

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

Added the implicit inference explanation here

There's another step towards conciseness that we can take in Idris – drop the implicit argument completely in function definitions! Idris will automatically insert them for us when it encounters unknown variables. *Note that by convention this will only happen for variables starting on a lowercase letter.*

```
repeat'' : (x : x_ty) → (count : Nat) → List x_ty
repeat'' x Z = Nil
repeat'' x (S count') = x :: repeat'' x count'
```

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

```
app : (l1, l2 : List x) → List x
app Nil l2 = l2
app (h::t) l2 = h :: app t l2

rev : (l : List x) → List x
rev [] = []
rev (h::t) = app (rev t) (h::Nil)

length : (l : List x) → Nat
length [] = Z
length (_::l') = S (length l')

test_rev1 : rev (1::2::[]) = 2::1::[]
test_rev1 = Refl

test_rev2 : rev (True::[]) = True::[]
test_rev2 = Refl

test_length1 : length (1::2::3::[]) = 3
test_length1 = Refl
```

1.1.5. *Supplying Type Arguments Explicitly.* One small problem with declaring arguments implicit is that, occasionally, Idris does not have enough local information to determine a type argument; in such cases, we need to tell Idris that we want to give the argument explicitly just this time. For example, suppose we write this:

```
λΠ> :let mynil = Nil
(input):Can't infer argument elem to []
```

Here, Idris 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 via `the` function (so that Idris has more information available when it gets to the "application" of `Nil`):

```
λΠ> :let mynil = the (List Nat) Nil
```

Alternatively, we can force the implicit arguments to be explicit by supplying them as arguments in curly braces.

```
λΠ> :let mynil' = Nil {elem=Nat}
```

Describe here how to bring variables from the type into definition scope via implicits?

Explain that Idris has built-in notation for lists instead?

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

Notation "x :: y" := (cons x y) (at level 60, right associativity). Notation "[ ]" := nil. Notation "[x ; .. ; y]" := (cons x .. (cons y []) ..). Notation "x ++ y" := (app x y) (at level 60, right associativity).

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

```
list123''' : List Nat
list123''' = [1, 2, 3]
```

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

```
app_nil_r : (l : List x) -> l ++ [] = l
app_nil_r l = ?app_nil_r_rhs

app_assoc : (l, m, n : List a) -> l ++ m ++ n = (l ++ m) ++ n
app_assoc l m n = ?app_assoc_rhs

app_length : (l1, l2 : List x) -> length (l1 ++ l2) = length l1 + length l2
app_length l1 l2 = ?app_length_rhs
```

□

1.1.7. *Exercise: 2 stars, optional (more_poly_exercises).* Here are some slightly more interesting ones…

```
rev_app_distr : (l1, l2 : List x) -> rev (l1 ++ l2) = rev l2 ++ rev l1
rev_app_distr l1 l2 = ?rev_app_distr_rhs

rev_involutive : (l : List x) -> rev (rev l) = l
rev_involutive l = ?rev_involutive_rhs
```

□

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

```
data Prod : (x, y : Type) -> Type where
  PPair : x -> y -> Prod x y
```

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

T

his sugar cannot be marked as private and messes up things when imported, consider changing the notation}

```
syntax "(" [x] "," [y] ")" = PPair x y
```

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

```
syntax [x_ty] "*" [y_ty] = Prod x_ty y_ty
```

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

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

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

```
fst : (p : x*y) → x
fst (x,y) = x

snd : (p : x*y) → y
snd (x,y) = y
```

Edit

The following function takes two lists and combines them into a list of pairs. In functional languages, it is usually called `zip` (though the Coq's standard library calls it `combine`).

```
zip : (lx : List x) → (ly : List y) → List (x*y)
zip [] _ = []
zip _ [] = []
zip (x::tx) (y::ty) = (x,y) :: zip tx ty
```

1.2.1. *Exercise: 1 star, optional (combine_checks).* Try answering the following questions on paper and checking your answers in Idris:

- What is the type of `zip` (i.e., what does `:t zip` print?)
- What does `combine [1,2] [False,False,True,True]` print?

□

1.2.2. *Exercise: 2 stars, recommended (split).* The function `split` is the right inverse of `zip`: 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.

```
split : (l : List (x*y)) → (List x) * (List y)
split l = ?split_rhs

test_split : split [(1,False),(2,False)] = ([1,2],[False,False])
test_split = ?test_split_rhs
```

□

1.2.3. *Polymorphic Options.* One last polymorphic type for now: *polymorphic options*, which generalize `NatOption` from the previous chapter:

```
data Option : (x : Type) → Type where
 Some : x → Option x
 None : Option x
```

In Idris' standard library this type is called `Maybe`, with constructors `Just x` and `Nothing`.

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

```
nth_error : (l : List x) → (n : Nat) → Option x
nth_error [] n = None
nth_error (a::l') n = if n == 0
                      then Some a
                      else nth_error l' (pred n)

test_nth_error1 : nth_error [4,5,6,7] 0 = Some 4
test_nth_error1 = Refl

test_nth_error2 : nth_error [[1],[2]] 1 = Some [2]
test_nth_error2 = Refl

test_nth_error3 : nth_error [True] 2 = None
test_nth_error3 = Refl
```

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

```
hd_error : (l : List x) → Option x
hd_error l = ?hd_error_rhs

test_hd_error1 : hd_error [1,2] = Some 1
test_hd_error1 = ?test_hd_error1_rhs

test_hd_error2 : hd_error [[1],[2]] = Some [1]
test_hd_error2 = ?test_hd_error2_rhs
```

□

2. Functions as Data

Like many other modern programming languages – including all functional languages (ML, Haskell, Scheme, Scala, Clojure etc.) – Idris treats functions as

first-class citizens, allowing them to be passed as arguments to other functions, returned as results, stored in data structures, etc.

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

```
doit3times : (f: x → x) → (n : x) → x
doit3times f n = 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`.

```
λΠ> :t doit3times
-- doit3times : (x → x) → x → x
```

Explain that the prefixes are needed to avoid the implicit scoping rule, seems that this fires up more often when passing functions as parameters to other functions

```
test_doit3times : doit3times Numbers.minusTwo 9 = 3
test_doit3times = Refl

test_doit3times' : doit3times Bool.not True = False
test_doit3times' = Refl
```

2.2. Filter. Here is a more useful higher-order function, taking a list of `x`s and a *predicate* on `x` (a function from `x` to `Bool`) and "filtering" the list, returning a new list containing just those elements for which the predicate returns `True`.

```
filter : (test : x → Bool) → (l: List x) → List x
filter test [] = []
filter test (h::t) = if test h
                     then h :: (filter test t)
                     else filter test t
```

(This is how it's defined in Idris's stdlib, too.)

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

```
test_filter1 : filter Numbers.evenb [1,2,3,4] = [2,4]
test_filter1 = Refl

length_is_1 : (l : List x) → Bool
length_is_1 l = length l == 1

test_filter2 : filter Poly.length_is_1
                      [ [1,2], [3], [4], [5,6,7], [], [8] ]
                    = [        [3], [4],              [8] ]
test_filter2 = Refl
```

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

```
countoddmembers' : (l: List Nat) -> Nat
countoddmembers' l = length (filter Numbers.oddb l)

test_countoddmembers'1 : countoddmembers' [1,0,3,1,4,5] = 4
test_countoddmembers'1 = Refl

test_countoddmembers'2 : countoddmembers' [0,2,4] = 0
test_countoddmembers'2 = Refl

test_countoddmembers'3 : countoddmembers' Nil = 0
test_countoddmembers'3 = Refl
```

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

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

Can't use * here due to the interference from our tuple sugar

```
test_anon_fun' : doit3times (\n => mult n n) 2 = 256
test_anon_fun' = Refl
```

The expression `\n => mult n n` can be read as "the function that, given a number `n`, yields `n * n`."

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

```
test_filter2' : filter (\l => length l == 1)
                       [ [1,2], [3], [4], [5,6,7], [], [8] ]
                     = [        [3], [4],             [8] ]
test_filter2' = Refl
```

2.3.1. *Exercise: 2 stars (filter_even_gt7).* Use `filter` (instead of function definition) to write an Idris 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`.

```
filter_even_gt7 : (l : List Nat) -> List Nat
filter_even_gt7 l = ?filter_even_gt7_rhs

test_filter_even_gt7_1 : filter_even_gt7 [1,2,6,9,10,3,12,8] = [10,12,8]
test_filter_even_gt7_1 = ?test_filter_even_gt7_1_rhs

test_filter_even_gt7_2 : filter_even_gt7 [5,2,6,19,129] = []
test_filter_even_gt7_2 = ?test_filter_even_gt7_2_rhs
```

□

2.3.2. *Exercise: 3 stars (partition).* Use `filter` to write an Idris function `partition`:

```
partition : (test : x → Bool) → (l : List x) → (List x) * (List x)
partition f xs = ?partition_rhs
```

Given a set `x`, a test function of type `x → Bool` and a `List x`, `partition` should return a pair of lists. The first member of the pair is the sublist of the original list containing the elements that satisfy the test, and the second is the sublist containing those that fail the test. The order of elements in the two sublists should be the same as their order in the original list.

```
test_partition1 : partition Numbers.oddb [1,2,3,4,5] = ([1,3,5], [2,4])
test_partition1 = ?test_partition1_rhs

test_partition2 : partition (\x ⇒ False) [5,9,0] = (([], [5,9,0]))
test_partition2 = ?test_partition2_rhs
```

□

2.4. Map. Another handy higher-order function is called `map`.

```
map : (f : x → y) → (l : List x) → List y
map f [] = []
map f (h::t) = (f h) :: map f t
```

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:

```
test_map1 : map (\x ⇒ plus 3 x) [2,0,2] = [5,3,5]
test_map1 = Refl
```

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:

```
test_map2 : map Numbers.oddb [2,1,2,5] = [False,True,False,True]
test_map2 = Refl
```

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:

```
test_map3 : map (\n ⇒ [evenb n, oddb n]) [2,1,2,5]
                = [[True,False],[False,True],[True,False],[False,True]]
test_map3 = Refl
```

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

```
map_rev : (f : x → y) → (l : List x) → map f (rev l) = rev (map f l)
map_rev f l = ?map_rev_rhs
```

□

2.4.2. *Exercise: 2 stars, recommended (flat_map).* The function `map` maps a `List x` to a `List y` using a function of type `x → y`. We can define a similar function, `flat_map`, which maps a `List x` to a `List y` using a function `f` of type `x → List y`. Your definition should work by 'flattening' the results of `f`, like so:

```
flat_map (\n ⇒ [n,n+1,n+2]) [1,5,10] = [1,2,3, 5,6,7, 10,11,12]

flat_map : (f : x → List y) → (l : List x) → List y
flat_map f l = ?flat_map_rhs

test_flat_map1 : flat_map (\n ⇒ [n,n,n]) [1,5,4] = [1,1,1, 5,5,5, 4,4,4]
test_flat_map1 = ?test_flat_map1_rhs
```

□

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

```
option_map : (f : x → y) → (xo : Option x) → Option y
option_map f None = None
option_map f (Some x) = Some (f x)
```

2.4.3. *Exercise: 2 stars, optional (implicit_args).* The definitions and uses of `filter` and `map` use implicit arguments in many places. Add explicit type parameters where necessary and use Idris 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.)

□

2.5. 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.

```
fold : (f : x → y → y) → (l : List x) → (b : y) → y
fold f [] b = b
fold f (h::t) b = f h (fold f t b)
```

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 (+) [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 (+) [1,2,3,4] 0
```

yields

```
1 + (2 + (3 + (4 + 0)))
```

Some more examples:

We go back to `andb` here because `(&&)`'s second parameter is lazy, with the left fold the return type is inferred to be lazy too, leading to type mismatch.

```
λΠ> :t fold andb
fold andb : List Bool → Bool → Bool
```

```
fold_example1 : fold (*) [1,2,3,4] 1 = 24
fold_example1 = Refl
```

```
fold_example2 : fold Booleans.andb [True,True,False,True] True = False
fold_example2 = Refl
```

```
fold_example3 : fold (++) [[1],[],[2,3],[4]] [] = [1,2,3,4]
fold_example3 = Refl
```

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

```
-- FILL IN HERE
```

□

2.6. 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.

```
constfun : (x : x_ty) → Nat → x_ty
constfun x = \k ⇒ x
```

```
ftrue : Nat → Bool
ftrue = constfun True
```

```
constfun_example1 : ftrue 0 = True
constfun_example1 = Refl
```

```
constfun_example2 : (constfun 5) 99 = 5
constfun_example2 = Refl
```

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`.

```
λΠ> :t plus
Prelude.Nat.plus : Nat → Nat → Nat
```

Each → in this expression is actually a *binary* operator on types. This operator is *right-associative*, so the type of `plus` is really a shorthand for `Nat → (Nat → Nat)` – i.e., it can be read as saying that "`plus` is a one-argument function that takes a `Nat` and returns a one-argument function that takes another `Nat` and returns a `Nat`." In the examples above, we have always applied `plus` to both of its arguments at once, but if we like we can supply just the first. This is called *partial application*.

```
plus3 : Nat → Nat
plus3 = plus 3
```

```
λΠ> :t plus3
```

```
test_plus3 : plus3 4 = 7
test_plus3 = Refl
```

```
test_plus3' : doit3times Poly.plus3 0 = 9
test_plus3' = Refl
```

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

3. Additional Exercises

```
namespace Exercises
```

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

```
  fold_length : (l : List x) → Nat
  fold_length l = fold (\_, n ⇒ S n) l 0
```

```
  test_fold_length1 : fold_length [4,7,0] = 3
  test_fold_length1 = Refl
```

Prove the correctness of `fold_length`.

```
  fold_length_correct : (l : List x) → fold_length l = length l
  fold_length_correct l = ?fold_length_correct_rhs
```

□

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

```
  fold_map : (f : x → y) → (l : List x) → List y
  fold_map f l = ?fold_map_rhs
```

Write down a theorem `fold_map_correct` in Idris stating that `fold_map` is correct, and prove it.

```
  fold_map_correct : ?fold_map_correct
```

□

3.0.3. *Exercise: 2 stars, advanced (currying).* In Idris, a function `f: a → b → c` really has the type `a → (b → c)`. That is, if you give `f` a value of type `a`, it will give you function `f' : b → c`. If you then give `f'` a value of type `b`, it will return a value of type `c`. This allows for partial application, as in `plus3`. Processing a list of arguments with functions that return functions is called *currying*, in honor of the logician Haskell Curry.

Conversely, we can reinterpret the type `a → b → c` as `(a * b) → c`. This is called *uncurrying*. With an uncurried binary function, both arguments must be given at once as a pair; there is no partial application.

We can define currying as follows:

```
prod_curry : (f : (x * y) → z) → (x_val : x) → (y_val : y) → z
prod_curry f x_val y_val = f (x_val, y_val)
```

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

```
prod_uncurry : (f : x → y → z) → (p : x * y) → z
prod_uncurry f p = ?prod_uncurry_rhs
```

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

Not sure what are they shortening here

```
test_map2' : map (\x ⇒ plus 3 x) [2,0,2] = [5,3,5]
test_map2' = Refl
```

Didn't we just write out these types explicitly?

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

```
ΛΠ> :t prod_curry
ΛΠ> :t prod_uncurry
```

```
uncurry_curry : (f : x → y → z) → (x_val : x) → (y_val : y) →
                prod_curry (prod_uncurry f) x_val y_val = f x_val y_val
uncurry_curry f x_val y_val = ?uncurry_curry_rhs

curry_uncurry : (f : (x * y) → z) → (p : x * y) →
                prod_uncurry (prod_curry f) p = f p
curry_uncurry f p = ?curry_uncurry_rhs
```

□

3.0.4. *Exercise: 2 stars, advanced (nth_error_informal).* Recall the definition of the `nth_error` function:

```
nth_error : (l : List x) → (n : Nat) → Option x
nth_error [] n = None
nth_error (a::l') n = if n == 0
                      then Some a
                      else nth_error l' (pred n)
```

Write an informal proof of the following theorem:

```
n → l → length l = n → nth_error l n = None
```

```
-- FILL IN HERE
```

□

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

```
namespace Church

  Nat' : {x : Type} → Type
  Nat' {x} = (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:

```
  one : Nat'
  one f x = f x
```

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

```
  two : Nat'
  two f 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.

```
  zero : Nat'
  zero f x = x
```

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

```
  three : Nat'
  three = doit3times
```

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

Successor of a natural number:

```
  succ' : (n : Nat' {x}) → Nat' {x}
  succ' n = ?succ__rhs
```

Even if you add `f x` on both sides of `=`, these "unit tests" don't seem to work neither with `Refl` nor with more advanced techniques currently

```
  succ'_1 : succ' zero = one
  succ'_1 = ?succ__1_rhs

  succ'_2 : succ' one = two
  succ'_2 = ?succ__2_rhs

  succ'_3 : succ' two = three
  succ'_3 = ?succ__3_rhs
```

Addition of two natural numbers:

```
plus' : (n, m : Nat' {x}) → Nat' {x}
plus' n m = ?plus__rhs

plus'_1 : plus' zero one = one
plus'_1 = ?plus__1_rhs

plus'_2 : plus' two three = plus' three two
plus'_2 = ?plus__2_rhs

plus'_3 : plus' (plus' two two) three = plus' one (plus' three three)
plus'_3 = ?plus__3_rhs
```

Multiplication:

```
mult' : (n, m : Nat' {x}) → Nat' {x}
mult' n m = ?mult__rhs

mult'_1 : mult' one one = one
mult'_1 = ?mult__1_rhs

mult'_2 : mult' zero (plus' three three) = zero
mult'_2 = ?mult__2_rhs

mult'_3 : mult' two three = plus' three three
mult'_3 = ?mult__3_rhs
```

Exponentiation:

Edit the hint. Can't make it work with `exp' : (n, m : Nat' {x}) → Nat' {x}`.

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

```
exp' : (n : Nat' {x}) → (m : Nat' {x=x→x}) → Nat' {x}
exp' n m = ?exp__rhs
```

This won't typecheck under this signature of `exp` because of 2 instances of `two`

```
-- exp'_1 : exp' two two = plus' two two
-- exp'_1 = ?exp__1_rhs

exp'_2 : exp' three two = plus' (mult' two (mult' two two)) one
exp'_2 = ?exp__2_rhs

exp'_3 : exp' three zero = one
exp'_3 = ?exp__3_rhs
```

□

CHAPTER 6

Logic : Logic in Idris

```
module Logic

import Basics
import Induction
import Tactics

%hide Basics.Numbers.pred
%hide Basics.Playground2.plus

%access public export
%default total
```

In previous chapters, we have seen many examples of factual claims (*propositions*) and ways of presenting evidence of their truth (*proofs*). In particular, we have worked extensively with equality propositions of the form `e1 = e2`, with implications (`p → q`), and with quantified propositions (`x → P(x)`). In this chapter, we will see how Idris can be used to carry out other familiar forms of logical reasoning.

Before diving into details, let's talk a bit about the status of mathematical statements in Idris. Recall that Idris is a *typed* language, which means that every sensible expression in its world has an associated type. Logical claims are no exception: any statement we might try to prove in Idris has a type, namely `Type`, the type of *propositions*. We can see this with the `:t` command:

```
λΠ> :t 3 = 3
3 = 3 : Type

λΠ> :t (n, m : Nat) → n + m = m + n
(n : Nat) → (m : Nat) → n + m = m + n : Type
```

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

Simply being a proposition is one thing; being provable is something else!

```
λΠ> :t (n : Nat) → n = 2
(n : Nat) → n = 2 : Type

λΠ> :t 3 = 4
3 = 4 : Type
```

Indeed, propositions don't just have types: they are *first-class objects* that can be manipulated in the same ways as the other entities in Idris's world. So far, we've seen one primary place that propositions can appear: in functions' type signatures.

```
plus_2_2_is_4 : 2 + 2 = 4
plus_2_2_is_4 = Refl
```

But propositions can be used in many other ways. For example, we can give a name to a proposition as a value on its own, just as we have given names to expressions of other sorts (you'll soon see why we start the name with a capital letter).

```
Plus_fact : Type
Plus_fact = 2+2=4
```

```
λΠ> :t Plus_fact
Plus_fact : Type
```

We can later use this name in any situation where a proposition is expected – for example, in a function declaration.

```
plus_fact_is_true : Plus_fact
plus_fact_is_true = Refl
```

(Here's the reason - recall that names starting with lowercase letters are considered implicits in Idris, so `plus_fact` would be considered a free variable!)

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:

```
is_three : Nat → Type
is_three n = n=3
```

```
λΠ> :t is_three
is_three : Nat → Type
```

In Idris, 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*.

```
Injective : (f : a → b) → Type
Injective {a} {b} f = (x, y : a) → f x = f y → x = y
```

```
succ_inj : Injective S
succ_inj x x Refl = Refl
```

The equality operator `=` is also a function that returns a `Type`.

The expression `n = m` is syntactic sugar for `(=) n m`, defined internally in Idris. Because `=` can be used with elements of any type, it is also polymorphic:

```
λΠ> :t (=)
(=) : A → B → Type
```

1. Logical Connectives

1.1. Conjunction. The *conjunction* (or *logical and*) of propositions `a` and `b` in Idris is the same as the pair of `a` and `b`, written `(a, b)`, representing the claim that both `a` and `b` are true.

```
and_example : (3 + 4 = 7, 2 * 2 = 4)
```

To prove a conjunction, we can use value-level pair syntax:

```
and_example = (Refl, Refl)
```

For any propositions `a` and `b`, if we assume that `a` is true and we assume that `b` is true, we can trivially conclude that `(a,b)` is also true.

```
and_intro : a → b → (a, b)
and_intro = MkPair
```

1.1.1. *Exercise: 2 stars (and_exercise).*

```
and_exercise : (n, m : Nat) → n + m = 0 → (n = 0, m = 0)
and_exercise n m prf = ?and_exercise_rhs
```

□

So much for proving conjunctive statements. To go in the other direction – i.e., to *use* a conjunctive hypothesis to help prove something else – we employ pattern matching.

If the proof context contains a hypothesis `h` of the form `(a,b)`, case splitting will replace it with a pair pattern `(a,b)`.

```
and_example2 : (n, m : Nat) → (n = 0, m = 0) → n + m = 0
and_example2 Z Z (Refl,Refl) = Refl
and_example2 (S _) _ (Refl,_) impossible
and_example2 _ (S _) (_,Refl) impossible
```

You may wonder why we bothered packing the two hypotheses `n = 0` and `m = 0` into a single conjunction, since we could have also stated the theorem with two separate premises:

```
and_example2' : (n, m : Nat) → n = 0 → m = 0 → n + m = 0
and_example2' Z Z Refl Refl = Refl
and_example2' (S _) _ Refl _ impossible
and_example2' _ (S _) _ Refl impossible
```

For this 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 bigger developments. Here's a simple example:

```
and_example3 : (n, m : Nat) -> n + m = 0 -> n * m = 0
and_example3 n m prf =
 let (nz, _) = and_exercise n m prf in
 rewrite nz in Refl
```

Remove lemma and exercise, use `fst` and `snd` directly?

Another common situation with conjunctions is that we know `(a,b)` but in some context we need just `a` (or just `b`). The following lemmas are useful in such cases:

```
proj1 : (p, q) -> p
proj1 = fst
```

1.1.2. *Exercise: 1 star, optional (proj2).*

```
proj2 : (p, q) -> q
proj2 x = ?proj2_rhs
```

□

Finally, we sometimes need to rearrange the order of conjunctions and/or the grouping of multi-way conjunctions. The following commutativity and associativity theorems are handy in such cases.

```
and_commut : (p, q) -> (q, p)
and_commut (p, q) = (q, p)
```

1.1.3. *Exercise: 2 stars (and_assoc).* Remove or demote to 1 star?

```
and_assoc : (p, (q, r)) -> ((p, q), r)
and_assoc x = ?and_assoc_rhs
```

□

1.2. Disjunction.

Hide *Basics.Booleans* analogues and make syntax synonyms (/\) and (\/) f and *Either*?

Another important connective is the *disjunction*, or *logical or* of two propositions: `a `Either` b` is true when either `a` or `b` is. The first case has be tagged with *Left*, and the second with *Right*.

To use a disjunctive hypothesis in a proof, we proceed by case analysis, which, as for *Nat* or other data types, can be done with pattern matching. Here is an example:

```
or_example : (n, m : Nat) -> ((n = 0) `Either` (m = 0)) -> n * m = 0
or_example Z _ (Left Refl) = Refl
or_example (S _) _ (Left Refl) impossible
or_example n Z (Right Refl) = multZeroRightZero n
or_example _ (S _) (Right Refl) impossible
```

Conversely, to show that a disjunction holds, we need to show that one of its sides does. This can be done via aforementioned *Left* and *Right* constructors. Here is a trivial use…

```
or_intro : a → a `Either` b
or_intro = Left
```

… and a slightly more interesting example requiring both *Left* and *Right*:

```
zero_or_succ : (n : Nat) → ((n = 0) `Either` (n = S (pred n)))
zero_or_succ Z = Left Refl
zero_or_succ (S _) = Right Refl
```

1.2.1. *Exercise: 1 star (mult_eq_0).*

```
mult_eq_0 : n * m = 0 → ((n = 0) `Either` (m = 0))
mult_eq_0 prf = ?mult_eq_0_rhs
```

□

1.2.2. *Exercise: 1 star (or_commut).*

```
or_commut : (p `Either` q) → (q `Either` p)
or_commut x = ?or_commut_rhs
```

□

1.3. Falsehood and Negation. So far, we have mostly been concerned with proving that certain things are *true* – addition is commutative, appending lists is associative, etc. Of course, we may also be interested in *negative* results, showing that certain propositions are *not* true. In Idris, such negative statements are expressed with the negation typelevel function *Not*.

Add hyperlink

To see how negation works, recall the discussion of the *principle of explosion* from the previous chapter; it asserts that, if we assume a contradiction, then any other proposition can be derived. Following this intuition, we could define *Not* p as q → (p → q). Idris actually makes a slightly different choice, defining *Not* p as p → *Void*, where *Void* is a *particular* contradictory proposition defined in the standard library as a data type with no constructors.

```
data Void : Type where

Not : Type → Type
Not a = a → Void
```

Discuss difference between void and absurd

Since *Void* is a contradictory proposition, the principle of explosion also applies to it. If we get *Void* into the proof context, we can call void or absurd on it to complete any goal:

```
ex_falso_quodlibet : Void → p
ex_falso_quodlibet = void
```

The Latin *ex falso quodlibet* means, literally, "from falsehood follows whatever you like"; this is another common name for the principle of explosion.

1.3.1. *Exercise: 2 stars, optional (not_implies_our_not).* Show that Idris's definition of negation implies the intuitive one mentioned above:

```
not_implies_our_not : Not p → (q → (p → q))
not_implies_our_not notp q p = ?not_implies_our_not_rhs
```

□

This is how we use `Not` to state that `0` and `1` are different elements of `Nat`:

Explain `Refl`-lambda syntax and `Uninhabited`, keep in mind https://github.com/idris-lang/Idris-dev/issues/3943

```
zero_not_one : Not (Z = S _)
zero_not_one = \Refl impossible
```

We could also rely on the `Uninhabited` instance in stdlib and write this as

```
zero_not_one = uninhabited
```

It takes a little practice to get used to working with negation in Idris. Even though you can see perfectly well why a statement involving negation is true, it can be a little tricky at first to get things into the right configuration so that Idris can understand it! Here are proofs of a few familiar facts to get you warmed up.

```
not_False : Not Void
not_False = absurd

contradiction_implies_anything : (p, Not p) → q
contradiction_implies_anything (p, notp) = absurd $ notp p

double_neg : p → Not $ Not p
double_neg p notp = notp p
```

1.3.2. *Exercise: 2 stars, advanced, recommended (double_neg_inf).* Write an informal proof of `double_neg`:

Theorem: `p` implies `Not $ Not p`, for any proposition `p`.

```
-- FILL IN HERE
```

□

1.3.3. *Exercise: 2 stars, recommended (contrapositive).*

```
contrapositive : (p → q) → (Not q → Not p)
contrapositive pq = ?contrapositive_rhs
```

□

1.3.4. *Exercise: 1 star (not_both_true_and_false).*

```
not_both_true_and_false : Not (p, Not p)
not_both_true_and_false = ?not_both_true_and_false_rhs
```

□

1.3.5. *Exercise: 1 star, advanced (informal_not_PNP).* Write an informal proof (in English) of the proposition `Not (p, Not p)`.

```
-- FILL IN HERE
```

□

Similarly, since inequality involves a negation, it requires a little practice to be able to work with it fluently. Here is one useful trick. If you are trying to prove a goal that is nonsensical (e.g., the goal state is `False = True`), apply `absurd` to change the goal to `Void`. This makes it easier to use assumptions of the form `Not p` that may be available in the context – in particular, assumptions of the form `Not (x=y)`.

```
not_true_is_false : (b : Bool) -> Not (b = True) -> b = False
not_true_is_false False h = Refl
not_true_is_false True h = absurd $ h Refl
```

1.4. Truth. Besides `Void`, Idris's standard library also defines `Unit`, a proposition that is trivially true. To prove it, we use the predefined constant `()`:

```
True_is_true : Unit
True_is_true = ()
```

Unlike `Void`, which is used extensively, `Unit` is used quite rarely in proofs, since it is trivial (and therefore uninteresting) to prove as a goal, and it carries no useful information as a hypothesis. But it can be quite useful when defining complex proofs using conditionals or as a parameter to higher-order proofs. We will see examples of such uses of `Unit` later on.

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

```
namespace MyIff

  iff : {p,q : Type} -> Type
  iff {p} {q} = (p -> q, q -> p)
```

Idris's stdlib has a more general form of this, `Iso`, in `Control.Isomorphism`.

```
  syntax [p] "<->" [q] = iff {p} {q}

  iff_sym : (p <-> q) -> (q <-> p)
  iff_sym (pq, qp) = (qp, pq)
```

```
not_true_iff_false : (Not (b = True)) <-> (b = False)
not_true_iff_false {b} = (not_true_is_false b, not_true_and_false b)
where
  not_true_and_false : (b : Bool) -> (b = False) -> Not (b = True)
  not_true_and_false False _ Refl impossible
  not_true_and_false True Refl _ impossible
```

1.5.1. *Exercise: 1 star, optional (iff_properties).* Using the above proof that <-> is symmetric (iff_sym) as a guide, prove that it is also reflexive and transitive.

```
iff_refl : p <-> p
iff_refl = ?iff_refl_rhs

iff_trans : (p <-> q) -> (q <-> r) -> (p <-> r)
iff_trans piq qir = ?iff_trans_rhs
```

□

1.5.2. *Exercise: 3 stars (or_distributes_over_and).*

```
or_distributes_over_and : (p `Either` (q,r)) <-> (p `Either` q, p `Either` r)
or_distributes_over_and = ?or_distributes_over_and_rhs
```

□

Edit the rest of the section. What to do with Setoids? We could probably just use profunctors here

Some of Idris's tactics treat iff statements specially, avoiding the need for some low-level proof-state manipulation. In particular, rewrite and reflexivity can be used with iff statements, not just equalities. To enable this behavior, we need to import a special Idris library that allows rewriting with other formulas besides equality (setoids).

Here is a simple example demonstrating how these tactics work with iff. First, let's prove a couple of basic iff equivalences…

```
mult_0 : (n * m = Z) <-> ((n = Z) `Either` (m = Z))
mult_0 {n} {m} = (to n m, or_example n m)
where
  to : (n, m : Nat) -> (n * m = Z) -> (n = 0) `Either` (m = 0)
  to Z _ Refl = Left Refl
  to (S _) Z _ = Right Refl
  to (S _) (S _) Refl impossible

or_assoc : (p `Either` (q `Either` r)) <-> ((p `Either` q) `Either` r)
or_assoc = (to, fro)
where
  to : Either p (Either q r) -> Either (Either p q) r
  to (Left p) = Left $ Left p
  to (Right (Left q)) = Left $ Right q
  to (Right (Right r)) = Right r
```

```
fro : Either (Either p q) r -> Either p (Either q r)
fro (Left (Left p)) = Left p
fro (Left (Right q)) = Right $ Left q
fro (Right r) = Right $ Right r
```

We can now use these facts with `rewrite` and *Refl* to give smooth proofs of statements involving equivalences. Here is a ternary version of the previous `mult_0` result:

```
mult_0_3 : (n * m * p = Z) <->
           ((n = Z) `Either` ((m = Z) `Either` (p = Z)))
mult_0_3 = (to, fro)
where
  to : (n * m * p = Z) -> ((n = Z) `Either` ((m = Z) `Either` (p = Z)))
  to {n} {m} {p} prf = let
    (nm_p_to, _) = mult_0 {n=(n*m)} {m=p}
    (n_m_to, _) = mult_0 {n} {m}
    (_, or_a_fro) = or_assoc {p=(n=Z)} {q=(m=Z)} {r=(p=Z)}
    in or_a_fro $ case nm_p_to prf of
         Left prf => Left $ n_m_to prf
         Right prf => Right prf
  fro : ((n = Z) `Either` ((m = Z) `Either` (p = Z))) -> (n * m * p = Z)
  fro (Left Refl) = Refl
  fro {n} (Right (Left Refl)) = rewrite multZeroRightZero n in Refl
  fro {n} {m} (Right (Right Refl)) = rewrite multZeroRightZero (n*m) in Refl
```

The apply tactic can also be used with <->. When given an equivalence as its argument, apply tries to guess which side of the equivalence to use.

```
apply_iff_example : (n, m : Nat) -> n * m = Z -> ((n = Z) `Either` (m = Z))
apply_iff_example n m = fst $ mult_0 {n} {m}
```

1.6. Existential Quantification. Another important logical connective is existential quantification. To say that there is some `x` of type `t` such that some property `p` holds of `x`, we write `(x : t ** p)`. The type annotation `: t` can be omitted if Idris is able to infer from the context what the type of `x` should be.

To prove a statement of the form `(x ** p)`, we must show that `p` holds for some specific choice of value for `x`, known as the *witness* of the existential. This is done in two steps: First, we explicitly tell Idris which witness `t` we have in mind by writing it on the left side of `**`. Then we prove that `p` holds after all occurrences of `x` are replaced by `t`.

```
four_is_even : (n : Nat ** 4 = n + n)
four_is_even = (2 ** Refl)
```

Conversely, if we have an existential hypothesis `(x ** p)` in the context, we can pattern match on it to obtain a witness `x` and a hypothesis stating that `p` holds of `x`.

```
exists_example_2 : (m : Nat ** n = 4 + m) -> (o : Nat ** n = 2 + o)
exists_example_2 (m ** pf) = (2 + m ** pf)
```

1.6.1. *Exercise: 1 star (dist_not_exists).* Prove that "p holds for all x" implies "there is no x for which p does not hold."

```
dist_not_exists : {p : a → Type} → ((x : a) → p x) → Not (x ** Not $ p x)
dist_not_exists f = ?dist_not_exists_rhs
```

□

1.6.2. *Exercise: 2 stars (dist_exists_or).* Prove that existential quantification distributes over disjunction.

```
dist_exists_or : {p, q : a → Type} → (x ** (p x `Either` q x)) ↔
                                     ((x ** p x) `Either` (x ** q x))
dist_exists_or = ?dist_exists_or_rhs
```

□

2. Programming with Propositions

The logical connectives that we have seen provide a rich vocabulary for defining complex propositions from simpler ones. To illustrate, let's look at how to express the claim that an element x occurs in a list l. Notice that this property has a simple recursive structure:

- If l is the empty list, then x cannot occur on it, so the property "x appears in l" is simply false.
- Otherwise, l has the form x' :: xs. In this case, x occurs in l if either it is equal to x' or it occurs in xs.

We can translate this directly into a straightforward recursive function from taking an element and a list and returning a proposition:

```
In : (x : a) → (l : List a) → Type
In x [] = Void
In x (x' :: xs) = (x' = x) `Either` In x xs
```

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

```
In_example_1 : In 4 [1, 2, 3, 4, 5]
In_example_1 = Right $ Right $ Right $ Left Refl
```

```
In_example_2 : In n [2, 4] → (n' : Nat ** n = 2 * n')
In_example_2 (Left Refl) = (1 ** Refl)
In_example_2 (Right $ Left Refl) = (2 ** Refl)
In_example_2 (Right $ Right prf) = absurd prf
```

(Notice the use of absurd to discharge the last case.)

We can also prove more generic, higher-level lemmas about *In*.

Note, in the next, how *In* starts out applied to a variable and only gets expanded when we do case analysis on this variable:

```
In_map : (f : a → b) → (l : List a) → (x : a) → In x l →
         In (f x) (map f l)
In_map _ [] _ ixl = absurd ixl
In_map f (x' :: xs) x (Left prf) = rewrite prf in Left Refl
In_map f (x' :: xs) x (Right r) = Right $ In_map f xs x r
```

This way of defining propositions recursively, though convenient in some cases, also has some drawbacks. In particular, it is subject to Idris's usual restrictions regarding the definition of recursive functions, e.g., the requirement that they be "obviously terminating." In the next chapter, we will see how to define propositions *inductively*, a different technique with its own set of strengths and limitations.

2.0.1. *Exercise: 2 stars (In_map_iff).*

```
In_map_iff : (f : a → b) → (l : List a) → (y : b) →
             (In y (map f l)) <-> (x ** (f x = y, In x l))
In_map_iff f l y = ?In_map_iff_rhs
```

□

2.0.2. *Exercise: 2 stars (in_app_iff).*

```
in_app_iff : (In a (l++l')) <-> (In a l `Either` In a l')
in_app_iff = ?in_app_iff_rhs
```

□

2.0.3. *Exercise: 3 stars (All).* Recall that functions returning propositions can be seen as *properties* of their arguments. For instance, if `p` has type `Nat → Type`, then `p n` states 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`.)

```
All : (p : t → Type) → (l : List t) → Type
All p l = ?All_rhs

All_In : ((x:t) → In x l → p x) <-> (All p l)
All_In = ?All_In_rhs
```

□

2.0.4. *Exercise: 3 stars (combine_odd_even).* Complete the definition of the `combine_odd_even` function below. It takes as arguments two properties of numbers, `podd` and `peven`, and it should return a property `p` such that `p n` is equivalent to `podd n` when `n` is odd and equivalent to `peven n` otherwise.

```
combine_odd_even : (podd, peven : Nat → Type) → (Nat → Type)
combine_odd_even podd peven = ?combine_odd_even_rhs
```

To test your definition, prove the following facts:

```
combine_odd_even_intro : (n : Nat) →
                         (oddb n = True → podd n) →
                         (oddb n = False → peven n) →
                         combine_odd_even podd peven n
combine_odd_even_intro n oddp evenp = ?combine_odd_even_intro_rhs

combine_odd_even_elim_odd : (n : Nat) →
                            combine_odd_even podd peven n →
                            oddb n = True →
                            podd n
combine_odd_even_elim_odd n x prf = ?combine_odd_even_elim_odd_rhs

combine_odd_even_elim_even : (n : Nat) →
                             combine_odd_even podd peven n →
                             oddb n = False →
                             peven n
combine_odd_even_elim_even n x prf = ?combine_odd_even_elim_even_rhs
```

□

3. Applying Theorems to Arguments

One feature of Idris that distinguishes it from many other proof assistants is that it treats *proofs* as first-class objects.

'nameref' the chapters when they're done

There is a great deal to be said about this, but it is not necessary to understand it in detail in order to use Idris. This section gives just a taste, while a deeper exploration can be found in the optional chapters `ProofObjects` and `IndPrinciples`.

We have seen that we can use the `:t` command to ask Idris to print the type of an expression. We can also use `:t` to ask what theorem a particular identifier refers to.

```
λΠ> :t plusCommutative
plusCommutative : (left : Nat) → (right : Nat) → left + right = right + left
```

Idris prints the *statement* of the `plusCommutative` theorem in the same way that it prints the *type* of any term that we ask it to check. Why?

The reason is that the identifier `plusCommutative` actually refers to a *proof object* – a data structure that represents a logical derivation establishing of the truth of the statement `(n, m : Nat) → n + m = m + n`. The type of this object *is* the statement of the theorem that it is a proof of.

Intuitively, this makes sense because the statement of a theorem tells us what we can use that theorem for, just as the type of a computational object tells us what we can do with that object – e.g., if we have a term of type `Nat → Nat → Nat`, we can give it two `Nat`s as arguments and get a `Nat` back. Similarly, if we have an

object of type `n = m → n + n = m + m` and we provide it an "argument" of type `n = m`, we can derive `n + n = m + m`.

Operationally, this analogy goes even further: by applying a theorem, as if it were a function, to hypotheses with matching types, we can specialize its result without having to resort to intermediate assertions. For example, suppose we wanted to prove the following result:

```
plus_comm3 : (n, m, p : Nat) → n + (m + p) = (p + m) + n
```

Edit, we have already done this in previous chapters (add a hyperlink?)

It appears at first sight that we ought to be able to prove this by rewriting with `plusCommutative` twice to make the two sides match. The problem, however, is that the second `rewrite` will undo the effect of the first.

```
Proof.
  intros n m p.
  rewrite plus_comm.
  rewrite plus_comm.
  (* We are back where we started... *)
Abort.
```

One simple way of fixing this problem, using only tools that we already know, is to use assert to derive a specialized version of plus_comm that can be used to rewrite exactly where we want.

```
Lemma plus_comm3_take2 :
  ∀n m p, n + (m + p) = (p + m) + n.
Proof.
  intros n m p.
  rewrite plus_comm.
  assert (H : m + p = p + m).
  { rewrite plus_comm. reflexivity. }
  rewrite H.
  reflexivity.
Qed.
```

A more elegant alternative is to apply `plusCommutative` 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.

```
plus_comm3 n m p = rewrite plusCommutative n (m+p) in
                   rewrite plusCommutative m p in Refl
```

You can "use theorems as functions" in this way with almost all tactics that take a theorem 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.

```
lemma_application_ex : (n : Nat) -> (ns : List Nat) ->
                       In n (map (\m => m * 0) ns) -> n = 0
lemma_application_ex _ [] prf = absurd prf
lemma_application_ex _ (y :: _) (Left prf) =
  rewrite sym $ multZeroRightZero y in sym prf
lemma_application_ex n (_ :: xs) (Right prf) =
  lemma_application_ex n xs prf
```

We will see many more examples of the idioms from this section in later chapters.

4. Idris vs. Set Theory

Edit, Idris's core is likely some variant of MLTT

Coq's logical core, the Calculus of Inductive Constructions, differs in some important ways from other formal systems that are used by mathematicians for writing down precise and rigorous proofs. For example, in the most popular foundation for mainstream paper-and-pencil mathematics, Zermelo-Fraenkel Set Theory (ZFC), a mathematical object can potentially be a member of many different sets; a term in Idris's logic, on the other hand, is a member of at most one type. This difference often leads to slightly different ways of capturing informal mathematical concepts, but these are, by and large, quite Natural and easy to work with. For example, instead of saying that a natural number `n` belongs to the set of even numbers, we would say in Idris that `ev n` holds, where `ev : Nat -> Type` is a property describing even numbers.

However, there are some cases where translating standard mathematical reasoning into Idris can be either cumbersome or sometimes even impossible, unless we enrich the core logic with additional axioms. We conclude this chapter with a brief discussion of some of the most significant differences between the two worlds.

4.1. Functional Extensionality. The equality assertions that we have seen so far mostly have concerned elements of inductive types (`Nat`, `Bool`, etc.). But since Idris's equality operator is polymorphic, these are not the only possibilities – in particular, we can write propositions claiming that two *functions* are equal to each other:

```
function_equality_ex1 : plus 3 = plus (pred 4)
function_equality_ex1 = Refl
```

In common mathematical practice, two functions `f` and `g` are considered equal if they produce the same outputs:

$$(\forall x, f(x) = g(x)) \to f = g$$

This is known as the principle of *functional extensionality*.

Informally speaking, an "extensional property" is one that pertains to an object's observable behavior. Thus, functional extensionality simply means that a function's identity is completely determined by what we can observe from it – i.e., in Idris terms, the results we obtain after applying it.

Functional extensionality is not part of Idris's basic axioms. This means that some "reasonable" propositions are not provable.

```
function_equality_ex2 : (\x ⇒ plus x 1) = (\x ⇒ plus 1 x)
function_equality_ex2 = ?stuck
```

Explain `believe_me` vs `really_believe_me`?

However, we can add functional extensionality to Idris's core logic using the `really_believe_me` command.

```
functional_extensionality : ((x : a) → f x = g x) → f = g
functional_extensionality = really_believe_me
```

Using `really_believe_me` has the same effect as stating a theorem and skipping its proof using a hole, but it alerts the reader (and type checker) that this isn't just something we're going to come back and fill in later!

We can now invoke functional extensionality in proofs:

```
function_equality_ex2 : (\x ⇒ plus x 1) = (\x ⇒ plus 1 x)
function_equality_ex2 = functional_extensionality $ \x ⇒ plusCommutative x 1
```

Naturally, we must be careful when adding new axioms into Idris's logic, as they may render it *inconsistent* – that is, they may make it possible to prove every proposition, including *Void*!

Unfortunately, there is no simple way of telling whether an axiom is safe to add: hard work is generally required to establish the consistency of any particular combination of axioms.

However, it is known that adding functional extensionality, in particular, *is* consistent.

Is there such a command in Idris?

To check whether a particular proof relies on any additional axioms, use the `Print Assumptions` command.

```
Print Assumptions function_equality_ex2.
(* ==>
    Axioms:
    functional_extensionality :
        forall (X Y : Type) (f g : X → Y),
               (forall x : X, f x = g x) → f = g *)
```

4.1.1. *Exercise: 4 stars (tr_rev).* One problem with the definition of the list-reversing function `rev` that we have is that it performs a call to `++` on each step; running `++` takes time asymptotically linear in the size of the list, which means that `rev` has quadratic running time.

We can improve this with the following definition:

```
rev_append : (l1, l2 : List x) -> List x
rev_append [] l2 = l2
rev_append (x :: xs) l2 = rev_append xs (x :: l2)

tr_rev : (l : List x) -> List x
tr_rev l = rev_append l []
```

(This is very similar to how `reverse` is defined in `Prelude.List`.)

This version is said to be *tail-recursive*, because the recursive call to the function is the last operation that needs to be performed (i.e., we don't have to execute `++` after the recursive call); a decent compiler will generate very efficient code in this case. Prove that the two definitions are indeed equivalent.

```
tr_rev_correct : (x : List a) -> tr_rev x = rev x
tr_rev_correct = ?tr_rev_correct_rhs
```

□

4.2. Propositions and Booleans. We've seen two different ways of encoding logical facts in Idris: with *booleans* (of type `Bool`), and with *propositions* (of type `Type`).

For instance, to claim that a number `n` is even, we can say either

- (1) that `evenb n` returns `True`, or
- (2) that there exists some `k` such that `n = double k`. Indeed, these two notions of evenness are equivalent, as can easily be shown with a couple of auxiliary lemmas.

We often say that the boolean `evenb n` *reflects* the proposition `(k ** n = double k)`.

```
evenb_double : evenb (double k) = True
evenb_double {k = Z} = Refl
evenb_double {k = (S k')} = evenb_double {k=k'}
```

4.2.1. *Exercise: 3 stars (evenb_double_conv).*

```
evenb_double_conv : (k ** n = if evenb n then double k else S (double k))
```

Hint: Use the `evenb_S` lemma from `Induction`.

```
evenb_double_conv = ?evenb_double_conv_rhs
```

□

```
even_bool_prop : (evenb n = True) <-> (k ** n = double k)
even_bool_prop = (to, fro)
where
  to : evenb n = True -> (k ** n = double k)
  to {n} prf =
    let (k ** p) = evenb_double_conv {n}
    in (k ** rewrite p in rewrite prf in Refl)
  fro : (k ** n = double k) -> evenb n = True
  fro {n} (k**prf) = rewrite prf in evenb_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`. These two notions are equivalent.

```
beq_nat_true_iff : (n1, n2 : Nat) -> (n1 == n2 = True) <-> (n1 = n2)
beq_nat_true_iff n1 n2 = (to, fro n1 n2)
where
  to : (n1 == n2 = True) -> (n1 = n2)
  to = beq_nat_true {n=n1} {m=n2}
  fro : (n1, n2 : Nat) -> (n1 = n2) -> (n1 == n2 = True)
  fro n1 n1 Refl = sym $ beq_nat_refl n1
```

However, while the boolean and propositional formulations of a claim are equivalent from a purely logical perspective, they need not be equivalent *operationally*. Equality provides an extreme example: knowing that `n == m = True` is generally of little direct help in the middle of a proof involving `n` and `m`; however, if we convert the statement to the equivalent form `n = m`, we can rewrite with it.

The case of even numbers is also interesting. Recall that, when proving the backwards direction of `even_bool_prop` (i.e., `evenb_double`, going from the propositional to the boolean claim), we used a simple induction on `k`. On the other hand, the converse (the `evenb_double_conv` exercise) required a clever generalization, since we can't directly prove `(k ** n = double k) -> evenb n = True`.

For these examples, the propositional claims are more useful than their boolean counterparts, but this is not always the case. For instance, we cannot test whether a general proposition is true or not in a function definition; as a consequence, the following code fragment is rejected:

```
is_even_prime : Nat -> Bool
is_even_prime n = if n = 2 then True else False
```

Idris complains that `n = 2` has type `Type`, while it expects an element of `Bool` (or some other inductive type with two elements). The reason for this error message has to do with the *computational* nature of Idris's core language, which is designed so that every function that it can express is computable and total. One reason for this is to allow the extraction of executable programs from Idris developments. As a consequence, `Type` in Idris does *not* have a universal case analysis operation telling whether any given proposition is true or false, since such an operation would allow us to write non-computable functions.

Although general non-computable properties cannot be phrased as boolean computations, it is worth noting that even many *computable* properties are easier to express using `Type` than `Bool`, since recursive function definitions are subject to significant restrictions in Idris. For instance, the next chapter shows how to define the property that a regular expression matches a given string using `Type`. Doing the same with `Bool` would amount to writing a regular expression matcher, which would be more complicated, harder to understand, and harder to reason about.

Conversely, an important side benefit of stating facts using booleans is enabling some proof automation through computation with Idris terms, a technique known as *proof by reflection*. Consider the following statement:

```
even_1000 : (k ** 1000 = double k)
```

The most direct proof of this fact is to give the value of `k` explicitly.

```
even_1000 = (500 ** Refl)
```

On the other hand, the proof of the corresponding boolean statement is even simpler:

```
even_1000' : evenb 1000 = True
even_1000' = Refl
```

What is interesting is that, since the two notions are equivalent, we can use the boolean formulation to prove the other one without mentioning the value 500 explicitly:

```
even_1000'' : (k ** 1000 = double k)
even_1000'' = (fst $ even_bool_prop {n=1000}) Refl
```

Add http://www.ams.org/journals/notices/200811/tx081101382p.pdf as a link

Although we haven't gained much in terms of proof size in this case, larger proofs can often be made considerably simpler by the use of reflection. As an extreme example, the Coq proof of the famous *4-color theorem* uses reflection to reduce the analysis of hundreds of different cases to a boolean computation. We won't cover reflection in great detail, but it serves as a good example showing the complementary strengths of booleans and general propositions.

4.2.2. *Exercise: 2 stars (logical_connectives).* The following lemmas relate the propositional connectives studied in this chapter to the corresponding boolean operations.

```
andb_true_iff : (b1, b2 : Bool) -> (b1 && b2 = True) <->
                                   (b1 = True, b2 = True)
andb_true_iff b1 b2 = ?andb_true_iff_rhs

orb_true_iff : (b1, b2 : Bool) -> (b1 || b2 = True) <->
                                  ((b1 = True) `Either` (b2 = True))
orb_true_iff b1 b2 = ?orb_true_iff_rhs
```

□

4.2.3. *Exercise: 1 star (beq_nat_false_iff).* The following theorem is an alternate "negative" formulation of `beq_nat_true_iff` that is more convenient in certain situations (we'll see examples in later chapters).

```
beq_nat_false_iff : (x, y : Nat) -> (x == y = False) <-> (Not (x = y))
beq_nat_false_iff x y = ?beq_nat_false_iff_rhs
```

□

4.2.4. *Exercise: 3 stars (beq_list).* Given a boolean operator `beq` for testing equality of elements of some type `a`, we can define a function `beq_list beq` for testing equality of lists with elements in `a`. Complete the definition of the `beq_list` function below. To make sure that your definition is correct, prove the lemma `beq_list_true_iff`.

```
beq_list : (beq : a -> a -> Bool) -> (l1, l2 : List a) -> Bool
beq_list beq l1 l2 = ?beq_list_rhs
```

```
beq_list_true_iff : (beq : a -> a -> Bool) ->
                    ((a1, a2 : a) -> (beq a1 a2 = True) <-> (a1 = a2)) ->
     ((l1, l2 : List a) -> (beq_list beq l1 l2 = True) <-> (l1 = l2))
beq_list_true_iff beq f l1 l2 = ?beq_list_true_iff_rhs
```

□

4.2.5. *Exercise: 2 stars, recommended (All_forallb).* Recall the function `forallb`, from the exercise `forall_exists_challenge` in chapter `Tactics`:

```
forallb : (test : x -> Bool) -> (l : List x) -> Bool
forallb _ [] = True
forallb test (x :: xs) = test x && forallb test xs
```

Prove the theorem below, which relates `forallb` to the *All* property of the above exercise.

```
forallb_true_iff : (l : List x) -> (forallb test l = True) <->
                                   (All (\x => test x = True) l)
forallb_true_iff l = ?forallb_true_iff_rhs
```

Are there any important properties of the function `forallb` which are not captured by this specification?

```
-- FILL IN HERE
```

□

4.3. Classical vs. Constructive Logic. We have seen that it is not possible to test whether or not a proposition `p` holds while defining a Idris function. You may be surprised to learn that a similar restriction applies to *proofs*! In other words, the following intuitive reasoning principle is not derivable in Idris:

```
excluded_middle : p `Either` (Not p)
```

To understand operationally why this is the case, recall that, to prove a statement of the form `p `Either` q`, we use the *Left* and *Right* pattern matches, which effectively require knowing which side of the disjunction holds. But the universally quantified `p` in `excluded_middle` is an arbitrary proposition, which we know nothing about. We don't have enough information to choose which of *Left* or *Right* to apply, just as Idris doesn't have enough information to mechanically decide whether `p` holds or not inside a function.

However, if we happen to know that `p` is reflected in some boolean term `b`, then knowing whether it holds or not is trivial: we just have to check the value of `b`.

```
restricted_excluded_middle : (p <-> b = True) -> p `Either` Not p
restricted_excluded_middle {b = True} (_, bp) = Left $ bp Refl
restricted_excluded_middle {b = False} (pb, _) = Right $ uninhabited . pb
```

In particular, the excluded middle is valid for equations `n = m`, between natural numbers `n` and `m`.

Is there a simpler way to write this? Maybe with setoids?

```
restricted_excluded_middle_eq : (n, m : Nat) -> (n = m) `Either` Not (n = m)
restricted_excluded_middle_eq n m =
  restricted_excluded_middle (to n m, fro n m)
where
  to : (n, m : Nat) -> (n=m) -> (n==m)=True
  to Z Z prf = Refl
  to Z (S _) Refl impossible
  to (S _) Z Refl impossible
  to (S k) (S j) prf = to k j (succInjective k j prf)
  fro : (n, m : Nat) -> (n==m)=True -> (n=m)
  fro Z Z Refl = Refl
  fro Z (S _) Refl impossible
  fro (S _) Z Refl impossible
  fro (S k) (S j) prf = rewrite fro k j prf in Refl
```

(Idris has a built-in version of this, called `decEq`.)

It may seem strange that the general excluded middle is not available by default in Idris; after all, any given claim must be either true or false. Nonetheless, there is an advantage in not assuming the excluded middle: statements in Idris can make stronger claims than the analogous statements in standard mathematics. Notably, if there is a Idris proof of `(x ** p x)`, it is possible to explicitly exhibit a value of `x` for which we can prove `p x` – in other words, every proof of existence is necessarily *constructive*.

Logics like Idris's, which do not assume the excluded middle, are referred to as *constructive logics*.

More conventional logical systems such as ZFC, in which the excluded middle does hold for arbitrary propositions, are referred to as *classical*.

The following example illustrates why assuming the excluded middle may lead to non-constructive proofs:

Use proper TeX?

Claim: There exist irrational numbers `a` and `b` such that `a ^ b` is rational.

Proof: It is not difficult to show that `sqrt 2` is irrational. If `sqrt 2 ^ sqrt 2` is rational, it suffices to take `a = b = sqrt 2` and we are done. Otherwise, `sqrt 2 ^ sqrt 2` is irrational. In this case, we can take `a = sqrt 2 ^ sqrt 2` and `b = sqrt 2`, since `a ^ b = sqrt 2 ^ (sqrt 2 * 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 that, we wind up knowing that such `a` and `b` exist but we cannot determine what their actual values are (at least, using this line of argument).

As useful as constructive logic is, it does have its limitations: There are many statements that can easily be proven in classical logic but that have much more complicated constructive proofs, and there are some that are known to have no constructive proof at all! Fortunately, like functional extensionality, the excluded middle is known to be compatible with Idris's logic, allowing us to add it safely as an axiom. However, we will not need to do so in this book: the results that we cover can be developed entirely within constructive logic at negligible extra cost.

It takes some practice to understand which proof techniques must be avoided in constructive reasoning, but arguments by contradiction, in particular, are infamous for leading to non-constructive proofs. Here's a typical example: suppose that we want to show that there exists `x` with some property `p`, i.e., such that `p x`. We start by assuming that our conclusion is false; that is, `Not (x : a ** p x)`. From this premise, it is not hard to derive `(x : a) -> Not $ p x`. If we manage to show that this intermediate fact results in a contradiction, we arrive at an existence proof without ever exhibiting a value of `x` for which `p x` holds!

The technical flaw here, from a constructive standpoint, is that we claimed to prove `(x ** p x)` using a proof of `Not $ Not (x ** p x)`. Allowing ourselves to remove double negations from arbitrary statements is equivalent to assuming the excluded middle, as shown in one of the exercises below. Thus, this line of reasoning cannot be encoded in Idris without assuming additional axioms.

4.3.1. *Exercise: 3 stars (excluded_middle_irrefutable).* The consistency of Idris with the general excluded middle axiom requires complicated reasoning that cannot be carried out within Idris 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* type `p`. Why? Because we cannot prove the negation of such an axiom; if we could, we would have both `Not (p `Either` Not p)` and `Not $ Not (p `Either` Not p)`, a contradiction.

```
excluded_middle_irrefutable : Not $ Not (p `Either` Not p)
excluded_middle_irrefutable = ?excluded_middle_irrefutable_rhs
```

□

4.3.2. *Exercise: 3 stars, advanced (not_exists_dist).* It is a theorem of classical logic that the following two assertions are equivalent:

```
Not (x : a ** Not p x)
(x : a) → p x
```

Add a hyperlink

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.

```
not_exists_dist : {p : a → Type} → Not (x ** Not $ p x) → ((x : a) → p x)
not_exists_dist prf x = ?not_exists_dist_rhs
  where
  excluded_middle : (a : Type) → a `Either` (Not a)
  excluded_middle p = really_believe_me p
```

□

4.3.3. *Exercise: 5 stars, optional (classical_axioms).* For those who like a challenge, here is an exercise taken from the Coq'Art book by Bertot and Casteran (p. 123). Each of the following four statements, together with `excluded_middle`, can be considered as characterizing classical logic. We can't prove any of them in Idris, but we can consistently add any one of them as an axiom if we wish to work in classical logic.

Prove that all five propositions (these four plus `excluded_middle`) are equivalent.

```
peirce : ((p → q) → p) → p

double_negation_elimination : Not $ Not p → p

de_morgan_not_and_not : Not (Not p, Not q) → p `Either` q

implies_to_or : (p → q) → ((Not p) `Either` q)

-- FILL IN HERE
```

□

CHAPTER 7

IndProp : Inductively Defined Propositions

```
module IndProp

import Basics
import Induction
import Tactics
import Logic

%hide Basics.Numbers.pred

%access public export
%default total
```

1. Inductively Defined Propositions

In the `Logic` chapter, we looked at several ways of writing propositions, including conjunction, disjunction, and quantifiers. In this chapter, we bring a new tool into the mix: *inductive definitions*.

Recall that we have seen two ways of stating that a number `n` is even: We can say (1) `evenb n = True`, or (2) `(k ** n = double k)`. Yet another possibility is to say that `n` is even if we can establish its evenness from the following rules:

- Rule `ev_0`: The number `0` is even.
- Rule `ev_SS`: If `n` is even, then `S (S n)` is even.

To illustrate how this definition of evenness works, let's imagine using it to show that `4` is even. By rule `ev_SS`, it suffices to show that `2` is even. This, in turn, is again guaranteed by rule `ev_SS`, as long as we can show that `0` is even. But this last fact follows directly from the `ev_0` rule.

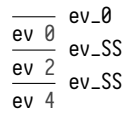
We will see many definitions like this one during the rest of the course. For purposes of informal discussions, it is helpful to have a lightweight notation that makes them easy to read and write. *Inference rules* are one such notation:

$$\frac{}{\texttt{ev 0}}\ \texttt{ev\_0}$$

$$\frac{\texttt{ev n}}{\texttt{ev (S (S n))}}\ \texttt{ev\_SS}$$

Each of the textual rules above is reformatted here as an inference rule; the intended reading is that, if the *premises* above the line all hold, then the conclusion below the line follows. For example, the rule `ev_SS` says that, if `n` satisfies `ev`, then `S (S n)` also does. If a rule has no premises above the line, then its conclusion holds unconditionally.

We can represent a proof using these rules by combining rule applications into a *proof tree*. Here's how we might transcribe the above proof that `4` is even:



Why call this a "tree" (rather than a "stack", for example)? Because, in general, inference rules can have multiple premises. We will see examples of this below.

Putting all of this together, we can translate the definition of evenness into a formal Idris definition using an `data` declaration, where each constructor corresponds to an inference rule:

```
data Ev : Nat → Type where
  Ev_0 : Ev Z
  Ev_SS : {n : Nat} → Ev n → Ev (S (S n))
```

This definition is different in one crucial respect from previous uses of `data`: its result is not a `Type`, but rather a function from `Nat` to `Type` – that is, a property of numbers. Note that we've already seen other inductive definitions that result in functions, such as `List`, whose type is `Type → Type`. What is new here is that, because the `Nat` argument of `Ev` appears unnamed, to the right of the colon, it is allowed to take different values in the types of different constructors: `Z` in the type of `Ev_0` and `S (S n)` in the type of `Ev_SS`.

In contrast, the definition of `List` names the `x` parameter globally, forcing the result of `Nil` and `(::)` to be the same (`List x`). Had we tried to name `Nat` in defining `Ev`, we would have seen an error:

```
data Wrong_ev : (n : Nat) → Type where
  Wrong_ev_0 : Wrong_ev Z
  Wrong_ev_SS : n → Wrong_ev n → Wrong_ev (S (S n))
```

```
When checking type of IndType.Wrong_ev_SS:
When checking argument n to IndType.Wrong_ev:
        Type mismatch between
                Type (Type of n)
        and
                Nat (Expected type)
```

Edit the explanation, it works fine if you remove the first `n →` in `Wrong_ev_SS`

("Parameter" here is Idris jargon for an argument on the left of the colon in an Inductive definition; "index" is used to refer to arguments on the right of the colon.)

We can think of the definition of `Ev` as defining a Idris property `Ev : Nat → Type`, together with theorems `Ev_0 : Ev Z` and `Ev_SS : n → Ev n → Ev (S (S n))`. Such "constructor theorems" have the same status as proven theorems. In particular, we can apply rule names as functions to each other to prove `Ev` for particular numbers…

```
ev_4 : Ev 4
ev_4 = Ev_SS {n=2} $ Ev_SS {n=0} Ev_0
```

We can also prove theorems that have hypotheses involving `Ev`.

```
ev_plus4 : Ev n → Ev (4 + n)
ev_plus4 x = Ev_SS $ Ev_SS x
```

More generally, we can show that any number multiplied by 2 is even:

1.0.1. *Exercise: 1 star (ev_double).*

```
ev_double : Ev (double n)
ev_double = ?ev_double_rhs
```

□

2. Using Evidence in Proofs

Besides *constructing* evidence that numbers are even, we can also *reason about* such evidence.

Introducing `Ev` with a `data` declaration tells Idris not only that the constructors `Ev_0` and `Ev_SS` are valid ways to build evidence that some number is even, but also that these two constructors are the *only* ways to build evidence that numbers are even (in the sense of `Ev`).

In other words, if someone gives us evidence `e` for the assertion `Ev n`, then we know that `e` must have one of two shapes:

- `e` is `Ev_0` (and `n` is `Z`), or
- `e` is `Ev_SS {n=n'} e'` (and `n` is `S (S n')`, where `e'` is evidence for `Ev n'`).

This suggests that it should be possible to analyze a hypothesis of the form `Ev n` much as we do inductively defined data structures; in particular, it should be possible to argue by *induction* and *case analysis* on such evidence. Let's look at a few examples to see what this means in practice.

2.1. Pattern Matching on Evidence.

Edit the whole section to talk about dependent pattern matching

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 the

`inversion` tactic, generating separate subgoals for the case where `n = Z` and the case where `n = S n'` for some `n'`. But for some proofs we may instead want to analyze the evidence that `Ev n` directly.

By the definition of `Ev`, there are two cases to consider:

- If the evidence is of the form `Ev_0`, we know that `n = Z`.
- Otherwise, the evidence must have the form `Ev_SS {n=n'} e'`, where `n = S (S n')` and `e'` is evidence for `Ev n'`.

We can perform this kind of reasoning in Idris, again using pattern matching. Besides allowing us to reason about equalities involving constructors, `inversion` provides a case-analysis principle for inductively defined propositions. When used in this way, its syntax is similar to `destruct`: We pass it a list of identifiers separated by | characters to name the arguments to each of the possible constructors.

```
ev_minus2 : Ev n -> Ev (pred (pred n))
ev_minus2 Ev_0       = Ev_0
ev_minus2 (Ev_SS e') = e'
```

In words, here is how the pattern match reasoning works in this proof:

- If the evidence is of the form `Ev_0`, we know that `n = Z`. Therefore, it suffices to show that `Ev (pred (pred Z))` holds. By the definition of `pred`, this is equivalent to showing that `Ev Z` holds, which directly follows from `Ev_0`.
- Otherwise, the evidence must have the form `Ev_SS {n=n'} e'`, where `n = S (S n')` and `e'` is evidence for `Ev n'`. We must then show that `Ev (pred (pred (S (S n'))))` holds, which, after simplification, follows directly from `e'`.

Suppose that we wanted to prove the following variation of `ev_minus2`:

```
evSS_ev : Ev (S (S n)) -> Ev n
```

Intuitively, we know that evidence for the hypothesis cannot consist just of the `Ev_0` constructor, since `Z` and `S` are different constructors of the type `Nat`; hence, `Ev_SS` is the only case that applies. Unfortunately, destruct is not smart enough to realize this, and it still generates two subgoals. Even worse, in doing so, it keeps the final goal unchanged, failing to provide any useful information for completing the proof.

The inversion tactic, on the other hand, can detect (1) that the first case does not apply, and (2) that the `n'` that appears on the `Ev_SS` case must be the same as `n`. This allows us to complete the proof

```
evSS_ev (Ev_SS e') = e'
```

By using dependent pattern matching, we can also apply the principle of explosion to “obviously contradictory” hypotheses involving inductive properties. For example:

```
one_not_even : Not (Ev 1)
one_not_even Ev_0 impossible
one_not_even (Ev_SS _) impossible
```

2.2. Exercise: 1 star (inversion_practice). Prove the following results using pattern matching.

```
SSSSev__even : Ev (S (S (S (S n)))) → Ev n
SSSSev__even e = ?SSSSev__even_rhs

even5_nonsense : Ev 5 → 2 + 2 = 9
even5_nonsense e = ?even5_nonsense_rhs
```

□

Edit

The way we've used `inversion` here may seem a bit mysterious at first. Until now, we've only used `inversion` on equality propositions, to utilize injectivity of constructors or to discriminate between different constructors. But we see here that `inversion` can also be applied to analyzing evidence for inductively defined propositions.

Here's how `inversion` works in general. Suppose the name `I` refers to an assumption `P` in the current context, where `P` has been defined by an `Inductive` declaration. Then, for each of the constructors of `P`, `inversion I` generates a subgoal in which `I` has been replaced by the exact, specific conditions under which this constructor could have been used to prove `P`. Some of these subgoals will be self-contradictory; `inversion` throws these away. The ones that are left represent the cases that must be proved to establish the original goal. For those, `inversion` adds all equations into the proof context that must hold of the arguments given to `P` (e.g., `S (S n') = n` in the proof of `evSS_ev`).

The `ev_double` exercise above shows that our new notion of evenness is implied by the two earlier ones (since, by `even_bool_prop` in chapter `Logic`, we already know that those are equivalent to each other). To show that all three coincide, we just need the following lemma:

```
ev_even : Ev n → (k ** n = double k)
```

We proceed by case analysis on `Ev n`. The first case can be solved trivially.

```
ev_even Ev_0 = (Z ** Refl)
```

Unfortunately, the second case is harder. We need to show `(k ** S (S n') = double k`, but the only available assumption is `e'`, which states that `Ev n'` holds. Since this isn't directly useful, it seems that we are stuck and that performing case analysis on `Ev n` was a waste of time.

If we look more closely at our second goal, however, we can see that something interesting happened: By performing case analysis on `Ev n`, we were able to reduce

the original result to an similar one that involves a *different* piece of evidence for `Ev n: e'`. More formally, we can finish our proof by showing that

```
(k' ** n' = double k')
```

which is the same as the original statement, but with `n'` instead of `n`. Indeed, it is not difficult to convince Idris that this intermediate result suffices.

```
ev_even (Ev_SS e') = I $ ev_even e'
  where
    I : (k' ** n' = double k') -> (k ** S (S n') = double k)
    I (k' ** prf) = (S k' ** cong {f=S} $ cong {f=S} prf)
```

2.3. Induction on Evidence.

Edit, we've already just done an induction-style proof, the following is basically replacing 'where' with 'let'

If this looks familiar, it is no coincidence: We've encountered similar problems in the `Induction` chapter, when trying to use case analysis to prove results that required induction. And once again the solution is... induction!

The behavior of `induction` on evidence is the same as its behavior on data: It causes Idris to generate one subgoal for each constructor that could have used to build that evidence, while providing an induction hypotheses for each recursive occurrence of the property in question.

Let's try our current lemma again:

```
ev_even' : Ev n -> (k ** n = double k)
ev_even' Ev_0       = (Z ** Refl)
ev_even' (Ev_SS e') =
  let
    (k**prf) = ev_even e'
    cprf = cong {f=S} $ cong {f=S} prf
  in
    rewrite cprf in (S k ** Refl)
```

Here, we can see that Idris 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.

```
ev_even_iff : (Ev n) <-> (k ** n = double k)
ev_even_iff = (ev_even, fro)
  where
    fro : (k ** n = double k) -> (Ev n)
    fro (k ** prf) = rewrite prf in ev_double {n=k}
```

As we will see in later chapters, induction on evidence is a recurring technique across many areas, and in particular when formalizing the semantics of programming languages, where many properties of interest are defined inductively.

The following exercises provide simple examples of this technique, to help you familiarize yourself with it.

2.3.1. *Exercise: 2 stars (ev_sum).*

```
ev_sum : Ev n → Ev m → Ev (n + m)
ev_sum x y = ?ev_sum_rhs
```

□

2.4. Exercise: 4 stars, advanced, optional (ev_alternate). In general, there may be multiple ways of defining a property inductively. For example, here's a (slightly contrived) alternative definition for *Ev*:

```
data Ev' : Nat → Type where
  Ev'_0 : Ev' Z
  Ev'_2 : Ev' 2
  Ev'_sum : Ev' n → Ev' m → Ev' (n + m)
```

Prove that this definition is logically equivalent to the old one. (You may want to look at the previous theorem when you get to the induction step.)

```
ev'_ev : (Ev' n) ↔ Ev n
ev'_ev = ?ev__ev_rhs
```

□

2.5. Exercise: 3 stars, advanced, recommended (ev_ev___ev). Finding the appropriate thing to do induction on is a bit tricky here:

```
ev_ev__ev : Ev (n+m) → Ev n → Ev m
ev_ev__ev x y = ?ev_ev__ev_rhs
```

□

2.5.1. *Exercise: 3 stars, optional (ev_plus_plus).* This exercise just requires applying existing lemmas. No induction or even case analysis is needed, though some of the rewriting may be tedious.

```
ev_plus_plus : Ev (n+m) → Ev (n+p) → Ev (m+p)
ev_plus_plus x y = ?ev_plus_plus_rhs
```

□

3. 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.

One useful example is the "less than or equal to" relation on numbers.

The following definition should be fairly intuitive. It says that there are two ways to give evidence that one number is less than or equal to another: either observe that they are the same number, or give evidence that the first is less than or equal to the predecessor of the second.

```
data Le : Nat → Nat → Type where
  Le_n : Le n n
  Le_S : Le n m → Le n (S m)

syntax [m] "≤'" [n] = Le m n
```

Proofs of facts about `≤'` 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 `≤'` goals (e.g., to show that `3≤'3` or `3≤'6`), and we can use pattern matching to extract information from `≤'` hypotheses in the context (e.g., to prove that `(2≤'1) → 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 – `Refl` doesn't do the job, because the proofs aren't just a matter of simplifying computations.)

```
test_le1 : 3 ≤' 3
test_le1 = Le_n

test_le2 : 3 ≤' 6
test_le2 = Le_S $ Le_S $ Le_S Le_n

test_le3 : (2≤'1) → 2+2=5
test_le3 (Le_S Le_n) impossible
test_le3 (Le_S (Le_S _)) impossible
```

The "strictly less than" relation `n < m` can now be defined in terms of `Le`.

```
Lt : (n, m : Nat) → Type
Lt n m = Le (S n) m

syntax [m] "<'" [n] = Lt m n
```

Here are a few more simple relations on numbers:

```
data Square_of : Nat → Nat → Type where
  Sq : Square_of n (n * n)

data Next_nat : Nat → Nat → Type where
  Nn : Next_nat n (S n)
```

```
data Next_even : Nat → Nat → Type where
  Ne_1 : Ev (S n) → Next_even n (S n)
  Ne_2 : Ev (S (S n)) → Next_even n (S (S n))
```

3.0.1. *Exercise: 2 stars, optional (total_relation).* Define an inductive binary relation `Total_relation` that holds between every pair of natural numbers.

```
-- FILL IN HERE
```

□

3.1. Exercise: 2 stars, optional (empty_relation). Define an inductive binary relation `Empty_relation` (on numbers) that never holds.

```
--FILL IN HERE
```

□

3.1.1. *Exercise: 3 stars, optional (le_exercises).* Here are a number of facts about the `≤'` and `<'` relations that we are going to need later in the course. The proofs make good practice exercises.

```
le_trans : (m ≤' n) → (n ≤' o) → (m ≤' o)
le_trans x y = ?le_trans_rhs

O_le_n : Z ≤' n
O_le_n = ?O_le_n_rhs

n_le_m__Sn_le_Sm : (n ≤' m) → ((S n) ≤' (S m))
n_le_m__Sn_le_Sm x = ?n_le_m__Sn_le_Sm_rhs

Sn_le_Sm__n_le_m : ((S n) ≤' (S m)) → (n ≤' m)
Sn_le_Sm__n_le_m x = ?Sn_le_Sm__n_le_m_rhs

le_plus_l : a ≤' (a + b)
le_plus_l = ?le_plus_l_rhs

plus_lt : ((n1 + n2) <' m) → (n1 <' m, n2 <' m)
plus_lt x = ?plus_lt_rhs

lt_S : (n <' m) → (n <' S m)
lt_S x = ?lt_S_rhs

lte_complete : lte n m = True → (n ≤' m)
lte_complete prf = ?lte_complete_rhs
```

Hint: The next one may be easiest to prove by induction on `m`.

```
lte_correct : (n ≤' m) → lte n m = True
lte_correct x = ?lte_correct_rhs
```

Hint: This theorem can easily be proved without using induction.

```
lte_true_trans : lte n m = True → lte m o = True → lte n o = True
lte_true_trans prf prf1 = ?lte_true_trans_rhs
```

3.1.2. *Exercise: 2 stars, optional (lte_iff).*

```
lte_iff : (lte n m = True) ↔ (n ≤' m)
lte_iff = ?lte_iff_rhs
```

□

```
namespace R
```

3.1.3. *Exercise: 3 stars, recommended (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:

```
  data R : Nat → Nat → Nat → Type where
    C1 : R 0 0 0
    C2 : R m n o → R (S m) n (S o)
    C3 : R m n o → R m (S n) (S o)
    C4 : R (S m) (S n) (S (S o)) → R m n o
    C5 : 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.

```
-- FILL IN HERE
```

□

3.1.4. *Exercise: 3 stars, optional (R_fact).* The relation `R` above actually encodes a familiar function. Figure out which function; then state and prove this equivalence in Idris?

```
  fR : Nat → Nat → Nat
  fR k j = ?fR_rhs

  R_equiv_fR : (R m n o) ↔ (fR m n = o)
  R_equiv_fR = ?R_equiv_fR_rhs
```

□

3.2. 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 type `Subseq` on `List Nat` that captures what it means to be a subsequence. (Hint: You'll need three cases.)
- Prove `subseq_refl` that subsequence is reflexive, that is, any list is a subsequence of itself.
- Prove `subseq_app` that for any lists `l1`, `l2`, and `l3`, if `l1` is a subsequence of `l2`, then `l1` is also a subsequence of `l2 ++ l3`.
- (Optional, harder) Prove `subseq_trans` that subsequence is transitive – that is, if `l1` is a subsequence of `l2` and `l2` is a subsequence of `l3`, then `l1` is a subsequence of `l3`. Hint: choose your induction carefully!

```
-- FILL IN HERE
```

□

3.2.1. *Exercise: 2 stars, optional (R_provability2).* Suppose we give Idris the following definition:

```
data R' : Nat → List Nat → Type where
  C1' : R' 0 []
  C2' : R' n l → R' (S n) (n :: l)
  C3' : 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]`

□

4. Case Study: Regular Expressions

The `Ev` property provides a simple example for illustrating inductive definitions and the basic techniques for reasoning about them, but it is not terribly exciting – after all, it is equivalent to the two non-inductive of evenness that we had already

seen, and does not seem to offer any concrete benefit over them. To give a better sense of the power of inductive definitions, we now show how to use them to model a classic concept in computer science: *regular expressions*.

Regular expressions are a simple language for describing strings, defined as follows:

```
data Reg_exp : (t : Type) → Type where
  EmptySet : Reg_exp t
  EmptyStr : Reg_exp t
  Chr : t → Reg_exp t
  App : Reg_exp t → Reg_exp t → Reg_exp t
  Union : Reg_exp t → Reg_exp t → Reg_exp t
  Star : Reg_exp t → Reg_exp t
```

Note that this definition is *polymorphic*: Regular expressions in `Reg_exp t` describe strings with characters drawn from`t` – that is, lists of elements of `t`.

(We depart slightly from standard practice in that we do not require the type `t` to be finite. This results in a somewhat different theory of regular expressions, but the difference is not significant for our purposes.)

We connect regular expressions and strings via the following rules, which define when a regular expression *matches* some string:

- The expression `EmptySet` does not match any string.
- The expression `EmptyStr` matches the empty string `[]`.
- The expression `Chr 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 ++ ... ++ s_k`, and the expression `re` matches each one of the strings `s_i`, then `Star re` matches `s`.

  As a special case, the sequence of strings may be empty, so `Star re` always matches the empty string `[]` no matter what `re` is.

We can easily translate this informal definition into a `data` one as follows:

```
data Exp_match : List t → Reg_exp t → Type where
  MEmpty : Exp_match [] EmptyStr
  MChar : Exp_match [x] (Chr x)
  MApp : Exp_match s1 re1 → Exp_match s2 re2 →
         Exp_match (s1 ++ s2) (App re1 re2)
  MUnionL : Exp_match s1 re1 →
            Exp_match s1 (Union re1 re2)
  MUnionR : Exp_match s2 re2 →
            Exp_match s2 (Union re1 re2)
  MStar0 : Exp_match [] (Star re)
  MStarApp : Exp_match s1 re →
```

```
    Exp_match s2 (Star re) →
    Exp_match (s1 ++ s2) (Star re)
```

Again, for readability, we can also display this definition using inference-rule notation. At the same time, let's introduce a more readable infix notation.

```
syntax [s] "=~" [re] = (Exp_match s re)
```

$$\frac{}{[] \text{ =~ } \mathit{EmptyStr}}\ \mathit{MEmpty}$$

$$\frac{}{[x] \text{ =~ } \mathit{Chr}\ x}\ \mathit{MChar}$$

$$\frac{s1 \text{ =~ } re1 \quad\quad s2 \text{ =~ } r2}{s1 \text{ ++ } s2 \text{ =~ } \mathit{App}\ re1\ re2}\ \mathit{MApp}$$

$$\frac{s1 \text{ =~ } re1}{s1 \text{ =~ } \mathit{Union}\ re1\ re2}\ \mathit{MUnionL}$$

$$\frac{s2 \text{ =~ } re2}{s2 \text{ =~ } \mathit{Union}\ re1\ re2}\ \mathit{MUnionR}$$

$$\frac{}{[] \text{ =~ } \mathit{Star}\ re}\ \mathit{MStar0}$$

$$\frac{s1 \text{ =~ } re \quad\quad s2 \text{ =~ } \mathit{Star}\ re}{s1 \text{ ++ } s2 \text{ =~ } \mathit{Star}\ re}\ \mathit{MStarApp}$$

Notice that these rules are not *quite* the same as the informal ones that we gave at the beginning of the section. First, we don't need to include a rule explicitly stating that no string matches *EmptySet*; we just don't happen to include any rule that would have the effect of some string matching *EmptySet*. (Indeed, the syntax of inductive definitions doesn't even *allow* us to give such a "negative rule.")

Second, the informal rules for *Union* and *Star* correspond to two constructors each: *MUnionL* / *MUnionR*, and *MStar0* / *MStarApp*. The result is logically equivalent to the original rules but more convenient to use in Idris, since the recursive occurrences of *Exp_match* are given as direct arguments to the constructors, making it easier to perform induction on evidence. (The `exp_match_ex1` and `exp_match_ex2` exercises below ask you to prove that the constructors given in the inductive declaration and the ones that would arise from a more literal transcription of the informal rules are indeed equivalent.)

Let's illustrate these rules with a few examples.

```
reg_exp_ex1 : [1] =~ (Chr 1)
reg_exp_ex1 = MChar

reg_exp_ex2 : [1,2] =~ (App (Chr 1) (Chr 2))
reg_exp_ex2 = MApp {s1=[1]} {s2=[2]} MChar MChar
```

Notice how the last example applies *MApp* to the strings [1] and [2] directly. While the goal mentions [1,2] instead of [1] ++ [2], Idris is able to figure out how to split the string on its own, so we can drop the implicits:

```
reg_exp_ex2 : [1,2] =~ (App (Chr 1) (Chr 2))
reg_exp_ex2 = MApp MChar MChar
```

Using pattern matching, we can also show that certain strings do not match a regular expression:

```
reg_exp_ex3 : Not ([1,2] =~ (Chr 1))
reg_exp_ex3 MEmpty impossible
reg_exp_ex3 MChar impossible
reg_exp_ex3 (MApp _ _) impossible
reg_exp_ex3 (MUnionL _) impossible
reg_exp_ex3 (MUnionR _) impossible
reg_exp_ex3 MStar0 impossible
reg_exp_ex3 (MStarApp _ _) impossible
```

We can define helper functions to help write down regular expressions. The `reg_exp_of_list` function constructs a regular expression that matches exactly the list that it receives as an argument:

```
reg_exp_of_list : List t -> Reg_exp t
reg_exp_of_list [] = EmptyStr
reg_exp_of_list (x :: xs) = App (Chr x) (reg_exp_of_list xs)

reg_exp_ex4 : [1,2,3] =~ (reg_exp_of_list [1,2,3])
reg_exp_ex4 = MApp MChar $ MApp MChar $ MApp MChar MEmpty
```

We can also prove general facts about *Exp_match*. For instance, the following lemma shows that every string `s` that matches `re` also matches *Star* `re`.

```
MStar1 : (s =~ re) -> (s =~ Star re)
MStar1 {s} h =
  rewrite sym $ appendNilRightNeutral s in
  MStarApp h MStar0
```

(Note the use of `appendNilRightNeutral` to change the goal of the theorem to exactly the same shape expected by *MStarApp*.)

4.0.1. *Exercise: 3 stars (exp_match_ex1).* The following lemmas show that the informal matching rules given at the beginning of the chapter can be obtained from the formal inductive definition.

```
empty_is_empty : Not (s =~ EmptySet)
empty_is_empty = ?empty_is_empty_rhs

MUnion' : (s =~ re1, s =~ re2) -> s =~ Union re1 re2
MUnion' m = ?MUnion__rhs
```

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 (++) ss []` is the result of concatenating them all together.

Copied from *Poly*, cannot import it due to tuple sugar issues

```
fold : (f : x → y → y) → (l : List x) → (b : y) → y
fold f [] b = b
fold f (h::t) b = f h (fold f t b)

MStar' : ((s : List t) → (In s ss) → (s =~ re)) →
         (fold (++) ss []) =~ Star re
MStar' f = ?MStar__rhs
```

□

4.0.2. *Exercise: 4 stars (reg_exp_of_list).* Prove that `reg_exp_of_list` satisfies the following specification:

```
reg_exp_of_list_spec : (s1 =~ reg_exp_of_list s2) ↔ (s1 = s2)
reg_exp_of_list_spec = ?reg_exp_of_list_spec_rhs
```

□

Since the definition of *Exp_match* has a recursive structure, we might expect that proofs involving regular expressions will often require induction on evidence. For example, suppose that we wanted to prove the following intuitive result: If a regular expression `re` matches some string `s`, then all elements of `s` must occur somewhere in `re`. To state this theorem, we first define a function `re_chars` that lists all characters that occur in a regular expression:

```
re_chars : (re : Reg_exp t) → List t
re_chars EmptySet       = []
re_chars EmptyStr       = []
re_chars (Chr x)        = [x]
re_chars (App re1 re2)  = re_chars re1 ++ re_chars re2
re_chars (Union re1 re2) = re_chars re1 ++ re_chars re2
re_chars (Star re)      = re_chars re

re_star : re_chars (Star re) = re_chars re
re_star = Refl
```

We can then phrase our theorem as follows:

Some unfortunate implicit plumbing

```
destruct : In x (s1 ++ s2) → (In x s1) `Either` (In x s2)
destruct {x} {s1} {s2} = fst $ in_app_iff {a=x} {l=s1} {l'=s2}

construct : (In x (re_chars re1)) `Either` (In x (re_chars re2)) →
            In x ((re_chars re1) ++ (re_chars re2))
```

```
construct {x} {re1} {re2} =
  snd $ in_app_iff {a=x} {l=(re_chars re1)} {l'=(re_chars re2)}

in_re_match : (s =~ re) -> In x s -> In x (re_chars re)
in_re_match MEmpty prf = prf
in_re_match MChar prf = prf
in_re_match (MApp m1 m2) prf = construct $ case destruct prf of
    Left prf1 => Left $ in_re_match m1 prf1
    Right prf2 => Right $ in_re_match m2 prf2
in_re_match (MUnionL ml) prf = construct $ Left $ in_re_match ml prf
in_re_match (MUnionR mr) prf = construct $ Right $ in_re_match mr prf
in_re_match MStar0 prf = absurd prf
in_re_match (MStarApp m ms) prf = case destruct prf of
```

Edit

Something interesting happens in the `MStarApp` case. We obtain two induction hypotheses: One that applies when `x` occurs in `s1` (which matches `re`), and a second one that applies when `x` occurs in `s2` (which matches `Star re`). This is a good illustration of why we need induction on evidence for `Exp_match`, as opposed to `re`: The latter would only provide an induction hypothesis for strings that match `re`, which would not allow us to reason about the case `In x s2`.

```
    Left prf' => in_re_match m prf'
    Right prfs => in_re_match ms prfs
```

4.0.3. *Exercise: 4 stars (re_not_empty).* Write a recursive function `re_not_empty` that tests whether a regular expression matches some string. Prove that your function is correct.

```
re_not_empty : (re : Reg_exp t) -> Bool
re_not_empty re = ?re_not_empty_rhs

re_not_empty_correct : (s ** s =~ re) <-> re_not_empty re = True
re_not_empty_correct = ?re_not_empty_correct_rhs
```

□

4.1. The remember Tactic.

Rewrite the section, dependent pattern matching figures all of this out

One potentially confusing feature of the induction tactic is that it happily lets you try to set up an induction over a term that isn't sufficiently general. The effect of this is to lose information (much as destruct can do), and leave you unable to complete the proof. Here's an example:

```
star_app : (s1 =~ Star re) -> (s2 =~ Star re) -> (s1 ++ s2) =~ Star re
star_app MStar0 m2 = m2
star_app {s2} (MStarApp {s1=s11} {s2=s21} m ms) m2 =
```

```
  rewrite sym $ appendAssociative s11 s21 s2 in
    MStarApp m (star_app ms m2)
```

Just doing an inversion on H1 won't get us very far in the recursive cases. (Try it!). So we need induction. Here is a naive first attempt:

```
  induction H1
    as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
        |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
        |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

But now, although we get seven cases (as we would expect from the definition of Exp_match), we 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…

```
  - (* MEmpty *)
    simpl. intros H. apply H.
```

… but most cases get stuck. For MChar, for instance, we must show that

```
    s2 =~ Char x' -> x' :: s2 =~ Char x',
```

which is clearly impossible.

```
  - (* MChar. Stuck... *)
Abort.
```

The problem is that induction over a Type hypothesis only works properly with hypotheses that are completely general, i.e., ones in which all the arguments are variables, as opposed to more complex expressions, such as Star re.

(In this respect, induction on evidence behaves more like destruct than like inversion.)

We can solve this problem by generalizing over the problematic expressions with an explicit equality:

```
Lemma star_app: forall T (s1 s2 : list T) (re re' : Reg_exp T),
  s1 =~ re' ->
  re' = Star re ->
  s2 =~ Star re ->
  s1 ++ s2 =~ Star re.
```

We can now proceed by performing induction over evidence directly, because the argument to the first hypothesis is sufficiently general, which means that we can discharge most cases by inverting the re' = Star re equality in the context.

This idiom is so common that Idris provides a tactic to automatically generate such equations for us, avoiding thus the need for changing the statements of our theorems.

Invoking the tactic remember e as x causes Idris to (1) replace all occurrences of the expression e by the variable x, and (2) add an equation x = e to the context. Here's how we can use it to show the above result:

```
Abort.

Lemma star_app: forall T (s1 s2 : list T) (re : Reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
Proof.
  intros T s1 s2 re H1.
  remember (Star re) as re'.
```

We now have Heqre' : re' = Star re.

```
  generalize dependent s2.
  induction H1
    as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
        |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
        |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

The Heqre' is contradictory in most cases, which allows us to conclude immediately.

```
  - (* MEmpty *) inversion Heqre'.
  - (* MChar *) inversion Heqre'.
  - (* MApp *) inversion Heqre'.
  - (* MUnionL *) inversion Heqre'.
  - (* MUnionR *) inversion Heqre'.
```

The interesting cases are those that correspond to Star. 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.

```
  - (* MStar0 *)
    inversion Heqre'. intros s H. apply H.

  - (* MStarApp *)
    inversion Heqre'. rewrite H0 in IH2, Hmatch1.
    intros s2 H1. rewrite ← app_assoc.
    apply MStarApp.
    + apply Hmatch1.
    + apply IH2.
      * reflexivity.
      * apply H1.
Qed.
```

4.1.1. *Exercise: 4 stars (exp_match_ex2).* The `MStar''` lemma below (combined with its converse, the `MStar'` exercise above), shows that our definition of `Exp_match` for `Star` is equivalent to the informal one given previously.

```
MStar'' : (s =~ Star re) ->
          (ss : List (List t) **
               (s = fold (++) ss [], (s': List t) -> In s' ss -> s' =~ re)
          )
MStar'' m = ?MStar___rhs
```

□

4.1.2. *Exercise: 5 stars, advanced (pumping).* One of the first really interesting theorems in the theory of regular expressions is the so-called *pumping lemma*, which states, informally, that any sufficiently long string `s` matching a regular expression `re` can be "pumped" by repeating some middle section of `s` an arbitrary number of times to produce a new string also matching `re`.

To begin, we need to define "sufficiently long." Since we are working in a constructive logic, we actually need to be able to calculate, for each regular expression `re`, the minimum length for strings `s` to guarantee "pumpability."

```
namespace Pumping

  pumping_constant : (re : Reg_exp t) -> Nat
  pumping_constant EmptySet        = 0
  pumping_constant EmptyStr        = 1
  pumping_constant (Chr _)         = 2
  pumping_constant (App re1 re2)   =
    pumping_constant re1 + pumping_constant re2
  pumping_constant (Union re1 re2) =
    pumping_constant re1 + pumping_constant re2
  pumping_constant (Star _)        = 1
```

Next, it is useful to define an auxiliary function that repeats a string (appends it to itself) some number of times.

```
  napp : (n : Nat) -> (l : List t) -> List t
  napp Z _     = []
  napp (S k) l = l ++ napp k l

  napp_plus: (n, m : Nat) -> (l : List t) ->
             napp (n + m) l = napp n l ++ napp m l
  napp_plus Z _ _     = Refl
  napp_plus (S k) m l =
   rewrite napp_plus k m l in
     appendAssociative l (napp k l) (napp m l)
```

Now, the pumping lemma itself says that, if `s =~ re` and if the length of `s` is at least the pumping constant of `re`, then `s` can be split into three substrings `s1 ++ s2 ++ s3` in such a way that `s2` can be repeated any number of times and the result, when combined with `s1` and `s3` will still match `re`. Since `s2` is also guaranteed not to be the empty string, this gives us a (constructive!) way to generate strings matching `re` that are as long as we like.

```
pumping : (s =~ re) -> ((pumping_constant re) ≤' (length s)) ->
          (s1 ** s2 ** s3 ** ( s = s1 ++ s2 ++ s3
                             , Not (s2 = [])
                             , (m:Nat) -> (s1 ++ napp m s2 ++ s3) =~ re
                             ))
```

Edit hint

To streamline the proof (which you are to fill in), the `omega` tactic, which is enabled by the following `Require`, is helpful in several places for automatically completing tedious low-level arguments involving equalities or inequalities over natural numbers. We'll return to `omega` in a later chapter, but feel free to experiment with it now if you like. The first case of the induction gives an example of how it is used.

```
pumping m le = ?pumping_rhs
```

5. Case Study: Improving Reflection

We've seen in the *Logic* chapter that we often need to relate boolean computations to statements in *Type*. But performing this conversion in the way we did it there can result in tedious proof scripts. Consider the proof of the following theorem:

```
filter_not_empty_In : {n : Nat} -> Not (filter ((==) n) l = []) -> In n l
filter_not_empty_In {l=[]} contra = contra Refl
filter_not_empty_In {l=(x::_)} {n} contra with (n == x) proof h
  filter_not_empty_In contra | True =
    Left $ sym $ beq_nat_true $ sym h
  filter_not_empty_In contra | False =
    Right $ filter_not_empty_In contra
```

In the second case we explicitly apply the `beq_nat_true` lemma to the equation generated by doing a dependent match on `n == x`, to convert the assumption `n == x = True` into the assumption `n = m`.

We can streamline this by defining an inductive proposition that yields a better case-analysis principle for `n == m`. Instead of generating an equation such as `n == m = True`, which is generally not directly useful, this principle gives us right away the assumption we really need: `n = m`.

We'll actually define something a bit more general, which can be used with arbitrary properties (and not just equalities):

Update the text: seems that additional (b=...) constructor parameter is needed for this to work in Idris.

```
data Reflect : Type -> Bool -> Type where
 ReflectT : (p : Type) -> (b=True) -> Reflect p b
 ReflectF : (p : Type) -> (Not p) -> (b=False) -> Reflect p b
```

Before explaining this, let's rearrange it a little: Since the types of both *ReflectT* and *ReflectF* begin with (p : *Type*), we can make the definition a bit more readable and easier to work with by making p a parameter of the whole data declaration.

```
data Reflect : (p : Type) → (b : Bool) → Type where
  ReflectT : p → (b=True) → Reflect p b
  ReflectF : (Not p) → (b=False) → Reflect p b
```

The reflect property takes two arguments: a proposition p and a boolean b. Intuitively, it states that the property p is *reflected* in (i.e., equivalent to) the boolean b: p holds if and only if b = *True*. To see this, notice that, by definition, the only way we can produce evidence that *Reflect* p *True* holds is by showing that p is true and using the *ReflectT* constructor. If we invert this statement, this means that it should be possible to extract evidence for p from a proof of *Reflect* p *True*. Conversely, the only way to show *Reflect* p *False* is by combining evidence for *Not* p with the *ReflectF* constructor.

It is easy to formalize this intuition and show that the two statements are indeed equivalent:

```
iff_reflect : (p ↔ (b = True)) → Reflect p b
iff_reflect {b = False} (pb, _) = ReflectF (uninhabited . pb) Refl
iff_reflect {b = True} (_, bp) = ReflectT (bp Refl) Refl
```

5.0.1. *Exercise: 2 stars, recommended (reflect_iff).*

```
reflect_iff : Reflect p b → (p ↔ (b = True))
reflect_iff x = ?reflect_iff_rhs
```

□

The advantage of *Reflect* over the normal "if and only if" connective is that, by destructing a hypothesis or lemma of the form *Reflect* p b, we can perform case analysis on b while at the same time generating appropriate hypothesis in the two branches (p in the first subgoal and *Not* p in the second).

```
beq_natP : {n, m : Nat} → Reflect (n = m) (n == m)
beq_natP {n} {m} = iff_reflect $ iff_sym $ beq_nat_true_iff n m
```

Edit - we basically trade the invocation of beq_nat_true in *Left* for an indirect rewrite in *Right*

The new proof of filter_not_empty_In now goes as follows. Notice how the calls to destruct and apply are combined into a single call to destruct.

(To see this clearly, look at the two proofs of filter_not_empty_In with Idris and observe the differences in proof state at the beginning of the first case of the destruct.)

```
filter_not_empty_In' : {n : Nat} → Not (filter ((==) n) l = []) → In n l
filter_not_empty_In' {l=[]} contra = contra Refl
filter_not_empty_In' {n} {l=(x::xs)} contra with (beq_natP {n} {m=x})
  filter_not_empty_In' _ | (ReflectT eq _) = Left $ sym eq
```

```
filter_not_empty_In' {n} {l=(x::xs)} contra | (ReflectF _ notbeq) =
  let
```

How to rewrite more neatly here?

```
    contra' = replace notbeq contra
                      {P = \a =>
                             Not ((if a
                                      then x :: filter ((=) n) xs
                                      else filter ((=) n) xs) = [])}
  in
    Right $ filter_not_empty_In' contra'
```

5.0.2. *Exercise: 3 stars, recommended (beq_natP_practice).* Use `beq_natP` as above to prove the following:

```
count : (n : Nat) -> (l : List Nat) -> Nat
count _ [] = 0
count n (x :: xs) = (if n == x then 1 else 0) + count n xs

beq_natP_practice : count n l = 0 -> Not (In n l)
beq_natP_practice prf = ?beq_natP_practice_rhs
```

□

This technique gives us only a small gain in convenience for the proofs we've seen here, but using *Reflect* consistently often leads to noticeably shorter and clearer scripts as proofs get larger. We'll see many more examples in later chapters.

Add http://math-comp.github.io/math-comp/ as a link

The use of the reflect property was popularized by `SSReflect`, a Coq library that has been used to formalize important results in mathematics, including as the 4-color theorem and the Feit-Thompson theorem. The name `SSReflect` stands for *small-scale reflection*, i.e., the pervasive use of reflection to simplify small proof steps with boolean computations.

6. Additional Exercises

6.0.1. *Exercise: 3 stars, recommended (nostutter).* Formulating inductive definitions of properties is an important skill you'll need in this course. Try to solve this exercise without any help at all.

We say that a list "stutters" if it repeats the same element consecutively. The property "*Nostutter* `mylist`" means that `mylist` does not stutter. Formulate an inductive definition for *Nostutter*. (This is different from the *NoDup* property in the exercise below; the sequence `[1,4,1]` repeats but does not stutter.)

```
data Nostutter : List t -> Type where
  -- FILL IN HERE
  RemoveMe : Nostutter [] -- needed for typechecking, data shouldn't be empty
```

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.)

```
test_nostutter_1 : Nostutter [3,1,4,1,5,6]
test_nostutter_1 = ?test_nostutter_1_rhs

(*
  Proof. repeat constructor; apply beq_nat_false_iff; auto.
  Qed.
*)

test_nostutter_2 : Nostutter []
test_nostutter_2 = ?test_nostutter_2_rhs

(*
  Proof. repeat constructor; apply beq_nat_false_iff; auto.
  Qed.
*)

test_nostutter_3 : Nostutter [5]
test_nostutter_3 = ?test_nostutter_3_rhs

(*
  Proof. repeat constructor; apply beq_nat_false; auto. Qed.
*)

test_nostutter_4 : Not (Nostutter [3,1,1,4])
test_nostutter_4 = ?test_nostutter_4_rhs

(*
  Proof. intro.
  repeat match goal with
    h: nostutter _ |- _ => inversion h; clear h; subst
  end.
  contradiction H1; auto. Qed.
*)
```

□

6.0.2. *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 `t`, a function `test : t->Bool`, and a list `l` of type `List t`. Suppose further that `l` is an in-order merge of two lists, `l1` and `l2`, such that every item in `l1` satisfies `test` and no item in `l2` satisfies `test`. Then `filter test l = l1`.

Translate this specification into a Idris theorem and prove it. (You'll need to begin by defining what it means for one list to be a merge of two others. Do this with an inductive `data` type, not a function.)

```
-- FILL IN HERE
```

□

6.0.3. *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.

```
-- FILL IN HERE
```

□

6.0.4. *Exercise: 4 stars, optional (palindromes).* A palindrome is a sequence that reads the same backwards as forwards.

- Define an inductive proposition `Pal` on `List t` that captures what it means to be a palindrome. (Hint: You'll need three cases. Your definition should be based on the structure of the list; just having a single constructor like

```
C : (l : List t) -> l = rev l -> Pal l
```

may seem obvious, but will not work very well.)

- Prove (`pal_app_rev`) that

```
(l : List t) -> Pal (l ++ rev l)
```

- Prove (`pal_rev`) that

```
(l : List t) -> Pal l -> l = rev l
```

```
-- FILL IN HERE
```

□

6.0.5. *Exercise: 5 stars, optional (palindrome_converse).* Again, the converse direction is significantly more difficult, due to the lack of evidence. Using your definition of `Pal` from the previous exercise, prove that

```
(l : List t) -> l = rev l -> Pal l
```

```
-- FILL IN HERE
```

□

6.1. 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:

```
In : (x : t) → (l : List t) → Type
In x [] = Void
In x (x' :: xs) = (x' = x) `Either` In x xs
```

Your first task is to use *In* to define a proposition *Disjoint* {t} l1 l2, which should be provable exactly when l1 and l2 are lists (with elements of type t) that have no elements in common.

```
-- FILL IN HERE
```

Next, use *In* to define an inductive proposition *NoDup* {t} l, which should be provable exactly when l is a list (with elements of type t) where every member is different from every other. For example, *NoDup* {t=*Nat*} [1,2,3,4] and *NoDup* {t=*Bool*} [] should be provable, while *NoDup* {t=*Nat*} [1,2,1] and *NoDup* {t=*Bool*} [*True*,True] should not be.

```
-- FILL IN HERE
```

Finally, state and prove one or more interesting theorems relating *Disjoint*, *NoDup* and (++) (list append).

```
-- FILL IN HERE
```

□

6.1.1. *Exercise: 4 stars, advanced, optional (pigeonhole principle).* The *pigeonhole principle* states a basic fact about counting: if we distribute more than n items into n pigeonholes, some pigeonhole must contain at least two items. As often happens, this apparently trivial fact about numbers requires non-trivial machinery to prove, but we now have enough…

First prove an easy useful lemma.

```
in_split : In x l → (l1 ** l2 ** l = l1 ++ x :: l2)
in_split prf = ?in_split_rhs
```

Now define a property *Repeats* such that *Repeats* {t} l asserts that l contains at least one repeated element (of type t).

```
data Repeats : List t → Type where
  -- FILL IN HERE
  RemoveMe' : Repeats [] -- needed for typechecking, data shouldn't be empty
```

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., `(In x l) `Either` (Not (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.

```
pigeonhole_principle : ((x : t) → In x l1 → In x l2) →
                       ((length l2) <' (length l1)) →
                       Repeats l1
pigeonhole_principle f prf = ?pigeonhole_principle_rhs
  where
    excluded_middle : (p : Type) → p `Either` (Not p)
    excluded_middle p = really_believe_me p
```

□

CHAPTER 8

Maps: Total and Partial Maps

```
module Maps

import Logic
import IndProp

%access public export
```

Maps (or dictionaries) are ubiquitous data structures both generally and in the theory of programming languages in particular; we're going to need them in many places in the coming chapters. They also make a nice case study using ideas we've seen in previous chapters, including building data structures out of higher-order functions (from `Basics` and `Poly`) and the use of reflection to streamline proofs (from `IndProp`).

We'll define two flavors of maps: *total* maps, which include a "default" element to be returned when a key being looked up doesn't exist, and *partial* maps, which return a *`Maybe`* to indicate success or failure. The latter is defined in terms of the former, using *`Nothing`* as the default element.

1. The Idris Standard Library

Edit

One small digression before we get to maps.

Unlike the chapters we have seen so far, this one does not `Require Import` the chapter before it (and, transitively, all the earlier chapters). Instead, in this chapter and from now, on we're going to import the definitions and theorems we need directly from Idris's standard library stuff. You should not notice much difference, though, because we've been careful to name our own definitions and theorems the same as their counterparts in the standard library, wherever they overlap.

```
Require Import Idris.Arith.Arith.
Require Import Idris.Bool.Bool.
Require Import Idris.Strings.String.
Require Import Idris.Logic.FunctionalExtensionality.
```

Documentation for the standard library can be found at https://www.idris-lang.org/docs/current/.

The :search command is a good way to look for theorems involving objects of specific types. Take a minute now to experiment with it.

2. Identifiers

First, we need a type for the keys that we use to index into our maps. For this purpose, we again use the type *Id* from the Lists chapter. To make this chapter self contained, we repeat its definition here, together with the equality comparison function for *Id* and its fundamental property.

```
data Id : Type where
  MkId : String -> Id

beq_id : (x1, x2 : Id) -> Bool
beq_id (MkId n1) (MkId n2) = decAsBool $ decEq n1 n2
```

Edit

(The function decEq comes from Idris's string library. If you check its result type, you'll see that it does not actually return a *Bool*, but rather a type that looks like *Either* (x = y) (*Not* (x = y)), called a {Dec}, which can be thought of as an "evidence-carrying boolean." Formally, an element of *Dec* (x=y) is either a proof that two things are equal or a proof that they are unequal, together with a tag indicating which. But for present purposes you can think of it as just a fancy *Bool*.)

```
beq_id_refl : (x : Id) -> True = beq_id x x
beq_id_refl (MkId n) with (decEq n n)
  beq_id_refl _ | Yes _     = Refl
  beq_id_refl _ | No contra = absurd $ contra Refl
```

The following useful property of beq_id follows from an analogous lemma about strings:

```
beq_id_true_iff : (beq_id x y = True) <-> x = y
beq_id_true_iff = (bto, bfro)
  where
    bto : (beq_id x y = True) -> x = y
    bto {x=MkId n1} {y=MkId n2} prf with (decEq n1 n2)
      bto Refl | Yes eq = cong {f=MkId} eq
      bto Refl | No _ impossible

    idInj : MkId x = MkId y -> x = y
    idInj Refl = Refl

    bfro : (x = y) -> beq_id x y = True
    bfro {x=MkId n1} {y=MkId n2} prf with (decEq n1 n2)
      bfro _   | Yes _     = Refl
      bfro prf | No contra = absurd $ contra $ idInj prf
```

Similarly:

```
beq_id_false_iff : (beq_id x y = False) <-> Not (x = y)
beq_id_false_iff = (to, fro)
  where
    to : (beq_id x y = False) -> Not (x = y)
    to beqf = (snd not_true_iff_false) beqf . (snd beq_id_true_iff)

    fro : (Not (x = y)) -> beq_id x y = False
    fro noteq = (fst not_true_iff_false) $ noteq . (fst beq_id_true_iff)
```

3. Total Maps

Our main job in this chapter will be to build a definition of partial maps that is similar in behavior to the one we saw in the `Lists` chapter, plus accompanying lemmas about its behavior.

This time around, though, we're going to use *functions*, rather than lists of key-value pairs, to build maps. The advantage of this representation is that it offers a more *extensional* view of maps, where two maps that respond to queries in the same way will be represented as literally the same thing (the very same function), rather than just "equivalent" data structures. This, in turn, simplifies proofs that use maps.

We build partial maps in two steps. First, we define a type of *total maps* that return a default value when we look up a key that is not present in the map.

```
TotalMap : Type -> Type
TotalMap a = Id -> a
```

Intuitively, a total map over an element type `a` is just a function that can be used to look up `Id`s, yielding `a`s.

The function `t_empty` yields an empty total map, given a default element; this map always returns the default element when applied to any id.

```
t_empty : (v : a) -> TotalMap a
t_empty v = \_ => v
```

We can also write this as:

```
t_empty = const
```

More interesting is the `update` function, which (as before) takes a map `m`, a key `x`, and a value `v` and returns a new map that takes `x` to `v` and takes every other key to whatever `m` does.

```
t_update : (x : Id) -> (v : a) -> (m : TotalMap a) -> TotalMap a
t_update x v m = \x' => if beq_id 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 `\x' => ...` that behaves like the desired map.

For example, we can build a map taking `Id`s to `Bool`s, where `Id 3` is mapped to `True` and every other key is mapped to `False`, like this:

Seems like a wrong description in the book here

```
examplemap : TotalMap Bool
examplemap = t_update (MkId "foo") False $
             t_update (MkId "bar") True $
             t_empty False
```

This completes the definition of total maps. Note that we don't need to define a `find` operation because it is just function application!

```
update_example1 : examplemap (MkId "baz") = False
update_example1 = Refl

update_example2 : examplemap (MkId "foo") = False
update_example2 = Refl

update_example3 : examplemap (MkId "quux") = False
update_example3 = Refl

update_example4 : examplemap (MkId "bar") = True
update_example4 = Refl
```

To use maps in later chapters, we'll need several fundamental facts about how they behave. Even if you don't work the following exercises, make sure you thoroughly understand the statements of the lemmas! (Some of the proofs require the functional extensionality axiom, which is discussed in the `Logic` chapter.)

3.0.1. *Exercise: 1 star, optional (t_apply_empty).* First, the empty map returns its default element for all keys:

```
t_apply_empty : t_empty v x = v
t_apply_empty = ?t_apply_empty_rhs
```

□

3.0.2. *Exercise: 2 stars, 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`:

```
t_update_eq : (t_update x v m) x = v
t_update_eq = ?t_update_eq_rhs
```

□

3.0.3. *Exercise: 2 stars, optional (t_update_neq).* On the other hand, if we update a map `m` at a key `x1` and then look up a *different* key `x2` in the resulting map, we get the same result that `m` would have given:

```
t_update_neq : Not (x1 = x2) -> (t_update x1 v m) x2 = m x2
t_update_neq neq = ?t_update_neq_rhs
```

□

3.0.4. *Exercise: 2 stars, optional (t_update_shadow).* If we update a map `m` at a key `x` with a value `v1` and then update again with the same key `x` and another value `v2`, the resulting map behaves the same (gives the same result when applied to any key) as the simpler map obtained by performing just the second `update` on `m`:

```
t_update_shadow : t_update x v2 $ t_update x v1 m = t_update x v2 m
t_update_shadow = ?t_update_shadow_rhs
```

□

For the final two lemmas about total maps, it's convenient to use the reflection idioms introduced in chapter `IndProp`. We begin by proving a fundamental *reflection lemma* relating the equality proposition on *Id*s with the boolean function `beq_id`.

3.0.5. *Exercise: 2 stars, optional (beq_idP).* Use the proof of `beq_natP` in chapter `IndProp` as a template to prove the following:

```
beq_idP : {x1, x2 : Id} → Reflect (x = y) (beq_id x y)
beq_idP = ?beq_idP_rhs
```

□

Now, given *Id*s `x1` and `x2`, we can use `with (beq_idP x1 x2)` to simultaneously perform case analysis on the result of `beq_id x1 x2` and generate hypotheses about the equality (in the sense of `=`) of `x1` and `x2`.

3.0.6. *Exercise: 2 stars (t_update_same).* With the example in chapter `IndProp` as a template, use `beq_idP` to prove the following theorem, which states that if we update a map to assign key `x` the same value as it already has in `m`, then the result is equal to `m`:

```
t_update_same : t_update x (m x) m = m
t_update_same = ?t_update_same_rhs
```

□

3.0.7. *Exercise: 3 stars, recommended (t_update_permute).* Use `beq_idP` to prove one final property of the `update` function: If we update a map `m` at two distinct keys, it doesn't matter in which order we do the updates.

```
t_update_permute : Not (x2 = x1) → t_update x1 v1 $ t_update x2 v2 m
                                 = t_update x2 v2 $ t_update x1 v1 m
t_update_permute neq = ?t_update_permute_rhs
```

□

4. Partial maps

Finally, we define *partial maps* on top of total maps. A partial map with elements of type `a` is simply a total map with elements of type *Maybe* `a` and default element *Nothing*.

```
PartialMap : Type → Type
PartialMap a = TotalMap (Maybe a)

empty : PartialMap a
empty = t_empty Nothing

update : (x : Id) → (v : a) → (m : PartialMap a) → PartialMap a
update x v m = t_update x (Just v) m
```

We now straightforwardly lift all of the basic lemmas about total maps to partial maps.

```
apply_empty : empty {a} x = Nothing {a}
apply_empty = Refl

update_eq : (update x v m) x = Just v
update_eq {v} = t_update_eq {v=Just v}

update_neq : Not (x2 = x1) → (update x2 v m) x1 = m x1
update_neq {x1} {x2} {v} = t_update_neq {x1=x2} {x2=x1} {v=Just v}

update_shadow : update x v2 $ update x v1 m = update x v2 m
update_shadow {v1} {v2} = t_update_shadow {v1=Just v1} {v2=Just v2}

update_same : m x = Just v → update x v m = m
update_same prf = rewrite sym prf in t_update_same

update_permute : Not (x2 = x1) → update x1 v1 $ update x2 v2 m
                               = update x2 v2 $ update x1 v1 m
update_permute {v1} {v2} = t_update_permute {v1=Just v1} {v2=Just v2}
```

CHAPTER 9

ProofObjects : The Curry-Howard Correspondence

```
module ProofObjects
```

"*Algorithms are the computational content of proofs.*"

– Robert Harper

```
import Logic
import IndProp
```

We have seen that Idris 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 Idris's programming and proving facilities are closely related. For example, the keyword `data` 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 Idris are almost the same thing. In this chapter we will study how this works.

We have already seen the fundamental idea: provability in Idris is represented by concrete *evidence*. When we construct the proof of a basic proposition, we are actually building a tree of evidence, which can be thought of as a data structure.

If the proposition is an implication like `A → B`, then its proof will be an evidence *transformer*: a recipe for converting evidence for `A` into evidence for `B`. So at a fundamental level, proofs are simply programs that manipulate evidence.

Question: If evidence is data, what are propositions themselves?

Answer: They are types!

Look again at the formal definition of the `Ev` property.

```
data Ev : Nat → Type where
  Ev_0 : Ev Z
  Ev_SS : {n : Nat} → Ev n → Ev (S (S n))
```

Suppose we introduce an alternative pronunciation of "`:`". Instead of "has type," we can say "is a proof of." For example, the second line in the definition of `Ev`

declares that `Ev_0 : Ev 0`. Instead of "*Ev_0* has type *Ev* 0," we can say that "*Ev_0* is a proof of *Ev* 0."

This pun between types and propositions — between `:` as "has type" and `:` as "is a proof of" or "is evidence for" — is called the Curry-Howard correspondence. It proposes a deep connection between the world of logic and the world of computation:

```
propositions  ~  types
proofs        ~  data values
```

Add http://dl.acm.org/citation.cfm?id=2699407 as a link

See [Wadler 2015] for a brief history and an up-to-date exposition.

Many useful insights follow from this connection. To begin with, it gives us a natural interpretation of the type of the *Ev_SS* constructor:

```
ΛΠ> :t Ev_SS
Ev_SS : Ev n → Ev (S (S n))
```

This can be read "*Ev_SS* is a constructor that takes two arguments — a number `n` and evidence for the proposition *Ev* `n` — and yields evidence for the proposition *Ev* (*S* (*S* `n`))."

Now let's look again at a previous proof involving *Ev*.

```
ev_4 : Ev 4
ev_4 = Ev_SS {n=2} $ Ev_SS {n=0} Ev_0
```

As with ordinary data values and functions, we can use the `:printdef` command to see the proof object that results from this proof script.

```
ΛΠ> :printdef ev_4
ev_4 : Ev 4
ev_4 = Ev_SS (Ev_SS Ev_0)
```

As a matter of fact, we can also write down this proof object directly, without the need for a separate proof script:

```
ΛΠ> Ev_SS $ Ev_SS Ev_0
Ev_SS (Ev_SS Ev_0) : Ev 4
```

The expression *Ev_SS* {`n=2`} $ *Ev_SS* {`n=0`} *Ev_0* can be thought of as instantiating the parameterized constructor *Ev_SS* with the specific arguments `2` and `0` plus the corresponding proof objects for its premises *Ev* `2` and *Ev* `0`. Alternatively, we can think of *Ev_SS* as a primitive "evidence constructor" that, when applied to a particular number, wants to be further applied to evidence that that number is even; its type,

```
{n : Nat} → Ev n → Ev (S (S n))
```

expresses this functionality, in the same way that the polymorphic type `{x : Type} → 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.

Edit or remove

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.
```

We can now see that this feature is a trivial consequence of the status the Idris grants to proofs and propositions: Lemmas and hypotheses can be combined in expressions (i.e., proof objects) according to the same basic rules used for programs in the language.

1. Proof Scripts

Rewrite, keep explanation about holes? Seems a bit late for that

The *proof objects* we've been discussing lie at the core of how Idris operates. When Idris 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 expression on the right hand side of `=` 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, Idris 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 stored in the global context under the name given in the type definition.

Tactic proofs are useful and convenient, but they are not essential: in principle, we can always construct the required evidence by hand, as shown above. Then we can use `Definition` (rather than `Theorem`) to give a global name directly to a piece of evidence.

```
Definition ev_4''' : ev 4 :=
  ev_SS 2 (ev_SS 0 ev_0).
```

All these different ways of building the proof lead to exactly the same evidence being saved in the global environment.

```
Print ev_4.
(* ⟹ ev_4    =   ev_SS 2 (ev_SS 0 ev_0) : ev 4 *)
Print ev_4'.
(* ⟹ ev_4'   =   ev_SS 2 (ev_SS 0 ev_0) : ev 4 *)
Print ev_4''.
(* ⟹ ev_4''  =   ev_SS 2 (ev_SS 0 ev_0) : ev 4 *)
Print ev_4'''.
(* ⟹ ev_4''' =   ev_SS 2 (ev_SS 0 ev_0) : ev 4 *)
```

1.0.1. *Exercise: 1 star (eight_is_even).*

Remove?

Give a tactic proof and a proof object showing that `Ev 8`.

```
ev_8 : Ev 8
ev_8 = ?ev_8_rhs
```

□

1.0.2. *Quantifiers, Implications, Functions.*

Edit the section

In Idris's computational universe (where data structures and programs live), there are two sorts of values with arrows in their types: *constructors* introduced by `data` definitions, and *functions*.

Similarly, in Idris's logical universe (where we carry out proofs), there are two ways of giving evidence for an implication: constructors introduced by `data`-defined propositions, and… functions!

For example, consider this statement:

```
ev_plus4 : Ev n → Ev (4 + n)
ev_plus4 x = Ev_SS $ Ev_SS x
```

What is the proof object corresponding to `ev_plus4`?

We're looking for an expression whose type is `{n: Nat} → 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' : forall n, ev n → ev (4 + n) :=
  fun (n : Nat) ⇒ fun (H : ev n) ⇒
    ev_SS (S (S n)) (ev_SS n H).
```

Recall that `\n ⇒ blah` means "the function that, given `n`, yields `blah`," and that Idris 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''.
(* ⟹ ev_plus4'' : forall n : Nat, ev n → ev (4 + n) *)
```

When we view the proposition being proved by `ev_plus4` as a function type, one aspect of it may seem a little unusual. The second argument's type, `Ev n`, mentions the *value* of the first argument, `n`. While such *dependent types* are not found in conventional programming languages, they can be useful in programming too, as the recent flurry of activity in the functional programming community demonstrates.

Reword?

Notice that both implication (`→`) and quantification (`(x : t) → f x`) correspond to functions on evidence. In fact, they are really the same thing: `→` is just a shorthand for a degenerate use of quantification where there is no dependency, i.e., no need to give a name to the type on the left-hand side of the arrow.

For example, consider this proposition:

```
ev_plus2 : Type
ev_plus2 = (n : Nat) → (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:

```
ev_plus2' : Type
ev_plus2' = (n : Nat) → Ev n → Ev (n + 2)
```

In general, "`p → q`" is just syntactic sugar for "`(_ : p) → q`".

2. Programming with Tactics

Edit and move to an appendix about ElabReflection/Pruviloj?

If we can build proofs by giving explicit terms rather than executing tactic scripts, you may be wondering whether we can build *programs* using *tactics* rather than explicit terms. Naturally, the answer is yes!

```
Definition add1 : Nat → Nat.
intro n.
Show Proof.
apply S.
Show Proof.
apply n. Defined.

Print add1.
(* ⟹
    add1 = fun n : Nat ⇒ S n
         : Nat → Nat
*)

Compute add1 2.
(* ⟹ 3 : Nat *)
```

Notice that we terminate the `Definition` with a `.` rather than with `:=` followed by a term. This tells Idris 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 Idris.

3. Logical Connectives as Inductive Types

Inductive definitions are powerful enough to express most of the connectives and quantifiers we have seen so far. Indeed, only universal quantification (and thus implication) is built into Idris; all the others are defined inductively. We'll see these definitions in this section.

3.1. Conjunction.

Edit

To prove that `(p,q)` holds, we must present evidence for both `p` and `q`. Thus, it makes sense to define a proof object for `(p,q)` as consisting of a pair of two proofs: one for `p` and another one for `q`. This leads to the following definition.

```
data And : (p, q : Type) → Type where
  Conj : p → q → And p q
```

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.

```
data Prod : (x, y : Type) → Type where
  PPair : x → y → Prod x y
```

This should clarify why pattern matching can be used on a conjunctive hypothesis. Case analysis allows us to consider all possible ways in which (p,q) was proved — here just one (the *Conj* constructor). Similarly, the split tactic actually works for any inductively defined proposition with only one constructor. In particular, it works for *And*:

```
and_comm : (And p q) <-> (And q p)
and_comm = (\(Conj x y) => Conj y x,
            \(Conj y x) => Conj x y)
```

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:

```
and_comm'_aux : And p q → And q p
and_comm'_aux (Conj x y) = Conj y x

and_comm' : (And p q) <-> (And q p)
and_comm' {p} {q} = (and_comm'_aux {p} {q}, and_comm'_aux {p=q} {q=p})
```

3.1.1. *Exercise: 2 stars, optional (conj_fact).* Construct a proof object demonstrating the following proposition.

```
conj_fact : And p q → And q r → And p r
conj_fact pq qr = ?conj_fact_rhs
```

□

3.2. Disjunction. The inductive definition of disjunction uses two constructors, one for each side of the disjunct:

```
data Or : (p, q : Type) → Type where
  IntroL : p → Or p q
  IntroR : q → Or p q
```

This declaration explains the behavior of pattern matching on a disjunctive hypothesis, since the generated subgoals match the shape of the *IntroL* and *IntroR* constructors.

Once again, we can also directly write proof objects for theorems involving *Or*, without resorting to tactics.

3.2.1. *Exercise: 2 stars, optional (or_comm).*

Edit

Try to write down an explicit proof object for `or_comm` (without using `Print` to peek at the ones we already defined!).

```
or_comm : Or p q -> Or q p
or_comm pq = ?or_comm_rhs
```

□

3.3. Existential Quantification. To give evidence for an existential quantifier, we package a witness `x` together with a proof that `x` satisfies the property `p`:

```
data Ex : (p : a -> Type) -> Type where
  ExIntro : (x : a) -> p x -> Ex p
```

This may benefit from a little unpacking. The core definition is for a type former *Ex* that can be used to build propositions of the form *Ex* `p`, where `p` itself is a function from witness values in the type `a` to propositions. The *ExIntro* constructor then offers a way of constructing evidence for *Ex* `p`, given a witness `x` and a proof of `p x`.

The more familiar form `(x ** p x)` desugars to an expression involving *Ex*:

Edit

```
Check ex (fun n ⇒ ev n).
(* ===> exists n : Nat, ev n
        : Prop *)
```

Here's how to define an explicit proof object involving *Ex*:

```
some_nat_is_even : Ex (\n ⇒ Ev n)
some_nat_is_even = ExIntro 4 (Ev_SS $ Ev_SS Ev_0)
```

3.3.1. *Exercise: 2 stars, optional (ex_ev_Sn).* Complete the definition of the following proof object:

```
ex_ev_Sn : Ex (\n ⇒ Ev (S n))
ex_ev_Sn = ?ex_ev_Sn_rhs
```

□

3.4. *Unit* and *Void*. The inductive definition of the *Unit* proposition is simple:

```
data Unit : Type where
  () : Unit
```

It has one constructor (so every proof of *Unit* is the same, so being given a proof of*Unit* is not informative.)

Void is equally simple — indeed, so simple it may look syntactically wrong at first glance!

Edit, this actually is wrong, stdlib uses `runElab` to define it

```
data Void : Type where
```

That is, *Void* is an inductive type with *no* constructors — i.e., no way to build evidence for it.

4. Equality

Edit, it actually is built in

Even Idris's equality relation is not built in. It has the following inductive definition. (Actually, the definition in the standard library is a small variant of this, which gives an induction principle that is slightly easier to use.)

```
data PropEq : {t : Type} → t → t → Type where
  EqRefl : PropEq x x

syntax [x] "='" [y] = PropEq x y
```

The way to think about this definition is that, given a set `t`, it defines a *family* of propositions "`x` is equal to `y`," indexed by pairs of values (`x` and `y`) from `t`. There is just one way of constructing evidence for each member of this family: applying the constructor *EqRefl* to a type `t` and a value `x : t` yields evidence that `x` is equal to `x`.

4.0.1. *Exercise: 2 stars (leibniz_equality).* The inductive definition of equality corresponds to *Leibniz equality*: what we mean when we say "`x` and `y` are equal" is that every property `p` that is true of `x` is also true of `y`.

```
leibniz_equality : (x =' y) → ((p : t → Type) → p x → p y)
leibniz_equality eq p px = ?leibniz_equality_rhs
```

□

Edit

We can use *EqRefl* 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 Idris treats as "the same" any two terms that are *convertible* according to a simple set of computation rules. These rules, which are similar to those used by `Compute`, include evaluation of function application, inlining of definitions, and simplification of `matches`.

```
four : (2 + 2) =' (1 + 3)
four = EqRefl
```

The *Refl* that we have used to prove equalities up to now is essentially just an application of an equality constructor.

Edit

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.

singleton : ([]++[x]) =' (x::[])
singleton = EqRefl

quiz6 : Ex (\x ⇒ (x + 3) =' 4)
quiz6 = ExIntro 1 EqRefl
```

4.1. Inversion, Again.

Edit/remove

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 = Z`), 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 (`Or p 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 (`And p 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 `PropEq`, there is again only one constructor, so only one subgoal gets generated. Now, though, the form of the `EqRefl` constructor does give us some extra information: it tells us that the two arguments to `PropEq` must be the same! The `inversion` tactic adds this fact to the context.

CHAPTER 10

Rel : Properties of Relations

```
module Rel
```

Add hyperlinks

This short (and optional) chapter develops some basic definitions and a few theorems about binary relations in Idris. The key definitions are repeated where they are actually used (in the `Smallstep` chapter), so readers who are already comfortable with these ideas can safely skim or skip this chapter. However, relations are also a good source of exercises for developing facility with Idris's basic reasoning facilities, so it may be useful to look at this material just after the `IndProp` chapter.

```
import Logic
import IndProp
```

A binary *relation* on a set `t` is a family of propositions parameterized by two elements of `t` — i.e., a proposition about pairs of elements of `t`.

```
Relation : Type → Type
Relation t = t → t → Type
```

Edit, there's n-relation *Data.Rel* in `contrib`, but no *Relation*

Confusingly, the Idris 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 Idris identifier `relation` will always refer to a binary relation between some set and itself, whereas the English word "relation" can refer either to the specific Idris 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.

There's a similar concept called *LTE* in *Prelude.Nat*, but it's defined by induction from zero

An example relation on *Nat* is *Le*, the less-than-or-equal-to relation, which we usually write `n1 ≤ n2`.

```
ΛΠ> the (Relation Nat) Le
Le : Nat → Nat → Type
```

Edit to show it (probably) doesn't matter in Idris

(Why did we write it this way instead of starting with `data Le : Relation Nat ...`? Because we wanted to put the first `Nat` to the left of the `:`, which makes Idris generate a somewhat nicer induction principle for reasoning about `≤'`.)

1. 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…

1.1. Partial Functions. A relation `r` on a set `t` 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`.

```
Partial_function : (r : Relation t) → Type
Partial_function {t} r = (x, y1, y2 : t) → r x y1 → r x y2 → y1 = y2
```

"Earlier" = in `IndProp`, add hyperlink?

For example, the `Next_nat` relation defined earlier is a partial function.

```
ΛΠ> the (Relation Nat) Next_nat
Next_nat : Nat → Nat → Type
```

```
next_nat_partial_function : Partial_function Next_nat
next_nat_partial_function x (S x) (S x) Nn Nn = Refl
```

However, the `≤'` relation on numbers is not a partial function. (Assume, for a contradiction, that `≤'` is a partial function. But then, since `0 ≤' 0` and `0 ≤' 1`, it follows that `0 = 1`. This is nonsense, so our assumption was contradictory.)

```
le_not_a_partial_function : Not (Partial_function Le)
le_not_a_partial_function f = absurd $ f 0 0 1 Le_n (Le_S Le_n)
```

1.1.1. *Exercise: 2 stars, optional.*

Again, "earlier" = `IndProp`

Show that the `Total_relation` defined in earlier is not a partial function.

```
-- FILL IN HERE
```

□

1.1.2. *Exercise: 2 stars, optional.* Show that the `Empty_relation` that we defined earlier is a partial function.

```
--FILL IN HERE
```

□

1.2. Reflexive Relations. A *reflexive* relation on a set `t` is one for which every element of `t` is related to itself.

```
Reflexive : (r : Relation t) → Type
Reflexive {t} r = (a : t) → r a a

le_reflexive : Reflexive Le
le_reflexive n = Le_n {n}
```

1.3. Transitive Relations. A relation `r` is *transitive* if `r a c` holds whenever `r a b` and `r b c` do.

```
Transitive : (r : Relation t) → Type
Transitive {t} r = (a, b, c : t) → r a b → r b c → r a c

le_trans : Transitive Le
le_trans _ _ _ lab Le_n = lab
le_trans a b (S c) lab (Le_S lbc) = Le_S $ le_trans a b c lab lbc

lt_trans : Transitive Lt
lt_trans a b c lab lbc = le_trans (S a) (S b) c (Le_S lab) lbc
```

1.3.1. *Exercise: 2 stars, optional.* We can also prove `lt_trans` more laboriously by induction, without using `le_trans`. Do this.

```
lt_trans' : Transitive Lt
-- Prove this by induction on evidence that a is less than c.
lt_trans' a b c lab lbc = ?lt_trans__rhs
```

□

1.3.2. *Exercise: 2 stars, optional.*

Not sure how is this different from `lt_trans'`?

Prove the same thing again by induction on `c`.

```
lt_trans'' : Transitive Lt
lt_trans'' a b c lab lbc = ?lt_trans___rhs
```

□

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)...

```
le_Sn_le : ((S n) ≤' m) → (n ≤' m)
le_Sn_le {n} {m} = le_trans n (S n) m (Le_S Le_n)
```

1.3.3. *Exercise: 1 star, optional.*

```
le_S_n : ((S n) ≤' (S m)) → (n ≤' m)
le_S_n less = ?le_S_n_rhs
```

□

1.3.4. *Exercise: 2 stars, optional (le_Sn_n_inf).* Provide an informal proof of the following theorem:

Theorem: For every n, `Not ((S n) ≤' n)`

A formal proof of this is an optional exercise below, but try writing an informal proof without doing the formal proof first.

Proof:

```
-- FILL IN HERE
```

□

1.3.5. *Exercise: 1 star, optional.*

```
le_Sn_n : Not ((S n) ≤' n)
le_Sn_n = ?le_Sn_n_rhs
```

□

Reflexivity and transitivity are the main concepts we'll need for later chapters, but, for a bit of additional practice working with relations in Idris, let's look at a few other common ones…

1.4. Symmetric and Antisymmetric Relations. A relation r is *symmetric* if r a b implies r b a.

```
Symmetric : (r : Relation t) → Type
Symmetric {t} r = (a, b : t) → r a b → r b a
```

1.4.1. *Exercise: 2 stars, optional.*

```
le_not_symmetric : Not (Symmetric Le)
le_not_symmetric = ?le_not_symmetric_rhs
```

□

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.

```
Antisymmetric : (r : Relation t) → Type
Antisymmetric {t} r = (a, b : t) → r a b → r b a → a = b
```

1.4.2. *Exercise: 2 stars, optional.*

```
le_antisymmetric : Antisymmetric Le
le_antisymmetric = ?le_antisymmetric_rhs
```

□

1.4.3. *Exercise: 2 stars, optional.*

```
le_step : (n <' m) → (m ≤' (S p)) → (n ≤' p)
le_step ltnm lemsp = ?le_step_rhs
```

□

1.5. Equivalence Relations. A relation is an *equivalence* if it's reflexive, symmetric, and transitive.

```
Equivalence : (r : Relation t) → Type
Equivalence r = (Reflexive r, Symmetric r, Transitive r)
```

1.6. Partial Orders and Preorders.

Edit

A relation is a *partial order* when it's reflexive, *anti*-symmetric, and transitive. In the Idris standard library it's called just "order" for short.

```
Order : (r : Relation t) → Type
Order r = (Reflexive r, Antisymmetric r, Transitive r)
```

A preorder is almost like a partial order, but doesn't have to be antisymmetric.

```
Preorder : (r : Relation t) → Type
Preorder r = (Reflexive r, Transitive r)
```

```
le_order : Order Le
le_order = (le_reflexive, le_antisymmetric, le_trans)
```

2. Reflexive, Transitive Closure

Edit

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 Idris standard library:

```
data Clos_refl_trans : (r : Relation t) → Relation t where
  Rt_step : r x y → Clos_refl_trans r x y
  Rt_refl : Clos_refl_trans r x x
  Rt_trans : Clos_refl_trans r x y → Clos_refl_trans r y z →
             Clos_refl_trans r x z
```

For example, the reflexive and transitive closure of the `Next_nat` relation coincides with the `Le` relation.

```
next_nat_closure_is_le : (n ≤' m) ↔ (Clos_refl_trans Next_nat n m)
next_nat_closure_is_le = (to, fro)
  where
    to : Le n m → Clos_refl_trans Next_nat n m
    to Le_n = Rt_refl
    to (Le_S {m} le) = Rt_trans {y=m} (to le) (Rt_step Nn)
    fro : Clos_refl_trans Next_nat n m → Le n m
    fro (Rt_step Nn) = Le_S Le_n
    fro Rt_refl = Le_n
```

```
    fro (Rt_trans {x=n} {y} {z=m} ny ym) =
      le_trans n y m (fro ny) (fro ym)
```

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:

```
data Clos_refl_trans_1n : (r : Relation t) → (x : t) → t → Type where
  Rt1n_refl : Clos_refl_trans_1n r x x
  Rt1n_trans : r x y → Clos_refl_trans_1n r y z → Clos_refl_trans_1n r x z
```

Edit

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.

```
rsc_R : r x y → Clos_refl_trans_1n r x y
rsc_R rxy = Rt1n_trans rxy Rt1n_refl
```

2.0.1. *Exercise: 2 stars, optional (rsc_trans).*

```
rsc_trans : Clos_refl_trans_1n r x y → Clos_refl_trans_1n r y z →
            Clos_refl_trans_1n r x z
rsc_trans crxy cryz = ?rsc_trans_rhs
```

□

Then we use these facts to prove that the two definitions of reflexive, transitive closure do indeed define the same relation.

2.0.2. *Exercise: 3 stars, optional (rtc_rsc_coincide).*

```
rtc_rsc_coincide : (Clos_refl_trans r x y) ↔ (Clos_refl_trans_1n r x y)
rtc_rsc_coincide = ?rtc_rsc_coincide_rhs
```

□

CHAPTER 11

Imp : Simple Imperative Programs

```
module Imp

import Logic
```

In this chapter, we begin a new direction that will continue for the rest of the course. Up to now most of our attention has been focused on various aspects of Idris itself, while from now on we'll mostly be using Idris to formalize other things. (We'll continue to pause from time to time to introduce a few additional aspects of Idris.)

Our first case study is a *simple imperative programming language* called Imp, embodying a tiny core fragment of conventional mainstream languages such as C and Java. Here is a familiar mathematical function written in Imp.

```
Z ::= X;;
Y ::= 1;;
WHILE not (Z == 0) DO
  Y ::= Y * Z;;
  Z ::= Z - 1
END
```

This chapter looks at how to define the *syntax* and *semantics* of Imp; the chapters that follow develop a theory of *program equivalence* and introduce *Hoare Logic*, a widely used logic for reasoning about imperative programs.

```
import Maps

%default total
%access public export
```

1. Arithmetic and Boolean Expressions

We'll present Imp in three parts: first a core language of *arithmetic and boolean expressions*, then an extension of these expressions with *variables*, and finally a language of *commands* including assignment, conditions, sequencing, and loops.

1.1. Syntax. These two definitions specify the *abstract syntax* of arithmetic and boolean expressions.

```
data AExp0 : Type where
  ANum0 : Nat → AExp0
  APlus0 : AExp0 → AExp0 → AExp0
  AMinus0 : AExp0 → AExp0 → AExp0
  AMult0 : AExp0 → AExp0 → AExp0

data BExp0 : Type where
  BTrue0 : BExp0
  BFalse0 : BExp0
  BEq0 : AExp0 → AExp0 → BExp0
  BLe0 : AExp0 → AExp0 → BExp0
  BNot0 : BExp0 → BExp0
  BAnd0 : BExp0 → BExp0 → BExp0
```

In this chapter, we'll elide the translation from the concrete syntax that a programmer would actually write to these abstract syntax trees — the process that, for example, would translate the string "1+2*3" to the AST

```
APlus0 (ANum0 1) (AMult0 (ANum0 2) (ANum0 3))
```

The optional chapter `ImpParser` develops a simple implementation of a lexical analyzer and parser that can perform this translation. You do *not* need to understand that chapter to understand this one, but if you haven't taken a course where these techniques are covered (e.g., a compilers course) you may want to skim it.

For comparison, here's a conventional BNF (Backus-Naur Form) grammar defining the same abstract syntax:

```
a ::= Nat
    | a + a
    | a - a
    | a * a

b ::= True
    | False
    | a = a
    | a ≤ a
    | not b
    | b and b
```

Compared to the Idris version above…

- The BNF is more informal — for example, it gives some suggestions about the surface syntax of expressions (like the fact that the addition operation is written `+` and is an infix symbol) while leaving other aspects of lexical analysis and parsing (like the relative precedence of `+`, `-`, and `*`, the use of parens to explicitly group subexpressions, etc.) unspecified. Some additional information (and human intelligence) would be required to turn this description into a formal definition, for example when implementing a compiler.

The Idris version consistently omits all this information and concentrates on the abstract syntax only.

- On the other hand, the BNF version is lighter and easier to read. Its informality makes it flexible, a big advantage in situations like discussions at the blackboard, where conveying general ideas is more important than getting every detail nailed down precisely.

  Indeed, there are dozens of BNF-like notations and people switch freely among them, usually without bothering to say which form of BNF they're using because there is no need to: a rough-and-ready informal understanding is all that's important.

It's good to be comfortable with both sorts of notations: informal ones for communicating between humans and formal ones for carrying out implementations and proofs.

1.2. Evaluation. *Evaluating* an arithmetic expression produces a number.

```
aeval0 : (a : AExp0) -> Nat
aeval0 (ANum0 n) = n
aeval0 (APlus0 a1 a2) = (aeval0 a1) + (aeval0 a2)
aeval0 (AMinus0 a1 a2) = (aeval0 a1) `minus` (aeval0 a2)
aeval0 (AMult0 a1 a2) = (aeval0 a1) * (aeval0 a2)

test_aeval1 : aeval0 (APlus0 (ANum0 2) (ANum0 2)) = 4
test_aeval1 = Refl
```

Similarly, evaluating a boolean expression yields a boolean.

```
beval0 : (b : BExp0) -> Bool
beval0 BTrue0 = True
beval0 BFalse0 = False
beval0 (BEq0 a1 a2) = (aeval0 a1) == (aeval0 a2)
beval0 (BLe0 a1 a2) = lte (aeval0 a1) (aeval0 a2)
beval0 (BNot0 b1) = not (beval0 b1)
beval0 (BAnd0 b1 b2) = (beval0 b1) && (beval0 b2)
```

1.3. 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., `(APlus0 (ANum0 0) e)`) into just `e`.

```
optimize_0plus : (a : AExp0) -> AExp0
optimize_0plus (ANum0 n) = ANum0 n
optimize_0plus (APlus0 (ANum0 Z) e2) =
  optimize_0plus e2
optimize_0plus (APlus0 e1 e2) =
  APlus0 (optimize_0plus e1) (optimize_0plus e2)
optimize_0plus (AMinus0 e1 e2) =
  AMinus0 (optimize_0plus e1) (optimize_0plus e2)
```

```
optimize_0plus (AMult0 e1 e2) =
  AMult0 (optimize_0plus e1) (optimize_0plus e2)
```

To make sure our optimization is doing the right thing we can test it on some examples and see if the output looks OK.

```
test_optimize_0plus :
  optimize_0plus (APlus0 (ANum0 2)
                         (APlus0 (ANum0 0)
                                 (APlus0 (ANum0 0) (ANum0 1))))
  = APlus0 (ANum0 2) (ANum0 1)
```

But if we want to be sure the optimization is correct — i.e., that evaluating an optimized expression gives the same result as the original — we should prove it.

```
optimize_0plus_sound : aeval0 (optimize_0plus a) = aeval0 a
optimize_0plus_sound {a=ANum0 _} = Refl
optimize_0plus_sound {a=APlus0 (ANum0 Z) y} =
  optimize_0plus_sound {a=y}
optimize_0plus_sound {a=APlus0 (ANum0 (S k)) y} =
  cong {f=\x=>S(k+x)} $ optimize_0plus_sound {a=y}
optimize_0plus_sound {a=APlus0 (APlus0 x z) y} =
  rewrite optimize_0plus_sound {a=APlus0 x z} in
  rewrite optimize_0plus_sound {a=y} in
  Refl
optimize_0plus_sound {a=APlus0 (AMinus0 x z) y} =
  rewrite optimize_0plus_sound {a=x} in
  rewrite optimize_0plus_sound {a=y} in
  rewrite optimize_0plus_sound {a=z} in
  Refl
optimize_0plus_sound {a=APlus0 (AMult0 x z) y} =
  rewrite optimize_0plus_sound {a=x} in
  rewrite optimize_0plus_sound {a=y} in
  rewrite optimize_0plus_sound {a=z} in
  Refl
optimize_0plus_sound {a=AMinus0 x y} =
  rewrite optimize_0plus_sound {a=x} in
  rewrite optimize_0plus_sound {a=y} in
  Refl
optimize_0plus_sound {a=AMult0 x y} =
  rewrite optimize_0plus_sound {a=x} in
  rewrite optimize_0plus_sound {a=y} in
  Refl
```

2. Coq Automation

Move the whole subsection to *Pruviloj* chapter?

The amount of repetition in this last proof is a little annoying. And if either the language of arithmetic expressions or the optimization being proved sound were significantly more complex, it would start to be a real problem.

So far, we've been doing all our proofs using just a small handful of Coq's tactics and completely ignoring its powerful facilities for constructing parts of proofs automatically. This section introduces some of these facilities, and we will see more over the next several chapters. Getting used to them will take some energy — Coq's automation is a power tool — but it will allow us to scale up our efforts to more complex definitions and more interesting properties without becoming overwhelmed by boring, repetitive, low-level details.

2.1. Tacticals. Tacticals is Coq's term for tactics that take other tactics as arguments — "higher-order tactics," if you will.

2.1.1. *The `try` Tactical.*

Exists in Pruviloj

If `T` is a tactic, then `try T` is a tactic that is just like `T` except that, if `T` fails, `try T` successfully does nothing at all (instead of failing).

```
Theorem silly1 : forall ae, aeval ae = aeval ae.
Proof. try Refl. (* this just does Refl *) Qed.

Theorem silly2 : forall (P : Type), P → P.
Proof.
  intros P HP.
  try Refl. (* just Refl would have failed *)
  apply HP. (* we can still finish the proof in some other way *)
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.

2.1.2. *The `;` Tactical (Simple Form).*

Approximated by `andThen` in Pruviloj

In its most common form, the `;` tactical takes two tactics as arguments. The compound tactic `T;T'` first performs `T` and then performs `T'` on each subgoal generated by `T`.

For example, consider the following trivial lemma:

```
Lemma foo : forall n, lte 0 n = True.
Proof.
  intros.
  destruct n.
    (* Leaves two subgoals, which are discharged identically...  *)
```

```
    - (* n=0 *) simpl. Refl.
    - (* n=Sn' *) simpl. Refl.
Qed.
```

We can simplify this proof using the ; tactical:

```
Lemma foo' : forall n, lte 0 n = True.
Proof.
  intros.
  (* destruct the current goal *)
  destruct n;
  (* then simpl each resulting subgoal *)
  simpl;
  (* and do Refl on each resulting subgoal *)
  Refl.
Qed.
```

Using try and ; together, we can get rid of the repetition in the proof that was bothering us a little while ago.

Mention Alternatives ?

```
Theorem optimize_0plus_sound': forall a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;
    (* Most cases follow directly by the IH... *)
    try (simpl; rewrite IHa1; rewrite IHa2; Refl).
    (* ... but the remaining cases -- ANum and APlus --
       are different: *)
  - (* ANum *) Refl.
  - (* APlus *)
    destruct a1;
      (* Again, most cases follow directly by the IH: *)
      try (simpl; simpl in IHa1; rewrite IHa1;
           rewrite IHa2; Refl).
    (* The interesting case, on which the try...
       does nothing, is when e1 = ANum n. In this
       case, we have to destruct n (to see whether
       the optimization applies) and rewrite with the
       induction hypothesis. *)
    + (* a1 = ANum n *) destruct n;
      simpl; rewrite IHa2; Refl. Qed.
```

Coq experts often use this "...; try..." idiom after a tactic like induction to take care of many similar cases all at once. Naturally, this practice has an analog in informal proofs. For example, here is an informal proof of the optimization theorem that matches the structure of the formal one:

Theorem: For all arithmetic expressions a,

```
aeval (optimize_0plus a) = aeval a.
```

Proof: By induction on a. Most cases follow directly from the IH. The remaining cases are as follows:

- Suppose a = ANum n for some n. We must show

```
aeval (optimize_0plus (ANum n)) = aeval (ANum n).
```

```
This is immediate from the definition of `optimize_0plus`.
```

- Suppose a = APlus a1 a2 for some a1 and a2. We must show

```
aeval (optimize_0plus (APlus a1 a2)) = aeval (APlus a1 a2).
```

```
Consider the possible forms of `a1`. For most of them, `optimize_0plus`
simply calls itself recursively for the subexpressions and rebuilds a new
expression of the same form as `a1`; in these cases, the result follows
directly from the `IH`. The interesting case is when `a1 = ANum n` for some
`n`. If `n = ANum 0`, then
```

```
optimize_0plus (APlus a1 a2) = optimize_0plus a2
```

```
and the `IH` for `a2` is exactly what we need. On the other hand, if `n = S
n'` for some `n'`, then again `optimize_0plus` simply calls itself
recursively, and the result follows from the `IH`. $\square$
```

However, this proof can still be improved: the first case (for a = ANum n) is very trivial — even more trivial than the cases that we said simply followed from the IH — yet we have chosen to write it out in full. It would be better and clearer to drop it and just say, at the top, "Most cases are either immediate or direct from the IH. The only interesting case is the one for APlus..." We can make the same improvement in our formal proof too. Here's how it looks:

```
Theorem optimize_0plus_sound'': forall a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;
    (* Most cases follow directly by the IH *)
    try (simpl; rewrite IHa1; rewrite IHa2; Refl);
    (* ... or are immediate by definition *)
    try Refl.
  (* The interesting case is when a = APlus a1 a2. *)
  - (* APlus *)
    destruct a1; try (simpl; simpl in IHa1; rewrite IHa1;
                      rewrite IHa2; Refl).
    + (* a1 = ANum n *) destruct n;
      simpl; rewrite IHa2; Refl. Qed.
```

2.1.3. *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']
```

2.1.4. *The `repeat` Tactical.*

Approximated by `repeatUntilFail` in Pruviloj

The `repeat` tactical takes another tactic and keeps applying this tactic until it fails. Here is an example showing that `10` is in a long list using `repeat`.

```
Theorem In10 : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (try (left; Refl); 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; Refl).
  repeat (right; try (left; Refl)).
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 forever, since `simpl` always succeeds). While evaluation in Coq's term language, Gallina, is guaranteed to terminate, tactic evaluation is not! This does not affect Coq's logical consistency, however, since the job of `repeat` and other tactics is to guide Coq in constructing proofs; if the construction process diverges, this simply means that we have failed to construct a proof, not that we have constructed a wrong one.

2.1.5. *Exercise: 3 stars (optimize_0plus_b).* 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 which performs that transformation on *BExp*s, and prove it is sound. Use the tacticals we've just seen to make the proof as elegant as possible.

```
optimize_0plus_b : (b : BExp0) → BExp0
optimize_0plus_b b = ?optimize_0plus_b_rhs
```

```
optimize_0plus_b_sound : beval0 (optimize_0plus_b b) = beval0 b
optimize_0plus_b_sound = ?optimize_0plus_b_sound_rhs
```

□

2.1.6. *Exercise: 4 stars, optional (optimizer). Design exercise*: The optimization implemented by our `optimize_0plus` function is only one of many possible optimizations on arithmetic and boolean expressions. Write a more sophisticated optimizer and prove it correct. (You will probably find it easiest to start small — add just a single, simple optimization and prove it correct — and build up to something more interesting incrementially.)

```
--FILL IN HERE
```

□

2.2. Defining New Tactic Notations. Coq also provides several ways of "programming" tactic scripts.

- The `Tactic Notation` idiom illustrated below gives a handy way to define "shorthand tactics" that bundle several tactics into a single command.
- For more sophisticated programming, Coq offers a built-in programming language called `Ltac` with primitives that can examine and modify the proof state. The details are a bit too complicated to get into here (and it is generally agreed that `Ltac` is not the most beautiful part of Coq's design!), but they can be found in the reference manual and other books on Coq, and there are many examples of `Ltac` definitions in the Coq standard library that you can use as examples.
- There is also an OCaml API, which can be used to build tactics that access Coq's internal structures at a lower level, but this is seldom worth the trouble for ordinary Coq users.

The `Tactic Notation` mechanism is the easiest to come to grips with, and it offers plenty of power for many purposes. Here's an example.

```
Tactic Notation "simpl_and_try" tactic(c) :=
  simpl;
  try c.
```

This defines a new tactical called `simpl_and_try` that takes one tactic `c` as an argument and is defined to be equivalent to the tactic `simpl; try c`. Now writing "`simpl_and_try Refl.`" in a proof will be the same as writing "`simpl; try Refl.`"

2.3. The `omega` Tactic.

Probably related to https://github.com/forestbelton/cooper

The `omega` tactic implements a decision procedure for a subset of first-order logic called *Presburger arithmetic*. It is based on the Omega algorithm invented in 1991 by William Pugh [Pugh 1991].

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 inequality (`≤`), and
- the logical connectives `/\`, `\/`, `Not`, and `→`,

then invoking `omega` will either solve the goal or tell you that it is actually false.

```
Example silly_presburger_example : forall m n o p,
  m + n ≤ n + o /\ o + 3 = p + 3 →
  m ≤ p.
Proof.
  intros. omega.
Qed.
```

2.4. A Few More Handy Tactics. Finally, here are some miscellaneous tactics that you may find convenient.

- `clear H`: Delete hypothesis `H` from the context.
- `subst x`: Find an assumption `x = e` or `e = x` in the context, replace `x` with `e` throughout the context and current goal, and clear the assumption.
- `subst`: Substitute away all assumptions of the form `x = e` or `e = x`.
- `rename... into...`: Change the name of a hypothesis in the proof context. For example, if the context includes a variable named `x`, then `rename x into y` will change all occurrences of `x` to `y`.
- `assumption`: Try to find a hypothesis `H` in the context that exactly matches the goal; if one is found, behave like `apply H`.
- `contradiction`: Try to find a hypothesis `H` in the current context that is logically equivalent to `Void`. 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 below.

3. Evaluation as a Relation

We have presented `aeval` and `beval` as functions. Another way to think about evaluation — one that we will see is often more flexible — is as a *relation* between expressions and their values. This leads naturally to `data` definitions like the following one for arithmetic expressions…

```
data AEvalR : AExp0 → Nat → Type where
  E_ANum : (n: Nat) → AEvalR (ANum n) n
  E_APlus : (e1, e2 : AExp0) → (n1, n2 : Nat) →
```

```
    AEvalR e1 n1 →
    AEvalR e2 n2 →
    AEvalR (APlus e1 e2) (n1 + n2)
  E_AMinus : (e1, e2 : AExp0) → (n1, n2 : Nat) →
    AEvalR e1 n1 →
    AEvalR e2 n2 →
    AEvalR (AMinus0 e1 e2) (n1 `minus` n2)
  E_AMult : (e1, e2: AExp0) → (n1, n2 : Nat) →
    AEvalR e1 n1 →
    AEvalR e2 n2 →
    AEvalR (AMult0 e1 e2) (n1 * n2)
```

Edit

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`.

In fact, Idris provides a way to use this notation in the definition of `AevalR` itself. This reduces confusion by avoiding situations where we're working on a proof involving statements in the form `e |/ n` but we have to refer back to a definition written using the form `AEvalR e n`.

We do this by first "reserving" the notation, then giving the definition together with a declaration of what the notation means.

```
infixl 5 |/

data (|/) : AExp0 → (n : Nat) → Type where
  E_ANum : (ANum0 n) |/ n
  E_APlus : (e1 |/ n1) → (e2 |/ n2) → (n = n1 + n2) →
            (APlus0 e1 e2) |/ n
```

We don't use - since it requires a proof of `LTE n1 n2`

```
  E_AMinus : (e1 |/ n1) → (e2 |/ n2) → (n = n1 `minus` n2) →
             (AMinus0 e1 e2) |/ n
  E_AMult : (e1 |/ n1) → (e2 |/ n2) → (n = n1 * n2) →
            (AMult0 e1 e2) |/ n

AEvalR : AExp0 → Nat → Type
AEvalR = (|/)
```

3.1. Inference Rule Notation.

Add hyperlink

In informal discussions, it is convenient to write the rules for `AEvalR` and similar relations in the more readable graphical form of inference rules, where the premises above the line justify the conclusion below the line (we have already seen them in the `IndProp` chapter).

For example, the constructor *E_APlus*...

```
E_APlus : (e1 |/ n1) → (e2 |/ n2) → (n = n1 + n2) →
          (APlus0 e1 e2) |/ n
```

...would be written like this as an inference rule:

$$\frac{\texttt{e1 |/ n1} \qquad \texttt{e2 |/ n2}}{\texttt{APlus e1 e2 |/ n1+n2}}\ \textit{E\_APlus}$$

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 function declaration. In informal prose, this is either elided or else indicated by saying something like "Let *AEvalR* be the smallest relation closed under the following rules...".

For example, |/ is the smallest relation closed under these rules:

$$\frac{}{\texttt{ANum n |/ n}}\ \textit{E\_ANum}$$

$$\frac{\texttt{e1 |/ n1} \qquad \texttt{e2 |/ n2}}{\texttt{APlus e1 e2 |/ n1+n2}}\ \textit{E\_APlus}$$

$$\frac{\texttt{e1 |/ n1} \qquad \texttt{e2 |/ n2}}{\texttt{AMinus e1 e2 |/ n1-n2}}\ \textit{E\_AMinus}$$

$$\frac{\texttt{e1 |/ n1} \qquad \texttt{e2 |/ n2}}{\texttt{AMult e1 e2 |/ n1*n2}}\ \textit{E\_AMult}$$

3.2. Equivalence of the Definitions. It is straightforward to prove that the relational and functional definitions of evaluation agree:

```
aeval_iff_aevalR : (a |/ n) ↔ aeval0 a = n
aeval_iff_aevalR = (to, fro)
where
  to : (a |/ n) → aeval0 a = n
  to E_ANum = Refl
  to (E_APlus x y xy) =
    rewrite xy in
    rewrite to x in
    rewrite to y in Refl
  to (E_AMinus x y xy) =
```

```
    rewrite xy in
    rewrite to x in
    rewrite to y in Refl
  to (E_AMult x y xy) =
    rewrite xy in
    rewrite to x in
    rewrite to y in Refl
  fro : (aeval0 a = n) -> (a || n)
  fro {a=ANum0 n} Refl = E_ANum
  fro {a=APlus0 x y} prf =
    E_APlus (fro {a=x} {n=aeval0 x} Refl)
            (fro {a=y} {n=aeval0 y} Refl)
            (sym prf)
  fro {a=AMinus0 x y} prf =
    E_AMinus (fro {a=x} {n=aeval0 x} Refl)
             (fro {a=y} {n=aeval0 y} Refl)
             (sym prf)
  fro {a=AMult0 x y} prf =
    E_AMult (fro {a=x} {n=aeval0 x} Refl)
            (fro {a=y} {n=aeval0 y} Refl)
            (sym prf)
```

We can make the proof quite a bit shorter by making more use of tacticals.

```
Theorem aeval_iff_aevalR' : forall a n,
  (a || n) <-> aeval a = n.
Proof.
  (* WORKED IN CLASS *)
  split.
  - (* -> *)
    intros H; induction H; subst; Refl.
  - (* <- *)
    generalize dependent n.
    induction a; simpl; intros; subst; constructor;
       try apply IHa1; try apply IHa2; Refl.
Qed.
```

3.2.1. *Exercise: 3 stars (bevalR).* Write a relation `BEvalR` in the same style as `AEvalR`, and prove that it is equivalent to `beval`.

```
-- data BevalR : BExp0 -> (b : Bool) -> Type where
  -- FILL IN HERE

-- beval_iff_bevalR : (BEvalR b bv) <-> beval b = bv
```

□

3.3. Computational vs. Relational Definitions. For the definitions of evaluation for arithmetic and boolean expressions, the choice of whether to use functional or relational definitions is mainly a matter of taste: either way works.

However, there are circumstances where relational definitions of evaluation work much better than functional ones.

For example, suppose that we wanted to extend the arithmetic operations by considering also a division operation:

```
data AExpD : Type where
  ANumD : Nat → AExpD
  APlusD : AExpD → AExpD → AExpD
  AMinusD : AExpD → AExpD → AExpD
  AMultD : AExpD → AExpD → AExpD
  ADivD : AExpD → AExpD → AExpD -- ⟵ new
```

Extending the definition of `aeval` to handle this new operation would not be straightforward (what should we return as the result of `ADiv (ANum 5) (ANum 0)`?). But extending `AEvalR` is straightforward.

```
infixl 5 |||

data (|||) : AExpD → (n : Nat) → Type where
  E_ANumD : (ANumD n) ||| n
  E_APlusD : (e1 ||| n1) → (e2 ||| n2) → (n = n1 + n2) →
    (APlusD e1 e2) ||| n
  E_AMinusD : (e1 ||| n1) → (e2 ||| n2) → (n = n1 `minus` n2) →
    (AMinusD e1 e2) ||| n
  E_AMultD : (e1 ||| n1) → (e2 ||| n2) → (n = n1 * n2) →
    (AMultD e1 e2) ||| n
  E_ADivD : (e1 ||| n1) → (e2 ||| n2) → (n2 `GT` 0) → (n1 = n2*n3) →
    (ADivD e1 e2) ||| n3

AEvalRD : AExpD → Nat → Type
AEvalRD = (|||)
```

Suppose, instead, that we want to extend the arithmetic operations by a nondeterministic number generator `any` that, when evaluated, may yield any number. (Note that this is not the same as making a *probabilistic* choice among all possible numbers — we're not specifying any particular distribution of results, but just saying what results are *possible*.)

```
data AExpA : Type where
  AAnyA : AExpA -- ⟵ new
  ANumA : Nat → AExpA
  APlusA : AExpA → AExpA → AExpA
  AMinusA : AExpA → AExpA → AExpA
  AMultA : AExpA → AExpA → AExpA
```

Again, extending `aeval` would be tricky, since now evaluation is *not* a deterministic function from expressions to numbers, but extending `AEvalR` is no problem:

```
infix 5 ||||/

data (||||/) : AExpA -> (n : Nat) -> Type where
  E_AnyA : AAnyA ||||/ n
  E_ANumA : (ANumA n) ||||/ n
  E_APlusA : (e1 ||||/ n1) -> (e2 ||||/ n2) -> (n = n1 + n2) ->
    (APlusA e1 e2) ||||/ n
  E_AMinusA : (e1 ||||/ n1) -> (e2 ||||/ n2) -> (n = n1 `minus` n2) ->
    (AMinusA e1 e2) ||||/ n
  E_AMultA : (e1 ||||/ n1) -> (e2 ||||/ n2) -> (n = n1 * n2) ->
    (AMultA e1 e2) ||||/ n

AEvalRA : AExpA -> Nat -> Type
AEvalRA = (||||/)
```

At this point you maybe wondering: which style should I use by default? The examples above show that relational definitions are fundamentally more powerful 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 choice. But what about when both styles are workable?

Edit

One point in favor of relational definitions is that some people feel they are more elegant and easier to understand. Another is that Idris automatically generates nice inversion and induction principles from function definitions.

On the other hand, functional definitions can often be more convenient:

- Functions are by definition deterministic and defined on all arguments; for a relation we have to show these properties explicitly if we need them.
- With functions we can also take advantage of Idris's computation mechanism to simplify expressions during proofs.

Furthermore, functions can be directly "extracted" to executable code in C or JavaScript.

Ultimately, the choice often comes down to either the specifics of a particular situation or simply a question of taste. Indeed, in large Idris 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.

4. Expressions With Variables

Let's turn our attention back to defining Imp. The next thing we need to do is to enrich our arithmetic and Boolean expressions with variables. To keep things simple, we'll assume that all variables are global and that they only hold numbers.

4.1. States. Since we'll want to look variables up to find out their current values, we'll reuse the type *Id* from the Maps chapter for the type of variables in Imp.

A *machine state* (or just *state*) represents the current values of *all* variables at some point in the execution of a program.

For simplicity, we assume that the state is defined for *all* variables, even though any given program is only going to mention a finite number of them. The state captures all of the information stored in memory. For Imp programs, because each variable stores a natural number, we can represent the state as a mapping from identifiers to *Nat*. For more complex programming languages, the state might have more structure.

```
State : Type
State = TotalMap Nat

empty_state : State
empty_state = t_empty 0
```

4.2. Syntax. We can add variables to the arithmetic expressions we had before by simply adding one more constructor:

```
data AExp : Type where
  ANum : Nat → AExp
  AId : Id → AExp -- <——— NEW
  APlus : AExp → AExp → AExp
  AMinus : AExp → AExp → AExp
  AMult : AExp → AExp → AExp
```

Defining a few variable names as notational shorthands will make examples easier to read:

```
W : Id
W = MkId "W"
X : Id
X = MkId "X"
Y : Id
Y = MkId "Y"
Z : Id
Z = MkId "Z"
```

Edit

(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 devoped to Imp, this overloading should not cause confusion.)

The definition of *BExp*s is unchanged (except for using the new *AExp*s):

```
data BExp : Type where
  BTrue : BExp
```

```
  BFalse : BExp
  BEq : AExp → AExp → BExp
  BLe : AExp → AExp → BExp
  BNot : BExp → BExp
  BAnd : BExp → BExp → BExp
```

4.3. Evaluation. The arith and boolean evaluators are extended to handle variables in the obvious way, taking a state as an extra argument:

```
aeval : (st : State) → (a : AExp) → Nat
aeval _ (ANum n) = n
aeval st (AId i) = st i
aeval st (APlus a1 a2) = (aeval st a1) + (aeval st a2)
aeval st (AMinus a1 a2) = (aeval st a1) `minus` (aeval st a2)
aeval st (AMult a1 a2) = (aeval st a1) * (aeval st a2)

beval : (st : State) → (b : BExp) → Bool
beval _ BTrue = True
beval _ BFalse = False
beval st (BEq a1 a2) = (aeval st a1) == (aeval st a2)
beval st (BLe a1 a2) = lte (aeval st a1) (aeval st a2)
beval st (BNot b1) = not (beval st b1)
beval st (BAnd b1 b2) = (beval st b1) && (beval st b2)

aexp1 : aeval (t_update X 5 Imp.empty_state)
              (APlus (ANum 3) (AMult (AId X) (ANum 2)))
        = 13
aexp1 = Refl

bexp1 : beval (t_update X 5 Imp.empty_state)
              (BAnd BTrue (BNot (BLe (AId X) (ANum 4))))
        = True
bexp1 = Refl
```

5. Commands

Now we are ready define the syntax and behavior of Imp *commands* (sometimes called *statements*).

5.1. Syntax. Informally, commands `c` are described by the following BNF grammar. (We choose this slightly awkward concrete syntax for the sake of being able to define Imp syntax using Idris's `syntax` mechanism. In particular, we use `IFB` to avoid conflicting with the `if` notation from the standard library.)

```
    c ::= SKIP | x ::= a | c ;; c | IFB b THEN c ELSE c FI
        | WHILE b DO c END
```

For example, here's factorial in Imp:

```
Z ::= X;;
Y ::= 1;;
WHILE not (Z = 0) DO
  Y ::= Y * Z;;
  Z ::= Z - 1
END
```

When this command terminates, the variable `Y` will contain the factorial of the initial value of `X`.

Here is the formal definition of the abstract syntax of commands:

```
data Com : Type where
  CSkip : Com
  CAss : Id → AExp → Com
  CSeq : Com → Com → Com
  CIf : BExp → Com → Com → Com
  CWhile : BExp → Com → Com
```

As usual, we can use a few `Notation` declarations to make things more readable. To avoid conflicts with Idris's built-in notations, we keep this light — in particular, we don't introduce any notations for `AExp`s and `BExp`s to avoid confusion with the numeric and boolean operators we've already defined.

Use do-notation and functions instead?

```
syntax SKIP = CSkip
syntax [x] "::=" [a] = CAss x a
syntax [c1] ";;" [c2] = CSeq c1 c2
syntax WHILE [b] DO [c] END = CWhile b c
syntax IFB [c1] THEN [c2] ELSE [c3] FI = CIf c1 c2 c3
```

For example, here is the factorial function again, written as a formal definition to Idris:

```
fact_in_idris : Com
fact_in_idris =
 (Z ::= AId X) ;;
 (Y ::= ANum 1) ;;
 WHILE BNot (BEq (AId Z) (ANum 0)) DO
   (Y ::= AMult (AId Y) (AId Z)) ;;
   (Z ::= AMinus (AId Z) (ANum 1))
 END
```

5.2. More Examples.

5.2.1. *Assignment:*

```
plus2 : Com
plus2 =
 X ::= (APlus (AId X) (ANum 2))
```

```
XtimesYinZ : Com
XtimesYinZ =
 Z ::= (AMult (AId X) (AId Y))

subtract_slowly_body : Com
subtract_slowly_body =
  (Z ::= AMinus (AId Z) (ANum 1)) ;;
  (X ::= AMinus (AId X) (ANum 1))
```

5.2.2. *Loops.*

```
subtract_slowly : Com
subtract_slowly =
 WHILE BNot (BEq (AId X) (ANum 0)) DO
   subtract_slowly_body
 END

subtract_3_from_5_slowly : Com
subtract_3_from_5_slowly =
  (X ::= ANum 3) ;;
  (Z ::= ANum 5) ;;
  subtract_slowly
```

5.2.3. *An infinite loop:*

```
loop : Com
loop =
  WHILE BTrue DO
    SKIP
  END
```

6. 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...

6.1. Evaluation as a Function (Failed Attempt). Here's an attempt at defining an evaluation function for commands, omitting the `WHILE` case.

```
ceval_fun_no_while : (st : State) → (c : Com) → State
ceval_fun_no_while st CSkip = st
ceval_fun_no_while st (CAss x a) = t_update x (aeval st a) st
ceval_fun_no_while st (CSeq c1 c2) =
  let st' = ceval_fun_no_while st c1
  in ceval_fun_no_while st' c2
ceval_fun_no_while st (CIf b c1 c2) =
  if beval st b
    then ceval_fun_no_while st c1
```

```
    else ceval_fun_no_while st c2
ceval_fun_no_while st (CWhile b c) = st   -- bogus
```

In a traditional functional programming language like OCaml or Haskell we could add the *WHILE* case as follows:

```
...
ceval_fun st (CWhile b c) =
  if (beval st b)
    then ceval_fun st (CSeq c $ CWhile b c)
    else st
```

Idris doesn't accept such a definition ("Imp.ceval_fun is possibly not total due to recursive path Imp.ceval_fun –> Imp.ceval_fun –> Imp.ceval_fun") 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 Idris is not just a functional programming language but also a consistent logic, any potentially non-terminating function needs to be rejected. Here is an (invalid!) program showing what would go wrong if Idris allowed non-terminating recursive functions:

Edit, discuss `partial`

```
loop_false : (n : Nat) -> Void
loop_false n = loop_false n
```

That is, propositions like *Void* would become provable (`loop_false 0` would be a proof of *Void*), which would be a disaster for Idris's logical consistency.

Thus, because it doesn't terminate on all inputs, `ceval_fun` cannot be written in Idris — at least not without additional tricks and workarounds (see chapter `ImpCEvalFun` if you're curious about what those might be).

6.2. Evaluation as a Relation. Here's a better way: define *CEval* as a *relation* rather than a *function* — i.e., define it with `data`, as we did for *AEvalR* above.

This is an important change. Besides freeing us from awkward workarounds, it gives us a lot more flexibility in the definition. For example, if we add nondeterministic features like `any` to the language, we want the definition of evaluation to be nondeterministic — i.e., not only will it not be total, it will not even be a function!

We'll use the notation `c / st |/ st'` for the *CEval* relation: `c / st |/ st'` means that executing program `c` in a starting state `st` results in an ending state `st'`. This can be pronounced "`c` takes state `st` to `st'`".

6.2.1. *Operational Semantics.* Here is an informal definition of evaluation, presented as inference rules for readability:

$$\frac{}{\textit{SKIP} \;/\; st \;|/\; st}\;\textit{E\_Skip}$$

$$\frac{\texttt{aeval st a1 = n}}{\texttt{x := a1 / st |/ (t\_update st x n)}}\ \textit{E\_Ass}$$

$$\frac{\texttt{c1 / st |/ st'} \quad \texttt{c2 / st' |/ st''}}{\texttt{c1;;c2 / st |/ st''}}\ \textit{E\_Seq}$$

$$\frac{\texttt{beval st b1 = True} \quad \texttt{c1 / st |/ st'}}{\textit{IF}\ \texttt{b1}\ \textit{THEN}\ \texttt{c1}\ \textit{ELSE}\ \texttt{c2}\ \textit{FI}\ \texttt{/ st |/ st'}}\ \textit{E\_IfTrue}$$

$$\frac{\texttt{beval st b1 = False} \quad \texttt{c2 / st |/ st'}}{\textit{IF}\ \texttt{b1}\ \textit{THEN}\ \texttt{c1}\ \textit{ELSE}\ \texttt{c2}\ \textit{FI}\ \texttt{/ st |/ st'}}\ \textit{E\_IfFalse}$$

$$\frac{\texttt{beval st b = False}}{\textit{WHILE}\ \texttt{b}\ \textit{DO}\ \texttt{c}\ \textit{END}\ \texttt{/ st |/ st}}\ \textit{E\_WhileEnd}$$

$$\frac{\texttt{beval st b = True} \quad \texttt{c / st |/ st'} \quad \textit{WHILE}\ \texttt{b}\ \textit{DO}\ \texttt{c}\ \textit{END}\ \texttt{/ st' |/ st''}}{\textit{WHILE}\ \texttt{b}\ \textit{DO}\ \texttt{c}\ \textit{END}\ \texttt{/ st |/ st''}}\ \textit{E\_WhileLoop}$$

Here is the formal definition. Make sure you understand how it corresponds to the inference rules.

```
data CEval : Com -> State -> State -> Type where
  E_Skip : CEval CSkip st st
  E_Ass : aeval st a1 = n -> CEval (CAss x a1) st (t_update x n st)
  E_Seq : CEval c1 st st' -> CEval c2 st' st'' ->
    CEval (CSeq c1 c2) st st''
  E_IfTrue : beval st b = True -> CEval c1 st st' ->
    CEval (CIf b c1 c2) st st'
  E_IfFalse : beval st b = False -> CEval c2 st st' ->
    CEval (CIf b c1 c2) st st'
  E_WhileEnd : beval st b = False ->
    CEval (CWhile b c) st st
  E_WhileLoop : beval st b = True ->
    CEval c st st' -> CEval (CWhile b c) st' st'' ->
    CEval (CWhile b c) st st''

syntax [c1] "/" [st] "|/" [st'] = CEval c1 st st'
```

The cost of defining evaluation as a relation instead of a function is that we now need to construct *proofs* that some program evaluates to some result state, rather than just letting Idris's computation mechanism do it for us.

```
ceval_example1 : (
  (X ::= ANum 2) ;;
  (IFB BLe (AId X) (ANum 1)
    THEN (Y ::= ANum 3)
    ELSE (Z ::= ANum 4)
   FI)
```

```
  ) / Imp.empty_state
   |/ (t_update Z 4 $ t_update X 2 Imp.empty_state)
ceval_example1 =
  E_Seq
    (E_Ass Refl)
    (E_IfFalse Refl
               (E_Ass Refl))
```

6.2.2. *Exercise: 2 stars (ceval_example2).*

```
ceval_example2 :
   ((X ::= ANum 0);; (Y ::= ANum 1);; (Z ::= ANum 2))
   / Imp.empty_state
  |/ (t_update Z 2 $ t_update Y 1 $ t_update X 0 Imp.empty_state)
ceval_example2 = ?ceval_example2_rhs
```

□

6.2.3. *Exercise: 3 stars, advanced (pup_to_n).* Write an Imp program that sums the numbers from 1 to X (inclusive: 1 + 2 + ... + X) in the variable Y. Prove that this program executes as intended for X = 2 (this is trickier than you might expect).

```
pup_to_n : Com
pup_to_n = ?pup_to_n_rhs

pup_to_2_ceval :
  Imp.pup_to_n / (t_update X 2 Imp.empty_state) |/
   (t_update X 0 $
    t_update Y 3 $
    t_update X 1 $
    t_update Y 2 $
    t_update Y 0 $
    t_update X 2 Imp.empty_state)
pup_to_2_ceval = ?pup_to_2_ceval_rhs
```

□

6.3. 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:

```
ceval_deterministic : (c / st |/ st1) -> (c / st |/ st2) -> st1 = st2
ceval_deterministic E_Skip E_Skip = Refl
ceval_deterministic (E_Ass aev1) (E_Ass aev2) =
  rewrite sym aev1 in
```

```
  rewrite sym aev2 in Refl
ceval_deterministic {st2} (E_Seq cev11 cev12)
                          (E_Seq {c2} cev21 cev22) =
  let ih = ceval_deterministic cev11 cev21
      cev22' = replace (sym ih) cev22 {P=\x⇒CEval c2 x st2}
  in ceval_deterministic cev12 cev22'
ceval_deterministic (E_IfTrue _ cev1) (E_IfTrue _ cev2) =
  ceval_deterministic cev1 cev2
ceval_deterministic (E_IfTrue prf1 _) (E_IfFalse prf2 _) =
  absurd $ replace prf1 prf2 {P=\x⇒x=False}
ceval_deterministic (E_IfFalse prf1 _) (E_IfTrue prf2 _) =
  absurd $ replace prf2 prf1 {P=\x⇒x=False}
ceval_deterministic (E_IfFalse _ cev1) (E_IfFalse _ cev2) =
  ceval_deterministic cev1 cev2
ceval_deterministic (E_WhileEnd _) (E_WhileEnd _) = Refl
ceval_deterministic (E_WhileEnd prf1) (E_WhileLoop prf2 _ _) =
  absurd $ replace prf2 prf1 {P=\x⇒x=False}
ceval_deterministic (E_WhileLoop prf1 _ _) (E_WhileEnd prf2) =
  absurd $ replace prf1 prf2 {P=\x⇒x=False}
ceval_deterministic {st2} (E_WhileLoop _ cev11 cev12)
                          (E_WhileLoop {b} {c} _ cev21 cev22) =
  let ih = ceval_deterministic cev11 cev21
      cev22' = replace (sym ih) cev22 {P=\x⇒CEval (CWhile b c) x st2}
  in ceval_deterministic cev12 cev22'
```

7. Reasoning About Imp Programs

We'll get deeper into systematic techniques for reasoning about Imp programs in the following chapters, but we can do quite a bit just working with the bare definitions. This section explores some examples.

```
plus2_spec : st X = n → (Imp.plus2 / st |/ st') → st' X = n + 2
```

Edit

Inverting `Heval` essentially forces Idris 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

```
plus2_spec prf (E_Ass aev) = rewrite sym aev in rewrite prf in Refl
```

7.0.1. *Exercise: 3 stars, recommendedM (XtimesYinZ_spec).* State and prove a specification of `XtimesYinZ`.

```
-- FILL IN HERE
```

□

7.0.2. *Exercise: 3 stars, recommended (loop_never_stops).*

```
loop_never_stops : Not (Imp.loop / st || st')
loop_never_stops contra = ?loop_never_stops_rhs
```

Edit the hint

```
Proof.
  intros st st' contra. unfold loop in contra.
  remember (WHILE BTrue 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 inversion).

(* FILL IN HERE *) Admitted.

□

7.0.3. *Exercise: 3 stars (no_whilesR).* Consider the following function:

```
no_whiles : (c : Com) → Bool
no_whiles CSkip = True
no_whiles (CAss _ _) = True
no_whiles (CSeq c1 c2) = (no_whiles c1) && (no_whiles c2)
no_whiles (CIf _ ct cf) = (no_whiles ct) && (no_whiles cf)
no_whiles (CWhile _ _) = False
```

This predicate yields `True` just on programs that have no while loops. Using `data`, 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`.

```
data No_whilesR : Com → Type where
  Remove_Me_No_whilesR : No_whilesR CSkip

no_whiles_eqv : (no_whiles c = True) ↔ (No_whilesR c)
no_whiles_eqv = ?no_whiles_eqv_rhs
```

□

7.0.4. *Exercise: 4 starsM (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.

```
-- FILL IN HERE
```

□

8. Additional Exercises

8.0.1. *Exercise: 3 stars (stack_compiler).* HP Calculators, programming languages like Forth and Postscript and abstract machines like the Java Virtual Machine all evaluate arithmetic expressions using a stack. For instance, the expression

```
(2*3)+(3*(4-2))
```

would be entered as

```
2 3 * 3 4 2 - * +
```

and evaluated like this (where we show the program being evaluated on the right and the contents of the stack on the left):

```
[ ]          |   2 3 * 3 4 2 - * +
[2]          |   3 * 3 4 2 - * +
[3, 2]       |   * 3 4 2 - * +
[6]          |   3 4 2 - * +
[3, 6]       |   4 2 - * +
[4, 3, 6]    |   2 - * +
[2, 4, 3, 6] |   - * +
[2, 3, 6]    |   * +
[6, 6]       |   +
[12]         |
```

The task of this exercise is to write a small compiler that translates `AExp`s into stack machine instructions.

The instruction set for our stack language will consist of the following instructions:

- `SPush n`: Push the number `n` on the stack.
- `SLoad x`: Load the identifier `x` from the store and push it on the stack
- `SPlus`: Pop the two top numbers from the stack, add them, and push the result onto the stack.
- `SMinus`: Similar, but subtract.
- `SMult`: Similar, but multiply.

```
data SInstr : Type where
  SPush : Nat → SInstr
  SLoad : Id → SInstr
  SPlus : SInstr
  SMinus : SInstr
  SMult : SInstr
```

Write a function to evaluate programs in the stack language. It should take as input a state, a stack represented as a list of numbers (top stack item is the head of the list), and a program represented as a list of instructions, and it should return the stack after executing the program. Test your function on the examples below.

Note that the specification leaves unspecified what to do when encountering an `SPlus`, `SMinus`, or `SMult` instruction if the stack contains less than two elements. In

a sense, it is immaterial what we do, since our compiler will never emit such a malformed program.

```
s_execute : (st : state) → (stack : List Nat) → (prog : List SInstr) →
            List Nat
s_execute st stack prog = ?s_execute_rhs

s_execute1 : s_execute Imp.empty_state [] [SPush 5, SPush 3, SPush 1, SMinus]
             = [2,5]
s_execute1 = ?s_execute1_rhs

s_execute2 : s_execute (t_update X 3 Imp.empty_state) [3,4]
                       [SPush 4, SLoad X, SMult, SPlus]
             = [15,4]
s_execute2 = ?s_execute2_rhs
```

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.

```
s_compile : (e : AExp) → List SInstr
s_compile e = ?s_compile_rhs
```

After you've defined `s_compile`, prove the following to test that it works.

```
s_compile1 : s_compile (AMinus (AId X) (AMult (ANum 2) (AId Y)))
             = [SLoad X, SPush 2, SLoad Y, SMult, SMinus]
s_compile1 = ?s_compile1_rhs
```

□

8.0.2. *Exercise: 4 stars, advanced (stack_compiler_correct).* Now we'll prove the correctness of the compiler implemented in the previous exercise. Remember that the specification left unspecified what to do when encountering an `SPlus`, `SMinus`, or `SMult` instruction if the stack contains less than two elements. (In order to make your correctness proof easier you might find it helpful to go back and change your implementation!)

Prove the following theorem. You will need to start by stating a more general lemma to get a usable induction hypothesis; the main theorem will then be a simple corollary of this lemma.

```
s_compile_correct : s_execute st [] (s_compile e) = [aeval st e]
s_compile_correct = ?s_compile_correct_rhs
```

□

8.0.3. *Exercise: 3 stars, optional (short_circuit).* Most modern programming languages use a "short-circuit" evaluation rule for boolean `and`: to evaluate `BAnd b1 b2`, first evaluate `b1`. If it evaluates to `False`, then the entire `BAnd` expression evaluates to `False` immediately, without evaluating `b2`. Otherwise, `b2` is evaluated to determine the result of the `BAnd` expression.

Write an alternate version of `beval` that performs short-circuit evaluation of *BAnd* in this manner, and prove that it is equivalent to `beval`.

```
-- FILL IN HERE
```

□

8.0.4. *Exercise: 4 stars, advanced (break_imp).* Imperative languages like C and Java often include a `break` or similar statement for interrupting the execution of loops. In this exercise we consider how to add `break` to Imp. First, we need to enrich the language of commands with an additional case.

```
data ComB : Type where
  CSkipB : ComB
  CBreakB : ComB  -- ⟵ new
  CAssB : Id → AExp → ComB
  CSeqB : ComB → ComB → ComB
  CIfB : BExp → ComB → ComB → ComB
  CWhileB : BExp → ComB → ComB
```

Also use do-notation here?

```
syntax SKIP' = CSkipB
syntax BREAK' = CBreakB
syntax [x] "::='" [a] = CAssB x a
syntax [c1] ";;'" [c2] = CSeqB c1 c2
syntax WHILE' [b] DO [c] END = CWhileB b c
syntax IFB' [c1] THEN [c2] ELSE [c3] FI = CIfB c1 c2 c3
```

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 not (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:

```
data Result : Type where
  SContinue : Result
  SBreak : Result
```

Intuitively, `c // st |/ s / st'` means that, if `c` is started in state `st`, then it terminates in state `st'` and either signals that the innermost surrounding loop (or the whole program) should exit immediately (`s = SBreak`) or that execution should continue normally (`s = SContinue`).

The definition of the "`c // st |/ s / st'`" relation is very similar to the one we gave above for the regular evaluation relation (`c / st |/ st'`) — we just need to handle the termination signals appropriately:

- If the command is *SKIP*, then the state doesn't change and execution of any enclosing loop can continue normally.
- If the command is *BREAK*, the state stays unchanged but we signal a *SBreak*.
- If the command is an assignment, then we update the binding for that variable in the state accordingly and signal that execution can continue normally.
- If the command is of the form *IFB* `b` *THEN* `c1` *ELSE* `c2` *FI*, then the state is updated as in the original semantics of Imp, except that we also propagate the signal from the execution of whichever branch was taken.
- If the command is a sequence `c1 ;; c2`, we first execute `c1`. If this yields a *SBreak*, we skip the execution of `c2` and propagate the *SBreak* signal to the surrounding context; the resulting state is the same as the one obtained by executing `c1` alone. Otherwise, we execute `c2` on the state obtained after executing `c1`, and propagate the signal generated there.
- Finally, for a loop of the form *WHILE* `b` *DO* `c` *END*, the semantics is almost the same as before. The only difference is that, when `b` evaluates to *True*, we execute `c` and check the signal that it raises. If that signal is *SContinue*, then the execution proceeds as in the original semantics. Otherwise, we stop the execution of the loop, and the resulting state is the same as the one resulting from the execution of the current iteration. In either case, since *BREAK* only terminates the innermost loop, *WHILE* signals *SContinue*.

Based on the above description, complete the definition of the *CEvalB* relation.

```
data CEvalB : ComB → State → Result → State → Type where
  E_SkipB : CEvalB CSkipB st SContinue st
  -- FILL IN HERE

syntax [c1] "//" [st] "|/" [s] "/" [st'] = CEvalB c1 st s st'
```

Now prove the following properties of your definition of *CEvalB*:

```
break_ignore : (((BREAK') ;;' (c)) // st |/ s / st') -> st = st'
break_ignore x = ?break_ignore_rhs

while_continue : ((WHILE' b DO c END) // st  |/ s / st') -> s = SContinue
while_continue x = ?while_continue_rhs

while_stops_on_break : beval st b = True ->
                       (c // st |/ SBreak / st') ->
                       ((WHILE' b DO c END) // st |/ SContinue / st')
while_stops_on_break prf x = ?while_stops_on_break_rhs
```

□

8.0.5. *Exercise: 3 stars, advanced, optional (while_break_true).*

```
while_break_true : ((WHILE' b DO c END) // st |/ SContinue / st') ->
                   beval st' b = True ->
                   (st'' ** c // st'' |/ SBreak / st')
while_break_true x prf = ?while_break_true_rhs
```

□

8.0.6. *Exercise: 4 stars, advanced, optional (cevalB_deterministic).*

```
cevalB_deterministic : (c // st |/ s1 / st1) ->
                       (c // st |/ s2 / st2) ->
                       (st1 = st2, s1 = s2)
cevalB_deterministic x y = ?cevalB_deterministic_rhs
```

□

8.0.7. *Exercise: 4 stars, optional (add_for_loop).* Add C-style `for` loops to the language of commands, update the `CEval` definition to define the semantics of `for` loops, and add cases for `for` loops as needed so that all the proofs in this file are accepted by Idris.

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.)

```
-- FILL IN HERE
```

□

CHAPTER 12

ImpParser: Lexing and Parsing in Idris

```
module ImpParser
```

The development of the Imp language in `Imp.lidr` 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 Idris'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.

```
import Maps
import Imp
```

1. Internals

1.1. Lexical Analysis.

```
data Chartype = White | Alpha | Digit | Other

classifyChar : (c : Char) -> Chartype
classifyChar c =
  if isSpace c then
    White
  else if isAlpha c then
    Alpha
  else if isDigit c then
    Digit
  else
    Other

Token : Type
Token = String
```

```
tokenizeHelper : (cls : Chartype) -> (acc, xs : List Char) -> List (List Char)
tokenizeHelper cls acc xs =
  case xs of
    []      => tk
    (x::xs') =>
      case (cls, classifyChar x, x) of
        (_, _, '(')       =>
          tk ++ ['('] :: (tokenizeHelper Other [] xs')
        (_, _, ')')       =>
          tk ++ [')'] :: (tokenizeHelper Other [] xs')
        (_, White, _)     =>
          tk ++ (tokenizeHelper White [] xs')
        (Alpha, Alpha, x) =>
          tokenizeHelper Alpha (x::acc) xs'
        (Digit, Digit, x) =>
          tokenizeHelper Digit (x::acc) xs'
        (Other, Other, x) =>
          tokenizeHelper Other (x::acc) xs'
        (_, tp, x)        =>
          tk ++ (tokenizeHelper tp [x] xs')
 where
  tk : List (List Char)
  tk = case acc of
         []     => []
         (_::_) => [reverse acc]

tokenize : (s : String) -> List String
tokenize s = map pack (tokenizeHelper White [] (unpack s))

tokenizeEx1 : tokenize "abc12==3  223*(3+(a+c))" = ["abc","12","==","3","223","*","(","3","+","(","a","+"
tokenizeEx1 = Refl
```

1.2. Parsing.

1.2.1. *Options With Errors.* An `Option` type with error messages:

```
data OptionE : (x : Type) -> Type where
  SomeE : x -> OptionE x
  NoneE : String -> OptionE x
```

Some interface instances to make writing nested match-expressions on `OptionE` more convenient.

Explain these/link to Haskell etc?

```
Functor OptionE where
  map f (SomeE x)   = SomeE (f x)
  map _ (NoneE err) = NoneE err
```

```
Applicative OptionE where
  pure = SomeE
  (SomeE f) <*> (SomeE x) = SomeE (f x)
  (SomeE _) <*> (NoneE e) = NoneE e
  (NoneE e) <*> _         = NoneE e

Alternative OptionE where
  empty = NoneE ""
  (SomeE x) <|> _ = SomeE x
  (NoneE _) <|> v = v

Monad OptionE where
  (NoneE e) >>= _ = NoneE e
  (SomeE x) >>= k = k x
```

1.2.2. *Generic Combinators for Building Parsers.*

```
Parser : (t : Type) -> Type
Parser t = List Token -> OptionE (t, List Token)

manyHelper : (p : Parser t) -> (acc : List t) -> (steps : Nat) -> Parser (List t)
manyHelper p acc Z _ = NoneE "Too many recursive calls"
manyHelper p acc (S steps') xs with (p xs)
 | NoneE _        = SomeE (reverse acc, xs)
 | SomeE (t', xs') = manyHelper p (t'::acc) steps' xs'
```

A (step-indexed) parser that expects zero or more `p`s:

```
many : (p : Parser t) -> (steps : Nat) -> Parser (List t)
many p steps = manyHelper p [] steps
```

A parser that expects a given token, followed by `p`:

```
firstExpect : (a : Token) -> (p : Parser t) -> Parser t
firstExpect a p (x::xs) = if x == a then p xs else NoneE ("Expected '" ++ a ++ "'")
firstExpect a _ [] = NoneE ("Expected '" ++ a ++ "'")
```

A parser that expects a particular token:

```
expect : (t : Token) -> Parser ()
expect t = firstExpect t (\xs => SomeE ((), xs))
```

1.2.3. *A Recursive-Descent Parser for Imp.* Identifiers:

```
parseIdentifier : Parser Id
parseIdentifier [] = NoneE "Expected identifier"
parseIdentifier (x::xs) =
  if all isLower (unpack x)
    then SomeE (MkId x, xs)
    else NoneE ("Illegal identifier:'" ++ x ++ "'")
```

Numbers:

```
parseNumber : Parser Nat
parseNumber [] = NoneE "Expected number"
parseNumber (x::xs) =
  if all isDigit (unpack x)
    then SomeE (foldl (\n, d => 10 * n + (cast (ord d - ord '0'))) 0 (unpack x), xs)
    else NoneE "Expected number"
```

Parse arithmetic expressions

```
mutual
  parsePrimaryExp : (steps : Nat) -> Parser AExp
  parsePrimaryExp Z _ = NoneE "Too many recursive calls"
  parsePrimaryExp (S steps') xs =
    (do (i, rest) <- parseIdentifier xs
        pure (AId i, rest))
    <|>
    (do (n, rest) <- parseNumber xs
        pure (ANum n, rest))
    <|>
    (do (e, rest)  <- firstExpect "(" (parseSumExp steps') xs
        (u, rest') <- expect ")" rest
        pure (e, rest'))

  parseProductExp : (steps : Nat) -> Parser AExp
  parseProductExp Z _ = NoneE "Too many recursive calls"
  parseProductExp (S steps') xs =
    do (e, rest) <- parsePrimaryExp steps' xs
       (es, rest') <- many (firstExpect "*" (parsePrimaryExp steps')) steps' rest
       pure (foldl AMult e es, rest')

  parseSumExp : (steps : Nat) -> Parser AExp
  parseSumExp Z _ = NoneE "Too many recursive calls"
  parseSumExp (S steps') xs =
    do (e, rest) <- parseProductExp steps' xs
       (es, rest') <- many psum steps' rest
       pure (foldl (\e0, term =>
                     case term of
                       (True,  e) => APlus e0 e
                       (False, e) => AMinus e0 e
                   ) e es, rest')
    where
    psum : Parser (Bool, AExp)
    psum xs =
      let p = parseProductExp steps' in
      (do (e, r) <- firstExpect "+" p xs
          pure ((True, e), r))
      <|>
      (do (e, r) <- firstExpect "-" p xs
```

```
          pure ((False, e), r))

parseAExp : (steps : Nat) → Parser AExp
parseAExp = parseSumExp
```

Parsing boolean expressions:

```
mutual
  parseAtomicExp : (steps : Nat) → Parser BExp
  parseAtomicExp Z _ = NoneE "Too many recursive calls"
  parseAtomicExp (S steps') xs =
    (do (_, rest) ← expect "true" xs
        pure (BTrue, rest))
    <|>
    (do (_, rest) ← expect "false" xs
        pure (BFalse, rest))
    <|>
    (do (e, rest) ← firstExpect "not" (parseAtomicExp steps') xs
        pure (BNot e, rest))
    <|>
    (do (e, rest) ← firstExpect "(" (parseConjunctionExp steps') xs
        (_, rest') ← expect ")" rest
        pure (e, rest'))
    <|>
    (do (e, rest) ← parseProductExp steps' xs
        ((do (e', rest') ← firstExpect "==" (parseAExp steps') rest
             pure (BEq e e', rest'))
         <|>
         (do (e', rest') ← firstExpect "≤" (parseAExp steps') rest
             pure (BLe e e', rest'))
         <|>
         (NoneE "Expected '==' or '≤' after arithmetic expression")))

  parseConjunctionExp : (steps : Nat) → Parser BExp
  parseConjunctionExp Z _ = NoneE "Too many recursive calls"
  parseConjunctionExp (S steps') xs =
    do (e, rest) ← parseAtomicExp steps' xs
       (es, rest') ← many (firstExpect "&&" (parseAtomicExp steps')) steps' rest
       pure (foldl BAnd e es, rest')

parseBExp : (steps : Nat) → Parser BExp
parseBExp = parseConjunctionExp

testParsing : (p : Nat → Parser t) → (s : String) → OptionE (t, List Token)
testParsing p s = p 100 (tokenize s)
```

The second one seems designed to fail

```
λΠ> testParsing parseProductExp "x*y*(x*x)*x"
```

```
λΠ> testParsing parseConjunctionExp "not((x==x||x*x≤(x*x)*x)&&x==x"
```

Parsing commands:

```
mutual
  parseSimpleCommand : (steps : Nat) -> Parser Com
  parseSimpleCommand Z _ = NoneE "Too many recursive calls"
  parseSimpleCommand (S steps') xs =
    (do (_, rest) <- expect "SKIP" xs
        pure (SKIP, rest))
    <|>
    (do (e, rest) <- firstExpect "IF" (parseBExp steps') xs
        (c, rest') <- firstExpect "THEN" (parseSequencedCommand steps') rest
        (c', rest'') <- firstExpect "ELSE" (parseSequencedCommand steps') rest'
        (_, rest''') <- expect "END" rest''
        pure (IFB e THEN c ELSE c' FI, rest'''))
    <|>
    (do (e, rest) <- firstExpect "WHILE" (parseBExp steps') xs
        (c, rest') <- firstExpect "DO" (parseSequencedCommand steps') rest
        (_, rest'') <- expect "END" rest'
        pure (WHILE e DO c END, rest''))
    <|>
    (do (i, rest) <- parseIdentifier xs;
        (e, rest') <- firstExpect ":=" (parseAExp steps') rest
        pure (i ::= e, rest'))

  parseSequencedCommand : (steps : Nat) -> Parser Com
  parseSequencedCommand Z _ = NoneE "Too many recursive calls"
  parseSequencedCommand (S steps') xs =
    do (c, rest) <- parseSimpleCommand steps' xs
       ((do (c', rest') <- firstExpect ";;" (parseSequencedCommand steps') rest
            pure (c ;; c', rest'))
        <|>
        (pure (c, rest)))

bignumber : Nat
bignumber = 1000

parse : (str : String) -> OptionE (Com, List Token)
parse str = parseSequencedCommand bignumber (tokenize str)
```

2. Examples

```
λΠ> parse "IF x == y + 1 + 2 - y * 6 + 3 THEN x := x * 1;; y := 0 ELSE SKIP END"
SomeE (CIf (BEq (AId (MkId "x")) (APlus (AMinus (APlus (APlus (AId (MkId "y")) (ANum 1)) (ANum 2)) (AMult
          (CSeq (CAss (MkId "x") (AMult (AId (MkId "x")) (ANum 1))) (CAss (MkId "y") (ANum 0)))
```

```
    CSkip,
  []) : OptionE (Com, List String)
```

```
λΠ> parse "SKIP;; z:=x*y*(x*x);; WHILE x==x DO IF z ≤ z*z && not x == 2 THEN x := z;; y := z ELSE SKIP EN
```

This one is repeated twice in the book for some reason

```
λΠ> parse "SKIP;; z:=x*y*(x*x);; WHILE x==x DO IF z ≤ z*z && not x == 2 THEN x := z;; y := z ELSE SKIP EN
SomeE (CSeq CSkip
        (CSeq (CAss (MkId "z") (AMult (AMult (AId (MkId "x")) (AId (MkId "y"))) (AMult (AId (MkId "x"))
              (CSeq (CWhile (BEq (AId (MkId "x")) (AId (MkId "x")))
                            (CSeq (CIf (BAnd (BLe (AId (MkId "z")) (AMult (AId (MkId "z")) (AId (MkId
                                       (CSeq (CAss (MkId "x") (AId (MkId "z"))) (CAss (MkId "y") (AId
                                       CSkip)
                                  CSkip))
                    (CAss (MkId "x") (AId (MkId "z"))))))),
  []) : OptionE (Com, List String)
```

Glossary

algebraic data type: . 9 define

computation rule: . 16 define

expression: . 16 define

first-class: . 9 define

fully certified: . 12 define

function type: . 15 define

idris-add-clause: (idris-add-clause PROOF)

Add clauses to the declaration at point. 11, 12

idris-case-split: (idris-case-split)

Case split the pattern variable at point. 11

idris-load-file: (idris-load-file &optional SET-LINE)

Pass the current buffer's file to the inferior Idris process.

A prefix argument restricts loading to the current line. 11

idris-proof-search: (idris-proof-search &optional ARG)

Invoke the proof search. A plain prefix argument causes the command to prompt for hints and recursion depth, while a numeric prefix argument sets the recursion depth directly. 12

induction: . 21 define

inductive rule: . 15 define

module system: . 15 define

pattern matching: . 9 define

polymorphic type system: . 9 define