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.

Contents

CHAPTER 1

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
```

`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*.

```
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | 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.

```
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.

```
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

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

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

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 : Bool) -> Bool
negb True  = False
negb 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
```

– 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 &&, ||

(&&) : Bool -> Bool -> Bool
(&&) = andb

(||) : Bool -> Bool -> Bool
(||) = 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 : Bool) -> (b2 : Bool) -> Bool
-- 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 `Bool`.

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

– 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 : Bool -> Bool
```

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

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

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`, `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:

```
-- 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. Althrough 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 recursivelt check whether `n-2` is even.

```
evenb : (n : Nat) -> Bool
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:

```
oddb : (n : Nat) -> Bool
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
```

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 `beqNat` function tests `Nat`ural numbers for `eq`uality, yielding a `b`oolean.

```
beqNat : (n, m : Nat) -> Bool
beqNat Z      Z      = True
beqNat Z      (S m') = False
beqNat (S n') Z      = False
beqNat (S n') (S m') = beqNat n' m'
```

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

```
leb : (n, m : Nat) -> Bool
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 (bltNat). The `bltNat` 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.

```
postulate
bltNat : (n, m : Nat) -> Bool
-- FILL IN HERE

postulate
testBltNat1 : bltNat 2 2 = False
-- FILL IN HERE
postulate
testBltNat2 : bltNat 2 4 = True
-- FILL IN HERE
postulate
testBltNat3 : bltNat 4 2 = False
-- FILL IN HERE
```

7. Proof by Simplification

– TODO: Translate this section and those that follow.