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.

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 significiantly 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, and Coq, among many others.

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

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

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

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

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

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

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

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

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. Note for Instructors

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

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

```
I hereby assign copyright in my past and future contributions
to the Software Foundations project to the Author of Record of
each volume or component, to be licensed under the same terms
as the rest of Software Foundations.  I understand that, at
present, the Authors of Record are as follows: For Volumes 1
```

```
and 2, known until 2016 as "Software Foundations" and from
2016 as (respectively) "Logical Foundations" and "Programming
Foundations," the Author of Record is Benjamin Pierce.  For
Volume 3, "Verified Functional Algorithms", the Author of
Record is Andrew W. Appel. For components outside of
designated Volumes (e.g., typesetting and grading tools and
other software infrastructure), the Author of Record is
Benjamin Pierce.
```

To get started, please send an email to Benjamin Pierce, describing yourself and how you plan to use the materials and including 1. the above copyright transfer text and 2. the result of doing "htpasswd -s -n NAME" where NAME is your preferred user name.

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

5. 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

import Prelude.Interfaces
import Prelude.Nat

%access public export

%default total
```

`postulate` is Idris's "escape hatch" that says accept this definition without proof. We use it to mark the 'holes' in the development that should be completed as part of your homework exercises. In practice, `postulate` is 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

– TODO: Edit the following.

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

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

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

2.1. 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."

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

Type the following:

```
nextWeekday : Day -> Day
```

Then with point on `nextWeekday`, call `idris-add-clause` (`M-RET d` in Spacemacs).

```
nextWeekday : Day -> Day
nextWeekday x = ?nextWeekday_rhs
```

With the point on `day`, call `idris-case-split` (`M-RET c` in Spacemacs).

```
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` (`M-RET r` in Spacemacs) to load the `Basics` module with the finished `nextWeekday` definition.

– TODO: Verify that top-level type signatures are optional.

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

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

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

```
Π> nextWeekday Friday
-- Monday : Day
```

```
Π> nextWeekday (nextWeekday Saturday)
-- Tuesday : Day
```

– TODO: 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 Idiris-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
```

– TODO: 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` (`M-RET d`) with the point on the name (`testNextWeekday`) in the type signature and then call `idris-proof-search` (`M-RET p`) with the point on the resultant hole to have Idris solve the proof for you.)

– TODO: 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 `B` of booleans, with members `False` and `True`.

```
||| Boolean Data Type
data B : Type where
 False : B
 True  : B
```

Although we are rolling our own booleans here for the sake of building up everything from scratch, Idiris 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:

```
negb : (b : B) -> B
negb True  = False
negb False = True

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

||| Boolean OR.
orb : (b1 : B) -> (b2 : B) -> B
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
```

– TODO: Edit this

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

```
infixl 4 &&, ||

(&&) : B -> B -> B
(&&) = andb
```

```
||| Boolean OR; infix alias for [`orb`](#Basics.Booleans.orb).
(||) : B -> B -> B
(||) = orb

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

3.1. Exercises: 1 star (nandb). Remove `postulate` and complete the following function; then make sure that the assertions belof can each be verified by Idris. (Remove `postulate` and fill in each proof, following the model of the `orb` tests above.) The function should return `True` if either or both of its inputs `False`.

```
postulate
nandb : (b1 : B) -> (b2 : B) -> B
-- FILL IN HERE

postulate
testNandb1 : (nandb True  False) = True
-- FILL IN HERE

postulate
testNandb2 : (nandb False False) = True
-- FILL IN HERE

postulate
testNandb3 : (nandb False True)  = True
-- FILL IN HERE

postulate
testNandb4 : (nandb True  True)  = False
-- FILL IN HERE
```

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 `negb True` is `B`.

```
Π> :type True
-- True : B
Π> :t negb True : B
-- negb True : B
```

– TODO: Confirm the "function types" wording.

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

```
Π> :t negb
-- negb : B -> B
```

The type of `negb`, written `B -> B` and pronounced "`B` arrow `B`," can be read, "Given an input of type `B`, this function produces an output of type `B`." Similarly, the type of `andb`, written `B -> B -> B`, can be read, "Given two inputs, both of type `B`, this function produces an output of type `B`."

5. Modules

– TODO: Flesh this out and discuss namespaces

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

```
-- FIXME: Figure out how to redefine `Nat` locally here.
-- namespace Playground1
--   %hide Prelude.Nat.Nat
```

6. Numbers

```
namespace Numbers
```

The types we have defined so far are examples of "enumerated types": their definitions explicitly enumerate a finit 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:

```
-- FIXME:
--   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`, `B`, 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 `B`. 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:

```
-- FIXME:
--   pred : (n : Nat) -> Nat
--   pred Z      = Z
--   pred (S n') = n'
```

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

```
minusTwo : (n : Nat) -> Nat
minusTwo Z           = Z
minusTwo (S Z)       = Z
minusTwo (S (S n')) = n'
```

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:

```
Π> S (S (S (S Z)))
-- 4 : Nat
Π> minusTwo 4
-- 2 : Nat
```

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

```
Π> :t S
Π> :t pred
Π> :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) -> B
evenb Z           = True
```

```
evenb (S Z)      = False
evenb (S (S n')) = evenb n'
```

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) -> B
oddb n = negb (evenb n)

testOddb1 : oddb 1 = True
testOddb1 = Refl
testOddb2 : oddb 4 = False
testOddb2 = Refl
```

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

```
-- FIXME: Figure out how to redefine `plus`, `mult` and `minus` locally here.
-- namespace Playground2
--   %hide Prelude.Nat.Nat

--   plus : (n : Nat) -> (m : Nat) -> Nat
--   plus Z      m = m
--   plus (S n') m = S (plus n' m)
```

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

```
Π> 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 n') = plus m (mult n' m)

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

You can match two expression at once:

```
--   minus (n, m : Nat) -> Nat
--   minus Z       _       = Z
--   minus n       Z       = n
--   minus (S n') (S m') = minus n' m'
```

– TODO: Verify this.

The _ in the first line is a *wilcard 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)
```

6.1. 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.

```
postulate
factorial : (n : Nat) -> Nat
-- FILL IN HERE

postulate
testFactorial1 : factorial 3 = 6
-- FILL IN HERE

postulate
testFactorial2 : factorial 5 = mult 10 12
-- FILL IN HERE
```

– FIXME: This breaks things… – We can make numerical expressions a little easier to read and write by – introducing *notations* for addition, multiplication, and subtraction.

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

```
Π> :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`.

The beq_nat function tests Natural numbers for equality, yielding a boolean.

```
|||  Test natural numbers for equality.
beq_nat : (n, m : Nat) -> B
beq_nat Z      Z      = True
beq_nat Z      (S m') = False
beq_nat (S n') Z      = False
beq_nat (S n') (S m') = beq_nat n' m'
```

The leb 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.
leb : (n, m : Nat) -> B
leb Z      m      = True
leb (S n') Z      = False
leb (S n') (S m') = leb n' m'

testLeb1 : leb 2 2 = True
testLeb1 = Refl
testLeb2 : leb 2 4 = True
testLeb2 = Refl
testLeb3 : leb 4 2 = False
testLeb3 = Refl
```

6.2. Exercise: 1 star (blt_nat). The blt_nat function tests Natural 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.

```
postulate
blt_nat : (n, m : Nat) -> B
-- FILL IN HERE

postulate
test_blt_nat_1 : blt_nat 2 2 = False
-- FILL IN HERE
postulate
test_blt_nat_2 : blt_nat 2 4 = True
-- FILL IN HERE
postulate
test_blt_nat_3 : blt_nat 4 2 = False
-- FILL IN HERE
```

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 pronounces "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

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.

```
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`.

– FIXME 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.

7.1. Exercise: 1 star (plus_id_exercise). Remove "`postulate`" and fill in the proof.

```
postulate
plus_id_exercise : (n, m, o : Nat) -> (n = m) -> (m = o)
                   -> n + m = m + o
```

The `postulate` command tells Idris that we want to skip trying to prove this theorem and just accept it as a given. This can be useful for developing longer proofs, since we can state subsidiary lemmas that we believe will be useful for making some larger argument, use `postulate` to accept them on faith for the moment, and continue working on the main argument until we are sure it makes sense; then we can go back and fill in the proofs we skipped. Be careful, though: every time you say `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.

7.2. Exercise: 2 starts (mult_S_1).

```
postulate
mult_S_1 : (n, m : Nat) -> (m = S n)
         -> m * (1 + n) = m * m
```

8. 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) -> beq_nat (n + 1) 0 = False
plus_1_neq_0_firsttry n = Refl -- does nothing!
```

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

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

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

– TODO: mention case splitting interactively in Emacs, Atom, etc.

```
plus_1_neq_0 : (n : Nat) -> beq_nat (n + 1) 0 = False
plus_1_neq_0 Z      = Refl
plus_1_neq_0 (S n') = 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 `beq_nat (S n' + 1) 0` to `False` by first rewriting `(S n' + 1)` to `S (n' + 1)`, then unfolding `beq_nat`, 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.
negb_involutive : (b : B) -> negb (negb b) = b
negb_involutive True  = Refl
negb_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 : B) -> andb b c = andb 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 : B) -> c = andb c True
andb_commutative'_rhs_1 True  = Refl
andb_commutative'_rhs_1 False = Refl

andb_commutative'_rhs_2 : (c : B) -> False = andb c False
andb_commutative'_rhs_2 True  = Refl
andb_commutative'_rhs_2 False = Refl

andb_commutative' : (b, c : B) -> andb b c = andb c b
andb_commutative' True  = andb_commutative'_rhs_1
andb_commutative' False = andb_commutative'_rhs_2
```

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

```
postulate
andb_true_elim_2 : (b, c : B) -> (andb b c = True) -> c = True
```

8.2. Exercise: 1 star (zero_nbeq_plus.

```
postulate
zero_nbeq_plus_1 : (n : Nat) -> beq_nat 0 (n + 1) = False
```

– TODO: discuss associativity

9. 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.

– TODO: verify the previous claims

10. More Exercises

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

```
postulate
identity_fn_applied_twice : (f : B -> B)
                          -> ((x : B) -> f x = x)
                          -> (b : B) -> f (f b) = b
```

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

```
postulate
andb_eq_orb : (b, c : B)
            -> (andb b c = orb b c)
            -> b = c
```

10.2. 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.

CHAPTER 3

Induction: Proof by 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 and Sublime Text 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.

1. Proof by Induction

We proved in the last chapter that `0` is a netural 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 `n'`, if `p in'` holds, then so does `p (S n')`;
- 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 n') =
  let inductiveHypothesis = plus_n_Z n' 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 n'` and the goal becomes `S n' = S (plus n' 0)`. Then we define the inductive hypothesis, `n' = n' + 0`, which can be written as `plus_n_Z n'`, and the goal follows from it.

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

1.1. 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
```

□