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

Basics

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

%hide (&&)
%hide (||)

%access public export

{- REMINDER:

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

(See the [Preface] for why.)

-}
```

– TODO: discuss postulate

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.

– 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.

Enumerated Types.

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.

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.

```
nextWeekday Friday
-- Monday : Day

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

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
```

(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: 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."

– 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.