Type-Driven Development with Idris

by Edwin Brady

Chapter 1

brief contents

1 Overview

This chapter covers

- Introducing type-driven development
- The essence of pure functional programming
- First steps with Idris

This book teaches a new approach to building robust software, *type-driven development*, using the Idris programming language. Traditionally, types are seen as a tool for checking for errors, with the programmer writing a complete program first and using either the compiler or the runtime system to detect type errors. In type-driven development, we use types as a tool for constructing programs. We put the *type* first, treating it as a plan for a program, and use the compiler and type checker as our assistant, guiding us to a complete and working program that satisfies the type. The more expressive the type is that we give up front, the more confidence we can have that the resulting program will be correct.

TYPES AND TESTS The name "type-driven development" suggests an analogy to test-driven development. There's a similarity, in that writing tests first helps establish a program's purpose and whether it satisfies some basic requirements. The difference is that, unlike tests, which can usually only be used to show the *presence* of errors, types (used appropriately) can show the *absence* of errors. But although types *reduce* the need for tests, they rarely eliminate it entirely.

Idris is a relatively young programming language, designed from the beginning to support type-driven development. A prototype implementation first appeared in 2008, with development of the current implementation beginning in 2011. It builds on decades of research into the theoretical and practical foundations of programming languages and type systems.

In Idris, types are a *first-class* language construct. Types can be manipulated, used, passed as arguments to functions, and returned from functions just like any other value, such as numbers, strings, or lists. This is a simple but powerful idea:

- It allows relationships to be expressed between values; for example, that two lists have the same length.
- It allows assumptions to be made explicit and checkable by the compiler. For example, if you assume that a list is non-empty, Idris can ensure this assumption always holds *before the program is run.*
- If desired, it allows program behavior to be formally stated and proven correct.

In this chapter, I'll introduce the Idris programming language and give a brief tour of its features and environment. I'll also provide an overview of type-driven development, discussing why types matter in programming languages and how they can be used to guide software development. But first, it's important to understand exactly what we mean when we talk about "types."

1.1 What is a type?

We're taught from an early age to recognize and distinguish *types* of objects. As a young child, you may have had a shape-sorter toy. This consists of a box with variously shaped holes in the top (see figure 1.1) and some shapes that fit through the holes. Sometimes they're equipped with a small plastic hammer. The idea is to fit each shape (think of this as a "value") into the appropriate hole (think of this as a "type"), possibly with coercion from the hammer.

In programming, types are a means of classifying values. For example, the values `94`, `"thing"`, and `[1,2,3,4,5]` could respectively be classified as an integer, a string, and a list of integers. Just as you can't put a square shape in a round hole in the shape sorter, you can't use a string like `"thing"` in a part of a program where you need an integer.

All modern programming languages classify values by type, although they differ enormously in when and how they do so (for example, whether they're checked statically at compile time or dynamically at runtime, whether conversions between types are automatic or not, and so on).

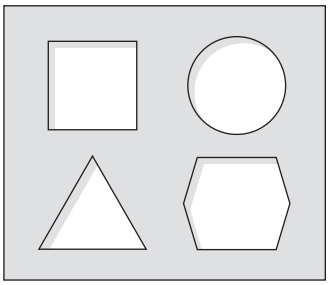


Figure 1.1 The top of a shape-sorter toy. The shapes correspond to the types of objects that will fit through the holes.

Types serve several important roles:

- For a *machine*, types describe how bit patterns in memory are to be interpreted.
- For a *compiler* or *interpreter*, types help ensure that bit patterns are interpreted consistently when a program runs.
- For a *programmer*, types help name and organize concepts, aiding documentation and supporting interactive editing environments.

From our point of view in this book, the most important purpose of types is the third. Types help programmers in several ways:

- By allowing for the naming and organization of concepts (such as `Square`, `Circle`, `Triangle`, and `Hexagon`)
- By providing explicit documentation of the purposes of variables, functions, and programs
- By driving code completion in an interactive editing environment

As you'll see, type-driven development makes extensive use of code completion in particular. Although all modern, statically typed languages support code completion to some extent, the expressivity of the Idris type system leads to powerful automatic code generation.

1.2 Introducing type-driven development

Type-driven development is a style of programming in which we write types first and use those types to guide the definition of functions. The overall process is to write the necessary data types, and then, for each function, do the following:

1. Write the input and output types.
2. Define the function, using the structure of the input types to guide the implementation.
3. Refine and edit the type and function definition as necessary.

In type-driven development, instead of thinking of types in terms of *checking*, with the type checker criticizing you when you make a mistake, you can think of types as being a *plan*, with the type checker acting as your guide, leading you to a working, robust program. Starting with a type and an empty function body, you gradually add details to the definition until it's complete, regularly using the

Types as models

When you write a program, you'll often have a conceptual model in your head (or, if you're disciplined, even on paper) of how it's supposed to work, how the components interact, and how the data is organized. This model is likely to be quite vague at first and will become more precise as the program evolves and your understanding of the concept develops.

Types allow you to make these models explicit in code and ensure that your implementation of a program matches the model in your head. Idris has an expressive type system that allows you to describe a model as precisely as you need, and to refine the model at the same time as developing the implementation.

compiler to check that the program so far satisfies the type. Idris, as you'll soon see, strongly encourages this style of programming by allowing *incomplete* function definitions to be checked, and by providing an expressive language for describing types.

To illustrate further, in this section I'll show some examples of how you can use types to describe in detail what a program is intended to do: matrix arithmetic, modeling an automated teller machine (ATM), and writing concurrent programs. Then, I'll summarize the process of type-driven development and introduce the concept of *dependent types*, which will allow you to express detailed properties of your programs.

1.2.1 Matrix arithmetic

A *matrix* is a rectangular grid of numbers, arranged in rows and columns. They have several scientific applications, and in programming they have applications in cryptography, 3D graphics, machine learning, and data analytics. The following, for example, is a 3 × 4 matrix:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

You can implement various arithmetic operations on matrices, such as addition and multiplication. To *add* two matrices, you add the corresponding elements, as you see here:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix} = \begin{pmatrix} 8 & 10 \\ 12 & 14 \\ 16 & 18 \end{pmatrix}$$

When programming with matrices, if you begin by defining a `Matrix` data type, then addition requires two inputs of type `Matrix` and gives an output of type `Matrix`. But because adding matrices involves adding corresponding elements of the inputs, what happens if the two inputs have different dimensions, as here?

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} = ???$$

It's likely that if you're trying to add matrices of different dimensions, then you've made a mistake somewhere. So, instead of using a `Matrix` type, you could *refine* the

type so that it includes the dimensions of the matrix, and require that the two input matrices have the same dimensions:

- The first example of a 3 × 4 matrix now has type `Matrix 3 4`.
- The first (correct) example of addition takes two inputs of type `Matrix 3 2` and gives an output of type `Matrix 3 2`.

By including the dimensions in the type of a matrix, you can describe the input and output types of addition in such a way that it's a *type error* to try to add matrices of different sizes. If you try to add a `Matrix 3 2` and a `Matrix 2 2`, your program won't compile, let alone run.

If you include the dimensions of a matrix in its type, then you need to think about the relationship between the dimensions of the input and output for *every* matrix operation. For example, transposing a matrix involves switching the rows to columns and vice versa, so if you transpose a 3 × 2 matrix, you'll end up with a 2 × 3 matrix:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \text{ ... transposed to ... } \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

The input type of this transposition is `Matrix 3 2`, and the output type is `Matrix 2 3`.

In general, rather than giving *exact* dimensions in the type, we'll use *variables* to describe the relationship between the dimensions of the inputs and the dimensions of the outputs. Table 1.1 shows the relationships between the dimensions of inputs and outputs for three matrix operations: addition, multiplication, and transposition.

Table 1.1 Input and output types for matrix operations. The names `x`, `y`, and `z` describe, in general, how the dimensions of the inputs and outputs are related.

Operation	Input types	Output type
Add	`Matrix x y, Matrix x y`	`Matrix x y`
Multiply	`Matrix x y, Matrix y z`	`Matrix x z`
Transpose	`Matrix x y`	`Matrix y x`

We'll look at matrices in depth in chapter 3, where we'll work through an implementation of matrix transposition in detail.

1.2.2 An automated teller machine

As well as using types to describe the relationships between the inputs and outputs of functions, as with matrix operations, you can describe precisely *when* operations are valid. For example, if you're implementing software to drive an ATM, you'll want to guarantee that the machine will dispense cash only after a user has entered a card and validated their personal identification number (PIN).

To see how this works, we'll need to consider the possible *states* that an ATM can be in:

- `Ready`—The ATM is ready and waiting for a user to insert a card.
- `CardInserted`—The ATM is waiting for a user, having inserted a card, to enter their PIN.
- `Session`—A validated session is in progress, with the ATM, having validated the user's PIN, ready to dispense cash.

An ATM supports several basic operations, each of which is valid only when the machine is in a specific state, and each of which might change the state of the machine, as illustrated in figure 1.2. These are the basic operations:

- `InsertCard`—Waits for the user to insert a card
- `EjectCard`—Ejects a card from the machine
- `GetPIN`—Prompts the user to enter a PIN
- `CheckPIN`—Checks whether an entered PIN is correct
- `Dispense`—Dispenses cash

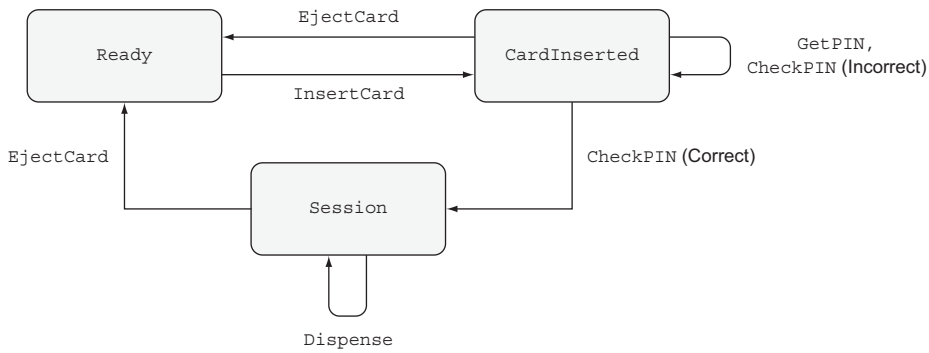


Figure 1.2 The states and valid operations on an ATM. Each operation is valid only in specific states and can change the state of the machine. `CheckPIN` changes the state only if the entered PIN is correct.

Whether an operation is valid or not depends on the state of the machine. For example, `InsertCard` is valid only in the `Ready` state, because that's the only state where there's no card already in the machine. Also, `Dispense` is valid only in the `Session` state, because that's the only state where there's a validated card in the machine.

Furthermore, executing one of these operations can *change* the state of the machine. For example, `InsertCard` changes the state from `Ready` to `CardInserted`, and `CheckPIN` changes the state from `CardInserted` to `Session`, provided that the entered PIN is correct.

STATE MACHINES AND TYPES Figure 1.2 illustrates a *state machine*, describing how operations affect the overall state of a system. State machines are often present, implicitly, in real-world systems. For example, when you `open`, `read`,

and then `close` a file, you change the state of the file with the `open` and `close` operations. As you'll see in chapter 13, types allow you to make these state changes explicit, guarantee that you'll execute operations only when they're valid, and help you use resources correctly.

By defining precise types for each of the operations on the ATM, you can guarantee, by type checking, that the ATM will execute only valid operations. If, for example, you try to implement a program that dispenses cash without validating a PIN, the program won't compile. By defining valid state transitions explicitly in types, you get strong and machine-checkable guarantees about the correctness of their implementation. We'll look at state machines in chapter 13, and then implement the ATM example in chapter 14.

1.2.3 Concurrent programming

A *concurrent* program consists of multiple processes running at the same time and coordinating with each other. Concurrent programs can be responsive and continue to interact with a user while a large computation is running. For example, a user can continue browsing a web page while a large file is downloading. Moreover, by writing concurrent programs we can take full advantage of the processor power of modern CPUs, dividing work among multiple processes on separate CPU cores.

In Idris, processes coordinate with each other by sending and receiving *messages*. Figure 1.3 shows one way this can work, with two processes, `main` and `adder`. The `adder` process waits for a *request* to add numbers from other processes. After it receives a message from `main` asking it to add two numbers, it sends a *response* back with the result.

Despite its advantages, however, concurrent programming is notoriously error prone. The need for processes to interact with each other can greatly increase a system's complexity. For each process, you need to ensure that the messages it sends and

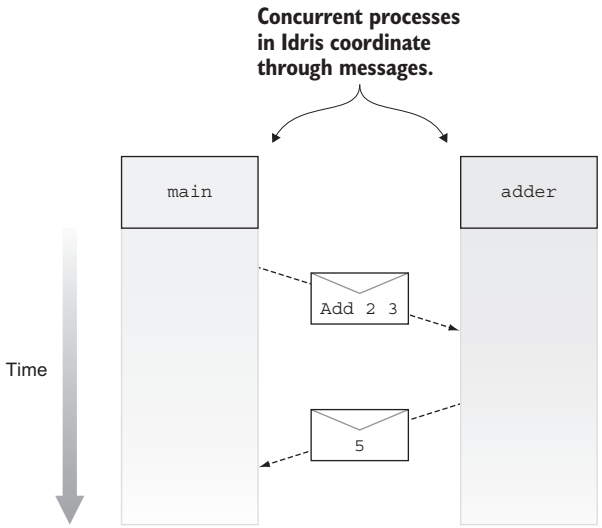


Figure 1.3 Two interacting concurrent processes, `main` and `adder`. The `main` process sends a request to `adder`, which then sends a response back to `main`.

receives are properly coordinated with other processes. If, for example, `main` and `adder` aren't properly coordinated and each is expecting to receive a message from the other at the same time, they'll *deadlock*.

> **TYPES VERSUS TESTING FOR CONCURRENT PROGRAMS** Testing a concurrent program is difficult because, unlike a purely sequential program, there's no guarantee about the order in which operations from different processes will execute. Even if two processes are correctly coordinated when you run a test once, there's no guarantee they'll be correctly coordinated when you next run the test. On the other hand, if you can express the coordination between processes in *types*, you can be sure that a concurrent program that type-checks has properly coordinated processes.

When you write concurrent programs, you'll ideally have a model of how processes should interact. Using types, you can make this model explicit in code. Then, if a concurrent program type-checks, you'll know that it correctly follows the model. In particular, you can do two things:

- Define an interface for `adder` that describes the form of messages it will handle.
- Define a protocol that defines the order of message passing, ensuring that `main` will *always* send a message to `adder` and then receive a reply, and `adder` will always do the opposite.

Concurrent programming is an extensive topic, and there are several ways you can use types to model coordination between processes. We'll look at one example of how to do this in chapter 15.

1.2.4 Type, define, refine: the process of type-driven development

In each of these introductory examples, we've discussed in general terms how we might *model* a system: by describing the valid forms of inputs and outputs for matrix operations, the valid states of an interactive system, or the order of transmission of messages between concurrent processes. In each case, to implement the system, you start by trying to find a type that captures the important details of the model, and then define functions to work with that type, refining the type as necessary.

To put it succinctly, you can characterize type-driven development as an iterative process of *type, define, refine*: writing a type, implementing a function to satisfy that type, and refining the type or definition as you learn more about the problem.

With matrix addition, for example, you do the following:

- *Type*—Write a `Matrix` data type, and use it as the input and output types for an addition function.
- *Define*—Write an addition function that satisfies its input and output types.
- *Refine*—Notice that the input and output types for your addition function allow you to give invalid inputs with different dimensions, and then make the type more precise by including the dimensions of the matrices.

In general, you'll write a type to represent the system you're modeling, define functions using that type, and then refine the type and definition as necessary to capture any missing properties. You'll see a lot more of this type-define-refine process throughout this book, both on a small scale when implementing individual functions, and on a larger scale when deciding how to write function and data types.

1.2.5 Dependent types

In the matrix arithmetic example, we began with a `Matrix` type and then refined it to include the number of rows and columns. This means, for example, that `Matrix 3 4` is the type of 3 × 4 matrices. In this type, `3` and `4` are ordinary values. A *dependent type*, such as `Matrix`, is a type that's calculated from some other values. In other words, it *depends on* other values.

By including values in a type like this, you can make types as precise as required. For example, some languages have a simple list type, describing lists of objects. You can make this more precise by parameterizing over the element type: a generic list of strings is more precise than a simple list and differs from a list of integers. You can be more precise still with a dependent type: a list of 4 strings differs from a list of 3 strings.

Table 1.2 illustrates how types in Idris can have differing levels of precision even for fundamental operations such as appending lists. Suppose you have two specific input lists of strings:

```
["a", "b", "c", "d"]
["e", "f", "g"]
```

When you append them, you'll expect the following output list:

```
["a", "b", "c", "d", "e", "f", "g"]
```

Using a *simple* type, both input lists have type `AnyList`, as does the output list. Using a *generic* type, you can specify that the input lists are both lists of strings, as is the output list. The more-precise types mean that, for example, the output is clearly related to the input in that the element type is unchanged. Finally, using a *dependent* type, you can specify the sizes of the input and output lists. It's clear from the type that the length of the output list is the sum of the lengths of the input lists. That is, a list of 3 strings appended to a list of 4 strings results in a list of 7 strings.

Table 1.2 Appending specific typed lists. Unlike simple types, where there's no difference between the input and output list types, dependent types allow the length to be encoded in the type.

	Input `["a", "b", "c", "d"]`	Input `["e", "f", "g"]`	Output type
Simple	`AnyList`	`AnyList`	`AnyList`
Generic	`List String`	`List String`	`List String`
Dependent	`Vect 4 String`	`Vect 3 String`	`Vect 7 String`

LISTS AND VECTORS The syntax for the types in table 1.2 is valid Idris syntax. Idris provides several ways of building list types, with varying levels of precision. In the table, you can see two of these, `List` and `Vect`. `AnyList` is included in the table purely for illustrative purposes and is not defined in Idris. `List` encodes generic lists with no explicit length, and `Vect` (short for "vector") encodes lists with the length explicitly in the type. You'll see much more of both these types throughout this book.

Table 1.3 illustrates how the input and output types of an `append` function can be written with increasing levels of precision in Idris. Using *simple* types, you can write the input and output types as `AnyList`, suggesting that you have no interest in the types of the elements of the list. Using *generic* types, you can write the input and output types as `List elem`. Here, `elem` is a *type variable* standing for the element types. Because the type variable is the same for both inputs and the output, the types specify that both the input lists and the output list have a consistent element type. If you append two lists of integers, the types guarantee that the output will also be a list of integers. Finally, using *dependent* types, you can write the inputs as `Vect n elem` and `Vect m elem`, where `n` and `m` are variables representing the *length* of each list. The output type specifies that the resulting length will be the sum of the lengths of the inputs.

Table 1.3 Appending typed lists, in general. Type variables describe the relationships between the inputs and outputs, even though the exact inputs and outputs are unknown.

	Input 1 type	Input 2 type	Output type
Simple	`AnyList`	`AnyList`	`AnyList`
Generic	`List elem`	`List elem`	`List elem`
Dependent	`Vect n elem`	`Vect m elem`	`Vect (n + m) elem`

TYPE VARIABLES Types often contain *type variables*, like `n`, `m`, and `elem` in table 1.3. These are very much like parameters to generic types in Java or C#, but they're so common in Idris that they have a very lightweight syntax. In general, concrete type names begin with an uppercase letter, and type variable names begin with a lowercase letter.

In the dependent type for the `append` function in table 1.3, the parameters `n` and `m` are ordinary numeric values, and the `+` operator is the normal addition operator. All of these could appear in programs just as they've appeared here in the types.

Introductory exercises

Throughout this book, exercises will help reinforce the concepts you've learned. As a warm-up, take a look at the following selection of function specifications, given purely in the form of input and output types. For each of them, suggest possible operations

that would satisfy the given input and output types. Note that there could be more than one answer in each case.

1. Input type: `Vect n elem`
   Output type: `Vect n elem`
2. Input type: `Vect n elem`
   Output type: `Vect (n * 2) elem`
3. Input type: `Vect (1 + n) elem`
   Output type: `Vect n elem`
4. Assume that `Bounded n` represents a number between zero and `n - 1`.
   Input types: `Bounded n`, `Vect n elem`
   Output type: `elem`

1.3 Pure functional programming

Idris is a *pure functional* programming language, so before we begin exploring Idris in depth, we should look at what it means for a language to be *functional*, and what we mean by the concept of *purity*. Unfortunately, there's no universally agreed-on definition of exactly what it means for a programming language to be *functional*, but for our purposes we'll take it to mean the following:

- Programs are composed of functions.
- Program execution consists of the evaluation of functions.
- Functions are a first-class language construct.

This differs from an *imperative* programming language primarily in that functional programming is concerned with the evaluation of functions, rather than the execution of statements.

In a *pure* functional language, the following are also true:

- Functions don't have side effects such as modifying global variables, throwing exceptions, or performing console input or output.
- As a result, for any specific inputs, a function will always give the same result.

You may wonder, very reasonably, how it's possible to write any useful software under these constraints. In fact, far from making it more difficult to write realistic programs, pure functional programming allows you to treat tricky concepts such as state and exceptions with the respect they deserve. Let's explore further.

1.3.1 Purity and referential transparency

The key property of a pure function is that the same inputs always produce the same result. This property is known as *referential transparency*. An expression (such as a function call) in a function is referentially transparent if it can be replaced with its result without changing the behavior of the function. If functions produce only results, with no side effects, this property is clearly true. Referential transparency is a very useful concept in type-driven development, because if a function has no side effects and is

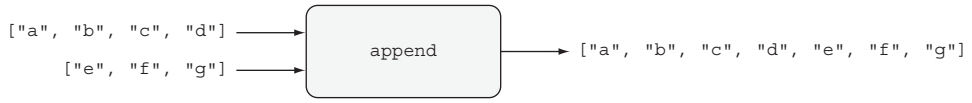


Figure 1.4 A pure function, taking inputs and producing outputs with no observable side effects

defined entirely by its inputs and outputs, then you can look at its input and output types and have a clear idea of the limits of what the function can do.

Figure 1.4 shows example inputs and outputs for the `append` function. It takes two inputs and produces a result, but there's no interaction with a user, such as reading from the keyboard, and no informative output, such as logging or progress bars.

Figure 1.5 shows pure functions in general. There can be no observable side effects when running these programs, other than perhaps making the computer slightly warmer or taking a different amount of time to run.

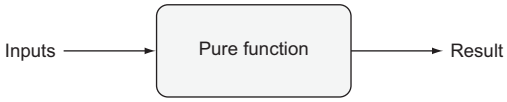


Figure 1.5 Pure functions, in general, take only inputs and have no observable side effects.

Pure functions are very common in practice, particularly for constructing and manipulating data structures. It's possible to reason about their behavior because the function always gives the same result for the same inputs; these functions are important components of larger programs. The preceding `append` function is pure, and it's a valuable component for any program that works with lists. It produces a list as a result, and because it's pure, you know that it won't require any input, output any logging, or do anything destructive like delete files.

1.3.2 Side-effecting programs

Realistically, programs must have side effects in order to be useful, and you're always going to have to deal with unexpected or erroneous inputs in practical software. At first, this would seem to be impossible in a pure language. There is a way, however: pure functions may not be able to perform side effects, but they can *describe* them.

Consider a function that reads two lists from a file, appends them, prints the resulting list, and returns it. The following listing outlines this function in imperative-style pseudocode, using simple types.

Listing 1.1 Appending lists read from a file (pseudocode)

```
List appendFromFile(File h) {
    list1 = readListFrom(h)
    list2 = readListFrom(h)

    result = append(list1, list2)
    print(result)
```

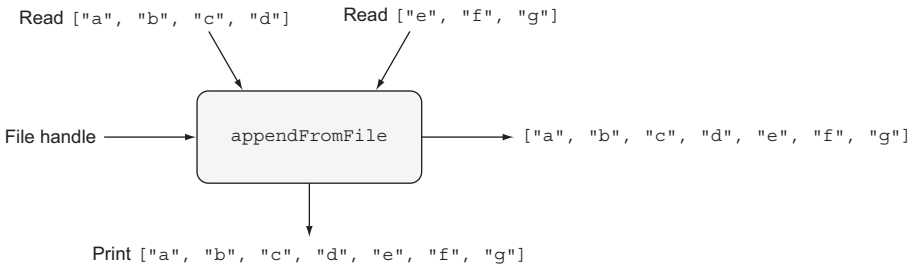


Figure 1.6 A side-effecting program, reading inputs from a file, printing the result, and returning the result

```
    return result
}
```

This program takes a file handle as an input and returns a `List` with some side effects. It reads two lists from the given file and prints the list before returning. Figure 1.6 illustrates this for the situation when the file contains the two lists `["a", "b", "c", "d"]` and `["e", "f", "g"]`.

The `appendFromFile` function doesn't satisfy the referential transparency property. Referential transparency requires that an expression can be replaced by its result without changing the program's behavior. Here, however, replacing a call to `appendFromFile` with its result means that nothing will be read from the file, and nothing will be output to the screen. The function's *input* and *output* types tell us that the input is a file and the output is a list, but nothing in the type describes the side effects the function may execute.

In pure functional programming in general, and Idris in particular, you can solve this problem by writing functions that *describe* side effects, rather than functions that *execute* them, and defer the details of execution to the compiler and runtime system. We'll explore this in greater detail in chapter 5; for now, it's sufficient to recognize that a program with side effects has a type that makes this explicit. For example, there's a distinction between the following:

- `String` is the type of a program that results in a `String` and is guaranteed to perform *no* input or output as side effects.
- `IO String` is the type of a program that describes a sequence of input and output operations that result in a `String`.

Type-driven development takes this idea much further. As you'll see from chapter 12 onward, you can define types that describe the specific side effects a program can have, such as console interaction, reading and writing global state, or spawning concurrent processes and sending messages.

1.3.3 Partial and total functions

Idris supports an even stronger property than purity for functions, making a distinction between *partial* and *total* functions. A *total* function is guaranteed to produce a result, meaning that it will return a value in a finite time for every possible well-typed input, and it's guaranteed not to throw any exceptions. A *partial* function, on the other hand, might not return a result for some inputs. Here are a couple of examples:

- The `append` function is *total* for finite lists, because it will always return a new list.
- The function that returns the first element of a list is *partial*, because it's not defined if the list is empty, and it will therefore crash.

TOTAL FUNCTIONS AND LONG-RUNNING PROGRAMS A total function is guaranteed to produce a *finite prefix* of a potentially infinite result. As you'll see in chapter 11, you can write command shells or servers as total functions that guarantee a response for every user input, indefinitely.

The distinction is important because knowing that a function is total allows you to make much stronger claims about its behavior based on its type. If you have a function with a return type of `String`, for example, you can make different claims depending on whether the function is partial or total.

- *If it's total*—It will return a value of type `String` in finite time.
- *If it's partial*—If it doesn't crash or enter an infinite loop, the value it returns will be a `String`.

In most modern languages, we must assume that functions are partial and can therefore only make the latter, weaker, claim. Idris checks whether functions are total, so we can therefore often make the former, stronger, claim.

Total functions and the halting problem

The *halting problem* is the problem of determining whether a program terminates for some specific input. Thanks to Alan Turing, we know that it's not possible to write a program that solves the halting problem in general. Given this, it's reasonable to wonder how Idris can determine that a function is total, which is essentially checking that a function terminates for *all* inputs.

Although it can't solve the problem in general, Idris can identify a large class of functions that *are* definitely total. You'll learn more about how it does so, along with some techniques for writing total functions, in chapters 10 and 11.

A useful pattern in type-driven development is to write a type that precisely describes the valid states of a system (like the ATM in section 1.2.2) and that constrains the oper-

ations the system is allowed to perform. A total function with that type is then *guaranteed* by the type checker to perform those operations as precisely as the type requires.

1.4 A quick tour of Idris

The Idris system consists of an interactive environment and a batch mode compiler. In the interactive environment, you can load and type-check source files, evaluate expressions, search libraries, browse documentation, and compile and run complete programs. We'll use these features extensively throughout this book.

In this section, I'll briefly introduce the most important features of the environment, which are evaluation and type checking, and describe how to compile and run Idris programs. I'll also introduce the two most distinctive features of the Idris language itself:

- *Holes*, which stand for incomplete programs
- The use of types as *first-class* language constructs

As you'll see, by using holes you can define functions incrementally, asking the type checker for contextual information to help complete definitions. Using first-class types, you can be very precise about what a function is intended to do, and even ask the type checker to fill in some of the details of functions for you.

1.4.1 The interactive environment

Much of your interaction with Idris will be through an interactive environment called the *read-eval-print loop*, typically abbreviated as REPL. As the name suggests, the REPL will *read* input from the user, usually in the form of an expression, *evaluate* the expression, and then *print* the result.

Once Idris is installed, you can start the REPL by typing `idris` at a shell prompt. You should see something like the following:

```
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 1.0
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Idris is free software with ABSOLUTELY NO WARRANTY.
For details type :warranty.
Idris>
```

INSTALLING IDRIS You can find instructions on how to download and install Idris for Linux, OS X, or Windows in appendix A.

You can enter expressions to be evaluated at the `Idris>` prompt. For example, arithmetic expressions work in the conventional way, with the usual precedence rules (that is, `*` and `/` have higher precedence than `+` and `-`):

```
Idris> 2 + 2
4 : Integer
```

```
Idris> 2.1 * 20
42.0 : Double

Idris> 6 + 8 * 11
94 : Integer
```

You can also manipulate `String`s. The `++` operator concatenates `String`s, and the `reverse` function reverses a `String`:

```
Idris> "Hello" ++ " " ++ "World!"
"Hello World!" : String

Idris> reverse "abcdefg"
"gfedcba" : String
```

Notice that Idris prints not only the result of evaluating the expression, but also its *type*. In general, if you see something of the form `x : T`—some expression `x`, a colon, and some other expression `T`—this can be read as "`x` has type `T`." In the previous examples, you have the following:

- `4` has type `Integer`.
- `42.0` has type `Double`.
- `"Hello World!"` has type `String`.

1.4.2 Checking types

The REPL provides a number of *commands*, all prefixed by a colon. One of the most commonly useful is `:t`, which allows you to check the *types* of expressions without evaluating them:

```
Idris> :t 2 + 2
2 + 2 : Integer

Idris> :t "Hello!"
"Hello!" : String
```

Types, such as `Integer` and `String`, can be manipulated just like any other value, so you can check their types too:

```
Idris> :t Integer
Integer : Type

Idris> :t String
String : Type
```

It's natural to wonder what the type of `Type` itself might be. In practice, you'll never need to worry about this, but for the sake of completeness, let's take a look:

```
Idris> :t Type
Type : Type 1
```

That is, `Type` has type `Type 1`, `Type 1` has type `Type 2`, and so on forever, as far as we're concerned. The good news is that Idris will take care of the details for you, and you can always write `Type` alone.

1.4.3 Compiling and running Idris programs

As well as evaluating expressions and inspecting the types of functions, you'll want to be able to compile and run complete programs. The following listing shows a minimal Idris program.

Listing 1.2 Hello, Idris World! (Hello.idr)

```
module Main                          <-- Module header

main : IO ()                         <-- Function declaration
main = putStrLn "Hello, Idris World!"    <-- Function definition
```

At this stage, there's no need to worry too much about the syntax or how the program works. For now, you just need to know that Idris source files consist of a module header and a collection of function and data type definitions. They may also import other source files.

> **WHITESPACE SIGNIFICANCE** Whitespace is significant in Idris, so when you type listing 1.2, make sure there are no spaces at the beginning of each line.

Here, the module is called `Main`, and there's only one function definition, called `main`. The entry point to any Idris program is the `main` function in the `Main` module.

To run the program, follow these steps:

1. Create a file called Hello.idr in a text editor.[1] Idris source files all have the extension .idr.
2. Enter the code in listing 1.2.
3. In the working directory where you saved Hello.idr, start up an Idris REPL with the command `idris Hello.idr`.
4. At the Idris prompt, type `:exec`.

If all is well, you should see something like the following:

```
$ idris Hello.idr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 1.0
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Idris is free software with ABSOLUTELY NO WARRANTY.
For details type :warranty.
Type checking ./Hello.idr
```

[1] I recommend Atom because it has a mode for interactive editing of Idris programs, which we'll use in this book.

```
*Hello> :exec
Hello, Idris World
```

Here, `$` stands for your shell prompt. Alternatively, you can create a standalone executable by invoking the `idris` command with the `-o` option, as follows:

```
$ idris Hello.idr -o Hello
$ ./Hello
Hello, Idris World
```

> **THE REPL PROMPT** The REPL prompt, by default, tells you the name of the file that's currently loaded. The `Idris>` prompt indicates that no file is loaded, whereas the prompt `*Hello>` indicates that the Hello.idr file is loaded.

1.4.4 Incomplete definitions: working with holes

Earlier, I compared working with types and values to inserting shapes into a shape-sorter toy. Much as the square shape will only fit through a square hole, the argument `"Hello, Idris World!"` will only fit into a function in a place where a `String` type is expected.

Idris functions themselves can contain *holes*, and a function with a hole is *incomplete*. Only a value of an appropriate type will fit into the hole, just as a square shape will only fit into a square hole in the shape sorter. Here's an incomplete implementation of the "Hello, Idris World!" program:

```
module Main

main : IO ()
main = putStrLn ?greeting
```

?greeting is a hole, standing for a missing part of the program.

If you edit Hello.idr to replace the string `"Hello, Idris World!"` with `?greeting` and load it into the Idris REPL, you should see something like the following:

```
Type checking ./Hello.idr
Holes: Main.greeting
*Hello>
```

The syntax `?greeting` introduces a *hole*, which is a part of the program yet to be written. You can type-check programs with holes and evaluate them at the REPL.

Here, when Idris encounters the `?greeting` hole, it creates a new name, `greeting`, that has a type but no definition. You can inspect the type using `:t` at the REPL:

```
*Hello> :t greeting
--------------------------------------
greeting : String
```

If you try to evaluate it, on the other hand, Idris will show you that it's a hole:

```
*Hello> greeting
?greeting : String
```

Reloading

Instead of exiting the REPL and restarting, you can also reload Hello.idr with the `:r` REPL command as follows:

```
*Hello> :r
Type checking ./Hello.idr
Holes: Main.greeting
*Hello>
```

Holes allow you to develop programs *incrementally*, writing the parts you know and asking the machine to help you by identifying the types for the parts you don't. For example, let's say you'd like to print a character (with type `Char`) instead of a `String`. The `putStrLn` function requires a `String` argument, so you can't simply pass a `Char` to it.

Listing 1.3 A program with a type error

```
module Main

main : IO ()
main = putStrLn 'x'
```

Type error, giving a character instead of a string

If you try loading this program into the REPL, Idris will report an error:

```
Hello.idr:4:17:When checking right hand side of main:
When checking an application of function Prelude.putStrLn:
        Type mismatch between
                Char (Type of 'x')
        and
                String (Expected type)
```

You have to convert a `Char` to a `String` somehow. Even if you don't know exactly how to do this at first, you can start by adding a hole to stand in for a conversion.

```
module Main

main : IO ()
main = putStrLn (?convert 'x')
```

Then you can check the type of the `convert` hole:

```
*Hello> :t convert
--------------------------------------
convert : Char -> String
```

This is a function type, taking a Char as input and returning a String.

The type of the hole, `Char -> String`, is the type of a function that takes a `Char` as an input and returns a `String` as an output. We'll discuss type conversions in more detail in chapter 2, but an appropriate function to complete this definition is `cast`:

```
main : IO ()
main = putStrLn (cast 'x')
```

1.4.5 First-class types

A *first-class* language construct is one that's treated as a value, with no syntactic restrictions on where it can be used. In other words, a first-class construct can be passed to functions, returned from functions, stored in variables, and so on.

In most statically typed languages, there are restrictions on where types can be used, and there's a strict syntactic separation between types and values. You can't, for example, say `x = int` in the body of a Java method or C function. In Idris, there are no such restrictions, and types are first-class; not only can types be used in the same way as any other language construct, but any construct can appear as part of a type.

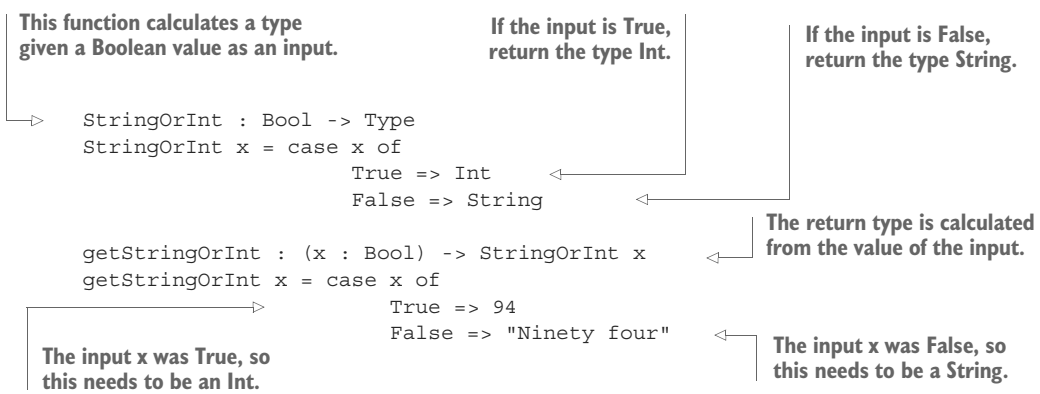
This means that you can write functions that compute types, and the return type of a function can differ depending on the input *value* to a function. This idea comes up regularly when programming in Idris, and there are several real-world situations where it's useful:

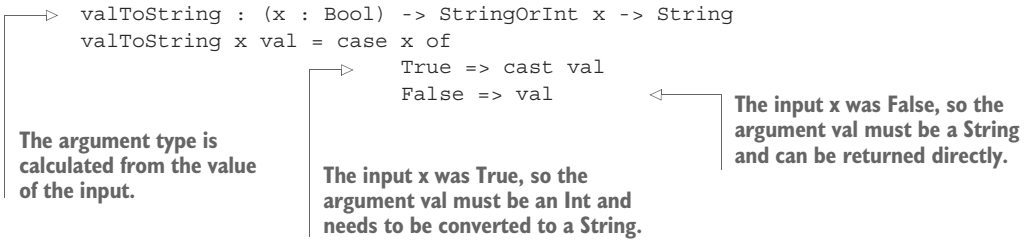
- A database schema determines the allowed forms of queries on a database.
- A form on a web page determines the number and type of inputs expected.
- A network protocol description determines the types of values that can be sent or received over a network.

In each of these cases, one piece of data tells you about the expected form of some other data. If you've programmed in C, you'll have seen a similar idea with the `printf` function, where one argument is a format string that describes the number and expected types of the remaining arguments. The C type system can't check that the format string is consistent with the arguments, so this check is often hardcoded into C compilers. In Idris, however, you can write a function similar to `printf` directly, by taking advantage of types as first-class constructs. You'll see this specific example in chapter 6.

The following listing illustrates the concept of first-class types with a small example: computing a type from a Boolean input.

Listing 1.4 Calculating a type, given a Boolean value (FCTypes.idr)





Function syntax

We'll go into much more detail on Idris syntax in the coming chapters. For now, just keep the following in mind:

- A *function* type takes the form `a -> b -> ... -> t`, where `a`, `b`, and so on, are the *input* types, and `t` is the *output* type. Inputs may also be annotated with names, taking the form `(x : a) -> (y : b) -> ... -> t`.
- `name : type` declares a new function, `name`, of type `type`.
- Functions are defined by *equations*:

```
square x = x * x
```

This defines a function called `square` that multiplies its input by itself.

Here, `StringOrInt` is a function that computes a type. Listing 1.4 uses it in two ways:

- In `getStringOrInt`, `StringOrInt` calculates the return type. If the input is `True`, `getStringOrInt` returns an `Int`; otherwise it returns a `String`.
- In `valToString`, `StringOrInt` calculates an argument type. If the first input is `True`, the second input must be an `Int`; otherwise it must be a `String`.

You can see in detail what's going on by introducing holes in the definition of `valToString`:

```
valToString : (x : Bool) -> StringOrInt x -> String
valToString x val = case x of
                         True => ?xtrueType
                         False => ?xfalseType
```

Inspecting the type of a hole with `:t` gives you not only the type of the hole itself, but also the types of any local variables in scope. If you check the type of `xtrueType`, you'll see the type of `val`, which is computed when `x` is known to be `True`:

```
*FCTypes> :t xtrueType
  x : Bool
  val : Int
--------------------------------------
xtrueType : String
```

So, if `x` is `True`, then `val` must be an `Int`, as computed by the `StringOrInt` function. Similarly, you can check the type of `xfalseType` to see the type of `val` when `x` is known to be `False`:

```
*FCTypes> :t xfalseType
  x : Bool
  val : String
-------------------------------------
xfalseType : String
```

This is a small example, but it illustrates a fundamental concept of type-driven development and programming with dependent types: the idea that the *type* of a variable can be computed from the *value* of another. In each case, Idris has used `StringOrInt` to refine the type of `val`, given what it knows about the value of `x`.

1.5 Summary

- Types are a means of classifying values. Programming languages use types to decide how to lay out data in memory, and to ensure that data is interpreted consistently.
- A type can be viewed as a specification, so that a language implementation (specifically, its type checker) can check whether a program conforms to that specification.
- Type-driven development is an iterative process of type, define, refine, creating a type to model a system, then defining functions, and finally refining the types as necessary.
- In type-driven development, a type is viewed more like a plan, helping an interactive environment guide the programmer to a working program.
- Dependent types allow you to give more-precise types to programs, and hence more informative plans to the machine.
- In a functional programming language, program execution consists of evaluating functions.
- In a purely functional programming language, additionally, functions have no *side effects*.
- Instead of writing programs that perform side effects, you can write programs that describe side effects, with the side effects stated explicitly in a program's type.
- A total function is guaranteed to produce a result for any well-typed input in finite time.
- Idris is a programming language that's specifically designed to support type-driven development. It's a purely functional programming language with first-class dependent types.
- Idris allows programs to contain holes that stand for incomplete programs.
- In Idris, types are first-class, meaning that they can be stored in variables, passed to functions, or returned from functions like any other value.