Functional Programming in

Paul Chiusano
Rúnar Bjarnason

MANNING

MEAP Edition
Manning Early Access Program
Functional Programming in Scala
version 10

For more information on this and other Manning titles go to
www.manning.com

brief contents

Preface

P.1 About this book

This is not a book about Scala. This book introduces the concepts and techniques of functional programming (FP)—we use Scala as the vehicle, but the lessons herein can be applied to programming in any language. Our goal is to give you the foundations to begin writing substantive functional programs and to comfortably absorb new FP concepts and techniques beyond those covered here. Throughout the book we rely heavily on programming exercises, carefully chosen and sequenced to guide you to discover FP for yourself. Expository text is often just enough to lead you to the next exercise. Do these exercises and you will learn the material. Read without doing and you will find yourself lost.

A word of caution: no matter how long you've been programming, learning FP is challenging. Come prepared to be a beginner once again. FP proceeds from a startling premise—that we construct programs using only pure functions, or functions that avoid *side effects* like writing to a database or reading from a file. In the first chapter, we will explain exactly what this means. From this single idea and its logical consequences emerges a very different way of building programs, one with its own body of techniques and concepts. We start by relearning how to write the simplest of programs in a functional way. From this foundation we will build the tower of techniques necessary for expressing functional programs of greater complexity. Some of these techniques may feel alien or unnatural at first and the exercises and questions can be difficult, even brain-bending at times. This is normal. Don't be deterred. Keep a beginner's mind, try to suspend judgment, and if

you must be skeptical, don't let this skepticism get in the way of learning. When you start to feel more fluent at expressing functional programs, then take a step back and evaluate what you think of the FP approach.

This book does not require any prior experience with Scala, but we won't spend a lot of time and space discussing Scala's syntax and language features. Instead we'll introduce them as we go, with a minimum of ceremony, mostly using short examples, and mostly as a consequence of covering other material. These minimal introductions to Scala should be enough to get you started with the exercises. If you have further questions about the Scala language while working on the exercises, you are expected to do some research and experimentation on your own or follow some of our links to further reading.

P.2 How to read this book

The book is organized into four parts, intended to be read sequentially. Part 1 introduces functional programming, explains what it is, why you should care, and walks through the basic low-level techniques of FP, including how to organize and structure small functional programs, define functional data structures, and handle errors functionally. These techniques will be used as the building blocks for all subsequent parts. Part 2 introduces functional design using a number of worked examples of functional libraries. It will become clear that these libraries follow certain patterns, which highlights the need for new cognitive tools for abstracting and generalizing code—we introduce these tools and explore concepts related to them in Part 3. Building on Part 3, Part 4 covers techniques and mechanisms for writing functional programs that perform I/O (like reading/writing to a database, files, or the screen) or writing to mutable variables.

Though the book can be read sequentially straight through, the material in Part 3 will make the most sense after you have a strong familiarity with the functional style of programming developed over parts 1 and 2. After Part 2, it may therefore be a good idea to take a break and try getting more practice writing functional programs beyond the shorter exercises we work on throughout the chapters. Part 4 also builds heavily on the ideas and techniques of Part 3, so a second break after Part 3 to get experience with these techniques in larger projects may be a good idea before moving on. Of course, how you read this book is ultimately up to you, and you are free to read ahead if you wish.

Most chapters in this book have similar structure. We introduce and explain some new idea or technique with an example, then work through a number of exercises, introducing further material via the exercises. The exercises thus serve

two purposes: to help you to understand the ideas being discussed and to guide you to discover for yourself new ideas that are relevant. Therefore we *strongly* suggest that you download the exercise source code and do the exercises as you go through each chapter. Exercises, hints and answers are all available at https://github.com/pchiusano/fpinscala. We also encourage you to visit the scala-functional Google group and the `#fp-in-scala` IRC channel on `irc.freenode.net` for questions and discussion.

Exercises are marked for both their difficulty and to indicate whether they are critical or noncritical. We will mark exercises that we think are *hard* or that we consider to be *critical* to understanding the material. The *hard* designation is our effort to give you some idea of what to expect—it is only our guess and you may find some unmarked questions difficult and some questions marked *hard* to be quite easy. The *critical* designation is applied to exercises that address concepts that we will be building on and are therefore important to understand fully. Noncritical exercises are still informative but can be skipped without impeding your ability to follow further material.

Examples are given throughout the book and they are meant to be *tried* rather than just read. Before you begin, you should have the Scala interpreter (REPL) running and ready. We encourage you to experiment on your own with variations of what you see in the examples. A good way to understand something is to change it slightly and see how the change affects the outcome.

Sometimes we will show a REPL session to demonstrate the result of running some code. This will be marked by lines beginning with the `scala>` prompt of the REPL. Code that follows this prompt is to be typed or pasted into the interpreter, and the line just below will show the interpreter's response, like this:

```
scala> println("Hello, World!")
          Hello, World!
```

SIDEBAR **Sidebars**

Occasionally throughout the book we will want to highlight the precise definition of a concept in a sidebar like this one. This lets us give you a complete and concise definition without breaking the flow of the main text with overly formal language, and also makes it easy to refer back to when needed.

There are chapter notes (which includes references to external resources) and

several appendix chapters after Part 4. Throughout the book we provide references to this supplementary material, which you can explore on your own if that interests you.

Have fun and good luck.

1 What is Functional Programming?

1.1 The fundamental premise of functional programming

Functional programming (FP) is based on a simple premise with far-reaching implications: We construct our programs using only *pure functions*. In other words, functions that have no *side effects*. What does this mean exactly? Performing any of the following actions directly would involve a side effect:

- Reassigning a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console or reading user input
- Reading from or writing to a file
- Drawing on the screen

Consider what programming would be like without the ability to do these things. It may be difficult to imagine. How is it even possible to write useful programs at all? If we can't reassign variables, how do we write simple programs like loops? What about working with data that changes, or handling errors without throwing exceptions? How can we perform I/O, like drawing to the screen or reading from a file?

The answer is that we can still write all of the same programs—programs that can do all of the above and more—without resorting to side effects. Functional programming is a restriction on *how* we write programs, but not on *what* programs we can write. And it turns out that accepting this restriction is tremendously

beneficial because of the increase in *modularity* that we gain from programming with pure functions. Because of their modularity, pure functions are easier to test, to reuse, to parallelize, to generalize, and to reason about.

But reaping these benefits requires that we revisit the act of programming, starting from the simplest of tasks and building upward from there. In many cases we discover how programs that seem to necessitate side effects have some purely functional analogue. In other cases we find ways to structure code so that effects occur but are not *observable* (For example, we can mutate data that is declared locally in the body of some function if we ensure that it cannot be referenced outside that function.) Nevertheless, FP is a truly radical shift in how programs are organized at every level—from the simplest of loops to high-level program architecture. The style that emerges is quite different, but it is a beautiful and cohesive approach to programming that we hope you come to appreciate.

In this book, you will learn the concepts and principles of FP as they apply to every level of programming. We begin in this chapter by explaining what a pure function is, as well as what it isn't. We also try to give you an idea of just why purity results in greater modularity and code reuse.

1.2 Exactly what is a (pure) function?

A function with input type `A` and output type `B` (written in Scala as a single type: `A => B`) is a computation which relates every value `a` of type `A` to exactly one value `b` of type `B` such that `b` is determined solely by the value of `a`.

For example, a function `intToString` having type `Int => String` will take every integer to a corresponding string. Furthermore, if it really is a *function*, it will do nothing else.

In other words, a function has no observable effect on the execution of the program other than to compute a result given its inputs; we say that it has no side effects. We sometimes qualify such functions as *pure* functions to make this more explicit. You already know about pure functions. Consider the addition (`+`) function on integers. It takes two integer values and returns an integer value. For any two given integer values it will *always return the same integer value*. Another example is the `length` function of a `String` in Java, Scala, and many other languages. For any given string, the same length is always returned and nothing else occurs.

We can formalize this idea of pure functions by using the concept of *referential transparency* (RT). This is a property of *expressions* in general and not just

functions. For the purposes of our discussion, consider an expression to be any part of a program that can be evaluated to a result, i.e. anything that you could type into the Scala interpreter and get an answer. For example, `2 + 3` is an expression that applies the pure function `+` to the values `2` and `3` (which are also expressions). This has no side effect. The evaluation of this expression results in the same value `5` every time. In fact, if you saw `2 + 3` in a program you could simply replace it with the value `5` and it would not change a thing about your program.

This is all it means for an expression to be referentially transparent—in any program, the expression can be replaced by its result without changing the meaning of the program. And we say that a function is *pure* if its body is RT, assuming RT inputs.

SIDEBAR **Referential transparency and purity**

An expression `e` is *referentially transparent* if for all programs `p`, all occurrences of `e` in `p` can be replaced by the result of evaluating `e`, without affecting the observable behavior of `p`. A function `f` is *pure* if the expression `f(x)` is referentially transparent for all referentially transparent `x`.[1]

Footnote 1 There are some subtleties to this definition, and we'll be refinining it later in this book. See the chapter notes for more discussion.

1.3 Functional and non-functional: an example

Referential transparency enables a mode of reasoning about program evaluation called *the substitution model*. When expressions are referentially transparent, we can imagine that computation proceeds very much like we would solve an algebraic equation. We fully expand every part of an expression, replacing all variables with their referents, and then reduce it to its simplest form. At each step we replace a term with an equivalent one; we say that computation proceeds by substituting *equals for equals*. In other words, RT enables *equational reasoning* about programs. This style of reasoning is *extremely* natural; you use it all the time when understanding programs, even in supposedly "non-functional" languages.

Let's look at two examples—one where all expressions are RT and can be reasoned about using the substitution model, and one where some expressions violate RT. There is nothing complicated here, part of our goal is to illustrate that we are just formalizing something you already likely understand on some level. Let's try the following in the Scala REPL:[2]

Footnote 2 In Java and in Scala, strings are immutable. If you wish to "modify" a string, you must create a copy of it.

```
scala> val x = "Hello, World"
x: java.lang.String = Hello, World

scala> val r1 = x.reverse
r1: String = dlroW ,olleH

scala> val r2 = x.reverse
r2: String = dlroW ,olleH
```

Suppose we replace all occurrences of the term `x` with the expression referenced by `x` (its definition), as follows:

```
scala> val r1 = "Hello, World".reverse
r1: String = dlroW ,olleH

val r2 = "Hello, World".reverse
r2: String = dlroW ,olleH
```

This transformation does not affect the outcome. The values of `r1` and `r2` are the same as before, so `x` was referentially transparent. What's more, `r1` and `r2` are referentially transparent as well, so if they appeared in some other part of a larger program, they could in turn be replaced with their values throughout and it would have no effect on the program.

Now let's look at a function that is *not* referentially transparent. Consider the `append` function on the `scala.collection.mutable.StringBuilder` class. This function operates on the `StringBuilder` in place. The previous state of the `StringBuilder` is destroyed after a call to `append`. Let's try this out:

```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val y = x.append(", World")
y: java.lang.StringBuilder = Hello, World

scala> val r1 = y.toString
r1: java.lang.String = Hello, World

scala> val r2 = y.toString
r2: java.lang.String = Hello, World
```

So far so good. Let's now see how this side effect breaks RT. Suppose we substitute the call to `append` like we did earlier, replacing all occurrences of `y` with the expression referenced by `y`:

```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val r1 = x.append(", World").toString
r1: java.lang.String = Hello, World

scala> val r2 = x.append(", World").toString
r2: java.lang.String = Hello, World, World
```

This transformation of the program results in a different outcome. We therefore conclude that `StringBuilder.append` is *not* a pure function. What's going on here is that while `r1` and `r2` look like they are the same expression, they are in fact referencing two different values of the same `StringBuilder`. By the time `r2` calls `x.append`, `r1` will have already mutated the object referenced by `x`. If this seems difficult to think about, that's because it is. Side effects make reasoning about program behavior more difficult.

Conversely, the substitution model is simple to reason about since effects of evaluation are purely local (they affect only the expression being evaluated) and we need not mentally simulate sequences of state updates to understand a block of code. Understanding requires only *local reasoning*. Even if you haven't used the name "substitution model", you have certainly used this mode of reasoning when thinking about your code.[3]

Footnote 3 In practice, programmers don't spend time mechanically applying substitution to determine if code is pure—it will usually be quite obvious.

1.4 Why functional programming?

We said that applying the discipline of FP buys us greater modularity. Why is this the case? Though this will become more clear over the course of the book, we can give some initial insight here.

A modular program consists of components that can be understood and reused independently of the whole, such that the meaning of the whole depends only on the meaning of the components and the rules governing their composition; that is, they are *composable*. A pure function is modular and composable because it separates the logic of the computation itself from "what to do with the result" and

"how to obtain the input"; it is a black box. Input is obtained in exactly one way: via the argument(s) to the function. And the output is simply computed and returned. By keeping each of these concerns separate, the logic of the computation is more reusable; we may reuse the logic wherever we want without worrying about whether the side effect being done with the result or the side effect being done to request the input is appropriate in all contexts. We also do not need to mentally track all the state changes that may occur before or after our function's execution to understand what our function will do; we simply look at the function's definition and substitute the arguments into its body.

Let's look at a case where factoring code into pure functions helps with reuse. This is a simple and contrived example, intended only to be illustrative. Suppose we are writing a computer game and are required to do the following:

If player 1's score property is greater than player 2's, notify the user that player 1 has won, otherwise notify the user that player 2 has won.

We may be tempted to write something like this:

```
case class Player(name: String, score: Int) ❶
def printWinner(p: Player): Unit = ❷
  println(p.name + " is the winner!")
def declareWinner(p1: Player, p2: Player): Unit = ❸
  if (p1.score > p2.score) printWinner(p1)
  else printWinner(p2)
```

❶ Declares a data type Player with two properties: name, which is a string, and score, an integer.
❷ Prints the name of the winner to the console.
❸ Takes two Players, compares their scores and declares the winner.

This declares a simple data type `Player` with two properties, `name`, which is a character string, and `score` which is an integer. The method `declareWinner` takes two `Player`s, compares their scores and declares the player with the higher score the winner (unfairly favoring the second player, granted). The `printWinner` method prints the name of the winner to the console. The result type of these methods is `Unit` indicating that they do not return a meaningful result but have a side effect instead.

Let's test this in the REPL:

```
scala> val sue = Player("Sue", 7)
sue: Player = Player(Sue, 7)
```

```
scala> val bob = Player("Bob", 8)
bob: Player = Player(Bob, 8)

scala> winner(sue, bob)
Bob is the winner!
```

While this code closely matches the earlier problem statement, it also intertwines the branching logic with that of displaying the result, which makes the reuse of the branching logic difficult. Consider trying to reuse the `declareWinner` method to compute and display the sole winner among `n` players instead of just two. In this case, the comparison logic is simple enough that we could just inline it, but then we are duplicating logic—what happens when playtesting reveals that our game unfairly favors one player, and we have to change the logic for determining the winner? We would have to change it in two places. And what if we want to use that same logic to sort a historical collection of past players to display a high score list?

Suppose we refactor the code as follows:

```
def winner(p1: Player, p2: Player): Player =  ❶
  if (p1.score > p2.score) p1 else p2

def declareWinner(p1: Player, p2: Player): Unit =
  printWinner(winner(p1, p2))
```

❶ A pure function that takes two players and returns the higher-scoring one.

This version separates the logic of computing the winner from the displaying of the result. Computing the winner in `winner` is referentially transparent and the impure part—displaying the result—is kept separate in `printWinner`. We can now reuse the logic of `winner` to compute the winner among a list of players:

```
val players = List(Player("Sue", 7),  ❶
                   Player("Bob", 8),
                   Player("Joe", 4))

val p = players.reduceLeft(winner)  ❷

printWinner(p)  ❸
```

❶ Constructs a list of players

2. Reduces the list to just the player with the highest score.
3. Prints the name of the winner to the console.

In this example, `reduceLeft` is a function on the `List` data type from the standard Scala library. The expression will compare all the players in the list and return the one with the highest score. Note that we are actually passing our `winner` function to `reduceLeft` as if it were a regular value. We will have a lot more to say about passing functions to functions, but for now just observe that because `winner` is a pure function, we are able to reuse it and combine it with other functions in ways that we didn't necessarily anticipate. In particular, this usage of `winner` would not have been possible when the side effect of displaying the result was interleaved with the logic for computing the winner.

This was just a simple example, meant to be illustrative, and the sort of factoring we did here is something you've perhaps done many times before. It's been said that functional programming, at least in small examples, is just normal separation of concerns and "good software engineering".

We will be taking the idea of FP to its logical endpoint in this book, and applying it in situations where is applicability is less obvious. As we'll learn, *any* function with side effects can be split into a pure function at the "core" and possibly a pair of functions with side effects; one on the input side, and one on the output side. This is what we did when we separated the declaration of the `winner` from our pure function `winner`. This transformation can be repeated to push side effects to the "outer layers" of the program. Functional programmers often speak of implementing programs with a pure core and a thin layer on the outside that handles effects. We will return to this principle again and again throughout the book.

1.5 Conclusion

In this chapter, we introduced functional programming and explained exactly what FP is and why you might use it. In subsequent chapters, we cover some of the fundamentals—how do we write loops in FP? Or implement data structures? How do we deal with errors and exceptions? We need to learn how to do these things and get comfortable with the low-level idioms of FP. We'll build on this understanding when we explore functional design techniques in parts 2 and 3.

Index Terms

composition
equals for equals
equational reasoning
expression substitution
modularity
program modularity
referential transparency
side effects
substitution
substitution model

2 Getting started

2.1 Introduction

Now that we have committed to using only pure functions, a question naturally emerges: how do we write even the simplest of programs? Most of us are used to thinking of programs as sequences of instructions that are executed in order, where each instruction has some kind of effect. In this chapter we will learn how to write programs in the Scala language just by combining pure functions.

This chapter is mainly intended for those readers who are new to Scala, to functional programming, or both. As with learning a foreign language, immersion is a very effective method, so we will start by looking at a small but complete Scala program. If you have no experience with Scala, you should not expect to understand the code at first glance. Therefore we will break it down piece by piece to look at what it does.

We will then look at working with higher-order functions. These are functions that take other functions as arguments, and may themselves return functions as their output. This can be brain-bending if you have a lot of experience programming in a language *without* the ability to pass functions around like that. Remember, it's not crucial that you internalize every single concept in this chapter, or solve every exercise. In fact, you might find it easier to skip whole sections and spiral back to them when you have more experience onto which to attach these concepts.

2.2 An example Scala program

The following is a complete program listing in Scala.

```
// A comment!
```

```
/* Another comment */
/** A documentation comment */
object MyModule {
  def abs(n: Int): Int =
    if (n < 0) -n
    else n

  private def formatAbs(x: Int) = {
    val msg = "The absolute value of %d is %d"
    msg.format(x, abs(x))
  }

  def main(args: Array[String]): Unit =
    println(formatAbs(-42))
}
```

We declare an object (also known as a "module") named `MyModule`. This is simply to give our code a place to live, and a name for us to refer to it later. The `object` keyword creates a new *singleton type*, which means that `MyModule` is the only value (or 'inhabitant') of that type. We put our code inside the object, between curly braces. We will discuss objects, modules, and namespaces in more detail shortly. For now, let's just look at this particular object.

The `MyModule` object has three methods: `abs`, `formatAbs`, and `main`. Each method is introduced by the `def` keyword, followed by the name of the method which is followed by the arguments in parentheses. In this case all three methods take only one argument. If there were more arguments they would be separated by commas. Following the closing parenthesis of the argument list, an optional type annotation indicates the type of the result (the colon is pronounced "has type").

The body of the method itself comes after an equals (`=`) sign. We will sometimes refer to the part of a method declaration that goes before the equals sign as the *left-hand side* or *signature*, and the code that comes after the equals sign we will sometimes refer to as the *right-hand side* or *definition*. Note the absence of an explicit `return` keyword. The value returned from a method is simply the value of its right-hand side.

Let's now go through these methods one by one. The `abs` method represents a pure function that takes an integer and returns its absolute value:[1]

Footnote 1 Astute readers might notice that this definition won't work for `Integer.MinValue`, the smallest negative 32-bit integer, which has no corresponding positive `Int`. We'll ignore this technicality here.

```
def abs(n: Int): Int = ❶
```

```
  if (n < 0) -n
  else n
```

(2)

(1) The abs method takes a single argument n of type Int, and this is declared with n: Int.
(2) The definition is a single Scala expression that uses the built-in if syntax to negate n if it's less than zero.

The `formatAbs` method represents another pure function:

```
private def formatAbs(x: Int) = {
  val msg = "The absolute value of %d is %d."
  msg.format(x, abs(x))
}
```

(1)

(1) The format method is a standard library method defined on String. Here we are calling it on the msg object, passing in the value of x along with the value of abs applied to x. This results in a new string with the occurrences of %d in msg replaced with the evaluated results of x and abs(x) respectively. Also see the sidebar on string interpolation below.

This method is declared `private`, which means that it cannot be called from any code outside of the `MyModule` object. This function takes an `Int` and returns a `String`, but note that the return type is not declared. Scala is usually able to infer the return types of methods, so they can be omitted, but it's generally considered good style to explicitly declare the return types of methods that you expect others to use. This method is private to our module, so we can omit the type annotation.

The body of the method contains more than one statement, so we put them inside curly braces. A pair of braces containing statements is called a *block*. Statements are separated by new lines or by semicolons. In this case we are using a new line to separate our statements.

The first statement in the block declares a `String` named `msg` using the `val` keyword. A `val` is an immutable variable, so inside the body of the `formatAs` method the name `msg` will always refer to the same `String` value. The Scala compiler will complain if you try to reassign `msg` to a different value in the same context.

Remember, a method simply returns the value of its right-hand side, which in this case is a block. And the value of a multi-statement block inside curly braces is simply the same as the value of its last statement. So the result of the `formatAbs`

method is just the value of `msg.format(x, abs(x))`.

SIDEBAR **String interpolation in Scala**

We could have written our `formatAbs` function using *string interpolation* (documentation) rather than the `format` method on `String`. Interpolated strings can reference Scala values in scope at the point where they are declared. An interpolated string has an `s` (for 'substitute') just before the first `"`, for example: `s"The absolute value of $x is ${abs(x)}`. See the documentation linked above for more details.

Finally, our `main` method is an "outer shell" that calls into our purely functional core and performs the effect of printing the answer to the console:

```
def main(args: Array[String]): Unit =
  println(formatAbs(-42))
```

The name `main` is special because when you run a program, Scala will look for a method named `main` with a specific signature. It has to take an `Array` of `String`s as its argument, and its return type must be `Unit`. The `args` array will contain the arguments that were given at the command line that ran the program. The return type of `Unit` indicates that this method does not return a meaningful value. There is only one value of type `Unit` and it has no inner structure. It's written `()`, pronounced "unit" just like the type. Usually a return type of `Unit` is a hint that the method has a side effect. But since the `main` method itself is called once by the operating environment and never from anywhere in our program, referential transparency is not violated.

2.3 Running our program

This section discusses the simplest possible way of running your Scala programs, suitable for short examples. More typically, you'll build and run your Scala code using sbt, the build tool for Scala, and/or an IDE like IntelliJ or Eclipse. See the book's source code repo on GitHub for more information on getting set up with sbt. Sbt is very smart about ensuring only the minimum number of files are recompiled when changes are made. It also has a number of other nice features which we won't discuss here.

But the simplest way we can run this Scala program (`MyModule`) is from the

command line, by invoking the Scala compiler directly ourselves. We start by putting the code in a file called `MyModule.scala` or something similar. We can then compile it to Java bytecode using the `scalac` compiler:

```
> scalac MyModule.scala
```

This will generate some files ending with the `.class` suffix. These files contain compiled code that can be run with the Java virtual machine. The code can be executed using the `scala` code runner:

```
> scala MyModule
The absolute value of -42 is 42.
```

Actually, it's not strictly necessary to compile the code first with `scalac`. A simple program like the one we have written here can just be run using the Scala interpreter by passing it to the `scala` code runner directly:

```
> scala MyModule.scala
The absolute value of -42 is 42.
```

This can be handy when using Scala for scripting. The code runner will look for any object within the file `MyModule.scala` that has a `main` method with the appropriate signature, and will then call it.

Lastly, an alternative way is to start the Scala interpreter's interactive mode, usually referred to as the read-evalulate-print-loop or REPL (pronounced "repple" like "apple"), and load the file from there (your actual console output may differ slightly):

```
> scala
Welcome to Scala.
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load MyModule.scala
Loading MyModule.scala...
defined module MyModule

scala> MyModule.main(Array()) ❶
The absolute value of -42 is 42.
```

❶ main takes an array as its argument and here we are simply passing it an empty array.

It's possible to simply copy and paste the code into the REPL. It also has a paste mode (accessed with the `:paste` command) specifically designed to paste code. It's a good idea to get familiar with the REPL and its features.

2.4 Modules, objects, and namespaces

In the above, notice that in order to refer to our `main` method, we had to say `MyModule.main` because `main` was defined in the `MyModule` object. Aside from a few technical corner cases, every value in Scala is what's called an "object". An object whose primary purpose is giving its members a namespace is sometimes called a *module*. An object may have zero or more *members*. A member can be a method declared with the `def` keyword, or it can be another object declared with `val` or `object`. Objects can also have other kinds of members that we will ignore for now.

We dereference the members of objects with the typical object-oriented dot-notation, which is a namespace (i.e. the name that refers to the object) followed by a dot (the period character) followed by the name of the member. For example, to call the `abs` method on the `MyModule` object we would say `MyModule.abs(42)`. To use the `toString` member on the object `42`, we would say `42.toString`. The implementations of members within an object can refer to each other unqualified (without prefixing the object name), but if needed they have access to their enclosing object using a special name: `this`.

Note that even an expression like `2 + 1` is just calling a member of an object. In that case what is being called is the `+` member of the object `2`. It is really syntactic sugar for the expression `2.+(1)`. We can in general omit the dot and parentheses like that when calling a method and applying it to a single argument. For example, instead of `MyModule.abs(42)` we can say `MyModule abs 42` and get the same result.

An object's member can be brought into scope by importing it, which allows us to call it unqualified from then on:

```
scala> import MyModule.abs
import MyModule.abs

scala> abs(-42)
res0: 42
```

We can bring all of an object's (non-private) members into scope by using the underscore syntax: `import MyModule._`

SIDEBAR **Packages**

In Scala, there is a language construct called a *package*, which is a namespace without an object. The difference between a package and a module is that a package cannot contain `val` or `def` members and can't be passed around as if it were an object.

For example, we can declare a package at the start of our Scala source file:

```
package mypackage

object MyModule {
  ...
}
```

And we can thereafter refer to `mypackage.MyModule` as a qualified name, or we can `import mypackage._` to be able to refer to `MyModule` unqualified. However, we cannot say `f(mypackage)` to pass the package to some function `f`, since a package is not a first-class value in Scala.

2.5 Function objects: passing functions to functions

In Scala, functions are objects too. They can be passed around like any other value, assigned to variables, stored in data structures, and so on. When writing purely functional programs, it becomes quite natural to want to accept functions as arguments to other functions. We are going to look at some rather simple examples just to illustrate this idea. In the chapters to come we'll see how useful this capability really is, and how it permeates our programming style. But to start, suppose we wanted to adapt our program to print out both the absolute value of a number *and* the factorial of another number. Here's a sample run of such a program:

```
The absolute value of -42 is 42
The factorial of 7 is 5040
```

First, let's write `factorial`, which also happens to be our first example of

how to write a loop without mutation:[2]

Footnote 2 We can also write this using an ordinary `while` loop and a mutable variable. See the chapter code for an example of this.

```
def factorial(n: Int): Int = {
  def go(n: Int, acc: Int): Int =
    if (n <= 0) acc
    else go(n-1, n*acc)

  go(n, 1)
}
```

1. Int is another primitive type in Scala, representing 32-bit integers
2. An inner or local function

The way we write loops in Scala is with a recursive function, by convention often called `go` (or sometimes `loop`) and which we'll often define local to another function (unlike Java, in Scala, we can define functions inside any block, including within another function definition). The arguments to `go` are the state for the loop (in this case, the remaining value `n`, and the current accumulated factorial, `acc`). To advance to the next iteration, we simply call `go` recursively with the new loop state (here, `go(n-1, n*acc)`), and to exit from the loop we return a value without a recursive call (here, we return `acc` in the case that `n <= 0`). Scala detects this sort of *self-recursion* and compiles it to the same sort of bytecode as would be emitted for a `while` loop, so long as the recursive call is in *tail position*. See the sidebar for the technical details on this, but the basic idea is that this optimization (called *tail call optimization*) is applied when there is no additional work left to do after the recursive call returns.[3]

Footnote 3 The name 'tail-call optimization' (TCO) is something of a misnomer. An 'optimization' usually connotes some nonessential performance improvement, but when we use tail calls to write loops, we generally rely on their being compiled as iterative loops that do not consume a call stack frame for each iteration (which would result in a `StackOverflowError` for large inputs).

SIDEBAR **Tail calls in Scala**

A call is said to be in 'tail position' if the caller does nothing other than return the value of the recursive call. For example, the recursive call to `go(n-1,n*acc)` above is in tail position, since the caller simply returns the value of this recursive call. If, on the other hand, we said `1 + go(n-1,n*acc)`, go would no longer be in tail position, since the caller would still have work to do when `go` returned its result (namely, adding `1` to it). Likewise if we said `f(go(n-1,n*acc))` for some function, `f`.

If all recursive calls made by a function are in tail position, Scala compiles the recursion to iterative loops that do not consume call stack frames for each iteration. If we are expecting this to occur for a recursive function we write, we can tell the Scala compiler about this assumption using an *annotation* (more information on this), so it can give us a compile error if it is not able to optimize the tail calls of the function. Here's the syntax for this:

```
def factorial(n: Int): Int = {
  @annotation.tailrec
  def go(n: Int, acc: Int): Int =
    if (n <= 0) acc
    else go(n-1, n*acc)
  go(n, 1)
}
```

We won't be talking much more about annotations in this book, but we'll use `@annotation.tailrec` extensively.

EXERCISE 1 (optional): Write a function to get the `n`th Fibonacci number. The first two Fibonacci numbers are `0` and `1`, and the next number is always the sum of the previous two. Your definition should use a local tail-recursive function.[4]

Footnote 4 Note that the nth Fibonacci number has a closed form solution. Using that would be cheating; the point here is just to get some practice writing loops using tail-recursive functions.

```
def fib(n: Int): Int
```

Now that we have `factorial`, let's edit our program from before:

The two functions, `formatAbs` and `formatFactorial`, are almost identical. If we like, we can generalize these to a single function, `formatResult`, which accepts as an argument *the function* to apply to its argument:

```
def formatResult(name: String, n: Int, f: Int => Int) = {
  val msg = "The %s of %d is %d."
  msg.format(n, f(n))
}

def main(args: Array[String]): Unit = {
  println(formatResult("absolute value", -42, abs))
  println(formatResult("factorial", 7, factorial))
}
```

There are a few new thing here. First, our `formatResult` function takes multiple arguments. To declare a function with multiple arguments, we just separate each argument by a comma. Second, our `formatResult` function now takes another function, which we call `f` (this is a common naming convention in FP; see the sidebar below). A function that takes another function as an argument is called a *higher-order function* (HOF). Like any other function parameter, we give a type to `f`, the type `Int => Int`, which indicates that `f` expects an `Int` and will also return an `Int`. (The type of a function expecting an `Int` and a `String` and returning an `Int` would be written as `(Int,String) => Int`.)

Next, notice that we call the function `f` using the same syntax as when we called `abs(x)` or `factorial(n)` directly. Lastly, notice that we can pass a reference to `abs` and `factorial` to the `formatResult` function. Our function `abs` accepts an `Int` and returns an `Int`, which matches the `Int => Int` requirement on `f` in `formatResult`. And likewise, `factorial` accepts an `Int` and returns an `Int`, which also matches the `Int => Int` requirement on `f`.

SIDEBAR **Variable naming conventions in FP**

It is a common convention to use `f`, `g`, and `h` as parameter names for functions passed to a HOF. In FP, we tend to use one-letter or very short variable names, especially when everything there is to say about a value is implied by its type. Since functions are usually quite short in FP, many functional programmers feel this makes the code easier to read, since it makes the structure of the code easier to see. We will introduce other conventions like this throughout the book.

This example isn't terribly exciting, but the same principles apply in larger examples, and we can use first-class functions to factor out duplication whenever we see it. We'll see many more examples of this throughout this book.

2.5.1 Annonymous functions

Functions get passed around so often in functional programming that it's convenient to have a lightweight way to declare a function, locally, without having to give it a name. Scala provides a syntax for declaring these nameless or *anonymous functions*. (Also often called *function literals*, *lambda functions*, *lambda expressions*, or just *lambdas*.[5])

Footnote 5 The name 'lambda' comes from the lambda calculus, another theoretical basis for computation.

Let's look at some examples of anonymous functions:

```
def main(args: Array[String]): Unit = {
  println(formatResult("absolute value", -42, abs))
  println(formatResult("factorial", 7, factorial))
  println(formatResult("increment", 7, (x: Int) => x + 1))
  println(formatResult("increment2", 7, (x) => x + 1))
  println(formatResult("increment3", 7, x => x + 1))
  println(formatResult("increment4", 7, _ + 1))
  println(formatResult("increment5", 7, x => { val r = x + 1; r }))
}
```

```
The absolute value of -42 is 42
The factorial of 7 is 5040
The increment of 7 is 8
The increment2 of 7 is 8
The increment3 of 7 is 8
The increment4 of 7 is 8
The increment5 of 7 is 8
```

In this code, `(x: Int) => x + 1`, `(x) => x + 1`, `x => x + 1`, and `_ + 1` are all alternate ways of writing the increment function, which has the type `Int => Int`. In this notation, the lambda expression has a left-hand side (`(x: Int)`, `(x)`, and `x`) and a right-hand side (`x + 1` and `{ val result = x + 1; result }`), separated by an arrow, `=>`. The left-hand side declares the argument(s) in order (`(x,y) => x + y` is an example of a two-argument anonymous function), and the right-hand side, the *body* of the function, is simply what the function will return. The body may of course refer to the arguments.[6]

Footnote 6 Note that in this case, Scala knows that `formatResult` is expecting an `Int => Int` and we can get away with not annotating the type of `x`; in other cases, Scala may not know the type of the argument and will force you to supply an annotation as in `(x: Int) => ...`.

We could declare a value of this type like so `val f = (x: Int) => x +`

`1`, but here we are not bothering to declare a local variable for the function, which is quite common in FP. In this last form `_ + 1`, sometimes called *underscore syntax* for a function literal, we are not even bothering to name the argument to the function, using _ represent the sole argument. When using this notation, we can only reference the function parameter once in the body of the function (if we mention _ again, it refers to another argument to the function).[7]

Footnote 7 There are various rules affecting the scope of an _ that we won't go over here. See the Scala Language Specification, section 6.23 for the full details. Generally, if you have to think about how an expression involving _'s will be interpreted, it's better to just use the named parameter syntax, as in `x => x + 1`.

SIDEBAR

Functions are ordinary objects

We have said that functions and methods are not exactly the same thing in Scala. When we define a function literal, what is actually being defined is an object with a method called `apply`. Scala has a special rule for this method name, so that objects that have an `apply` method can be called as if they were themselves methods. When we define a function literal like `(a, b) => a < b` this is really syntax sugar for object creation:

```
val lessThan = new Function2[Int, Int, Boolean] {
  def apply(a: Int, b: Int) = a < b
}
```

`lessThan` has type `Function2[Int,Int,Boolean]`, which is usually written `(Int,Int) => Boolean`. Note that the `Function2` interface (known in Scala as a "trait") has a single method called `apply`. And when we call the `lessThan` function with `lessThan(10, 20)`, it is really syntax sugar for calling its `apply` method:

```
scala> val b = lessThan.apply(10, 20)
b: Boolean = true
```

`Function2` is just an ordinary trait (i.e. an interface) provided by the standard Scala library (API docs link) to represent function objects that take two arguments. Also provided are `Function1`, `Function3`, and others, taking a number of arguments indicated by the name. Because functions are really just ordinary Scala objects, we say that they are *first-class* values. We will often use "function" to refer to either such a first-class function or a method, depending on context.

2.6 Polymorphic functions: abstracting over types

So far we have been defining only *monomorphic* functions. That is, functions that operate on only one type of data. For example, `abs`, and `factorial` are specific to arguments of type `Int`, and the higher-order function `formatResult` is also fixed to operate on functions that take arguments of type `Int`. Very often, we want to write code which works for *any* type it is given. As an example, here's a definition of binary search, specialized for searching for a `Double` in an `Array[Double]`. `Double` is another primitive type in Scala, representing double precision floating point numbers. And `Array[Double]` is the type representing an array of `Double` values.

```
def binarySearch(ds: Array[Double], key: Double): Int = {
  @annotation.tailrec
  def go(low: Int, mid: Int, high: Int): Int = {
    if (low > high) -mid - 1
    else {
      val mid2 = (low + high) / 2
      val d = ds(mid2) // 1
      if (d == key) mid2
      else if (d > key) go(low, mid2, mid2-1)
      else go(mid2 + 1, mid2, high)
    }
  }
  go(0, 0, ds.length - 1)
}
```

1. We index into an array using the same syntax as function application

The details of the algorithm aren't too important here. What is important is that the code for `binarySearch` is going to look almost identical if we are searching for a `Double` in an `Array[Double]`, an `Int` in an `Array[Int]`, a `String` in an `Array[String]`, or an `A` in an `Array[A]`. We can write `binarySearch` more generally for any type `A`, by accepting a function to use for testing whether an `A` value is greater than another:

```
def binarySearch[A](as: Array[A], key: A, gt: (A,A) => Boolean): Int = {
  @annotation.tailrec
  def go(low: Int, mid: Int, high: Int): Int = {
    if (low > high) -mid - 1
    else {
      val mid2 = (low + high) / 2
      val a = as(mid2)
```

```
      val greater = gt(a, key)
      if (!greater && !gt(key,a)) mid2
      else if (greater) go(low, mid2, mid2-1)
      else go(mid2 + 1, mid2, high)
    }
  }
  go(0, 0, as.length - 1)
}
```

This is an example of a *polymorphic* function.[8] We are *abstracting over the type* of the array, and the comparison function used for searching it. To write a polymorphic function, we introduce a comma-separated list of *type parameters*, surrounded by `[]` (here, just a single `[A]`), following the name of the function, in this case `binarySearch`. We can call the type parameters anything we want—`[Foo, Bar, Baz]` and `[TheParameter, another_good_one]` are valid type parameter declarations—though by convention we typically use short, one letter type parameter names, like `[A,B,C]`.

Footnote 8 We are using the term 'polymorphism' in a slightly different way than mainstream object-oriented programming, where that term usually connotes some form of subtyping. There are no interfaces or subtyping here in this example. One will occasionally see the term *parametric polymorphism* used to refer to this form of polymorphism.

The type parameter list introduces *type variables* (or sometimes *type parameters*) that can be referenced in the rest of the type signature (exactly analogous to how variables introduced in the arguments to a function can be referenced in the body of the function). Here, the type variable `A` is referenced in three places—the search key is required to have the type `A`, the values of the array are required to have the type `A` (since it is an `Array[A]`), and the `gt` function must accept two arguments both of type `A` (since it is an `(A,A) => Boolean`). The fact that the same type variable is referenced in all three places in the type signature enforces that the type must be the same for all three arguments, and the compiler will enforce this fact anywhere we try to call `binarySearch`. If we try to search for a `String` in an `Array[Int]`, for instance, we'll get a type mismatch error.[9]

Footnote 9 Unfortunately, Scala's use of subtyping means we sometimes get rather cryptic compile errors, since Scala will try to find a common supertype to use for the `A` type parameter, and will fall back to using `Any`, the supertype of all types.

EXERCISE 2: Implement `isSorted`, which checks whether an `Array[A]` is sorted according to a given comparison function.

```
def isSorted[A](as: Array[A], gt: (A,A) => Boolean): Boolean
```

SIDEBAR **Boxed types and specialization in Scala**

A function that is polymorphic in some type is generally forced to represent values of these types as *boxed*, or non-primitive values, meaning they are stored as a pointer to a value on the heap. It is possible to instruct the Scala compiler to produce specialized versions of a function for each of the primitive types, just by adding an annotation to that type parameter:

```
def binarySearch[@specialized A](as: Array[A], key: A,
                                 gt: (A,A) => Boolean): Int
```

This can potentially be much more efficient, though the mechanism is rather fragile, since the polymorphic values will get boxed as soon as they are passed to any other polymorphic function or data type which is unspecialized in this way.

As you might have seen when writing `isSorted`, the universe of possible implementations is significantly reduced when implementing a polymorphic function. If a function is polymorphic in some type, `A`, the only operations that can be performed on that `A` are those passed into the function as arguments (or that can be defined in terms of these given operations).[10] In some cases, you'll find that the universe of possibilities for a given polymorphic type is constrained such that there is only a single implementation!

Footnote 10 Technically, all values in Scala can be compared for equality (using `==`), and we can compute a hash code for them as well. But this is something of a wart inherited from Java.

Let's look at an example of this, a higher-order function for doing what is called *partial application*. This function, `partial1`, takes a value and a function of two arguments, and returns a function of one argument as its result. The name comes from the fact that the function is being applied to some but not all of its required arguments.

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C
```

EXERCISE 3 (hard): Implement `partial1` and write down a concrete usage of it. There is only one possible implementation that compiles. We don't have any

concrete types here, so we can only stick things together using the local 'rules of the universe' established by the type signature. The style of reasoning required here is very common in functional programming—we are simply manipulating symbols in a very abstract way, similar to how we would reason when solving an algebraic equation.

EXERCISE 4 (hard): Let's look at another example, *currying*, which converts a function of *N* arguments into a function of one argument that returns another function as its result.[11] Here again, there is only one implementation that typechecks.

Footnote 11 This is named after the mathematician Haskell Curry, who discovered the principle. It was independently discovered earlier by Moses Schoenfinkel, but "Schoenfinkelization" didn't catch on.

```
def curry[A,B,C](f: (A, B) => C): A => (B => C)
```

EXERCISE 5 (optional): Implement `uncurry`, which reverses the transformation of `curry`. Note that since `=>` associates to the right, `A => (B => C)` can be written as `A => B => C`.

```
def uncurry[A,B,C](f: A => B => C): (A, B) => C
```

Let's look at a final example, *function composition*, which feeds the output of one function in as the input to another function. Again, the implementation of this function is fully determined by its type signature.

EXERCISE 6: Implement the higher-order function that composes two functions.

```
def compose[A,B,C](f: B => C, g: A => B): A => C
```

This is such a common thing to want to do that Scala's standard library provides `compose` as a method on `Function1`. To compose two functions `f` and `g`, you simply say `f compose g`[12]. It also provides an `andThen` method. `f andThen g` is the same as `g compose f`:

Footnote 12 Solving the `compose` exercise by using this library function is considered cheating.

```
scala> val f = (x: Double) => math.Pi / 2 - x
```

```
f: Double => Double = <function1>

scala> val cos = f andThen math.sin
cos: Double => Double = <function1>
```

Interestingly, functions like `compose` do not care whether they are operating on huge functions backed by millions of lines of code, or a couple of one-line functions. Polymorphic, higher-order functions often end up being extremely widely applicable, precisely because they say nothing about any particular domain and are simply abstracting over a common pattern that occurs in many contexts. We'll be writing many more such functions over the course of this book, and this is just a short taste of the style of reasoning and thinking you'll use when writing such functions.

2.7 Conclusion

In this chapter we have learned some preliminary functional programming concepts, and enough Scala to get going. We learned how to define simple functions and programs, including how we can express loops using recursion, then introduced the idea of higher-order functions and got some practice writing polymorphic functions in Scala. We saw how the implementations of polymorphic functions are often significantly constrained, such that one can often simply 'follow the types' to the correct implementation. This is something we'll see a lot more of in the chapters ahead.

Although we haven't yet written any large or complex programs, the principles we have discussed here are scalable and apply equally well to programming in the large as they do to programming in the small.

Next up we will look at using pure functions to manipulate data.

Index Terms

3 Functional data structures

3.1 Introduction

We said in the introduction that functional programs do not update variables or modify data structures. This raises pressing questions—what sort of data structures *can* we use in functional programming, how do we define them in Scala, and how do we operate over these data structures? In this chapter we will learn the concept of a *functional data structure* and how to define and work with such structures. We'll use this as an opportunity to introduce how data types are defined in functional programming, learn about the related technique of *pattern matching*, and get practice writing and generalizing pure functions.

This chapter has a lot of exercises, particularly to help with this last point—writing and generalizing pure functions. Some of these exercises may be challenging. As always, if you need to, consult the hints or the answers, or ask for help online.

3.2 Defining functional data structures

A functional data structure is (not surprisingly!) operated on using only pure functions. Remember, a pure function may only accept some values as input and yield a value as output. It may not change data in place or perform other side effects. *Therefore, functional data structures are immutable.* For example, the empty list, (denoted `List()` or `Nil` in Scala) is as eternal and immutable as the integer values `3` or `4`. And just as evaluating `3 + 4` results in a new number `7` without modifying either `3` or `4`, concatenating two lists together (the syntax for this is `a ++ b` for two lists `a` and `b`) yields a new list and leaves the two inputs unmodified.

Doesn't this mean we end up doing a lot of extra copying of the data?

Somewhat surprisingly, the answer is 'no'. We will return to this issue after examining the definition of what is perhaps the most ubiquitous of functional data structures, the singly-linked list. The definition here is identical in spirit to (though simpler than) the `List` data type defined in Scala's standard library. This code listing makes use of a lot of new syntax and concepts, so don't worry if not everything makes sense at first—we will talk through it in detail.[1]

Footnote 1 Note—the implementations of `sum` and `product` here are not tail recursive. We will be writing tail recursive versions of these functions later in the chapter.

Listing 3.1 Singly-linked lists

```
package fpinscala.datastructures

sealed trait List[+A] ❶
case object Nil extends List[Nothing] ❷
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List { ❸
  def sum(ints: List[Int]): Int = ints match { ❹
    case Nil => 0
    case Cons(x,xs) => x + sum(xs)
  }

  def product(ds: List[Double]): Double = ds match {
    case Nil => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(x,xs) => x * product(xs)
  }

  def apply[A](as: A*): List[A] = ❺
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))

  val example = Cons(1, Cons(2, Cons(3, Nil))) ❻
  val example2 = List(1,2,3)
  val total = sum(example)
}
```

❶ List data type
❷ data constructor for List
❸ List companion object
❹ Pattern matching example
❺ Variadic function syntax
❻ Creating lists

Let's look first at the definition of the data type, which begins with the keywords `sealed trait`. In general, we introduce a data type with the `trait` keyword. A `trait` is an abstract interface that may optionally contain implementations of some methods. Here we are declaring a `trait`, called `List`, with no methods on it. Adding `sealed` in front means that all implementations of our `trait` must be declared in this file.[2]

Footnote 2 We could also say `abstract class` here instead of `trait`. Technically, an `abstract class` can contain *constructors*, in the OO sense, which is what separates it from a `trait`, which cannot contain constructors. This distinction is not really relevant for our purposes right now.

There are two such implementations or *data constructors* of `List` (each introduced with the keyword `case`) declared next, to represent each of the two possible forms a `List` can take—it can be *empty*, denoted by the data constructor `Nil`, or it can be nonempty (the data constructor `Cons`, traditionally short for 'construct'), in which case it consists of an initial element, `head`, followed by a `List` (possibly empty) of remaining elements (the `tail`).

Listing 3.2 The data constructors of List

```
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

Just as functions can be polymorphic, data types can be as well, and by adding the type parameter `[+A]` after `sealed trait List` and then using that `A` parameter inside of the `Cons` data constructor, we have declared the `List` data type to be polymorphic in the type of elements it contains, which means we can use this same definition for a list of `Int` elements (denoted `List[Int]`), `Double` elements (denoted `List[Double]`), `String` elements (`List[String]`), and so on (the `+` indicates that the type parameter, `A` is *covariant*—see sidebar 'More about variance' for more info).

A data constructor declaration gives us a function to construct that form of the data type (the `case object Nil` lets us write `Nil` to construct an empty `List`, and the `case class Cons` lets us write `Cons(1, Nil)`, `Cons(1, Cons(2, Nil))`, and so on for nonempty lists), but also introduces a *pattern* that can be used for *pattern matching*, as in the functions `sum` and `product`.

SIDEBAR **More about variance**

In the declaration `trait List[+A]`, the `+` in front of the type parameter `A` is a *variance annotation* which signals that `A` is a *covariant* or 'positive' parameter of `List`. This means that, for instance, `List[Dog]` is considered a subtype of `List[Animal]`, assuming `Dog` is a subtype of `Animal`. (More generally, for all types `X` and `Y`, if `X` is a subtype of `Y` then `List[X]` is a subtype of `List[Y]`). We could leave out the `+` in front of the `A`, which would make `List` *invariant* in that type parameter.

But notice now that `Nil` extends `List[Nothing]`. `Nothing` is a subtype of all types, which means that in conjunction with the *variance annotation*, `Nil` can be considered a `List[Int]`, a `List[Double]`, and so on, exactly as we want.

These concerns about variance are not very important for the present discussion and are more of an artifact of how Scala encodes data constructors via subtyping, so don't worry if this is not completely clear right now.[3]

Footnote 3 It is certainly possible to write code without using variance annotations at all, and function signatures are sometimes simpler (while type inference often gets worse). Unless otherwise noted, we will be using variance annotations throughout this book, but you should feel free to experiment with both approaches.

3.2.1 Pattern matching

Let's look in detail at the functions `sum` and `product`, which we place in the `object List`, sometimes called the *companion object* to `List` (see sidebar). Both these definitions make use of pattern matching.

```
def sum(ints: List[Int]): Int = ints match {
  case Nil => 0
  case Cons(x,xs) => x + sum(xs)
}

def product(ds: List[Double]): Double = ds match {
  case Nil => 1.0
  case Cons(0.0, _) => 0.0
  case Cons(x, xs) => x * product(xs)
}
```

As you might expect, the `sum` function states that the sum of an empty list is `0`,

and the sum of a nonempty list is the first element, `x`, plus the sum of the remaining elements, `xs`.[4] Likewise the `product` definition states that the product of an empty list is `1.0`, the product of any list starting with `0.0` is `0.0`,[5] and the product of any other nonempty list is the first element multiplied by the product of the remaining elements. Notice these are recursive definitions, which are quite common when writing functions that operate over recursive data types like `List` (which refers to itself recursively in its `Cons` data constructor).

Footnote 4 We could call `x` and `xs` anything there, but it is a common convention to use `xs`, `ys`, `as`, `bs` as variable names for a sequence of some sort, and `x`, `y`, `z`, `a`, or `b` as the name for a single element of a sequence. Another common naming convention is `h` for the first element of a list (the "head" of the list), `t` for the remaining elements (the "tail"), and `l` for an entire list.

Footnote 5 LISTS

---

[6] This isn't the most robust test—pattern matching on `0.0` will match only the exact value `0.0`, not `1e-102` or any other value very close to `0`.

Footnote 6 LISTS

Pattern matching works a bit like a fancy `switch` statement that may descend into the structure of the expression it examines and extract subexpressions of that structure (we'll explain this shortly). It is introduced with an expression (the *target* or *scrutinee*), like `ds` followed by the keyword `match`, and a `{}`-wrapped sequence of *cases*. Each case in the match consists of a *pattern* (like `Cons(x,xs)`) to the left of the `=>` and a *result* (like `x * product(xs)`) to the right of the `=>`. If the target *matches* the pattern in a case (see below), the result of that case becomes the result of the entire match expression. If multiple patterns match the target, Scala chooses the first matching case.

SIDEBAR **Companion objects in Scala**

We will often declare a *companion object* in addition to our data type and its data constructors. This is just an `object` with the same name as the data type (in this case `List`) where we put various convenience functions for creating or working with values of the data type.

If, for instance, we wanted a function `def fill[A](n: Int, a: A): List[A]`, that created a `List` with n copies of the element a, the `List` companion object would be a good place for it. Companion objects are more of a convention in Scala.[7] We could have called this module `Foo` if we wanted, but calling it `List` makes it clear that the module contains functions relevant to working with lists, and also gives us the nice `List(1,2,3)` syntax when we define a variadic `apply` function (see sidebar 'Variadic functions in Scala').

Footnote 7 There is some special support for them in the language that isn't really relevant for our purposes.

Let's look at a few more examples of pattern matching:

- `List(1,2,3) match { case _ => 42 }` results in `42`. Here we are using a variable pattern, _, which matches any expression. We could say `x` or `foo` instead of _ but we usually use _ to indicate a variable whose value we ignore in the result of the case.[8]

Footnote 8 The _ variable pattern is treated somewhat specially in that it may be mentioned multiple times in the pattern to ignore multiple parts of the target.

- `List(1,2,3) match { case Cons(h,t) => h }` results in `1`. Here we are using a data constructor pattern in conjunction with variables to *capture* or *bind* a subexpression of the target.
- `List(1,2,3) match { case Cons(_,t) => t }` results in `List(2,3)`.
- `List(1,2,3) match { case Nil => 42 }` results in a `MatchError` at runtime. A `MatchError` indicates that none of the cases in a match expression matched the target.

What determines if a pattern matches an expression? A pattern may contain *literals*, like `0.0` or `"hi"`, *variables* like `x` and `xs` which match anything, indicated by an identifier starting with a lowercase letter or underscore, and data constructors like `Cons(x,xs)` or `Nil`, which match only values of the corresponding form (`Nil` as a pattern matches only the value `Nil`, and `Cons(h,t)` or `Cons(x,xs)` as a pattern only match `Cons` values). These components of a pattern may be nested arbitrarily—`Cons(x1, Cons(x2, Nil))` and `Cons(y1, Cons(y2, Cons(y3, _)))` are valid patterns. A

pattern *matches* the target if there exists an assignment of variables in the pattern to subexpressions of the target that make it *structurally equivalent* to the target. The result expression for a matching case will then have access to these variable assignments in its local scope.

EXERCISE 1: What will the result of the following match expression be?

```
val x = List(1,2,3,4,5) match {
  case Cons(x, Cons(2, Cons(4, _))) => x
  case Nil => 42
  case Cons(x, Cons(y, Cons(3, Cons(4, _)))) => x + y
  case Cons(h, t) => h + sum(t)
  case _ => 101
}
```

You are strongly encouraged to try experimenting with pattern matching in the REPL to get a sense for how it behaves.

SIDEBAR **Variadic functions in Scala**

The function `List.apply` in the listing above is a *variadic function*, meaning it accepts zero or more arguments of type `A`. For data types, it is a common idiom to have a variadic `apply` method in the companion object to conveniently construct instances of the data type. By calling this function `apply` and placing it in the companion object, we can invoke it with syntax like `List(1,2,3,4)` or `List("hi","bye")`, with as many values as we want separated by commas (we sometimes call this the *list literal* or just *literal* syntax).

Variadic functions are just providing a little syntax sugar for creating and passing a `Seq` of elements explicitly. `Seq` is the interface in Scala implemented by sequence-like data structures like lists, queues, vectors, and so forth. Inside `apply`, `as` will be bound to a `Seq[A]` (documentation link), which has the functions `head` (returns the first element) and `tail` (returns all elements but the first).

We can convert a `Seq[A]`, `x`, back into something that can be passed to a variadic function using the syntax `x: _*`, where `x` can be any expression, for instance: `List(x: _*)` or even `List(List(1,2,3): _*)`.

3.3 Functional data structures and data sharing

When data is immutable, how do we write functions that, for example, add or remove elements from a list? The answer is simple. When we add an element `1` to the front of an existing list, say `xs`, we return a new list, in this case `Cons(1,xs)`. Since lists are immutable, we don't need to actually copy `xs`; we can just reuse it. This property of immutable data is called *data sharing* or just *sharing*. Data sharing of immutable data often lets us implement functions more efficiently; we can always return immutable data structures without having to worry about subsequent code modifying our data. There's no need to pessimistically make copies to avoid modification or corruption.[9]

Footnote 9 This pessimistic copying can become a problem in large programs, when data may be passed through a chain of loosely components, each of which may be forced to make copies of this data. Using immutable data structures means never having to copy that data just to share it between two components of a system, which promotes keeping these components loosely coupled. We find that *in the large*, FP can often achieve greater efficiency than approaches that rely on side effects, due to much greater sharing of data and computation.

In the same way, to "remove" an element from the front of a list `val mylist = Cons(x,xs)`, we simply return `xs`. There is no real removing going on. The original list, `mylist` is still available, unharmed. We say that functional data structures are *persistent*, meaning that existing references are never changed by operations on the data structure.

Let's try implementing a few different functions for "modifying" lists in different ways. You can place this and other functions we write inside the `List` companion object.

EXERCISE 2: Implement the function `tail` for "removing" the first element of a `List`. Notice the function takes constant time. What are different choices you could make in your implementation if the `List` is `Nil`? We will return to this question in the next chapter.

EXERCISE 3: Generalize `tail` to the function `drop`, which removes the first `n` elements from a list.

EXERCISE 4: Implement `dropWhile`,[10] which removes elements from the `List` prefix as long as they match a predicate. Again, notice these functions take time proportional only to the number of elements being dropped—we do not need to make a copy of the entire `List`.

Footnote 10 `dropWhile` has two argument lists to improve type inference. See sidebar.

```
def drop[A](l: List[A], n: Int): List[A]
```

```
def dropWhile[A](l: List[A])(f: A => Boolean): List[A]
```

SIDEBAR **Type inference in Scala**

When writing functions like `dropWhile`, we will often place the `List` in the first argument group, and any functions, `f` that receive elements of the `List` in a later argument group. We can call this function with two sets of parentheses, like `dropWhile(xs)(f)`, or we can partially apply it by supplying only the first argument `dropWhile(xs)`. This returns a function that accepts the other argument, `f`. The main reason for grouping the arguments this way is to assist with type inference. If we do this, Scala can determine the type of `f` without any annotation, based on what it knows about the type of the `List`, which makes the function more convenient to use, especially when passing a function literal like `x => x > 34` in for `f`, which would otherwise require an annotation like `(x: Int) => x > 34`. (Here it is not so bad, but when working with more complicated types, it is a pain to have to write out these types each time we pass a function literal into a higher-order function like `dropWhile`). This is an unfortunate restriction of the Scala compiler; other functional languages like Haskell and OCaml provide *complete* inference, meaning type annotations are almost never required.[11]

Footnote 11 See the notes for this chapter for more information and links to further reading.

EXERCISE 5: Using the same idea, implement the function `setHead` for replacing the first element of a `List` with a different value.

Data sharing often brings some more surprising efficiency gains. For instance, here is a function that adds all the elements of one list to the end of another:

```
def append[A](a1: List[A], a2: List[A]): List[A] =
  a1 match {
    case Nil => a2
    case Cons(h,t) => Cons(h, append(t, a2))
  }
```

Notice that this definition only copies values until the first list is exhausted, so

its runtime is determined only by the length of `a1`. The remaining list then just points to `a2`. If we were to implement this same function for two arrays, we would be forced to copy all the elements in both arrays into the result.

EXERCISE 6: Not everything works out so nicely. Implement a function, `init`, which returns a `List` consisting of all but the last element of a `List`. So, given `List(1,2,3,4)`, `init` will return `List(1,2,3)`. Why can't this function be implemented in constant time like `tail`?

```
def init[A](l: List[A]): List[A]
```

Because of the structure of a singly-linked list, any time we want to replace the `tail` of a `Cons`, even if it is the last `Cons` in the list, we must copy all the previous `Cons` objects. Writing purely functional data structures that support different operations efficiently is all about finding clever ways to exploit data sharing, which often means working with more tree-like data structures. We are not going to cover these data structures here; for now, we are content to use the functional data structures others have written. As an example of what's possible, in the Scala standard library, there is a purely functional sequence implementation, `Vector` (documentation link), with constant-time random access, updates, `head`, `tail`, `init`, and constant-time additions to either the front or rear of the sequence.

3.4 Recursion over lists and generalizing to higher-order functions

Let's look again at the implementations of sum and product. We've simplified the `product` implementation slightly, so as not to include the "short-circuiting" logic of checking for `0.0`:

```
def sum(ints: List[Int]): Int = ints match {
  case Nil => 0
  case Cons(x,xs) => x + sum(xs)
}

def product(ds: List[Double]): Double = ds match {
  case Nil => 1.0
  case Cons(x, xs) => x * product(xs)
}
```

Notice how similar these two definitions are. The only things that differ are the

value to return in the case that the list is empty (`0` in the case of `sum`, `1.0` in the case of `product`), and the operation to apply to combine results (`+` in the case of `sum`, `*` in the case of `product`). Whenever you encounter duplication like this, as we've discussed before, you can generalize it away by pulling subexpressions out into function arguments. If a subexpression refers to any local variables (the `+` operation refers to the local variables `x` and `xs` introduced by the pattern, similarly for `product`), turn the subexpression into a function that accepts these variables as arguments. Putting this all together for this case, our function will take as arguments the value to return in the case of the empty list, and the function to add an element to the result in the case of a nonempty list:[12]

Footnote 12 In the Scala standard library, `foldRight` is a method on `List` and its arguments are curried similarly for better type inference.

Listing 3.3 Right folds and simple uses

```
def foldRight[A,B](l: List[A], z: B)(f: (A, B) => B): B =  ❶
  l match {
    case Nil => z
    case Cons(x, xs) => f(x, foldRight(xs, z)(f))
  }

def sum2(l: List[Int]) =
  foldRight(l, 0.0)(_ + _)

def product2(l: List[Double]) =
  foldRight(l, 1.0)(_ * _)
```

❶ Again, placing f in its own argument group after l and z lets type inference determine the input types to f. See earlier sidebar.

`foldRight` is not specific to any one type of element, and the value that is returned doesn't have to be of the same type as the elements either. One way of describing what `foldRight` does is that it replaces the constructors of the list, `Nil` and `Cons` with `z` and `f`, respectively. So the value of `foldRight(Cons(a, Nil), z)(f)` becomes `f(a, z)`, and `foldRight(Cons(a, Cons(b, Nil)), z)(f)` becomes `f(a, f(b, z))`.

Let's look at an example. We are going to *trace* the evaluation of `foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)(_ + _)`, by repeatedly subsituting the definition of `foldRight` in for its usages. We'll use

program traces like this throughout this book.

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)(_ + _)
1 + foldRight(Cons(2, Cons(3, Nil)), 0)(_ + _)
1 + (2 + foldRight(Cons(3, Nil), 0)(_ + _))
1 + (2 + (3 + (foldRight(Nil, 0)(_ + _))))
1 + (2 + (3 + (0)))
6
```

Notice that `foldRight` must traverse all the way to the end of the list (pushing frames onto the call stack as we go) before it can begin collapsing it.

EXERCISE 7: Can `product` implemented using `foldRight` immediately halt the recursion and return `0.0` if it encounters a `0.0`? Why or why not? Consider how any short-circuiting might work if you call `foldRight` with a large list. This is a deeper question that we'll return to a few chapters from now.

EXERCISE 8: See what happens when you pass `Nil` and `Cons` themselves to `foldRight`, like this: `foldRight(List(1,2,3), Nil:List[Int])(Cons(_,_))`.[13] What do you think this says about the relationship between `foldRight` and the data constructors of `List`?

Footnote 13 The type annotation `Nil:List[Int]` is needed here, because otherwise Scala infers the `B` type parameter in `foldRight` as `List[Nothing]`.

EXERCISE 9: Compute the length of a list using `foldRight`.

```
def length[A](l: List[A]): Int
```

EXERCISE 10: `foldRight` is not tail-recursive and will `StackOverflow` for large lists. Convince yourself that this is the case, then write another general list-recursion function, `foldLeft` that is tail-recursive, using the techniques we discussed in the previous chapter. Here is its signature:[14]

Footnote 14 Again, `foldLeft` is defined as a method of `List` in the scala standard library, and it is curried similarly for better type inference, so you can write `mylist.foldLeft(0.0)(_ + _)`.

```
def foldLeft[A,B](l: List[A], z: B)(f: (B, A) => B): B
```

EXERCISE 11: Write `sum`, `product`, and a function to compute the length of a list using `foldLeft`.

EXERCISE 12: Write a function that returns the reverse of a list (so given `List(1,2,3)` it returns `List(3,2,1)`). See if you can write it using a fold.

EXERCISE 13 (hard): Can you write `foldLeft` in terms of `foldRight`? How about the other way around?

EXERCISE 14: Implement `append` in terms of either `foldLeft` or `foldRight`.

EXERCISE 15 (hard): Write a function that concatenates a list of lists into a single list. Its runtime should be linear in the total length of all lists. Try to use functions we have already defined.

3.4.1 More functions for working with lists

There are many more useful functions for working with lists. We are going to cover a few more here. After finishing this chapter, we recommend looking through the scala API documentation to see what other functions there are. If you find yourself writing an explicit recursive function for doing some sort of list manipulation, check the `List` API to see if something like the function you need already exists.

After finishing this section, you're not going to emerge with an automatic sense of when to use each of these functions. Just get in the habit of looking for possible ways to generalize any explicit recursive functions you write to process lists. If you do this, you'll (re)discover these functions for yourself and build more of a sense for when you'd use each one.

SIDEBAR **Lists in the standard library**

`List` exists in the Scala standard library (API documentation), and we'll use the standard library version in subsequent chapters.

The main difference between the `List` developed here and the standard library version is that `Cons` is called `::`, which is right-associative (all operators ending in `:` are right-associative), so `1 :: 2 :: Nil` is equal to `List(1,2,3)`. When pattern matching, `case Cons(h,t)` becomes `case h :: t`, which avoids having to nest parentheses if writing a pattern like `case h :: h2 :: t` to extract more than just the first element of the `List`.

EXERCISE 16: Write a function that transforms a list of integers by adding `1` to each element. (Reminder: this should be a pure function that returns a new `List`!)

EXERCISE 17: Write a function that turns each value in a `List[Double]` into a `String`.

EXERCISE 18: Write a function `map`, that generalizes modifying each element in a list while maintaining the structure of the list. Here is its signature:[15]

Footnote 15 In the standard library, `map` and `flatMap` are methods of `List`.

```
def map[A,B](l: List[A])(f: A => B): List[B]
```

EXERCISE 19: Write a function `filter` that removes elements from a list unless they satisfy a given predicate. Use it to remote all odd numbers from a `List[Int]`.

EXERCISE 20: Write a function `flatMap`, that works like `map` except that the function given will return a list instead of a single result, and that list should be inserted into the final resulting list. Here is its signature:

```
def flatMap[A,B](l: List[A])(f: A => List[B]): List[B]
```

For instance `flatMap(List(1,2,3))(i => List(i,i))` should result in `List(1,1,2,2,3,3)`.

EXERCISE 21: Can you use `flatMap` to implement `filter`?

EXERCISE 22: Write a function that accepts two lists and constructs a new list by adding corresponding elements. For example, `List(1,2,3)` and `List(4,5,6)` becomes `List(5,7,9)`.

EXERCISE 23: Generalize the function you just wrote so that it's not specific to integers or addition.

There are a number of other useful methods on lists. You may want to try experimenting with these and other methods in the REPL after reading the API documentation. These are defined as methods on `List[A]`, rather than as standalone functions as we've done in this chapter.

- `def take(n: Int): List[A]`: returns a list consisting of the first `n` elements of `this`.
- `def takeWhile(f: A => Boolean): List[A]`: returns a list consisting of the longest valid prefix of `this` whose elements all pass the predicate `f`.
- `def forall(f: A => Boolean): Boolean`: returns `true` if and only if all elements of `this` pass the predicate `f`.
- `def exists(f: A => Boolean): Boolean`: returns `true` if any element of `this` passes the predicate `f`.

- `scanLeft` and `scanRight` are like `foldLeft` and `foldRight`, but they return the `List` of partial results, rather than just the final accumulated value.

One of the problems with `List` is that while we can often express operations and algorithms in terms of very general-purpose functions, the resulting implementation isn't always efficient—we may end up making multiple passes over the same input, or else have to write explicit recursive loops to allow early termination.

EXERCISE 24 (hard): As an example, implement `hasSubsequence` for checking whether a `List` contains another `List` as a subsequence. For instance, `List(1,2,3,4)` would have `List(1,2)`, `List(2,3)`, and `List(4)` as subsequences, among others. You may have some difficulty finding a concise purely functional implementation that is also efficient. That's okay. Implement the function however comes most naturally. We will return to this implementation in a couple of chapters and hopefully improve on it. Note: any two values, `x`, and `y`, can be compared for equality in Scala using the expression `x == y`.

```
def hasSubsequence[A](l: List[A], sub: List[A]): Boolean
```

3.5 Trees

`List` is just one example of what is called an *algebraic data type* (ADT). (Somewhat confusingly, ADT is sometimes used in OO to stand for "abstract data type".) An ADT is just a data type defined by one or more data constructors, each of which may contain zero or more arguments. We say that the data type is the *sum* or *union* of its data constructors, and each data constructor is the *product* of its arguments, hence the name *algebraic* data type.[16]

Footnote 16 The naming is not coincidental. There is actually a deep connection, beyond the scope of this book, between the "addition" and "multiplication" of types to form an ADT and addition and multiplication of numbers.

When you encode a data type as an ADT, the data constructors and associated patterns form part of that type's API, and other code may be written in terms of explicit pattern

SIDEBAR **Tuple types in Scala**

Pairs and other arity tuples are also algebraic data types. They work just like the ADTs we've been writing here, but have special syntax:

```
scala> val p = ("Bob", 42)
p: (java.lang.String, Int) = (Bob,42)

scala> p._1
res0: java.lang.String = Bob

scala> p._2
res1: Int = 42

scala> p match { case (a,b) => b }
res2: Int = 42
```

In this example, `("Bob", 42)` is a pair whose type is `(String,Int)`, which is syntax sugar for `Tuple2[String,Int]` (API link). We can extract the first or second element of this pair (a `Tuple3` will have a method _3 and so on), and we can pattern match on this pair much like any other `case class`. Higher arity tuples work similarly—try experimenting with them on the REPL if you're interested.

Algebraic data types can be used to define other data structures. Let's define a simple binary tree data structure:

```
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

Pattern matching again provides a convenient way of operating over elements of our ADT. Let's try writing a few functions.

EXERCISE 25: Write a function `size` that counts the number of nodes in a tree.

EXERCISE 26: Write a function `maximum` that returns the maximum element in a `Tree[Int]`. (Note: in Scala, you can use `x.max(y)` or `x max y` to compute the maximum of two integers `x` and `y`.)

EXERCISE 27: Write a function `depth` that returns the maximum path length from the root of a tree to any leaf.

EXERCISE 28: Write a function `map`, analogous to the method of the same

name on `List`, that modifies each element in a tree with a given function.

SIDEBAR **ADTs and encapsulation**

One might object that algebraic data types violate encapsulation by making public the internal representation of a type. In FP, we approach concerns about encapsulation a bit differently—we don't typically have delicate mutable state which could lead to bugs or violation of invariants if exposed publicly. Exposing the data constructors of a type is often fine, and the decision to do so is approached much like any other decision about what the public API of a data type should be.[17]

Footnote 17 It is also possible in Scala to expose patterns like `Nil` and `Cons` *independent of* the actual data constructors of the type.

We do typically use ADTs for cases where the set of cases is *closed* (known to be fixed). For `List` and `Tree`, changing the set of data constructors would significantly change what these data types are. List *is* a singly-linked list—that is its nature—and the two cases, `Nil` and `Cons` form part of its useful public API. We can certainly write code which deals with a more abstract API than `List` (we will see examples of this later in the book), but this sort of information hiding can be handled as a separate layer rather than being baked into `List` directly.

EXERCISE 29: Generalize `size`, `maximum`, `depth`, and `map`, writing a new function `fold` that abstracts over their similarities. Reimplement them in terms of this more general function. Can you draw an analogy between this `fold` function and the left and right folds for `List`?

3.6 Summary

In this chapter we covered a number of important concepts. We introduced algebraic data types and pattern matching and showed how to implement purely functional data structures, including the singly-linked list. Also, through the exercises in this chapter, we hope you got more comfortable writing pure functions and generalizing them. We will continue to develop this skill in the chapters ahead.[18]

Footnote 18 As you work through more of the exercises, you may want to read appendix Todo discussing different techniques for generalizing functions.

Index Terms

4 Handling errors without exceptions

4.1 Introduction

In chapter 1 we said that throwing an exception breaks referential transparency. Let's look at an example:

```
def failingFn(i: Int): Int = {
  val x: Int = throw new Exception("fail!")
  try {
    val y = 42 + 5
    x + y
  }
  catch { case e: Exception => 43 }
}
```

Unlike the expression `42 + 5`, which produces a result of `47`, the expression `throw new Exception("fail!")` does not produce a value at all—its "result" is to jump to the nearest enclosing `catch`, which depends on the context in which it is evaluated. The use of `throw` and `catch` means we can no longer reason about our program purely locally by substituting terms with their definitions—if we replace substitute `x` in `x + y` with `throw new Exception("fail!") + y`, our program has a different result.

How then do we write functional programs which handle errors? That is what you will learn in this chapter. The technique is based on a simple idea: instead of throwing an exception, we return a value indicating an exceptional condition has occurred. This idea might be familiar to anyone who has used return codes in C to handle exceptions, although in FP it works a bit differently, as we'll see.

4.2 Possible alternatives to exceptions

Let's consider a more realistic situation where we might use an exception and look at different approaches we could use instead. Here is an implementation of a function that computes the mean of a list, which is undefined if the list is empty:

```
def mean(xs: Seq[Double]): Double =                          (1)
  if (xs.isEmpty)
    throw new ArithmeticException("mean of empty list!")
  else xs.sum / xs.length                                    (2)
```

(1) Seq is the common interface of various linear sequence-like collections. Check the API docs for more information.
(2) sum is defined as a method on Seq using some magic (that we won't get into here) that makes the method available only if the elements of the sequence are numeric.

`mean` is an example of what is called a *partial function*: it is not defined for some inputs. A function is typically partial because it makes some assumptions about its inputs that are not implied by the input types.[1] You may be used to throwing exceptions in this case, but we have a few other options. Let's look at these for our `mean` example:

Footnote 1 A function may also be partial if it does not terminate for some inputs. We aren't going to discuss this form of partiality here—a running program cannot recover from or detect this nontermination internally, so there's no question of how best to handle it.

The first possibility is to return some sort of bogus value of type `Double`. We could simply return `xs.sum / xs.length` in all cases, and have it result in `0.0/0.0` when the input is empty, which is `Double.NaN`, or we could return some other sentinel value. In other situations we might return `null` instead of a value of the needed type. We reject this solution for a few reasons:

- It allows errors to silently propagate—the caller can forget to check this condition and will not be alerted by the compiler, which might result in subsequent code not working properly. Often the error won't be detected until much later in the code.
- It is not applicable to polymorphic code. For some output types, we might not even *have* a sentinel value of that type even if we wanted to! Consider a function like `max` which finds the maximum value in a sequence according to a custom comparison function: `def max[A](xs: Seq[A])(greater: (A,A) => Boolean): A`. If the input were empty, we cannot invent a value of type `A`. Nor can `null` be used here since `null` is only valid for non-primitive types, and `A` is completely unconstrained by this signature.
- It demands a special policy or calling convention of callers—proper use of the `mean` function now requires that callers do something other than simply call `mean` and make use of the result. Giving functions special policies like this makes it difficult to pass them to

> higher-order functions, who must treat all functions they receive as arguments uniformly and will generally only be aware of the types of these functions, not any special policy or calling convention.

The second possibility is to force the caller to supply an argument which tells us what to do in case we don't know how to handle the input:

```
def mean_1(xs: IndexedSeq[Double], onEmpty: Double): Double =
  if (xs.isEmpty) onEmpty
  else xs.sum / xs.length
```

This makes `mean` into a total function, but it has drawbacks—it requires that immediate callers have direct knowledge of how to handle the undefined case and limits them to returning a `Double`. What if `mean` is called as part of a larger computation and we would like to abort that computation if `mean` is undefined? Or perhaps we would like to take some completely different branch in the larger computation in this case? Simply passing an `onEmpty` parameter doesn't give us this freedom.

We need a way to defer the decision of how to handle undefined cases so that they can be dealt with at the most appropriate level.

4.3 The Option data type

The solution is to represent explicitly in the return type that we may not always have a defined value. We can think of this as deferring to the caller for the error handling strategy. We introduce a new type, `Option`:

```
sealed trait Option[+A]
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

`Option` has two cases: it can be defined, in which case it will be a `Some`, or it can be undefined, in which case it will be `None`. We can use this for our definition of `mean` like so:

```
def mean(xs: Seq[Double]): Option[Double] =
  if (xs.isEmpty) None
  else Some(xs.sum / xs.length)
```

The return type now reflects the possibility that the result is not always defined.

We still always return a result of the declared type (now `Option[Double]`) from our function, so `mean` is now a *total function*. It takes each value of the input type to exactly one value of the output type.

4.3.1 Usage patterns for Option

Partial functions abound in programming, and `Option` (and related data types we will discuss shortly) is typically how this partiality is dealt with in FP. You'll see `Option` used throughout the Scala standard library, for instance:

- Map lookup for a given key returns `Option`
- `headOption` and `lastOption` defined for lists and other iterables return `Option` containing the first or last elements of a sequence if it is nonempty.

These aren't the only examples—we'll see `Option` come up in many different situations. What makes `Option` convenient is that we can factor out common patterns of error handling via higher order functions, freeing us from writing the usual boilerplate that comes with exception-handling code.

`Option` can be thought of like a `List` that can contain at most one element, and many of the `List` functions we saw earlier have analogous functions on `Option`. Let's look at some of these functions. We are going to do something slightly different than last chapter. Last chapter we put all the functions that operated on `List` in the `List` companion object. Here we are going to place our functions, when possible, inside the body of the `Option` trait, so they can be called with OO syntax (`obj.fn(arg1)` or `obj fn arg1` instead of `fn(obj, arg1)`). This is a stylistic choice with no real significance, and we'll use both styles throughout this book.[2]

Footnote 2 In general, we'll use the OO style where possible for functions that have a single, clear operand (like `List.map`), and the standalone function style otherwise.

```
trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
  def getOrElse[B >: A](default: => B): B
  def orElse[B >: A](ob: => Option[B]): Option[B]
  def filter(f: A => Boolean): Option[A]
}
```

There are a couple new things here. The `default: => B` type annotation in `getOrElse` (and the similar annotation in `orElse`) indicates the argument will

not be evaluated until it is needed by the function. Don't worry about this for now—we'll talk much more about it in the next chapter. Also, the `B>:A` parameter on these functions indicates that `B` must be a *supertype* of `A`. It's needed to convince Scala that it is still safe to declare `Option[+A]` as covariant in `A` (which lets the compiler assume things like `Option[Dog]` is a subtype of `Option[Animal]`). Likewise for `orElse`. This isn't really important to our purposes; it's mostly an artifact of the OO style of placing the functions that operate on a type within the body of the `trait`.

EXERCISE 1: We'll explore when you'd use each of these next. But first, as an exercise, implement all of the above functions on `Option`. As you implement each function, try to think about what it means and in what situations you'd use it. Here are a few hints:

- It is fine to use pattern matching, though you should be able to implement all the functions besides `map` and `getOrElse` without resorting to pattern matching.
- For `map` and `flatMap`, the type signature should be sufficient to determine the implementation.
- `getOrElse` returns the result inside the `Some` case of the `Option`, or if the `Option` is `None`, returns the given default value.
- `orElse` returns the first `Option` if it is defined, otherwise, returns the second `Option`.

Although we can explicitly pattern match on an `Option`, we will almost always use the above higher order functions. We'll try to give some guidance for when to use each of them, but don't worry if it's not totally clear yet. The purpose here is mostly to get some basic familiarity so you can recognize these patterns as you start to write more functional code.

The `map` function can be used to transform the result inside an `Option`, if it exists. We can think of it as proceeding with a computation on the assumption that an error has not occurred—it is also a way of deferring the error handling to later code:

```
case class Employee(name: String, department: String)

val employeesByName: Map[String, Employee] =                    (1)
  List(Employee("Alice", "R&D"), Employee("Bob", "Accounting")).
  map(e => (e.name, e)).toMap                                   (2)

val dept: Option[String] = employeesByName.get("Joe").map(_.dept)
```

(1) A dictionary or associative container with String as the key type and Employee as

the value type
❷ Convenient method for converting a List[(A,B)] to a Map[A,B]

Here, `employeesByName.get` returns an `Option[Employee]`, which we transform using `map` to pull out the `Option[String]` representing the department.

`flatMap` is similar, except that the function we provide to transform the result can itself fail.

EXERCISE 2: Implement the `variance` function (if the mean is `m`, variance is the mean of `math.pow(x - m, 2)`, see definition) in terms of `mean` and `flatMap`.[3]

Footnote 3 Variance can actually be computed in one pass, but for pedagogical purposes we will compute it using two passes. The first will compute the mean of the data set, and the second will compute the mean squared difference from this mean.

```
def variance(xs: Seq[Double]): Option[Double]
```

`filter` can be used to convert successes into failures if the successful values don't match the given predicate. A common pattern is to transform an `Option` via calls to `map`, `flatMap`, and/or `filter`, then use `getOrElse` to do error handling at the end.

```
val dept: String =
  employeesByName.get("Joe").
  map(_.dept).
  filter(_ != "Accounting").
  getOrElse("Default Dept")
```

`getOrElse` is used here to convert from an `Option[String]` to a `String`, by providing a default department in case the key `"Joe"` did not exist in the `Map` or if Joe's department was `"Accounting"`.

`orElse` is similar to `getOrElse`, except that we return another `Option` if the first is undefined. This is often useful when we need to chain together possibly failing computations, trying the second if the first hasn't succeeded.

A common idiom is to do `o.getOrElse(throw AnException("FAIL"))` to convert the `None` case of an `Option` back to an

exception. The general rule of thumb is that we use exceptions only if no reasonable program would ever catch the exception—if for some callers the exception might be a recoverable error, we use `Option` to give them flexibility.

4.3.2 Option composition and lifting

It may be easy to jump to the conclusion that once we start using `Option`, it infects our entire code base. One can imagine how any callers of methods that take or return `Option` will have to be modified to handle either `Some` or `None`. But this simply doesn't happen, and the reason why is that we can *lift* ordinary functions to become functions that operate on `Option`.

For example, the `map` function lets us operate on values of type `Option[A]` using a function of type `A => B`, returning `Option[B]`. Another way of looking at this is that `map` turns a function `f` of type `A => B` into a function of type `Option[A] => Option[B]`. It might be more clear if we make this more explicit in the type signature:

```
def lift[A,B](f: A => B): Option[A] => Option[B] = _ map f
```

As an example of when you might use `map`, let's look at one more example of a function that returns `Option`:

```
import java.util.regex._

def pattern(s: String): Option[Pattern] =
  try {
    Some(Pattern.compile(s))
  } catch {
    case e: PatternSyntaxException => None
  }
```

This example uses the Java standard library's regex package to parse a string into a regular expression pattern.[4] If there is a syntax error in the pattern (it's not a valid regular expression), we catch the exception thrown by the library function and return `None`. Methods on the `Pattern` class don't need to know anything about `Option`. We can simply *lift* them using the `map` function:

Footnote 4 Scala runs on the Java Virtual Machine and is completely compatible with all existing Java libraries. We can therefore call `Pattern.compile`, a Java function, exactly as we would any Scala function.

```
def mkMatcher(pat: String): Option[String => Boolean] =
  pattern(pat) map (p => (s: String) => p.matcher(s).matches) (1)
```

(1) The details of this API don't matter too much, but p.matcher(s).matches will check if the string s matches the pattern p.

Here, the `pattern(pat)` call will return an `Option[Pattern]`, which will be `None` if `pat` is not valid. Notice how we are using the `matcher` and `matches` functions *inside* the `map`. Because they are inside the `map`, they don't need to be aware of the outer containing `Option`. If you don't feel you fully grasp how this works yet, use the substitution model to execute this on paper step by step, both for the case where `pat` is valid and where it is invalid.

It's also possible to lift a function by using a *for-comprehension*, which in Scala is a convenient syntax for writing a sequence of nested calls to `map` and `flatMap`. We'll explain the translation in a minute. First, here are some examples:

```
def mkMatcher_1(pat: String): Option[String => Boolean] =
  for {
    p <- pattern(pat)
  } yield ((s: String) => p.matcher(s).matches)

def doesMatch(pat: String, s: String): Option[Boolean] =
  for {
    p <- mkMatcher_1(pat)
  } yield p(s)
```

So far we are only lifting functions that take one argument. But some functions take more than one argument and we would like to be able to lift them too. The for-comprehension makes this easy, and we can combine as many options as we want:

```
def bothMatch(pat: String, pat2: String, s: String): Option[Boolean] =
  for {
    f <- mkMatcher(pat)
    g <- mkMatcher(pat2)
  } yield f(s) && g(s)
```

A for-comprehension like this is simply syntax sugar. Internally, Scala will translate the above to ordinary method calls to `map` and `flatMap`:

```
def bothMatch_1(pat: String, pat2: String, s: String): Option[Boolean] =
```

```
  mkMatcher(pat) flatMap (f =>
  mkMatcher(pat2) map     (g =>
  f(s) && g(s)))
```

EXERCISE 3: `bothMatch` is an instance of a more general pattern. Write a generic function `map2`, that combines two `Option` values using a binary function. If either `Option` value is `None`, then the return value is too. Here is its signature:

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C]
```

EXERCISE 4: Re-implement `bothMatch` above in terms of this new function, to the extent possible.

```
def bothMatch_2(pat1: String, pat2: String, s: String): Option[Boolean]
```

EXERCISE 5: Write a function `sequence`, that combines a list of Options into one option containing a list of all the `Some` values in the original list. If the original list contains `None` even once, the result of the function should be `None`, otherwise the result should be `Some` with a list of all the values. Here is its signature:[5]

Footnote 5 This is a clear instance where it's not possible to define the function in the OO style. This should not be a method on `List` (which shouldn't need to know anything about `Option`), and it can't be a method on `Option`.

```
def sequence[A](a: List[Option[A]]): Option[List[A]]
```

Sometimes we will want to map over a list using a function that might fail, returning `None` if applying it to any element of the list returns None. For example, parsing a whole list of strings into a list of patterns. In that case, we can simply sequence the results of the `map`:

```
def parsePatterns(a: List[String]): Option[List[Pattern]] =
  sequence(a map pattern)
```

Unfortunately, this is a little inefficient, since it traverses the list twice. Wanting to sequence the results of a `map` this way is a common enough occurrence to

warrant a new generic function `traverse`, with the following signature:

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]]
```

EXERCISE 6: Implement this function. It is straightforward to do using `map` and `sequence`, but try for a more efficient implementation that only looks at the list once. In fact, implement `sequence` in terms of `traverse`.

4.3.3 The Either data type

The big idea in this chapter is that we can represent failures and exceptions with ordinary values, and we can write functions that abstract out common patterns of error handling and recovery. `Option` is not the only data type we could use for this purpose, and although it gets used quite frequently, it's rather simplistic. One thing you may have noticed with `Option` is that it doesn't tell us very much about what went wrong in the case of an exceptional condition. All it can do is give us `None` indicating that there is no value to be had. But sometimes we want to know more. For example, we might want a `String` that gives more information, or if an exception was raised, we might want to know what that error actually was.

We can craft a data type that encodes whatever information we want about failures. Sometimes just knowing whether a failure occurred is sufficient in which case we can use `Option`; other times we want more information. In this section, we'll walk through a simple extension to `Option`, the `Either` data type, which lets us track a *reason* for the failure. Let's look at its definition:

```
sealed trait Either[+E, +A]
case class Left[+E](value: E) extends Either[E, Nothing]
case class Right[+A](value: A) extends Either[Nothing, A]
```

`Either` has only two cases, just like `Option`. The essential difference is that both cases carry a value. The `Either` data type represents, in a very general way, values that can be one of two things. We can say that it is a *disjoint union* of two types. When we use it to indicate success or failure, by convention the `Left` constructor is reserved for the failure case.[6]

Footnote 6 `Either` is also often used more generally to encode one of two possibilities, in cases where it isn't worth defining a fresh data type. We'll see some examples of this throughout the book.

SIDEBAR **Option and Either in the standard library**

Both `Option` and `Either` exist in the Scala standard library (Option API link and Either API link), and most of the functions we've defined here in this chapter exist for the standard library versions. There are a few missing functions, though, notably `sequence`, `traverse`, and `map2`.

You are encouraged to read through the API for `Option` and `Either` to understand the differences. `Either` in particular, does not define a right-biased `flatMap` directly like we did here, but relies on explicit `left` and `right` projection functions. Read the API for details.

Let's look at the `mean` example again, this time returning a `String` in case of failure:

```
def mean(xs: IndexedSeq[Double]): Either[String, Double] =
  if (xs.isEmpty)
    Left("mean of empty list!")
  else
    Right(xs.sum / xs.length)
```

Sometimes we might want to include more information about the error, for example a stack trace showing the location of the error in the source code. In such cases we can simply return the exception in the `Left` side of an `Either`:

```
def safeDiv(x: Double, y: Double): Either[Exception, Double] =
  try {
    Right(x / y)
  } catch {
    case e: Exception => Left(e)
  }
```

EXERCISE 7: Implement versions of `map`, `flatMap`, `orElse`, and `map2` on `Either` that operate on the `Right` value.

```
trait Either[+E, +A] {
  def map[B](f: A => B): Either[E, B]
  def flatMap[EE >: E, B](f: A => Either[EE, B]): Either[EE, B]
  def orElse[EE >: E,B >: A](b: => Either[EE, B]): Either[EE, B]
  def map2[EE >: E, B, C](b: Either[EE, B])(f: (A, B) => C):
    Either[EE, C]
}
```

Note that with these definitions, `Either` can now be used in for-comprehensions, for instance:

```
for {
  age <- Right(42)
  name <- Left("invalid name")
  salary <- Right(1000000.0)
} yield employee(name, age, salary)
```

This will result in `Left("invalid name")`. Of course, `Left("invalid name")` could be an arbitrary expression like `foo(x,y,z)` that happens to result in a `Left`.

EXERCISE 8: Implement `sequence` and `traverse` for `Either`.

As a final example, here is an application of `map2`, where the function `mkPerson` validates both the given name and the given age before constructing a valid `Person`:

```
case class Person(name: Name, age: Age)
sealed class Name(val value: String)
sealed class Age(val value: Int)

def mkName(name: String): Either[String, Name] =
  if (name == "" || name == null) Left("Name is empty.")
  else Right(new Name(name))

def mkAge(age: Int): Either[String, Age] =
  if (age < 0) Left("Age is out of range.")
  else Right(new Age(age))

def mkPerson(name: String, age: Int): Either[String, Person] =
  mkName(name).map2(mkAge(age))(Person(_, _))
```

EXERCISE 9: In this implementation, `map2` is only able to report one error, even if both the name and the age are invalid. What would you need to change in order to report *both* errors? Would you change `map2` or the signature of `mkPerson`? Or could you create a new data type that captures this requirement better than `Either` does, with some additional structure? How would `orElse`, `traverse`, and `sequence` behave differently for that data type?

4.4 Conclusion

Using algebraic data types such as `Option` and `Either`, we can handle errors in a way that is modular, compositional, and simple to reason about. In this chapter, we have developed a number of higher-order functions that manipulate errors in ways that we couldn't otherwise if we were just throwing exceptions. With these new tools in hand, exceptions should be reserved only for truly unrecoverable conditions in our programs.

Index Terms

disjoint union
for-comprehension
lifting
lifting

5 Strictness and laziness

In chapter 3 we talked about purely functional data structures, using singly-linked lists as an example. We covered a number of bulk operations on lists — `map`, `filter`, `foldLeft`, `foldRight`, `zip`, etc. We noted that each of these operations makes its own pass over the input and constructs a fresh list for the output.

Imagine if you had a deck of cards and you were asked to remove the odd-numbered cards and then remove all the queens. Ideally, you would make a single pass through the deck, looking for queens and odd-numbered cards at the same time. This is more efficient than removing the odd cards and then looking for queens in the remainder. And yet the latter is what Scala is doing in the following code:[1]

Footnote 1 We are using the Scala standard library's `List` type here, where `map` and `filter` are methods on `List` rather than standalone functions like those we wrote in chapter 3.

```
scala> List(1,2,3,4) map (_ + 10) filter (_ % 2 == 0) map (_ * 3)
List(36,42)
```

In this expression, `map (_ + 10)` will produce an intermediate list that then gets passed to `filter (_ % 2 == 0)`, which in turn constructs a list which gets passed to `map (_ * 3)` which then produces the final list. In other words, each transformation will produce a temporary list that only ever gets used as input to the next transformation and is then immediately discarded.

Think about how this program will be evaluated. If we manually produce a trace of its evaluation, the steps would look something like this:

```
List(1,2,3,4) map (_ + 10) filter (_ % 2 == 0) map (_ * 3)
List(11,12,13,14) filter (_ % 2 == 0) map (_ * 3)
List(12,14) map (_ * 3)
List(36,42)
```

Here we are showing the result of each substitution performed to evaluate our expression (for example, to go from the first line to the second, we have substituted `List(1,2,3,4) map (_ + 10)` with `List(11,12,13,14)`, based on the definition of `map`).[2] This view makes it clear how the calls to `map` and `filter` each perform their own traversal of the input and allocate lists for the output. Wouldn't it be nice if we could somehow fuse sequences of transformations like this into a single pass and avoid creating temporary data structures? We could rewrite the code into a while-loop by hand, but ideally we'd like to have this done automatically while retaining the same high-level compositional style. We want to use higher-order functions like `map` and `filter` instead of manually fusing passes into loops.

Footnote 2 With program traces like these, it is often more illustrative to not fully trace the evaluation of every subexpression. For instance, in this case, we've omitted the full expansion of `List(1,2,3,4) map (_ + 10)`. We could "enter" the definition of `map` and trace its execution but we chose to omit this level of detail in this trace.

It turns out that we can accomplish this through the use of *non-strictness* (or more informally, "laziness"). In this chapter, we will explain what exactly this means, and we'll work through the implementation of a lazy list type that fuses sequences of transformations. Although building a "better" list is the motivation for this chapter, we'll see that non-strictness is a fundamental technique for improving on the efficiency and modularity of functional programs in general.

5.1 Strict and non-strict functions

Before we get to our example of lazy lists, we need to cover some basics. What is strictness and non-strictness, and how are these concepts expressed in Scala?

SIDEBAR **Termination and strictness**

If the evaluation of an expression runs forever or throws an error instead of returning a definite value, we say that the expression does not *terminate*, or that it evaluates to *bottom*. A function `f` is `strict` if the expression `f(x)` evaluates to bottom for all `x` that evaluate to bottom.

Non-strictness is a property of a function. To say a function is non-strict just

means that the function may choose *not* to evaluate one or more of its arguments. In contrast, a *strict* function always evaluates its arguments. Strict functions are the norm in most programming languages and most languages don't even provide a way to define non-strict functions. Unless you tell it otherwise, any function definition in Scala will be strict (and all the functions we have defined so far have been strict). As an example, consider the following function:

```
def square(x: Double): Double = x * x
```

When you invoke `square(41.0 + 1.0)` square will receive the evaluated value of `42.0` because it is strict. If you were to invoke `square(sys.error("failure"))`, you would get an exception, since the `sys.error("failure")` expression will be evaluated before entering the body of `square`.

Although we haven't yet learned the syntax for indicating non-strictness in scala, you are almost certainly familiar with non-strictness *as a concept*. For example, the Boolean functions `&&` and `||` are non-strict. You may be used to thinking of `&&` and `||` as built-in syntax, part of the language, but we can also think of them as functions that may choose not to evaluate their arguments. The function `&&` takes two `Boolean` arguments, but only evaluates the second argument if the first is `true`:

```
scala> false && { println("!!"); true } // does not print anything
res0: Boolean = false
```

And `||` only evaluates its second argument if the first is `false`:

```
scala> true || { println("!!") // doesn't print anything either }
res1: Boolean = true
```

Another example of non-strictness is the `if` control construct in Scala:

```
val result = if (input.isEmpty) sys.error("empty input") else input
```

The `if` language construct could also be thought of as a function accepting three parameters: a condition of type `Boolean`, an expression of some type `A` to

return in the case that the condition is true, and another expression of the same type `A` to return if the condition is false. This `if` function would be non-strict, since it will not evaluate all of its arguments. To be more precise, we would say that the `if` function is strict in its condition parameter, since it will always evaluate the condition to determine which branch to take, and non-strict in the two branches for the `true` and `false` cases, since it will only evaluate one or the other based on the condition.

In Scala, we can write non-strict functions by accepting some of our arguments unevaluated, using the following syntax:

```
def if2[A](cond: Boolean, onTrue: => A, onFalse: => A): A =
  if (cond) onTrue else onFalse
```

The arguments we would like to pass unevaluated have `=>` immediately before their type. In the body of the function, we do not need to do anything to evaluate an argument annotated with `=>`. We just reference the identifier. We also call this function with the usual function call syntax:

```
scala> if2(false, sys.error("fail"), 3)
res2: Int = 3
```

Scala will take care of making sure that the second and third arguments are passed unevaluated to the body of `if2`.[3]

Footnote 3 The unevaluated form of an expression is often called a *thunk*. Thunks are represented at runtime in Scala as a value of type `scala.Function0`, which you can see if you're curious by inspecting the signature of non-strict functions in the `.class` file the Scala compiler generates.

An argument that is passed unevaluated to a function will be evaluated once for each place it is referenced in the body of the function. That is, Scala will not (by default) cache the result of evaluating an argument:

```
scala> def pair(i: => Int) = (i, i)
scala> pair { println("hi"); 1 + 41 }
hi
hi
val res3: (Int, Int) = (42,42)
```

Here, `i` is referenced twice in the body of `pair`, and we have made it

particularly obvious that it is evaluated each time by passing the block `println("hi"); 1 + 41` which prints `hi` as a side effect before returning a result of `42`. The expression `1 + 41` will be computed twice as well. Luckily we can cache the value explicitly if we wish to only evaluate the result once, by using the `lazy` keyword:

```
scala> def pair2(i: => Int) = { lazy val j = i; (j, j) }
scala> pair2 { println("hi"); 1 + 41 }
hi
val res4: (Int, Int) = (42,42)
```

Adding the `lazy` keyword to a `val` declaration will cause Scala to delay evaluation of the right hand side until it is first referenced and will also cache the result so that subsequent references of `j` don't trigger repeated evaluation. In this example, we were going to evaluate `i` on the next line anyway, so we could have just written `val j = i`. Despite receiving its argument unevaluated, `pair2` is still considered a strict function since it always ends up evaluating its argument. In other situations, we can use `lazy val` when we don't know if subsequent code will evaluate the expression and simply want to cache the result if it is ever demanded.

As a final bit of terminology, a non-strict function that evaluates its arguments each time it references them is said to evaluate those arguments *by name*; if it evaluates them only once and then caches their value, it is said to evaluate *by need*, or it's said to be *lazy*. We'll often refer to unevaluated parameters in Scala as *by-name parameters*. Note also that the terms *laziness* or *lazy evaluation* are sometimes used informally to refer to any sort of non-strict evaluation, not necessarily evaluation by need. When you encounter the word "lazy" in this book, you can assume that we are using an informal definition.

5.2 An extended example: lazy lists

Let's now return to the problem posed at the beginning of this chapter. We are going to explore how laziness can be used to improve the efficiency and modularity of functional programs, using *lazy lists*, or *streams* as an example. We'll see how chains of transformations on streams are fused into a single pass, through the use of laziness. Here is a simple `Stream` definition:[4]

Footnote 4 There are some subtle possible variations on this definition of `Stream`. We'll touch briefly on some of these variations later in this chapter.

```
trait Stream[+A] {
  def uncons: Option[(A, Stream[A])]
  def isEmpty: Boolean = uncons.isEmpty
}

object Stream {

def empty[A]: Stream[A] =
  new Stream[A] { def uncons = None }

def cons[A](hd: => A, tl: => Stream[A]): Stream[A] =
  new Stream[A] {
    lazy val uncons = Some((hd, tl))
  }

def apply[A](as: A*): Stream[A] =
  if (as.isEmpty) empty
  else cons(as.head, apply(as.tail: _*))
}
```

Notice the `cons` function is non-strict in its arguments. Thus the head and tail of the stream will not be evaluated until first requested. We'll sometimes write `cons` using the infix, right associative operator `#::`, so `1 #:: 2 #:: empty` is equivalent to `cons(1, cons(2, empty))`. We could add it to the `Stream` trait in the same way that we added `::` to the `List` trait in the code for chapter 3, though there are some annoying additional hoops to jump through to make it properly non-strict. See the associated code for this chapter if you're interested in exactly how this syntax is implemented.

Before continuing, let's write a few helper functions to make inspecting streams easier.

EXERCISE 1: Write a function to convert a `Stream` to a `List`, which will force its evaluation and let us look at it in the REPL. You can convert to the regular `List` type in the standard library. You can place this and other functions that accept a `Stream` inside the `Stream` trait.

```
def toList: List[A]
```

EXERCISE 2: Write a function `take` for returning the first `n` elements of a `Stream`.

```
def take(n: Int): Stream[A]
```

EXERCISE 3: Write the function `takeWhile` for returning all starting elements of a `Stream` that match the given predicate.

```
def takeWhile(p: A => Boolean): Stream[A]
```

You can use `take` and `toList` together to inspect the streams we'll be creating. For example, try printing `Stream(1,2,3).take(2).toList`.

5.3 Separating program description from evaluation

Laziness lets us separate the description of an expression from the evaluation of that expression. This gives us a powerful ability — we may choose to describe a "larger" expression than we need, then evaluate only a portion of it. As an example, consider `foldRight` — we can implement this for `Stream` much like we did for `List`, but we can implement it lazily:

```
def foldRight[B](z: => B)(f: (A, => B) => B): B =
  uncons match {
    case Some((h, t)) => f(h, t.foldRight(z)(f))
    case None => z
  }
```

This looks very similar to the `foldRight` we wrote for `List`, but notice how our combining function, `f`, is non-strict in its second parameter. If `f` chooses not to evaluate its second parameter, this terminates the traversal early. We can see this by using `foldRight` to implement `exists`, which checks to see if any value in the `Stream` matches a given predicate.

```
def exists(p: A => Boolean): Boolean =
  foldRight(false)((a, b) => p(a) || b)
```

Since `foldRight` can terminate the traversal early, we can reuse it to implement `exists` rather than writing an explicit recursive function to handle early termination. This is a simple example where separating the concerns of *describing* a computation from the concern of *evaluation* makes our descriptions more reusable than when these concerns are intertwined. This kind of separation of concerns is a central theme in functional programming.

EXERCISE 4: Implement `forAll`, which checks that all elements in the

`Stream` match a given predicate. Your implementation should terminate the traversal as soon as it encounters a non-matching value.

```
def forAll(p: A => Boolean): Boolean
```

EXERCISE 5: Use `foldRight` to implement `takeWhile`. This will construct a stream incrementally, and only if the values in the result are demanded by some other expression.

EXERCISE 6: Implement `map`, `filter`, `append`, and `flatMap` using `foldRight`.

Because the implementations are incremental, chains of transformations will avoid fully instantiating the intermediate data structures. Let's look at a simplified program trace for (a fragment of) the motivating example we started this chapter with, `Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0)`. Take a minute to work through this trace to understand what's happening. It's a bit more challenging than the trace we looked at earlier in this chapter.

```
Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0)
(11 #:: Stream(2,3,4).map(_ + 10)).filter(_ % 2 == 0)          ❶
Stream(2,3,4).map(_ + 10).filter(_ % 2 == 0)                   ❷
(12 #:: Stream(3,4).map(_ + 10)).filter(_ % 2 == 0)            ❸
12 #:: Stream(3,4).map(_ + 10).filter(_ % 2 == 0)              ❹
12 #:: (13 #:: Stream(4).map(_ + 10)).filter(_ % 2 == 0)
12 #:: Stream(4).map(_ + 10).filter(_ % 2 == 0)
12 #:: (14 #:: Stream().map(_ + 10)).filter(_ % 2 == 0)
12 #:: 14 #:: Stream().map(_ + 10).filter(_ % 2 == 0)
12 #:: 14 #:: Stream()
```

❶ Apply map to first element
❷ Apply filter to first element
❸ Apply map to second element
❹ Apply filter to second element

Notice how the `filter` and `map` transformations are interleaved—the computation alternates between generating a single element of the output of `map`, and `filter` testing to see if that element is divisible by `2` (adding it to the output stream if it is), exactly as if we had interleaved these bits of logic in a special-purpose loop that combined both transformations. Notice we do not fully

instantiate the intermediate stream that results from the `map`. For this reason, people sometimes describe streams as "first-class loops" whose logic can be combined using higher-order functions like `map` and `filter`.

The incremental nature of stream transformations also has important consequences for memory usage. In a sequence of stream transformations like this, the garbage collector can usually reclaim the space needed for each intermediate stream element, *as soon as* that element is passed on to the next transformation. Here, for instance, the garbage collector can reclaim the space allocated for the value `13` emitted by `map` as soon as `filter` determines it isn't needed. Of course, this is a simple example; in other situations we might be dealing with larger numbers of elements, and the stream elements themselves could be large objects that retain significant amounts of memory. Being able to reclaim this memory as quickly as possible can cut down on the amount of memory required by your program as a whole.[5]

Footnote 5 We will have a lot more to say about defining memory-efficient streaming calculations, in particular calculations that require I/O, in part 4 of this book.

5.4 Infinite streams and corecursion

Because they are incremental, the functions we've written also work fine for *infinite streams*. Here is an example of an infinite `Stream` of `1`s:

```
val ones: Stream[Int] = cons(1, ones)
```

Although `ones` is infinite, the functions we've written so far only inspect the portion of the stream needed to generate the demanded output. For example:

```
scala> ones.take(5).toList
res0: List[Int] = List(1, 1, 1, 1, 1)

scala> ones.exists(_ % 2 != 0)
res1: Boolean = true
```

Try playing with a few other examples:

- `ones.map(_ + 1).exists(_ % 2 == 0)`
- `ones.takeWhile(_ == 1)`
- `ones.forAll(_ != 1)`

In each case, we get back a result immediately. Be careful though, since it's easy to write an expression that never terminates. For example `ones.forAll(_ == 1)` will forever need to inspect more of the series since it will never encounter an element that allows it to terminate with a definite answer.

Let's see what other functions we can discover for generating streams.

EXERCISE 7: Generalize `ones` slightly to the function `constant` which returns an infinite `Stream` of a given value.

```
def constant[A](a: A): Stream[A]
```

EXERCISE 8: Write a function that generates an infinite stream of integers, starting from `n`, then `n + 1`, `n + 2`, etc.

```
def from(n: Int): Stream[Int]
```

EXERCISE 9: Write a function `fibs` that generates the infinite stream of Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, and so on.

EXERCISE 10: We can write a more general stream building function. It takes an initial state, and a function for producing both the next state and the next value in the generated stream. It is usually called `unfold`:

```
def unfold[A, S](z: S)(f: S => Option[(A, S)]): Stream[A]
```

Option is used to indicate when the `Stream` should be terminated, if at all. The function `unfold` is the most general `Stream`-building function. Notice how closely it mirrors the structure of the `Stream` data type.

`unfold` and the functions we can implement with it are examples of what is sometimes called a *corecursive* function. While a recursive function consumes data and eventually terminates, a corecursive function produces data and *coterminates*. We say that such a function is *productive*, which just means that we can always evaluate more of the result in a finite amount of time (for `unfold`, we just need to run the function `f` one more time to generate the next element). Corecursion is also sometimes called *guarded recursion*. These terms aren't that important to our

discussion, but you will hear them used sometimes in the context of functional programming. If you are curious to learn where they come from and understand some of the deeper connections, follow the references in the chapter notes.

EXERCISE 11: Write `fibs`, `from`, `constant`, and `ones` in terms of `unfold`.

EXERCISE 12: Use `unfold` to implement `map`, `take`, `takeWhile`, `zip` (as in chapter 3), and `zipAll`. The `zipAll` function should continue the traversal as long as either stream has more elements — it uses `Option` to indicate whether each stream has been exhausted.

Now that we have some practice writing stream functions, let's return to the exercise we covered at the end of chapter 3 — a function, `hasSubsequence`, to check whether a list contains a given subsequence. With strict lists and list-processing functions, we were forced to write a rather tricky monolithic loop to implement this function without doing extra work. Using lazy lists, can you see how you could implement `hasSubsequence` by combining some other functions we have already written? Try to think about it on your own before continuing.

EXERCISE 13 (hard): implement `startsWith` using functions you've written. It should check if one `Stream` is a prefix of another. For instance, `Stream(1,2,3) starsWith Stream(1,2)` would be `true`.

```
def startsWith[A](s: Stream[A], s2: Stream[A]): Boolean
```

EXERCISE 14: implement `tails` using `unfold`. For a given `Stream`, `tails` returns the `Stream` of suffixes of the input sequence, starting with the original `Stream`. So, given `Stream(1,2,3)`, it would return `Stream(Stream(1,2,3), Stream(2,3), Stream(3), Stream.empty)`.

```
def tails: Stream[Stream[A]]
```

We can now implement `hasSubsequence` using functions we've written:

```
def hasSubsequence[A](s1: Stream[A], s2: Stream[A]): Boolean =
  s1.tails exists (startsWith(_,s2))
```

This implementation performs the same number of steps as a more monolithic implementation using nested loops with logic for breaking out of each loop early. By using laziness, we can compose this function from simpler components and still retain the efficiency of the more specialized (and verbose) implementation.

EXERCISE 15 (hard, optional): Generalize `tails` to the function `scanRight`, which is like a `foldRight` that returns a stream of the intermediate results. For example:

```
scala> Stream(1,2,3).scanRight(0)(_ + _).toList
res0: List[Int] = List(6,5,3,0)
```

This example would be equivalent to the expression `List(1+2+3+0, 2+3+0, 3+0, 0)`. Your function should reuse intermediate results so that traversing a `Stream` with `n` elements always takes time linear in `n`. Can it be implemented using `unfold`? How, or why not? Could it be implemented using another function we have written?

5.5 Summary

In this chapter we have introduced non-strictness as a fundamental technique for implementing efficient and modular functional programs. As we have seen, while non-strictness can be thought of as a technique for recovering some efficiency when writing functional code, it's also a much bigger idea — non-strictness can improve modularity by separating the description of an expression from the "how and when" of its evaluation. Keeping these concerns separate lets us reuse a description in multiple contexts, evaluating different portions of our expression to obtain different results. We were not able to do that when description and evaluation were intertwined as they are in strict code. We saw a number of examples of this principle in action over the course of the chapter and we will see many more in the remainder of the book.

Index Terms

non-strict
strict
thunk

6 Purely functional state

In this chapter, we will be introducing how to write programs that manipulate state in a purely functional way, using the very simple domain of *random number generation* as motivation. Although this is an unambitious domain on its own, we'll be building on what we develop here in chapters to come, and its simplicity makes it a good place to explore some fundamental issues that you will likely encounter as you start writing your own functional APIs.

A word before getting started: don't worry if not everything in this chapter sinks in at first. The goal is more to give you the basic pattern for how to make stateful APIs purely functional. We will say a lot more about dealing with state and effects in later parts of the book.

6.1 Generating random numbers using side-effects

If you need to generate random numbers in Scala, there's a class in Java's standard library, `java.util.Random` (API link), with a pretty typical imperative API that relies on side effects. Here's an example of its use:

```
cala> val rng = new java.util.Random
rng: java.util.Random = java.util.Random@caca6c9

scala> rng.nextDouble
res1: Double = 0.9867076608154569

scala> rng.nextDouble
res2: Double = 0.8455696498024141

scala> rng.nextInt
res3: Int = -623297295

scala> rng.nextInt
res4: Int = 1989531047
```

And here's an excerpt of the API, transcribed to Scala:

```
trait Random {
  def nextInt: Int
  def nextBoolean: Boolean
  def nextDouble: Double
  ...
}
```

Even if we didn't know anything about what happens inside a `java.util.Random`, we can assume that this class has some internal state that gets updated after each invocation, since we would otherwise get the same "random" value each time. Because these state updates are performed as a side effect, these methods are not referentially transparent.

6.2 Purely functional random number generation

What can we do about this? The key to recovering referential transparency is to make these state updates *explicit*. That is, do not update the state as a side effect, but simply return the new state along with the value we are generating. Here's one possible interface:

```
trait RNG {
  def nextInt: (Int, RNG)                    (1)
}
```

(1) Should generate a random Int. We will later define other functions in terms of nextInt.

Rather than returning only the generated random number (as is done in `java.util.Random`) and updating some internal state by *mutating* it in place, we return the random number and the new state, leaving the old state unmodified. In effect, we separate the *computing* of the next state from the concern of *propagating* that state throughout the program. There is no global mutable memory being used—we simply return the next state back to the caller. This leaves the caller of `nextInt` in complete control of what to do with the new state. Notice that we are still *encapsulating* the state, in the sense that users of this API do not need to know anything about the implementation of the random number generator itself.

Here is a simple implementation using the same algorithm as

`java.util.Random`, which happens to be a kind of random number generator called a linear congruential generator. The details of this implementation aren't really important, but notice that `nextInt` returns both the generated value and a new `RNG` to use for generating the next value.

```
object RNG {
def simple(seed: Long): RNG = new RNG {
  def nextInt = {
    val seed2 = (seed*0x5DEECE66DL + 0xBL) &                    (1)
                ((1L << 48) - 1)                                (2)
    ((seed2 >>> 16).asInstanceOf[Int],                          (3)
     simple(seed2))
  }
}
}
```

(1) & is bitwise AND
(2) << is left binary shift
(3) >>> is right binary shift with zero fill

This problem of making seemingly stateful APIs pure, and its solution, of having the API *compute* the next state rather than actually mutate anything, is not unique to random number generation. It comes up quite frequently, and we can always deal with it in this same way.[1]

Footnote 1 There is an efficiency loss that comes with computing next states using pure functions, because it means we cannot actually mutate the data in place. (Here, it is not really a problem since the state is just a single `Long` that must be copied.) This loss of efficiency can be mitigated by using efficient purely functional data structures. It's also possible in some cases to mutate the data in place without breaking referential transparency. We'll be discussing this in Part 4.

For instance, suppose you have a class like this:

```
class Foo {
  var s: FooState = ...
  def bar: Bar
  def baz: Int
}
```

Suppose `bar` and `baz` each mutate `s` in some way. We can mechanically translate this to the purely functional API:

```
trait Foo {
  def bar: (Bar, Foo)
  def baz: (Int, Foo)
```

```
}
```

In both of these cases, we are making state propagation explicit.

Whenever we use this pattern, we make users of the API responsible for threading the computed next state through their programs. For the pure `RNG` interface above, if you reuse a previous `RNG`, it will always generate the same value it generated before. For instance:

```
def randomPair(rng: RNG): (Int,Int) = {
  val (i1,_) = rng.nextInt
  val (i2,_) = rng.nextInt
  (i1,i2)
}
```

Here, `i1` and `i2` will be the same! If we want to generate two distinct numbers, we need to use the `RNG` returned by the first call to `nextInt` to generate the second `Int`.

```
def randomPair(rng: RNG): ((Int,Int), RNG) = {
  val (i1,rng2) = rng.nextInt
  val (i2,rng3) = rng2.nextInt                                    (1)
  ((i1,i2), rng3)                                                 (2)
}
```

(1) Notice use of rng2 here.
(2) Notice we return the final state, after generating the two random numbers. This lets the caller generate more random values using the new state.

You can see the general pattern, and perhaps you can also see how it might get somewhat tedious to use this API directly. Let's write a few functions to generate random values and see if we notice any patterns we can factor out.

EXERCISE 1: Write a function to generate a random positive integer. Note: you can use `x.abs` to take the absolute value of an `Int`, `x`. Make sure to handle the corner case `Int.MinValue`, which doesn't have a positive counterpart.

```
def positiveInt(rng: RNG): (Int, RNG)
```

SIDEBAR **Dealing with awkwardness in FP**

As you write more functional programs, you'll sometimes encounter situations like this where the functional way of expressing a program feels awkward or tedious. Does this imply that FP is the programming equivalent of trying to write an entire novel without using the letter 'e'? Of course not. Awkwardness like this is almost always a sign of some missing abstraction waiting to be discovered!

When you encounter these situations, we encourage you to plow ahead and look for common patterns you can factor out. Most likely, this is a problem others have encountered, and you may even rediscover the "standard" solution yourself. Even if you get stuck, struggling to puzzle out a clean solution yourself will help you to better understand what solutions others have discovered to deal with the same or similar problems.

With practice, experience, and more familiarity with the idioms of FP, expressing a program functionally will become effortless and natural.[2]

Footnote 2 Of course, good design is still hard, but programming using pure functions becomes easy with experience.

EXERCISE 2: Write a function to generate a `Double` between `0` and `1`, not including `1`. Note: you can use `Int.MaxValue` to obtain the maximum positive integer value and you can use `x.toDouble` to convert an `Int`, `x`, to a `Double`.

```
def double(rng: RNG): (Double, RNG)
```

EXERCISE 3: Write functions to generate an `(Int, Double)` pair, a `(Double, Int)` pair, and a `(Double, Double, Double)` 3-tuple. You should be able to reuse the functions you've already written.

```
def intDouble(rng: RNG): ((Int,Double), RNG)
def doubleInt(rng: RNG): ((Double,Int), RNG)
def double3(rng: RNG): ((Double,Double,Double), RNG)
```

EXERCISE 4: Write a function to generate a list of random integers.

```
def ints(count: Int)(rng: RNG): (List[Int], RNG)
```

6.3 A better API for state actions

Looking back at our implementations, we notice a common pattern: each of our functions has a type of the form `RNG => (A, RNG)` for some type `A`. Functions of this type describe *state actions* that transform `RNG` states, and these state actions can be built up and combined using general-purpose functions.

To make them convenient to talk about, let's make a type alias for the `RNG` state action data type:

```
type Rand[+A] = RNG => (A, RNG)
```

We can now turn methods such as `RNG`'s `nextInt` into values of this type:

```
val int: Rand[Int] = _.nextInt
```

We want to start writing combinators that let us avoid explicitly passing along the `RNG` state. This will become a kind of domain-specific language that does all of this passing for us. For example, a simple `RNG`-transition is the `unit` action, which passes the `RNG` state through without using it, always returning a constant value rather than a random value.

```
def unit[A](a: A): Rand[A] =
  rng => (a, rng)
```

There is also `map`, for transforming the output of a state action without modifying the state itself. Remember, `Rand[A]` is just a type alias for a function type `RNG => (A, RNG)`, so this is just a kind of function composition.

```
def map[A,B](s: Rand[A])(f: A => B): Rand[B] =
  rng => {
    val (a, rng2) = s(rng)
    (f(a), rng2)
  }
```

EXERCISE 5: Use `map` to generate an `Int` between `0` and `n`, inclusive:

```
def positiveMax(n: Int): Rand[Int]
```

EXERCISE 6: Use `map` to reimplement `RNG.double` in a more elegant way.

EXERCISE 7: Unfortunately, `map` is not powerful enough to implement `intDouble` and `doubleInt` from before. What we need is a new combinator `map2`, that can combine two RNG actions into one using a binary rather than unary function. Write its implementation and then use it to reimplement the `intDouble` and `doubleInt` functions.

```
def map2[A,B,C](ra: Rand[A], rb: Rand[B])(f: (A, B) => C): Rand[C]
```

EXERCISE 8 (hard): If we can combine two RNG transitions, we should be able to combine a whole list of them. Implement `sequence`, for combining a `List` of transitions into a single transition. Use it to reimplement the `ints` function you wrote before. For the latter, you can use the standard library function `List.fill(n)(x)` to make a list with `x` repeated `n` times.

```
def sequence[A](fs: List[Rand[A]]): Rand[List[A]]
```

We're starting to see a pattern: We're progressing towards implementations that don't explicitly mention or pass along the `RNG` value. The `map` and `map2` combinators allowed us to implement, in a rather succinct and elegant way, functions that were otherwise tedious and error-prone to write. But there are some functions that we can't very well write in terms of `map` and `map2`.

Let's go back to `positiveInt` and see if it can be implemented in terms of `map`. It's possible to get most of the way there, but what do we do in the case that `Int.MinValue` is generated? It doesn't have a positive counterpart and we can't just select an arbitrary number:

```
def positiveInt: Rand[Int] = {
  map(int) { i =>
    if (i != Int.MinValue) i.abs else ??                    (1)
  }
}
```

(1) What goes here?

We want to retry the generator in the case of `Int.MinValue`, but we don't actually have an `RNG`! Besides, anything except an `Int` there would have the

wrong type. So we clearly need a more powerful combinator than `map`.

EXERCISE 9: Implement `flatMap`, then use it to reimplement `positiveInt`.

```
def flatMap[A,B](f: Rand[A])(g: A => Rand[B]): Rand[B]
```

EXERCISE 10: Reimplement `map` and `map2` in terms of `flatMap`.

The functions you've just written, `unit`, `map`, `map2`, `flatMap`, and `sequence`, are not really specific to random number generation at all. They are general-purpose functions for working with state actions, and don't actually care about the type of the state. Notice that, for instance, `map` does not care that it is dealing with `RNG` state actions and we can give it a more general signature:

```
def map[S,A,B](a: S => (A,S))(f: A => B): S => (B,S)
```

Changing this signature doesn't require modifying the implementation of `map`! The more general signature was there all along, we just didn't see it.

We should then come up with a more general type than `Rand`, for handling any type of state:

```
type State[S,+A] = S => (A,S)
```

Here, `State` is short for "state action" (or even "state transition"). We might even want to write it as its own class, wrapping the underlying function like this:

```
case class State[S,+A](run: S => (A,S))
```

The representation doesn't matter too much. What is important is that we have a single, general-purpose type and using this type we can write general-purpose functions for capturing common patterns of handling and propagating state.

In fact, we could just make `Rand` a type alias for `State`:

```
type Rand[A] = State[RNG, A]
```

EXERCISE 11: Generalize the functions `unit`, `map`, `map2`, `flatMap`, and

`sequence`. Add them as methods on the `State` case class where possible. Otherwise you should put them in a `State` companion object.

The functions we've written here capture only a few of the most common patterns. As you write more functional code, you'll likely encounter other patterns and discover other functions to capture them.

6.4 Purely functional imperative programming

In the sections above, we were writing functions that followed a definite pattern. We would run a state action, assign its result to a `val`, then run another state action that used that `val`, assign its result to another `val`, and so on. It looks a lot like *imperative* programming.

In the imperative programming paradigm, a program is a sequence of statements where each statement may modify the program state. That's exactly what we have been doing, except that our "statements" are really state actions, which are really functions. As functions, they read the current program state simply by receiving it in their argument, and they write to the program state simply by returning a value.

SIDEBAR **Aren't imperative and functional programming opposites?**

Absolutely not. Remember, functional programming is simply programming without side-effects. Imperative programming is about programming with statements that modify some program state, and as we've seen it's entirely reasonable to maintain state without side-effects.

Functional programming has excellent support for writing imperative programs, with the added benefit that such programs can be reasoned about equationally because they are referentially transparent.

We implemented some combinators like `map`, `map2`, and ultimately `flatMap`, to handle the propagation of the state from one statement to the next. But in doing so, we seem to have lost a bit of the imperative mood.

Consider as an example the following (which assumes that we have made `Rand[A]` a type alias for `State[RNG, A]`):

```
int.flatMap(x =>
int.flatMap(y =>
ints(x).map(xs =>
xs.map(_ % y))))
```

It's not very clear what's going on here. But since we have `map` and `flatMap` defined, we can use `for`-comprehension to recover the imperative style:

```
for {
  x <- int
  y <- int
  xs <- ints(x)
} yield xs.map(_ % y)
```

This code is much easier to read (and write), and it looks like what it is—an imperative program that maintains some state. But it is *the same code*. We get the next `Int` and assign it to `x`, get the next `Int` after that and assign it to `y`, then generate a list of length `x`, and finally return the list with all of its elements wrapped around the modulus `y`.

To facilitate this kind of imperative programming with `for`-comprehensions (or `flatMap`s), we really only need two primitive `State` combinators—one for reading the state and one for writing the state. If we imagine that we have a combinator `get` for getting the current state, and a combinator `set` for setting a new state, we could implement a combinator that can modify the state in arbitrary ways:

```
def modify[S](f: S => S): State[S, Unit] = for {
  s <- get                                          (1)
  _ <- set(f(s))                                    (2)
} yield ()
```

(1) Gets the current state and assigns it to `s`.
(2) Sets the new state to `f` applied to `s`.

This method returns a `State` action that modifies the current state by the function `f`. It yields `Unit` to indicate that it doesn't have a return value other than the state.

EXERCISE 12: Come up with the signatures for `get` and `set`, then write their implementations.

EXERCISE 13 (hard): To gain experience with the use of `State`, implement a simulation of a simple candy dispenser. The machine has two types of input: You can insert a coin, or you can turn the knob to dispense candy. It can be in one of two states: locked or unlocked. It also tracks how many candies are left and how many coins it contains.

```
sealed trait Input
case object Coin extends Input
case object Turn extends Input

case class Machine(locked: Boolean, candies: Int, coins: Int)
```

The rules of the machine are as follows:

- Inserting a coin into a locked machine will cause it to unlock if there is any candy left.
- Turning the knob on an unlocked machine will cause it to dispense candy and become locked.
- Turning the knob on a locked machine or inserting a coin into an unlocked machine does nothing.
- A machine that is out of candy ignores all inputs.

The method `simulateMachine` should operate the machine based on the list of inputs and return the number of coins left in the machine at the end. Note that if the input `Machine` has `10` coins in it, and a net total of `4` coins are added in the inputs, the output will be `14`.

```
def simulateMachine(inputs: List[Input]): State[Machine, Int]
```

6.5 Summary

In this chapter, we touched on the subject of how to deal with state and state propagation. We used random number generation as the motivating example, but the overall pattern comes up in many different domains, and this chapter illustrated the basic idea of how to handle state in a purely functional way. The idea is very simple: we use a pure function that accepts a state as its argument, and it returns the new state alongside its result. Next time you encounter an imperative API that relies on side effects, see if you can provide a purely functional version of it, and use some of the functions we wrote here to make working with it more convenient.

7 Purely functional parallelism

7.1 Introduction

In this chapter, we are going to build a library for creating and composing parallel and asynchronous computations. We are going to work iteratively, refining our design and implementation as we gain a better understanding of the domain and the design space.

Before we begin, let's think back to the libraries we wrote in Part 1, for example the functions we wrote for `Option` and `Stream`. In each case we defined a data type and wrote a number of useful functions for creating and manipulating values of that type. But there was something interesting about the functions we wrote. For instance consider `Stream`—if you look back, you'll notice we wrote only a few *primitive* functions (like `foldRight` and `unfold`) which required knowledge of the internal representation of `Stream` (consisting of its `uncons` method). We then wrote a large number of *derived operations* or *combinators* without introducing additional primitives, just by combining existing functions.

In Part 1, very little design effort went into creating these nicely compositional libraries. We created our data types and found, perhaps surprisingly, that it was possible to define a large number of useful operations over these data types, just by combining existing functions. When you create a library for a new domain, the design process won't always be this easy. You will need to choose data types and functions that *facilitate this compositional structure*, and this is what makes functional design both challenging and interesting.

SIDEBAR **What is a combinator library?**

The libraries we wrote in Part 1, and the libraries we will write in Part 2 are sometimes called *combinator libraries*. This is a somewhat informal term in FP. Generally, it's used to describe a library consisting of one or more data types, along with a collection of (often higher-order) functions for creating, manipulating, and combining values of these types. In particular, the name is usually applied to libraries with a very compositional structure, where functions are combined in different ways to produce richer and more complex functionality. Though because this style of organization is so common in FP, we sometimes don't bother to distinguish between an ordinary functional library and a "combinator library".

7.2 Choosing data types and functions

Our goal in this section is to discover a data type and a set of primitive functions for our domain, and derive some useful combinators. This will be a somewhat meandering journey. Functional design can be a messy, iterative process. We hope to show at least a stylized view of this messiness that nonetheless gives some insight into how functional design proceeds in the real world. Don't worry if you don't follow absolutely every bit of discussion throughout this process. This chapter is a bit like peering over the shoulder of someone as they think through possible designs. And because no two people approach this process the same way, the particular path we walk here might not strike you as the most natural one—perhaps it considers issues in what seems like an odd order, skips too fast or goes too slow. Keep in mind that when you design your own functional libraries, you get to do it at your own pace, take whatever path you want, and whenever questions come up about design choices, you get to think through the consequences in whatever way makes sense for you, which could include running little experiments, creating prototypes, and so on.

With that as disclaimer, why don't we get started? When you begin designing a functional library, you usually have some ideas about what you generally want to be able *to do*, and the difficulty in the design process is in refining these ideas and finding a data type that enables the functionality you want. In our case, we'd like to be able to "create parallel computations", but what does that mean exactly? Let's

try to refine this into something we can implement by examining a simple, parallelizable computation—summing a sequence of values. Here's a sequential function for this in Scala:

```
def sum(as: IndexedSeq[Int]): Int =
  if (as.size <= 1) as.headOption getOrElse 0                    (1)
  else {
    val (l,r) = as.splitAt(as.length/2)
    sum(l) + sum(r)
  }
```

(1) headOption is a method defined on all collections in Scala (API docs). It returns the first element in the collection, or None if the collection is empty.

This implementation isn't the usual left fold, `as.foldLeft(0)(_ + _)`; instead we are dividing the sequence in half using the `splitAt` function, recursively summing both halves, then combining their results. And unlike the `foldLeft`-based implementation, this implementation can be parallelized—the two halves can be summed in parallel.[1]

Footnote 1 Assuming we have *n* elements and *n* CPU cores, we could in principle compute `sum` in *O(log n)* time. This is meant to be a simple example (see sidebar) though; in this instance the overhead of parallelization is unlikely to pay for itself.

SIDEBAR **The importance of simple examples**

Summing integers is in practice probably so fast that parallelization imposes more overhead than it saves. But simple examples like this are *exactly* the sort that are most helpful to consider when designing a functional library. Complicated examples include all sorts of incidental structure and extraneous detail that can confuse the initial design process. We are trying to understand the essence of the problem domain, and a good way to do this is to work with very small examples, factor out common concerns across these examples, and gradually add complexity. In functional design, our goal is to achieve expressiveness not with mountains of special cases, but by building a simple and *composable* set of core data types and functions.

As we think about how what sort of data types and functions could enable parallelizing this computation, we can shift our perspective. Rather than focusing on *how* this parallelism will ultimately be implemented (likely using `java.lang.Thread` and `java.lang.Runnable` and related types) and forcing ourselves to work with those APIs directly, we are instead going to design

our 'ideal' API as illuminated by our examples and work backwards from there to an implementation.

Look at the line `sum(l) + sum(r)`, which invokes `sum` on the two halves recursively. Just from looking at this single line, we can see that *whatever* data type we choose to represent our parallel computations needs to be able to *contain a result*, that result will have some meaningful type (in this case `Int`), and we require some way of extracting this result. Let's apply this newfound knowledge to our implementation. For now, let's just *invent* a container type for our result, `Par[A]` (for "parallel"), and assume the existence of the functions we need:

- `def unit[A](a: => A): Par[A]`, for taking an unevaluated `A` and returning a parallel computation that yields an `A`.
- `def get[A](a: Par[A]): A`, for extracting the resulting value from a parallel computation.

Can we really just do this? Yes, of course! For now, we don't need to worry about what other functions we require, what the internal representation of `Par` might be, or how these functions are implemented. We are simply reading off the needed data types and functions by inspecting our simple example. Let's update this example now:

```
def sum(as: IndexedSeq[Int]): Int =
  if (as.size <= 1) as.headOption getOrElse 0
  else {
    val (l,r) = as.splitAt(as.length/2)
    val sumL: Par[Int] = Par.unit(sum(l))
    val sumR: Par[Int] = Par.unit(sum(r))
    Par.get(sumL) + Par.get(sumR)
  }
```

We've wrapped the two recursive calls to `sum` in calls to `unit`, and we are calling `get` to extract the two results from the two subcomputations.

SIDEBAR **The problem with using threads directly**

What of `java.lang.Thread`, or `Runnable`? Let's take a look at these class Here is a partial excerpt of their API, transcribed to Scala:

```
trait Runnable { def run: Unit }

class Thread(r: Runnable) {
  def start: Unit
  def join: Unit
}
```

(1) (2)

(1) Begin executing r in a separate thread.
(2) Wait until r finishes executing before returning.

Already, we can see a problem with both of these types—none of the metho return a meaningful value. Therefore, if we want to get any information out of `Runnable`, it has to have some side effect, like mutating some internal state t we know about. This is bad for composability—we cannot manipulate `Runnab` objects generically, we need to always know something about their inter behavior to get any useful information out of them. `Thread` also has downside that it maps directly onto operating system threads, which are a sca resource. We'd prefer to create as many 'logical' parallel computations as natural for our problem, and later deal with mapping these logical computatio onto actual OS threads.

We now have a choice about the meaning of `unit` and `get`—`unit` could begin evaluating its argument immediately in a separate (logical) thread,[2] or it could simply hold onto its argument until `get` is called and begin evaluation then. But notice that in this example, if we want to obtain any degree of parallelism, we require that `unit` begin evaluating its argument immediately. Can you see why?[3]

Footnote 2 We'll use the term "logical thread" somewhat informally throughout this chapter, to mean a chunk of computation that runs concurrent to the main execution thread of our program. There need not be a one-to-one correspondence between logical threads and OS threads. We may have a large number of logical threads mapped onto a smaller number of OS threads via thread pooling, for instance.

Footnote 3 Function arguments in Scala are strictly evaluated from left to right, so if `unit` delays execution until `get` is called, we will both spawn the parallel computation and wait for it to finish before spawning the second parallel computation. This means the computation is effectively sequential!

But if `unit` begins evaluating its argument immediately, then calling `get` arguably breaks referential transparency. We can see this by replacing `sumL` and

`sumR` with their definitions—if we do so, we still get the same result, but our program is no longer parallel:

```
Par.get(Par.unit(sum(l))) + Par.get(Par.unit(sum(r)))
```

Given what we have decided so far, `unit` will start evaluating its argument right away. And the very next thing to happen is that `get` will wait for that evaluation to complete. So the two sides of the `+` sign will not run in parallel if we simply inline the `sumL` and `sumR` variables. Here we can see that `unit` has a very definite side-effect, but only *with regard to* `get`. That is, `unit` simply returns a `Par[Int]` in this case, representing an asynchronous computation. But as soon as we pass that `Par` to `get`, we explicitly wait for it, exposing the side-effect. So it seems that we want to avoid calling `get`, or at least delay calling it until the very end. We want to be able to combine asynchronous computations without waiting for them to finish.

Before we continue, notice what we have done. First, we conjured up a simple, almost trivial example. We next explored this example a bit to uncover a design choice. Then, via some experimentation, we discovered an interesting consequence of one option and in the process learned something fundamental about the nature of our problem domain! The overall design process is a series of these little adventures. You don't need any special license to do this sort of exploration, and you don't need to be an expert in functional programming either. Just dive in and see what you find.

Let's see if we can avoid the above pitfall of combining `unit` and `get`. If we don't call `get`, that implies that our `sum` function must return a `Par[Int]`. What consequences does this change reveal? Again, let's just invent functions with the required signatures:

```
def sum(as: IndexedSeq[Int]): Par[Int] =
  if (as.size <= 1) Par.unit(as.headOption getOrElse 0)
  else {
    val (l,r) = as.splitAt(as.length/2)
    Par.map2(sum(l), sum(r))(_ + _)
  }
```

EXERCISE 1: `Par.map2` is a new higher-order function for combining the result of two parallel computations. What is its signature? Give the most general

signature possible (that is, do not assume it works only for `Par[Int]`).

Observe that we are no longer calling `unit` in the recursive case, and it isn't clear whether `unit` should accept its argument lazily anymore. In this example, accepting the argument lazily doesn't seem to provide any benefit, but perhaps this isn't always the case. Let's try coming back to this question in a minute.

What about `map2`—should it take its arguments lazily? It would make sense for `map2` to run both sides of the computation in parallel, giving each side equal opportunity to run (it would seem arbitrary for the order of the `map2` arguments to matter—we simply want `map2` to indicate that the two computations being combined are independent, and can be run in parallel). What choice lets us implement this meaning? As a simple test case, consider what happens if `map2` is strict in both arguments, and we are evaluating `sum(IndexedSeq(1,2,3,4))`. Take a minute to work through and understand the following (somewhat stylized) program trace:

```
sum(IndexedSeq(1,2,3,4))
map2(
  sum(IndexedSeq(1,2)),
  sum(IndexedSeq(3,4)))(_ + _)
map2(
  map2(
    sum(IndexedSeq(1)),
    sum(IndexedSeq(2)))(_ + _),
  sum(IndexedSeq(3,4)))(_ + _)
map2(
  map2(
    unit(1),
    unit(2))(_ + _),
  sum(IndexedSeq(3,4)))(_ + _)
map2(
  map2(
    unit(1),
    unit(2))(_ + _),
  map2(
    sum(IndexedSeq(3)),
    sum(IndexedSeq(4)))(_ + _))(_ + _)
...
```

In this trace, to evaluate `sum(x)`, we substitute `x` into the definition of `sum`, as we've done in previous chapters. Because `map2` is strict, and Scala evaluates arguments left to right, whenever we encounter `map2(sum(x),sum(y))(_ + _)`, we have to then evaluate `sum(x)`, and so on recursively. This has the rather unfortunate consequence that we will strictly construct the entire left half of the

tree of summations first before moving on to (strictly) constructing the right half (here, `sum(IndexedSeq(1,2))` gets fully expanded before we consider `sum(IndexedSeq(3,4))`). And if `map2` begins evaluating its arguments immediately (using whatever resource is being used to implement the parallelism, like a thread pool), that implies the left half of our computation will begin its execution before we even begin constructing the right half of our computation.

What if we keep `map2` strict, but *don't* have it begin execution immediately? Does this help? If `map2` doesn't begin evaluation immediately, this implies a `Par` value is merely constructing a *description* of what needs to be computed in parallel. Nothing actually occurs until we *evaluate* this description, perhaps using a `get`-like function. The problem is that if we construct our descriptions strictly, they are going to be rather heavyweight objects. Looking back at our trace, our description is going to have to contain the full tree of operations to be performed:

```
map2(
  map2(
    unit(1),
    unit(2))(_ + _),
  map2(
    unit(3),
    unit(4))(_ + _))(_ + _)
```

Whatever data structure we use to store this description, it will likely occupy more space than the original list itself! It would be nice if our descriptions were a bit more lightweight.

It seems we should make `map2` lazy and have it begin immediate execution of both sides in parallel (this also addresses the problem of giving each side equal "weight"), but something still doesn't feel right about this. Is it *always* the case that we want to evaluate the two arguments to `map2` in parallel? Probably not. Consider this simple hypothetical example:

```
Par.map2(Par.unit(1), Par.unit(1))(_ + _)
```

In this case, we happen to know that the two computations we're combining will execute so quickly that there isn't much point in spawning off a separate logical thread to evaluate them. But our API doesn't give us any way of providing this sort of information. That is, our current API is very *implicit* about when computations get forked off the main thread—the programmer does not get to

specify where this forking should occur. What if we make this forking more explicit? We can do that by inventing another function, `def fork[A](a: => Par[A]): Par[A]`, which we can take to mean that the given `Par` should be run in a separate logical thread:

```
def sum(as: IndexedSeq[Int]): Par[Int] =
  if (as.isEmpty) Par.unit(0)
  else {
    val (l,r) = as.splitAt(as.length/2)
    Par.map2(Par.fork(sum(l)), Par.fork(sum(r)))(_ + _)
  }
```

With `fork`, we can now make `map2` strict, leaving it up to the programmer to wrap arguments if they wish. A function like `fork` solves the problem of instantiating our parallel computations too strictly (as an exercise, try revisiting the hypothetical trace from earlier), but more fundamentally it makes the parallelism more explicit and under programmer control. There are really two separate concerns being addressed here. The first is that we need some way to indicate that the results of the two parallel tasks should be combined. Separate from this, we have the choice of whether a particular task should be performed asynchronously. By keeping these concerns separate, we avoid having any sort of global policy for parallelism attached to `map2` and other combinators we write, which would mean making tough (and ultimately arbitrary) choices about what global policy is best. Such a policy may in practice be inappropriate in many cases.

Let's now return to the question of whether `unit` should be strict or lazy. With `fork`, we can now make `unit` strict without any loss of expressiveness. A non-strict version of it, let's call it `async`, can be implemented using `unit` and `fork`.

```
def unit[A](a: A): Par[A]
def async[A](a: => A): Par[A] = fork(unit(a))
```

The function `async` is a simple example of a *derived* combinator, as opposed to a *primitive* combinator like `unit`. We were able to define `async` just in terms of other operations. Later, when we pick a representation for `Par`, `async` will not need to know anything about this representation—its only knowledge of `Par` is through the operations `fork` and `unit` that are defined on `Par`.[4]

Footnote 4 This sort of indifference to representation is a hint that the operations are actually more general, and can be abstracted to work for types other than just `Par`. We will explore this topic in detail in part 3.

We still have the question of whether `fork` should begin evaluating its argument immediately, or wait until the computation is *forced* later using something like `get`. When you are unsure about a meaning to assign to some function in your API, you can always continue with the design process—at some point later the tradeoffs of different choices of meaning may become clear. Here, we make use of a helpful trick—we are going to think about what *sort of information* is required to implement `fork` with various meanings.

If `fork` begins evaluating its argument immediately in parallel, the implementation must clearly know something (either directly or indirectly) about how to create threads or submit tasks to some sort of thread pool. Moreover, if `fork` is just a standalone function (as it currently is on `Par`), it implies that whatever resource is used to implement the parallelism must be *globally accessible*. This means we lose the ability to control the parallelism strategy used for different parts of our program. And while there's nothing inherently wrong with having a global resource for executing parallel tasks, we can imagine how it would be useful to have more fine-grained control over what implementations are used where (we might like for each subsystem of a large application to get its own thread pool with different parameters, say).

Notice that coming to these conclusions didn't require knowing exactly how `fork` would be implemented, or even what the representation of `Par` will be. We just reasoned informally about the sort of information required to actually spawn a parallel task, and examined the consequences of having `Par` values know about this information.

In contrast, if `fork` simply holds onto the computation until later, this requires no access to the mechanism for implementing parallelism. Let's tentatively assume this meaning then for `fork`. With this model, `Par` itself does not know how to actually *implement* the parallelism. It is more a *description* of a parallel computation. This is a big shift from before, where we were considering `Par` to be a "container" of a value that we could "get". Now it's more of a first-class *program* that we can *run*. So let's rename our `get` function to `run`.

```
def run[A](a: Par[A]): A
```

Because `Par` now just a pure data structure, we will assume that `run` has some means of implementing the parallelism, whether it spawns new threads, delegates tasks to a thread pool, or uses some other mechanism.

7.2.1 Picking a representation

Just by exploring this simple example and thinking through the consequences of different choices, we've sketched out the following API:

```
def unit[A](a: A): Par[A]
def map2[A,B,C](a: Par[A], b: Par[B])(f: (A,B) => C): Par[C]
def fork[A](a: => Par[A]): Par[A]
def async[A](a: => A): Par[A] = fork(unit(a))
def run[A](a: Par[A]): A
```

We've also loosely assigned meaning to these various functions:

- `unit` injects a constant into a parallel computation.
- `map2` combines the results of two parallel computations with a binary function.
- `fork` spawns a parallel computation. The computation will not be spawned until forced by `run`.
- `run` extracts a value from a `Par` by actually performing the computation.

At any point while sketching out an API, you can start thinking about possible *representations* for the abstract types that appear.

EXERCISE 2: Before continuing, try to come up with representations for `Par` and `Strategy` that make it possible to implement the functions of our API.

Let's see if we can come up with a representation. We know `run` needs to execute asynchronous tasks somehow. We could write our own API, but there's already a class for this in `java.util.concurrent`, `ExecutorService`. Here it its API, excerpted and transcribed to Scala:

```
class ExecutorService {
  def submit[A](a: Callable[A]): Future[A]
}
trait Future[A] {
  def get: A
  def get(timeout: Long, unit: TimeUnit): A
  def cancel(evenIfRunning: Boolean): Boolean
  def isDone: Boolean
  def isCancelled: Boolean
}
```

So, `ExecutorService` lets us submit a `Callable` value (in Scala we'd probably just use a lazy argument to `submit`), and get back a corresponding `Future`. We can block to obtain a value from a `Future` with its `get` method, and it has some extra features for cancellation, only blocking for a certain amount of time, and so on.

Let's try assuming that our `run` function has an `ExecutorService` and see if that suggests anything about the representation for `Par`:

```
def run[A](s: ExecutorService)(a: Par[A]): A
```

The simplest possible model for `Par[A]` might just be `ExecutorService => A`. This would obviously make `run` trivial to implement. But it might be nice to defer the decision of how long to wait for a computation or whether to cancel it to the caller of `run`:

```
type Par[A] = ExecutorService => Future[A]

def run[A](s: ExecutorService)(a: Par[A]): Future[A] = a(s)
```

Is it really that simple? Let's assume it is for now, and revise our model if we decide it doesn't allow some functionality we'd like.

7.2.2 Exploring and refining the API

The way we've worked so far is actually a bit artificial. In practice, there aren't such clear boundaries between designing your API and choosing a representation, and one does not necessarily precede the other. Ideas for a representation can inform the API you develop, the API you develop can inform the choice of representation, and it's natural to shift fluidly between these two perspectives, run experiments as questions arise, build prototypes, and so on.

We are going to devote the rest of this section to exploring our API. Though we got a lot of mileage out of considering a simple example, before we add any new primitive operations let's try to learn more about what is expressible using those we already have. With our primitives and choices of meaning for them, we have carved out a little universe for ourselves. We now get to discover what ideas are expressible in this universe. This can and should be a fluid process—we can

change the rules of our universe at any time, make a fundamental change to our representation or introduce a new primitive, and explore how our creation then behaves.

EXERCISE 3: Let's begin by implementing the functions of the API we've developed so far. Now that we have a representation for `Par`, we should be able to fill these in. Optional (hard): try to ensure your implementations respect the contract of the `get` method on `Future` that accepts a timeout.

```
def unit[A](a: A): Par[A]
def map2[A,B,C](a: Par[A], b: Par[B])(f: (A,B) => C): Par[C]
def fork[A](a: => Par[A]): Par[A]
```

You can place these functions and other functions we write inside an `object` called `Par`, like so:

```
object Par {
  /* Functions go here */
}
```

SIDEBAR **Adding infix syntax using implicit conversions**

If `Par` were an actual data type, functions like `map2` could be placed in the class body and then called with infix syntax like `x.map2(y)(f)` (much like we did for `Stream` and `Option`). But since `Par` is just a type alias we can't do this directly. There is, however, a trick to add infix syntax to *any* type using *implicit conversions*. We won't discuss that here since it isn't all that relevant to what we're trying to cover, but if you're interested, check out the code associated with this chapter and also the appendix.

EXERCISE 4: This API already enables a rich set of operations. Here's a simple example: using `async`, write a function to convert any function `A => B` to one that evaluates its result asynchronously:

```
def asyncF[A,B](f: A => B): A => Par[B]
```

What else can we express with our existing combinators? Let's look at a more concrete example.

Suppose we have a `Par[List[Int]]` representing a parallel computation

producing a `List[Int]` and we would like to convert this to a `Par[List[Int]]` whose result is now sorted:

```
def sortPar(l: Par[List[Int]]): Par[List[Int]]
```

We could of course `run` the `Par`, sort the resulting list, and re-package it in a `Par` with `unit`. But we want to avoid calling `run`. The only other combinator we have that allows us to manipulate the value of a `Par` in any way is `map2`. So if we passed `l` to one side of `map2`, we would be able to gain access to the `List` inside and sort it. And we can pass whatever we want to the other side of `map2`, so let's just pass a no-op:

```
def sortPar(l: Par[List[Int]]): Par[List[Int]] =
  map2(l, unit(()))((a, _) => a.sorted)
```

Nice. We can now tell a `Par[List[Int]]` that we would like that list sorted. But we can easily generalize this further. We can "lift" any function of type `A => B` to become a function that takes `Par[A]` and returns `Par[B]`. That is, we can `map` any function over a `Par`:

```
def map[A,B](fa: Par[A])(f: A => B): Par[B] =
  map2(fa, unit(()))((a,_) => f(a))
```

For instance, `sortPar` is now simply this:

```
def sortPar(l: Par[List[Int]]) = map(l)(_.sorted)
```

This was almost too easy. We just combined the operations to make the types line up. And yet, if you look at the implementations of `map2` and `unit`, it should be clear this implementation of `map` *means* something sensible.

Was it cheating to pass a bogus value, `unit(())` as an argument to `map2`, only to ignore its value? Not at all! It shows that `map2` is strictly more powerful than `map`. This sort of thing can be a hint that `map2` can be further decomposed into primitive operations. And when we consider it, `map2` is actually doing two things—it is creating a parallel computation that waits for the result of two other

computations *and then* it is combining their results using some function. We could split this into two functions, `product` and `map`:

```
def product[A,B](fa: Par[A], fb: Par[B]): Par[(A,B)]
def map[A,B](fa: Par[A])(f: A => B): Par[B]
```

EXERCISE 5 (optional): Implement `product` and `map` as primitives, then define `map2` in terms of them.

This is sort of an interesting little discovery that we can factor things this way. Is it an improvement? On the one hand, `product` is "doing one thing only", and `map` is now completely orthogonal. But if you look at your implementation of `product`, you can almost see `map2` hiding inside, except that you are always supplying a function of type `(A,B) => (A,B)` that sticks its arguments in a pair. If we're going to have to write that function, we might as well expose the most general version of it, `map2`. So let's keep `map2` as primitive for now, and define `map` in terms of it as above. This is one example where there is a choice of which things we consider primitive and which things are derived.

What else can we implement using our API? Could we `map` over a list in parallel? Unlike `map2`, which combines two parallel computations, `parMap` (let's call it) needs to combine *N* parallel computations. Still, it seems like this should somehow be expressible.

EXERCISE 6: Note that we could always just write `parMap` as a new primitive. See if you can implement it this way. Remember that `Par[A]` is simply an alias for `ExecutorService => Future[A]`. Here is the signature for `parMap`:

```
def parMap[A,B](l: List[A])(f: A => B): Par[List[B]]
```

There's nothing wrong with implementing operations as new primitives. In some cases we can even implement the operations more efficiently, by assuming something about the underlying representation of the data types we are working with. But we're interested in exploring what operations are expressible using our existing API, and understanding the relationships between the various operations

we've defined. Knowledge of what combinators are truly primitive will become more important in Part 3, when we learn to abstract over common patterns across libraries.[5]

Footnote 5 In this case, there's another good reason not to implement `parMap` as a new primitive—it's challenging to do correctly, particularly if we want to properly respect timeouts. It's frequently the case that primitive combinators encapsulate some rather tricky logic, and reusing them means we don't have to duplicate this logic.

Let's see how far we can get implementing `parMap` in terms of existing combinators:

```
def parMap[A,B](l: List[A])(f: A => B): Par[List[B]] = {
  val fbs: List[Par[B]] = l.map(asyncF(f))
  ...
}
```

Remember, `asyncF` converts an `A => B` to a `A => Par[B]`, by forking a parallel computation to produce the result. So we can fork off our *N* parallel computations pretty easily, but we need some way of collecting up their results. Are we stuck? Well, just from inspecting the types, we can see that we need some way of converting our `List[Par[B]]` to the `Par[List[B]]` required by the return type of `parMap`.

EXERCISE 7 (hard): Let's write this function, typically called `sequence`. No additional primitives are required.

```
def sequence[A](l: List[Par[A]]): Par[List[A]]
```

Once we have `sequence`, we can complete our implementation of `parMap`:

```
def parMap[A,B](l: List[A])(f: A => B): Par[List[B]] = fork {
  val fbs: List[Par[B]] = l.map(asyncF(f))
  sequence(fbs)
}
```

Notice that we've wrapped our implementation in a call to `fork`. With this implementation, `parMap` will return immediately, even for a huge input list. When we later call `run`, it will fork a single asynchronous computation which itself spawns *N* parallel computations then waits for these computations to finish, collecting their results up into a list. If you look back at your previous

implementation of `parMap`, the one that had knowledge of the internal representation of `Par`, you'll see that it's doing the same thing.

EXERCISE 8: Implement `parFilter`, which filters elements of a list in parallel.

```
def parFilter[A](l: List[A])(f: A => Boolean): Par[List[A]]
```

Can you think of any other useful functions to write? Experiment with writing a few parallel computations of your own to see which ones can be expressed without additional primitives. Here are some ideas to try:

- Is there a more general version of the parallel summation function we wrote at the beginning of this chapter? Try using it to find the maximum value of an `IndexedSeq` in parallel.
- Write a function that takes a list of paragraphs (a `List[String]`), and returns the total number of words across all paragraphs, in parallel. Generalize this function as much as possible.
- Implement `map3`, `map4`, and `map5`, in terms of `map2`.

7.2.3 The algebra of an API

As the previous section demonstrates, we often get quite far by treating this all as a game of (what seems like) meaningless symbol manipulation! We write down the type signature for an operation we want, then "follow the types" to an implementation. Quite often when working this way we can almost forget the concrete domain (for instance, when we implemented `map` in terms of `map2` and `unit`) and just focus on lining up types. This isn't cheating; it's a natural but different style of reasoning, analogous to the reasoning one does when simplifying an algebraic equation like *x*. We are treating the API as an *algebra,*[6] or an abstract set of operations along with a set of *laws* or properties we assume true, and simply doing formal symbol manipulation following the "rules of the game" specified by this algebra.

Footnote 6 We do mean algebra in the mathematical sense of one or more sets, together with a collection of functions operating on objects of these sets, and a set of *axioms*. (Axioms are statements assumed true, from which we can derive other *theorems* that must also be true.) In our case, the sets are values with particular types, like `Par[A]`, `List[Par[A]]`, the functions are operations like `map2`, `unit`, and `sequence`.

Up until now, we have been reasoning somewhat informally about our API. There's nothing wrong with this, but it can be helpful to take a step back and formalize what laws you expect to hold (or would like to hold) for your API.[7]

Without realizing it, you have probably mentally built up a model of what properties or laws you expect. Actually writing these down and making them precise can highlight design choices that wouldn't be otherwise apparent when reasoning informally.

Footnote 7 We'll have much more to say about this throughout the rest of this book. In the next chapter, we'll be designing a declarative testing library that lets us define properties we expect functions to satisfy, and automatically generates test cases to check these properties. And in Part 3 we'll introduce abstract interfaces specified *only* by sets of laws.

Like any design choice, choosing laws has *consequences*—it places constraints on what the operations can mean, what implementation choices are possible, affects what other properties can be true or false, and so on. Let's look at an example. We are going to simply *conjure up* a possible law that seems reasonable. This might be used as a test case if we were creating unit tests for our library:

```
map(unit(1))(_ + 1) == unit(2)
```

We are saying that mapping over `unit(1)` with the `_ + 1` function is in some sense equivalent to `unit(2)`. (Laws often start out this way, as concrete examples of *identities*[8] we expect to hold.) In what sense are they equivalent? This is somewhat of an interesting question. For now, let's say two `Par` objects are equivalent if *for any valid* `ExecutorService` argument, the `Future` they return results in the same value.

Footnote 8 Here we mean 'identity' in the mathematical sense of a statement that two expressions are identical or equivalent.

We can check that this holds for a particular `ExecutorService` with a function like:

```
def equal[A](e: ExecutorService)(p: Par[A], p2: Par[A]): Boolean =
  p(e).get == p2(e).get
```

Laws and functions share much in common. Just as we can generalize functions, we can generalize laws. For instance, the above could be generalized:

```
map(unit(x))(f) == unit(f(x))
```

Here we are saying this should hold for *any* choice of `x` and `f`. This places

some constraints on our implementation. Our implementation of `unit` cannot, say, inspect the value it receives and decide to return a parallel computation with a result of `42` when the input is `1`—it can only pass along whatever it receives. Similarly for our `ExecutorService`—when we submit `Callable` objects to it for execution, it cannot make any assumptions or change behavior based on the values it receives.[9] More concretely, this law disallows downcasting or `isInstanceOf` checks (often grouped under the term *typecasing*) in the implementations of `map` and `unit`.

Footnote 9 Hints and standalone answers

Much like we strive to define functions in terms of simpler functions, each of which *do* just one thing, we can define laws in terms of simpler laws that each *say* just one thing. Let's see if we can simplify this law further. We said we wanted this law to hold for *any* choice of `x` and `f`. Something interesting happens if we substitute the identity function for `f`[10]. We can simplify both sides of the equation and get a new law which is considerably simpler:[11]

Footnote 10 The identity function has the signature `def id[A](a: A): A = a`.

Footnote 11 This is the same sort of substitution and simplification one might do when solving an algebraic equation.

```
map(unit(x))(f) == unit(f(x))
map(unit(x))(id) == unit(id(x))                    (1)
map(unit(x))(id) == unit(x)                        (2)
map(y)(id) == y                                    (3)
```

(1) Substitute identity function for f
(2) Simplify
(3) Substitute y for unit(x) on both sides

Fascinating! Our new, simpler law talks only about `map`—apparently the mention of `unit` was an extraneous detail. To get some insight into what this new law is saying, let's think about what `map` *cannot* do. It cannot, say, throw an exception and crash the computation before applying the function to the result (can you see why this violates the law?). All it can do is apply the function `f` to the result of `y`, which of course, leaves `y` unaffected in the case that function is `id`.[12] Even more interesting, given `map(y)(id) == y`, we can perform the substitutions in the other direction to get back our original, more complex law.

(Try it!) Logically, we have the freedom to do so because `map` cannot possibly behave differently for different function types it receives. Thus, given `map(y)(id) == y`, it must be true that `map(unit(x))(f) == unit(f(x))`. Since we get this second law or theorem "for free", simply because of the parametricity of `map`, it is sometimes called a *free theorem*.[13]

Footnote 12 We say that `map` is required to be *structure-preserving*, in that it does not alter the structure of the parallel computation, only the value "inside" the computation.

Footnote 13 The idea of free theorems was introduced by Philip Wadler in a classic paper called Theorems for free!

EXERCISE 9 (hard, optional): Given `map(y)(id) == y`, it is a free theorem that `map(map(y)(g))(f) == map(y)(f compose g)`. (This is sometimes called *map fusion*, and it can be used as an optimization—rather than spawning a separate parallel computation to compute the second mapping, we can fold it into the first mapping.)[14] Can you construct a proof? You may want to read the paper Theorems for Free! to better understand the "trick" of free theorems.

Footnote 14 Our representation of `Par` does not give us the ability to implement this optimization, since it is an opaque function. If it were reified as a data type, we could pattern match and discover opportunities to apply this rule. You may want to try experimenting with this idea on your own.

As interesting as all this is, these laws don't do much to constrain our implementation. You have probably been assuming these properties without even realizing it (it would be rather strange to have any special cases in the implementations of `map`, `unit` or `ExecutorService.submit`, or have `map` randomly throwing exceptions). Let's consider a stronger property, namely that `fork` should not affect the result of a parallel computation:

```
fork(x) == x
```

This seems like it should be obviously true of our implementation, and it is clearly a desirable property, consistent with our expectation of how `fork` should work. `fork(x)` should 'do the same thing' as `x`, but asynchronously, in a logical thread separate from the main thread. If this law didn't always hold, we'd have to somehow know when it was safe to call without changing meaning, without any help from the type system.

Surprisingly, this simple property places very strong constraints on our implementation of `fork`. After you've written down a law like this, take off your

implementer hat, put on your debugging hat, and try to break your law. Think through any possible corner cases, try to come up with counterexamples, and even construct an informal proof that the law holds—at least enough to convince a skeptical fellow programmer.

Let's try this mode of thinking. We are expecting that `fork(x) == x` for *all* choices of `x`, and any choice of `ExecutorService`. We have a pretty good sense of what `x` could be—it's some expression making use of `fork`, `unit`, and `map2` (and other combinators derived from these). What about `ExecutorService`? What are some possible implementations of it? There's a good listing of different implementations in the class `Executors` (API link).

EXERCISE 10 (hard, optional): Take a look through the various static methods in `Executors` to get a feel for the different implementations of `ExecutorService` that exist. Then, before continuing, go back and revisit your implementation of `fork` and try to find a counterexample or convince yourself that the law holds for your implementation.

SIDEBAR **Why laws about code and proofs are important**

It may seem unusual to state and prove properties about an API. This certainly isn't something typically done in ordinary programming. Why is it important in FP?

In functional programming, it is easy, and expected, that we will factor out common functionality into generic, reusable, components that can be *composed*. Side effects hurt compositionality, but more generally, any hidden or out-of-band assumptions or behavior that prevent us from treating our components (be they functions or anything else) as *black boxes* make composition difficult or impossible.

In our example of the law for `fork`, we can see that if the law we posited did not hold, many of our general purpose combinators, like `parMap`, would no longer be sound (and their usage might be dangerous, since they could, depending on the broader parallel computation they were used in, result in deadlocks).

Giving our APIs an algebra, with laws that are meaningful and aid reasoning, make the API more usable for clients, but also mean we can treat the objects of our API as black boxes. As we'll see in Part 3, this is crucial for our ability to factor out common patterns across the different libraries we've written.

There's actually a problem with most implementations of `fork`. It's a rather

subtle deadlock that occurs when using an `ExecutorService` backed by a thread pool of bounded size (see `ExecutorService.newFixedThreadPool`).[15] Suppose we have an `ExecutorService` backed by a thread pool where the maximum number of threads is bounded to 1. Try running the following example using your current implementation:

Footnote 15 In the next chapter we'll be writing a combinator library for testing that can help discover problems like these automatically.

```
val a = async(42 + 1)
val S = Executors.newFixedSizeThreadPool(1)
println(Par.equal(S)(a, fork(a)))
```

Most implementations of `fork` will result in this code deadlocking. Can you see why? Most likely, your implementation of `fork` looks something like this:[16]

Footnote 16 There's actually another minor problem with this implementation—we are just calling `get` on the inner `Future` returned from `fa`. This means we are not properly respecting any timeouts that have been placed on the outer `Future`.

```
def fork_simple[A](a: => Par[A]): Par[A] =
  es => es.submit(new Callable[A] {
    def call = a(es).get
  })
```

The bug is somewhat subtle. Notice that we are submitting the `Callable` first, and *within that callable*, we are submitting another `Callable` to the `ExecutorService` and blocking on its result (recall that `a(es)` will submit a `Callable` to the `ExecutorService` and get back a `Future`). This is a problem if our thread pool has size 1. The outer `Callable` gets submitted and picked up by the sole thread. Within that thread, before it will complete, we submit and block waiting for the result of another `Callable`. But there are no threads available to run this `Callable`. Our code therefore deadlocks.

EXERCISE 11 (hard, optional): Can you show that any fixed size thread pool can be made to deadlock given this implementation of `fork`?

When you find counterexamples like this, you have two choices—you can try to fix your implementation such that the law holds, or you can refine your law a bit, to state more explicitly the conditions under which it holds (we could simply

stipulate that we require thread pools that can grow unbounded). Even this is a good exercise—it forces you to document invariants or assumptions that were previously implicit.

We are going to try to fix our implementation, since being able to run our parallel computations on fixed size thread pools seems like a useful capability. The problem with the above implementation of `fork` is that we are invoking `submit` *inside* a callable, and we are *blocking* on the result of what we've submitted. This leads to deadlock when there aren't any remaining threads to run the task we are submitting. So it seems we have a simple rule we can follow to avoid deadlock:

A `Callable` should never submit and then block on the result of a `Callable`.

You may want to take a minute to prove to yourself that our parallel tasks cannot deadlock, even with a fixed-size thread pool, so long as this rule is followed.

Let's look at a different implementation of `fork`:

```
def fork[A](fa: => Par[A]): Par[A] =
  es => fa(es)
```

This certainly avoids deadlock. The only problem is that we aren't actually forking a separate logical thread to evaluate `fa`. So, `fork(hugeComputation)(es)` for some `ExecutorStrategy`, `es`, runs `hugeComputation` in the main thread, which is exactly what we wanted to avoid by calling `fork`. This is still a useful combinator, though, since it lets us delay instantiation of a parallel computation until it is actually needed. Let's give it a name, `delay`:

```
def delay[A](fa: => Par[A]): Par[A] =
  es => fa(es)
```

EXERCISE 12 (hard, optional): Can you figure out a way to still evaluate `fa` in a separate logical thread, but avoid deadlock? You may have to resort to some mutation or imperative tricks behind the scenes. There's absolutely nothing wrong with doing this, so long as these local violations of referential transparency aren't

observable to users of the API. The details of this are quite finicky to get right. The nice thing is that we are confining this detail to one small block of code, rather than forcing users to have to think about these issues throughout their use of the API.

Taking a step back from these details, the purpose here is not necessarily to figure out the best, nonblocking implementation of `fork`, but more to show that laws are important. They give us another angle to consider when thinking about the design of a library. If we hadn't tried writing some of these laws out, we may not have discovered this behavior of `fork` until much later.

In general, there are multiple approaches you can consider when choosing laws for your API. You can think about your conceptual model, and reason from there to postulate laws that should hold. You can also *conjure up* laws you think are *useful for reasoning or compositionality* (like we did with our `fork` law), and see if it is possible and sensible to ensure they hold for your model and implementation. And lastly, you can look at your *implementation* and come up with laws you expect to hold based on your implementation.[17]

Footnote 17 This last way of generating laws is probably the weakest, since it can be a little too easy to just have the laws reflect the implementation, even if the implementation is buggy or requires all sorts of unusual side conditions that make composition difficult.

EXERCISE 13: Can you think of other laws that should hold for your implementation of `unit`, `fork`, and `map2`? Do any of them have interesting consequences?

7.2.4 Expressiveness and the limitations of an algebra

Functional design is an iterative process. After you've written down your API and have at least a prototype implementation, try using it for progressively more complex or realistic scenarios. Often you'll find that these scenarios require only some combination of existing primitive or derived combinators, and this is a chance to factor out common usage patterns into other combinators; occasionally you'll find situations where your existing primitives are insufficient. We say in this case that the API is not expressive enough.

Let's look at an example of this.

EXERCISE 14: Try writing a function to choose between two forking computations based on the result of an initial computation. Can this be implemented in terms of existing combinators or is a new primitive required?

```
def choice[A](a: Par[Boolean])(ifTrue: Par[A], ifFalse: Par[A]): Par[A]
```

Notice what happens when we try to define this using only `map`. If we try `map(a)(a => if (a) ifTrue else ifFalse)`, we obtain the type `Par[Par[A]]`. If we had an `ExecutorStrategy` we could force the outer `Par` to obtain a `Par[A]`, but we'd like to keep our `Par` values agnostic to the `ExecutorStrategy`. That is, we don't want to actually execute our parallel computation, we simply want to *describe* a parallel computation that *first* runs one parallel computation and *then* uses the result of that computation to choose what computation to run next.

This is a case where our existing primitives are insufficient. When you encounter these situations, you can respond by simply introducing the exact combinator needed (for instance, we could simply write `choice` as a new primitive, using the fact that `Par[A]` is merely an alias for `ExecutorService => Future[A]` rather than relying only on `unit`, `fork`, and `map2`). But quite often, the example that motivates the need for a new primitive will not be *minimal*—it will have some incidental features that aren't really relevant to the essence of the API's limitation. It's a good idea to try to explore some related examples around the particular one that cannot be expressed, to see if a common pattern emerges. Let's do that here.

If it's useful to be able to choose between *two* parallel computations based on the results of a first, it should be useful to choose between *N* computations:

```
def choiceN[A](a: Par[Int])(choices: List[Par[A]]): Par[A]
```

Let's say that `choiceN` runs `a`, then uses that to select a parallel computation from `choices`. This is a bit more general than `choice`.

EXERCISE 15: Implement `choiceN` and then `choice` in terms of `choiceN`.

EXERCISE 16: Still, let's keep looking at some variations. Try implementing the following combinator. Here, instead of a list of computations, we have a `Map` of them:[18]

Footnote 18 `Map` (API link) is a purely functional data structure.

```
def choiceMap[A,B](a: Par[A])(choices: Map[A,Par[B]]): Par[B]
```

If you want, stop reading here and see if you can come up with a combinator

that generalizes these examples.

Something about these combinators seems a bit arbitrary. The `Map` encoding of the set of possible choices feels too specific. If you look at your implementation of `choiceMap`, you can see you aren't really using much of the API of `Map`. Really, we are just using the `Map[A,Par[B]]` to provide a function, `A => Par[B]`. And looking back at `choice` and `choiceN`, we can see that for `choice`, the pair of arguments was just being used as a `Boolean => Par[A]` (where the boolean selects either the first or second element of the pair), and for `choiceN`, the list was just being used as an `Int => Par[A]`.

Let's take a look at the signature.

```
def chooser[A,B](a: Par[A])(choices: A => Par[B]): Par[B]
```

EXERCISE 17: Implement this new primitive `chooser`, then use it to implement `choice` and `choiceN`.

Whenever you generalize functions like this, take a look at the result when you're finished. Although the function you've written may have been motivated by some very specific use case, the signature and implementation may have a more general meaning. In this case, `chooser` is perhaps no longer the most appropriate name for this operation, which is actually quite general—it is a parallel computation that, when run, will run an initial computation whose result is used to determine a second computation. Nothing says that this second computation needs to even exist before the first computation's result is available. Perhaps it is being *generated* from whole cloth using the result of the first computation. This function, which comes up quite often in combinator libraries, is usually called `bind` or `flatMap`:

```
def flatMap[A,B](a: Par[A])(f: A => Par[B]): Par[B]
```

Is `flatMap` really the most primitive possible function? Let's play around with it a bit more. Recall when we first tried to implement `choice`, we ended up with a `Par[Par[A]]`. From there we took a step back, tried some related examples, and eventually discovered `flatMap`. But suppose instead we simply *conjured* another combinator, let's call it `join`, for converting `Par[Par[A]]` to `Par[A]`:

```

```

```
def join[A](a: Par[Par[A]]): Par[A]
```

We'll call it `join` since conceptually, it is a parallel computation that when run, will run the inner computation, wait for it to finish (much like `Thread.join`), then return its result. Again, we are just following the types here. We have an example that demands a function with the given signature, and so we just bring it into existence.

EXERCISE 18: Implement `join`. Optional: can it be implemented in a way that avoids deadlock, even when run on bounded thread pools as in `fork`? Can you see how to implement `flatMap` using `join`? And can you implement `join` using `flatMap`?

We are going to stop here, but you are encouraged to try exploring this algebra further. Try more complicated examples, discover new combinators, and see what you find! Here are some questions to consider:

- Can you implement a function with the same signature as `map2`, but using `bind` and `unit`? How is its meaning different than that of `map2`?
- Can you think of laws relating `join` to the other primitives of the algebra?
- Are there parallel computations that cannot be expressed using this algebra? Can you think of any computations that cannot even be expressed by adding new primitives to the algebra?
- In this chapter, we've chosen a "pull" model for our parallel computations. That is, when we run a computation, we get back a `Future`, and we can block to obtain the result of this `Future`. Are there alternative models for `Par` that don't require us to ever block on a `Future`?

7.3 Conclusion

In this chapter, we worked through the design of a library for defining parallel and asynchronous computations. Although this domain is interesting, the goal of this chapter was to give you a window into the process of functional design, to give you a sense of the sorts of issues you're likely to encounter and to give you ideas for how you can handle them. If you didn't follow absolutely every part of this, or if certain conclusions felt like logical leaps, don't worry. No two people take the same path when designing a library, and as you get more practice with functional design, you'll start to develop your own tricks and techniques for exploring a problem and possible designs.

In the next chapter, we are going to look at a completely different domain, and take yet another meandering journey toward discovering an API for that domain.

Index Terms

free theorem
map fusion
parametricity
shape
structure-preserving
typecasing

8 Property-based testing

8.1 Introduction

In the last chapter, we worked through the design of a functional library for expressing parallel computations. There, we introduced the idea of an API forming an *algebra*—that is, a collection of data types, functions over these data types, and importantly, *laws* or *properties* that express relationships between these functions. We also hinted at the idea that it might be possible to somehow check these laws automatically.

This chapter will work up to the design and implementation of a simple but powerful *property-based testing* library. What does this mean? The general idea of such a library is to decouple the specification of program behavior from the creation of test cases. The programmer focuses on specifying the behavior and giving high-level constraints on the test cases; the framework then handles generating (often *random*) test cases satisfying the constraints and checking that programs behave as specified for each case.

8.2 A brief tour of property-based testing

As an example, in ScalaCheck, a property-based testing library for Scala, a property looks something like:

```
val intList = Gen.listOf(Gen.choose(0,100))                          (1)
val prop =                                                           (2)
  forAll(intList)(l => l.reverse.reverse == l) &&                    (3)
  forAll(intList)(l => l.headOption == l.reverse.lastOption)         (4)
val failingProp = forAll(intList)(l => l.reverse == l)               (5)
```

(1) A generator of lists of integers between 0 and 100
(2) A property which specifies the behavior of the List.reverse method

3. Check that reversing a list twice gives back the original list
4. Check that the first element becomes the last element after reversal
5. A property which is obviously false

And we can check properties like so:

```
scala> prop.check
+ OK, passed 100 tests.

scala> failingProp.check
! Falsified after 6 passed tests.
> ARG_0: List("0", "1")
```

Here, `intList` is not a `List[Int]`, but a `Gen[List[Int]]`, which is something that knows how to generate test data of type `List[Int]`. We can *sample* from this generator, and it will produce lists of different lengths, filled with random numbers between `0` and `100`. Generators in a property-based testing library have a rich API. We can combine and compose generators in different ways, reuse them, and so on.

The function `forAll` creates a *property* by combining a `Gen[A]` with some predicate `A => Boolean` to check for each value generated. Like `Gen`, properties can also have a rich API. Here in this simple example we have used `&&` to combine two properties. The resulting property will hold only if neither property can be *falsified* by any of the generated test cases. Together, the two properties form a *partial* specification of the correct behavior of the `reverse` method.[1]

Footnote 1 Hints and standalone answers

When we invoke `prop.check`, ScalaCheck will randomly generate `List[Int]` values and ensure that each passes the predicates we have supplied. The output indicates that ScalaCheck has generated 100 test cases (of type `List[Int]`) and that the predicates were satisfied for each. Properties can of course fail—the output of `failingProp.check` indicates that the predicate tested false for some input, which is helpfully printed out to facilitate further testing or debugging.

EXERCISE 1: To get more of a feel for how to approach testing in this way, try thinking of properties to specify the implementation of a `sum: List[Int] => Int` function. Don't try writing your properties down as executable ScalaCheck code, an informal description is fine. Here are some ideas to get you started:

- Reversing a list and summing it should give the same result as summing the original, non-reversed list.
- What should the sum be if all elements of the list are the same value?

Can you think of other properties?

EXERCISE 2: What are properties that specify the function that finds the maximum of a `List[Int]`?

Property-based testing libraries often come equipped with other useful features. We'll talk more about these features later, but just to give an idea of what is possible:

- *Test case minimization*: In the event of discovering a failing test, the test runner tries smaller sizes until finding the *minimal* size test case that also fails, which is more illuminating for debugging purposes. For instance, if a property fails for a list of size 10, the test runner checks smaller lists and reports the smallest failure.
- *Exhaustive test case generation*: We call the set of possible values that could be produced by some `Gen[A]` the *domain*.[2] When the domain is small enough (for instance, suppose the domain were all even integers less than 100), we may exhaustively generate all its values, rather than just randomly from it. If the property holds for all values in a domain, we have an actual proof, rather than just the absence of evidence to the contrary.

Footnote 2 This is the same usage of 'domain' as the domain of a function—generators describe possible inputs to functions we would like to test. Note that we will also still sometimes use 'domain' in the more colloquial sense, to refer to a subject or area of interest, e.g. "the domain of functional parallelism" or "the error-handling domain".

ScalaCheck is just one property-based testing library. And while there's nothing wrong with it, we are going to be deriving our own library in this chapter, starting from scratch. This is partially for pedagogical purposes, but there's another reason: we want to encourage the view that no existing library (even one designed by supposed experts) is authoritative. Don't treat existing libraries as a cookbook to be followed. Most libraries contain a whole lot of *arbitrary* design choices, many made unintentionally. Look back to the previous chapter—notice how on several occasions, we did some informal reasoning to rule out entire classes of possible designs. This sort of thing is an inevitable part of the design process (it is impossible to fully explore every conceivable path), but it means it's easy to miss out on workable designs.

When you start from scratch, you get to revisit all the fundamental assumptions that went into designing the library, take a different path, and discover things about

the domain and the problem space that others may not have even considered. As a result, you might arrive at a design that's much better for your purposes. But even if you decide you like the existing library's solution, spending an hour or two of playing with designs and writing down some type signatures is a great way to learn more about a domain, understand the design tradeoffs, and improve your ability to think through design problems.

8.3 Choosing data types and functions

In this section, we will embark on another messy, iterative process of discovering data types and a set of primitive functions and combinators for doing property-based testing. As before, this is a chance to peer over the shoulder of someone working through possible designs. The particular path we take and the library we arrive at isn't necessarily the same as what you would discover. If property-based testing is unfamiliar to you, even better; this is a chance to explore a new domain and its design space, and make your own discoveries about it. If at any point, you're feeling inspired or have ideas of your own about how to design a library like this, don't wait for an exercise to prompt you—*put the book down* and go off to implement and play with your ideas. You can always come back to this chapter if you want ideas or get stuck on how to proceed.

With that said, let's get started. What data types should we use for our testing library? What primitives should we define, and what should their meanings be? What laws should our functions satisfy? As before, we can look at a simple example and "read off" the needed data types and functions, and see what we find. For inspiration, let's look at the ScalaCheck example we showed earlier:

```
val intList = Gen.listOf(Gen.choose(0,100))
val prop =
  forAll(intList)(l => l.reverse.reverse == l) &&
  forAll(intList)(l => l.headOption == l.reverse.lastOption)
```

Without knowing anything about the implementation of `Gen.choose` or `Gen.listOf`, we can guess that whatever data type they return (let's call it `Gen`, short for "generator") must be parametric in some type. That is, `Gen.choose(0,100)` probably returns a `Gen[Int]`, and `Gen.listOf` is then a function with the signature `Gen[Int] => Gen[List[Int]]`. But

since it doesn't seem like `Gen.listOf` should care about the type of the `Gen` it receives as input (it would be odd to require separate combinators for creating lists of `Int`, `Double`, `String`, and so on), let's go ahead and make it polymorphic:

```
def listOf[A](a: Gen[A]): Gen[List[A]]
```

We can learn many things by looking at this signature. Notice what we are *not* specifying—the size of the list to generate. For this to be implementable, this implies our generator must either assume or be told this size. Assuming a size seems a bit inflexible—whatever we assume is unlikely to be appropriate in all contexts. So it seems that generators must be told the size of test cases to generate. We could certainly imagine an API where this were explicit:

```
def listOfN[A](n: Int, a: Gen[A]): Gen[List[A]]
```

This would certainly be a useful combinator, but *not* having to explicitly specify sizes is powerful as well—it means the test runner has the freedom to choose test case sizes, which opens up the possibility of doing the test case minimization we mentioned earlier. If the sizes are always fixed and specified by the programmer, the test runner won't have this flexibility. Keep this concern in mind as we get further along in our design.

What about the rest of this example? The `forAll` function looks interesting. We can see that it accepts a `Gen[List[Int]]` and what looks to be a corresponding predicate, `List[Int] => Boolean`. But again, it doesn't seem like `forAll` should care about the types of the generator and the predicate, as long as they match up. We can express this with the type:

```
def forAll[A](a: Gen[A])(f: A => Boolean): Prop
```

Here, we've simply invented a new type, `Prop` (short for "property", following the ScalaCheck naming), for the result of binding a `Gen` to a predicate. We might not know the internal representation of `Prop` or what other functions it supports but based on this example, we can see that it has an && operator, so let's introduce that:

```

```

```
trait Prop { def &&(p: Prop): Prop }
```

8.3.1 The meaning and API of properties

Now that we have a few fragments of an API, let's discuss what we want our types and functions to *mean*. First, consider `Prop`. We know there exist functions `forAll` (for creating a property) and `check` (for running a property). In ScalaCheck, this `check` method has a side effect of printing to console. This is probably fine to expose as a convenience function, but it's not a basis for composition. For instance, we couldn't implement `&&` for `Prop` if its only API were the `check` method:[3]

Footnote 3 This might remind you of similar problems that we discussed last chapter, when we looked at directly using `Thread` and `Runnable` for parallelism.

```
trait Prop {
  def check: Unit
  def &&(p: Prop): Prop = ???
}
```

In order to combine `Prop` values using combinators like `&&`, we need `check` (or whatever function "runs" properties) to return some meaningful value. What type should that value be? Well, let's consider what sort of information we'd like to get out of checking our properties. At a minimum, we need to know whether the property succeeded or failed. This lets us implement `&&`.

EXERCISE 3: Assuming the following definition of `Prop`, implement `&&` as a method of `Prop`:

```
trait Prop { def check: Boolean }
```

In this representation, `Prop` is nothing more than a non-strict `Boolean`, and any of the usual `Boolean` functions ('and', 'or', 'not', 'xor', etc) can be defined for `Prop`. But a `Boolean` alone is probably insufficient. If a property fails, we'd like to perhaps know how many tests succeeded first, and what the arguments were that resulted in the failure. And if it succeeded, we might like to know how many tests it ran. Let's try to return an `Either` to indicate this success or failure:

```
object Prop {
```

```
  type SuccessCount = Int                                    (1)
  ...
}
trait Prop { def check: Either[???,SuccessCount] }
```

(1) Type aliases like this can help readability of an API

There's a problem, what type do we return in the failure case? We don't know anything about the type of the test cases being generated internal to the `Prop`. Should we add a type parameter to `Prop`, make it a `Prop[A]`? Then `check` could return `Either[A,Int]`. Before going too far down this path, let's ask ourselves, do we really care though about the *type* of the value that caused the property to fail? Not exactly. We would only care about the type if we were going to do further computation with this value. Most likely we are just going to end up printing this value to the screen, for inspection by the person running these tests. After all, the goal here is a) to find bugs, and b) to indicate to a person what test cases trigger those bugs, so they can go fix them. This suggests we can get away with the following type:

```
object Prop {
  type FailedCase = String
  type SuccessCount = Int
}

trait Prop { def check: Either[FailedCase,SuccessCount] }
```

In the case of failure, `check` returns a `Left(s)`, where `s` is some `String` representation of the value that caused the property to fail. As a general rule, whenever you return a value of some type `A` from a function, think about what callers of your function are likely to do with that value. Will any of them care that the value is of type `A`, or will they always convert your `A` to some other uniform representation (like `String` in this case)? If you have a good understanding of all the ways callers will use your function, and they all involve converting your value to some other type like `String`, there's often no loss in expressiveness to simply return that `String` directly. The uniformity of representation makes composition easier.

That takes care of the return value of `check`, at least for now, but what about the arguments to `check`? Right now, the `check` method takes no arguments. Is this sufficient? We can think about what information `Prop` will have access to just

by inspecting the way `Prop` values are created. In particular, let's look at `forAll`:

```
def forAll[A](a: Gen[A])(f: A => Boolean): Prop
```

Without knowing more about the representation of `Gen`, it's hard to say whether there's enough information here to be able to generate values of type `A` (which is what we need to implement `check`). So for now let's turn our attention to `Gen`, to get a better idea of what it means and what its dependencies might be.

8.3.2 The meaning and API of generators

Let's take a step back to reflect on what we've learned so far. We've certainly made some progress. By inspecting a simple example, we learned that our library deals with at least two fundamental types, `Gen`, and `Prop`, and we've loosely assigned meanings to these types. In looking at `Gen`, we made what seems like an important distinction between generators whose "size" is chosen explicitly by the programmer (as in `listOfN`), and generators where the testing framework is allowed to pick sizes (as in `listOf`). We noted this as something to keep in mind for later.

Somewhat arbitrarily, we then chose to look at `Prop` first, and determined we couldn't commit to a concrete representation for `Prop` without first knowing the representation of `Gen`. We've made a note of this, and plan on returning to `Prop` shortly. Have we made a mistake by starting with `Prop`? Not at all. This sort of thing happens all the time. It doesn't matter much where we begin our inquiry—the domain will inexorably guide us to make all the design choices that are required. As you do more functional design, you'll develop a better intuition for where a good place is to start.

TWO WAYS OF GENERATING VALUES

Let's press on. We determined earlier that a `Gen[A]` was something that knows how to generate values of type `A`. What are some ways it could do that? Well, it could *randomly* generate these values. Look back at the example from chapter six—there, we gave an interface for a purely functional random number generator and showed how to make it convenient to combine computations that made use of it. We could have `Gen` build on this:[4]

Footnote 4 Recall `case class State[S,A](run: S => (A,S))`.

```
type Gen[A] = State[RNG,A]
```

EXERCISE 4: Implement `Gen.choose` using this representation of `Gen`. Feel free to use functions you've already written.

```
def choose(start: Int, stopExclusive: Int): Gen[Int]
```

But in addition to randomly generating values, it might be nice when possible to exhaustively enumerate a sequence of values, and be notified somehow that all possible values have been generated. If we can get through all possible values without finding a failing case, this is an actual *proof* that our property holds over its domain. Clearly exhaustive generation won't always be feasible, but in some cases it might be. As a simple example, a `Gen[Boolean]` could first generate `true`, then `false`, then report it was done generating values. Likewise for the expression `Gen.choose(1,10)`. Note that if we can't exhaustively enumerate all possible values, we'll likely want to sample our values differently—for instance, if we have `Gen.choose(1,10)`, we could generate `1, 2, ... 10` in sequence, but if we have `Gen.choose(0,1000000000)`, we won't be able to enumerate all possible values and should probably sample our random values more uniformly from the range.

If we want to support both modes of test case generation (random and exhaustive), we need to extend `Gen`. `State[RNG,A]` gives us all we need to support random test case generation, but what about exhaustive generation? Well, the simplest type we could use to encode a generator for an exhaustive list of values is... a `List` of these values:

```
type Gen[+A] = (State[RNG,A], List[A])
```

The first element of the pair is the generator of random values, the second is an exhaustive list of values. Note that with this representation, the test runner will likely have to choose between the two modes based on the number of test cases it is running. For instance, if the test runner is running 1000 tests, it could 'spend' up to the first 300 of these tests working through the domain exhaustively, then switch to randomly sampling the domain if the domain has not been fully enumerated. We'll get to writing this logic a bit later, after we nail down exactly how to represent our "dual-mode" generators.

There's a few problems with the current encoding of these dual-mode generators. For one, it would be a shame to have to exhaustively generate all these values if we end up having to resort to random test case generation (and the full set of values may be huge or even infinite!). So let's use a `Stream` in place of `List`. We are going to use the `Stream` type we developed last chapter and also promote our type alias to a data type.[5]

Footnote 5 We aren't using Scala's standard library streams here. They have an unfortunate "off-by-one" error—they strictly evaluate their first element. Generic code like what we are writing in this chapter can't assume it is desireable to evaluate any part of the stream until it is explicitly requested.

```
case class Gen[+A](sample: State[RNG,A], exhaustive: Stream[A])
```

EXERCISE 5: Let's see what we can implement using this representation of `Gen`. Try implementing `unit`, `boolean`, `choose`, and `listOfN`.

```
def unit[A](a: => A): Gen[A]
```

```
def boolean: Gen[Boolean]
```

```
def choose(start: Int, stopExclusive: Int): Gen[Int]
```

```
/** Generate lists of length n, using the given generator. */
def listOfN[A](n: Int, g: Gen[A]): Gen[List[A]]
```

So far so good. But these domains are all finite. What should we do about infinite domains, like a `Double` generator in some range:

```
/** Between 0 and 1, not including 1. */
def uniform: Gen[Double]
```

```
/** Between `i` and `j`, not including `j`. */
def choose(i: Double, j: Double): Gen[Double]
```

To randomly sample from these domains is straightforward, but what should we

do for `exhaustive`? Return the empty `Stream`? No, probably not. Previously, we made a choice about the meaning of an empty stream—we interpreted it to mean that we have finished exhaustively generating values in our domain and there are no more values to generate. We could change its meaning to "the domain is infinite, use random sampling to generate test cases", but then we lose the ability to determine that we have exhaustively enumerated our domain, or that the domain is simply empty. How can we distinguish these cases? One simple way to do this is with `Option`:

```
case class Gen[+A](sample: State[RNG,A], exhaustive: Stream[Option[A]])
```

We'll adopt the convention that a `None` in `exhaustive` signals to the test runner should switch to random sampling, because the domain is infinite or otherwise not worth fully enumerating.[6] If the domain can be fully enumerated, `exhaustive` will be a finite stream of `Some` values. Note that this is a pretty typical usage of `Option`. Although we introduced `Option` as a way of doing error handling, `Option` gets used a lot whenever we need a simple way of encoding one of two possible cases.

Footnote 6 We could also choose to put the `Option` on the outside: `Option[Stream[A]]`. You may want to explore this representation on your own. You will find that it doesn't work out so well as it requires that we be able to decide *up front* that the domain is not worth enumerating. We will see examples later of generators where it isn't possible to make this determination.

SIDEBAR **Using the simplest possible types**

We could create a new data type to wrap our `Stream[Option[A]]` representing our domain for `exhaustive`, but we will sometimes hold off on doing this until it feels justified (say, when we accumulate some functions over this representation that aren't trivially defined in terms of `Option` and `Stream`, or we really want to hide this representation in the interest of modularity). When using a type like this with no particular meaning attached to it, it can be a good practice to define type aliases and functions which help codify and document the meaning you have assigned to the type:

```
type Domain[+A] = Stream[Option[A]]

def bounded[A](a: Stream[A]): Domain[A] = a map (Some(_))
def unbounded: Domain[Nothing] = Stream(None)
```

This is pretty low cost—it doesn't require us to reimplement or replicate the API of `Option` or `Stream`. But it helps with documentation of the API and makes it easier to implement a refactoring later which promotes `Domain` to its own data type.

EXERCISE 6: With this representation, reimplement the operations `boolean`, `choose` (the `Int` version), and `listOfN` (hard) from before, then implement `uniform` and `choose` (the `Double` version).

REFININING THE PRIMITIVES

As we discussed last chapter, we are interested in understanding what operations are *primitive*, and what operations are *derived*, and in finding a small yet expressive set of primitives. A good way to explore what is expressible with a given set of primitives is to pick some concrete examples you'd like to express, and see if you can assemble the functionality you want. As you do so, look for patterns, try factoring out these patterns into combinators, and refine your set of primitives. We encourage you to stop reading here and simply *play* with the primitives and combinators we've written so far. If you want some concrete examples to inspire you, here are some ideas:

- If we can generate a single `Int` in some range, do we need a new primitive to generate an `(Int,Int)` pair in some range?
- Can we produce a `Gen[Option[A]]` from a `Gen[A]`? What about a `Gen[A]` from a `Gen[Option[A]]`?

- Can we generate strings somehow using our existing primitives?

SIDEBAR **The importance of play**

You don't have to wait around for a concrete example to force exploration of the design space. In fact, if you rely exclusively on concrete, obviously "useful" or "important" examples to design your API, you'll often miss out on aspects of the design space and generate designs with ad hoc, overly specific features. We don't want to *overfit* our API to the particular examples we happen to think of *right now*. We want to reduce the problem to its *essence*, and sometimes the best way to do this is *play*. Don't try to solve important problems or produce useful functionality. Not right away. Just experiment with different representations, primitives, and operations, let questions naturally arise, and explore whatever piques your curiosity. ("These two functions seem similar. I wonder if there's some more general operation hiding inside." or "Would it make sense to make this data type polymorphic?" or "What would it mean to change this aspect of the representation from a single value to a `List` of values?") There's no right or wrong way to do this, but there are so many different design choices that it's impossible *not* to run headlong into fascinating little questions to play with.

Here, we are going to take a bit of a shortcut. Notice that `Gen` is composed of a few other types, `Stream`, `State`, and `Option`. This can often be a hint that the API of `Gen` is going to have many of the same operations as these types. Let's see if there's some familiar operations from `Stream`, `State`, and `Option` that we can also define for `Gen`.

EXERCISE 7: Aha! Our `Gen` data type supports both `map` and `map2`.[7] See if you can implement these. Your implementation should be almost trivially defined in terms of the `map` and `map2` functions on `Stream`, `State`, and `Option`.[8] You can add them as methods of the `Gen` type, as we've done before, or write them as standalone functions in the `Gen` companion object. After you've implemented `map`, you may want to revisit your implementation of `choose` for `Double` and define it in terms of `uniform` and `map`.

Footnote 7 You've probably noticed by now that many data types support `map`, `map2`, and `flatMap`. We'll be discussing how to abstract over these similarities in part 3 of the book.

Footnote 8 In part 3 we will also learn how to derive these implementations automatically. That is, by composing our data types in certain well-defined ways, we can obtain the implementations of `map`, `map2`, and so on for free, without having to write any additional code!

```
def map[B](f: A => B): Gen[B]
```

```
def map2[B,C](g: Gen[B])(f: (A,B) => C): Gen[C]
```

So far so good. But `map` and `map2` are not expressive enough to encode some generators. Suppose we'd like a `Gen[(Int,Int)]` where both integers are odd, or both are even. Or a `Gen[List[Int]]` where we first generate a length between 0 and 11, then generate a `List[Double]` of the chosen length. In both these cases there is a dependency—we generate a value, then use that value to determine what generator to use next.[9] For this we need `flatMap`, another function we've seen before.

Footnote 9 Technically, this first case can be implemented by generating the two integers separately, and using `map2` to make them both odd or both even. But a more natural way is to choose an even or odd generator based on the first value generated.

EXERCISE 8: Implement `flatMap`, then use it to implement the generators mentioned above. You can make `flatMap` and this version of `listOfN` in the `Gen` class, and `sameParity` in the `Gen` companion object.

```
def flatMap[B](f: A => Gen[B]): Gen[B]
```

```
def sameParity(from: Int, to: Int): Gen[(Int,Int)]
```

```
def listOfN(size: Gen[Int]): Gen[List[A]]
```

EXERCISE 9 (hard, optional): Try implementing the version of `listOfN` which picks its size up front purely in terms of other primitives. Is it possible? If yes, give an implementation. Otherwise, explain why not:

```
def listOfN[A](n: Int, g: Gen[A]): Gen[List[A]]
```

EXERCISE 10 (hard): Implement `union`, for combining two generators of the same type into one, by pulling values from each generator with equal likelihood. What are some possible ways you could combine the two exhaustive streams? Can

you think of any reason to prefer one implementation over another?

```
def union(g1: Gen[A], g2: Gen[A]): Gen[A]
```

EXERCISE 11 (hard, optional): Implement `weighted`, a version of `union` which accepts a weight for each `Gen` and generates values from each `Gen` with probability proportional to its weight.

```
def weighted(g1: (Gen[A],Double), g2: (Gen[A],Double)): Gen[A]
```

REFININING THE PROP DATA TYPE

Now that we know more about our representation of generators, let's return to our definition of `Prop`. Our `Gen` representation has revealed information about the requirements for `Prop`. First, we notice that our properties can succeed in one of two ways—they can be *proven* correct, by exhaustive enumeration, or they can succeed when no counterexamples are found via random generation. Our current definition of `Prop` doesn't distinguish between these possibilities:

```
trait Prop { def check: Either[FailedCase,SuccessCount] }
```

It only includes two cases, one for success, and one for failure. We can create a new data type, `Status`, to represent the two ways a test can succeed:

```
trait Status
case object Proven extends Status
case object Unfalsified extends Status

trait Prop { def run: Either[FailedCase,(Status,SuccessCount)] }
```

A test can succeed by being *proven*, if the domain has been fully enumerated and no counterexamples found, or it can be merely *unfalsified*, if the test runner had to resort to random sampling.

`Prop` is now nothing more than a non-strict `Either`. But `Prop` is still missing some information—we have not specified how many test cases to examine before we consider the property to be passed. We could certainly hardcode something, but it would be better to propagate this dependency out:

```
type TestCases = Int
type Result = Either[FailedCase, (Status,SuccessCount)]
case class Prop(run: TestCases => Result)
```

Is this sufficient? Let's take another look at `forAll`. Can `forAll` be implemented? Why not?

```
def forAll[A](a: Gen[A])(f: A => Boolean): Prop
```

We can see that `forAll` does not have enough information to return a `Prop`. Besides the number of test cases to run, `Prop` must have all the information needed for generated test cases to return a `Status`, but if it needs to generate random test cases, it is going to need an `RNG`. Let's go ahead and propagate that dependency to `Prop`:

```
case class Prop(run: (TestCases,RNG) => Result)
```

We can start to see a pattern here. There are certain parameters that go into generating test cases, and if we think of other parameters besides the number of test cases and the source of randomness, we can just add these as extra arguments to `Prop`.

We now have enough information to actually implement `forAll`. Here is a simple implementation. We have a choice about whether to use exhaustive or random test case generation. For our implementation, we'll spend a third of our test cases examining elements from the exhaustive `Stream`. If we reach the end of that `Stream` or find a counterexample, we return immediately, otherwise, we fall back to generating random test cases for our remaining test cases:[10]

Footnote 10 Here is a question to explore—might there be a way to track the expected size of the exhaustive stream, such that the decision to use random data could be made up front? For some primitives, it is certainly possible, but is it possible for all our primitives?

```
def forAll[A](a: Gen[A])(f: A => Boolean): Prop = Prop {
  (n,rng) => {
    def go(i: Int, j: Int, s: Stream[Option[A]], onEnd: Int => Result):
    Result =
      if (i == j) Right((Unfalsified, i))
      else s.uncons match {
        case Some((Some(h),t)) =>
          try { if (f(h)) go(i+1,j,s,onEnd)
```

```
                else Left(h.toString) }
          catch { case e: Exception => Left(buildMsg(h, e)) }
        case Some((None,_)) => Right((Unfalsified,i))
        case None           => onEnd(i)
      }
    go(0, n/3, a.exhaustive, i => Right((Proven, i))) match {
      case Right((Unfalsified,_)) =>
        val rands = randomStream(a)(rng).map(Some(_))
        go(n/3, n, rands, i => Right((Unfalsified, i)))
      case s => s                                                    (1)
    }
  }
}
```

(1) If proven or failed, stop immediately

```
def buildMsg[A](s: A, e: Exception): String =
  "test case: " + s + "\n" +
  "generated an exception: " + e.getMessage + "\n" +
  "stack trace:\n" + e.getStackTrace.mkString("\n")
```

Notice we are catching exceptions and reporting them as test failures, rather than bringing down the test runner (which would lose information about what argument triggered the failure).

EXERCISE 12: Now that we have a representation of `Prop`, implement &&, and `||` for manipulating `Prop` values. While we can implement &&, notice that in the case of failure we aren't informed which property was responsible, the left or the right. (Optional): Can you devise a way of handling this, perhaps by allowing `Prop` values to be assigned a tag or label which gets displayed in the event of a failure?

```
def &&(p: Prop): Prop
```

```
def ||(p: Prop): Prop
```

8.3.3 Test case minimization

Earlier, we mentioned the idea of test case minimization. That is, we would ideally like our framework to find the *smallest* or simplest failing test case, to better illustrate the problem and facilitate debugging. Let's see if we can tweak our representations to support this. There are two general approaches we could take:

- *Shrinking*: After we have found a failing test case, we can run a separate procedure to minimize the test case, by successively decreasing its "size" until it no longer fails. This is called *shrinking*, and it usually requires us to write separate code for each data type to implement this minimization process.
- *Sized generation:* Rather than shrinking test cases after the fact, we simply generate our test cases in order of increasing size and complexity. So, we start small and increase size until finding a failure. This idea can be extended in various ways, to allow the test runner to make larger jumps in the space of possible sizes while still making it possible to find the smallest failing test.

ScalaCheck, incidentally, takes this first approach of *shrinking*. There's nothing wrong with this approach (it is also used by the Haskell library QuickCheck that ScalaCheck is based on) but we are going to see what we can do with sized generation. It's a bit simpler and in some ways more modular—our generators only need to be knowledgeable about how to generate a test case of a given size, they don't need to be aware of the 'schedule' used to search the space of test cases and the test runner therefore has the freedom to choose this schedule. We'll see how this plays out shortly.

We are going to do something different. Rather than modifying our `Gen` data type, for which we've already written a number of useful combinators, we are going to introduce sized generation as a separate layer in our library. That is, we are going to introduce a type, `SGen`, for representing sized generators:

```
case class SGen[+A](forSize: Int => Gen[A])
```

EXERCISE 13: Implement helper functions for converting `Gen` to `SGen`. You can add this as a method to `Gen`.

```
def unsized: SGen[A]
```

EXERCISE 14 (optional): Not surprisingly, `SGen` at a minimum supports many of the same operations as `Gen`, and the implementations are rather mechanical. You may want to define some convenience functions on `SGen` that simply delegate to the corresponding functions on `Gen`.[11]

Footnote 11 Again, we are going to discuss in Part 3 ways of factoring out this sort of duplication.

EXERCISE 15: We can now implement a `listOf` combinator that does not accept an explicit size. It can return an `SGen` instead of a `Gen`. The

implementation can generate lists of the requested size.

```
def listOf[A](g: Gen[A]): SGen[List[A]]
```

Let's see how `SGen` affects the definition of `Prop` and `Prop.forAll`. The `SGen` version of `forAll` looks like this:

```
def forAll[A](g: SGen[A])(f: A => Boolean): Prop
```

Can you see how this function is not possible to implement? `SGen` is expecting to be told a size, but `Prop` does not receive any size information. Much like we did with the source of randomness and number of test cases, we simply need to propagate this dependency to `Prop`. But rather than just propagating this dependency *as is* to the caller of `Prop`, we are going to have `Prop` accept a *maximum* size. This puts `Prop` in charge of invoking the underlying generators with various sizes, up to and including the maximum specified size, which means it can also search for the smallest failing test case. Let's see how this works out:[12]

Footnote 12 This rather simplistic implementation gives an equal number of test cases to each size being generated, and increases the size by `1` starting from `0`. We could imagine a more sophisticated implementation that does something more like a binary search for a failing test case size—starting with sizes `0,1,2,4,8,16...` then narrowing in on smaller sizes in the event of a failure.

```
case class Prop(run: (MaxSize,TestCases,RNG) => Status)
```

```
def forAll[A](g: SGen[A])(f: A => Boolean): Prop =
  forAll(g.forSize)(f)

def forAll[A](g: Int => Gen[A])(f: A => Boolean): Prop = Prop {
  (max,n,rng) =>
    val casesPerSize = n / max + 1
    val props: List[Prop] =
      Stream.from(0).take(max+1).map(i => forAll(g(i))(f)).toList
    props.map(p => Prop((max,n,rng) => p.run(max,casesPerSize,rng))).
          reduceLeft(_ && _)(max,n,rng)
}
```

This implementation highlights a couple minor problems with our representation of `Prop`. For one, there are actually now *three* ways that a property can succeed. It can be *proven*, if the domain of the generator has been fully

examined (for instance, a `SGen[Boolean]` can only ever generate two distinct values, regardless of size). It can be *exhausted*, if the domain of the generator has been fully examined, but only up through the maximum size. Or it could be merely *unfalsified*, if we had to resort to random generation and no counterexamples were found. Let's add this to our `Status` representation:

```
case object Exhausted extends Status
case object Proven extends Status
case object Unfalsified extends Status
```

EXERCISE 16: Try to reimplement `forAll`, assuming this definition of `Status`. Notice that we lack a way to distinguish between `Proven` and `Exhausted`. Why is that? Can you see how to fix it?

The problem is that `SGen` is a totally opaque function from `Size` to `Gen`, and so the test runner in `forAll` cannot distinguish a generator that has been *fully* exhausted from a generator that has merely been exhausted for the given size. We can add this distinction by making `SGen` into a data type with two cases:

```
trait SGen[+A]
case class Sized[+A](forSize: Size => Gen[A]) extends SGen[A]
case class Unsized[+A](get: Gen[A]) extends SGen[A]
```

EXERCISE 17: Implement `forAll` for this representation of `SGen` and any other functions you've implemented that require updating to reflect these changes in representation. Notice that we only need to update primitive combinators—derived combinators get their new behavior "for free", based on the updated implementation of primitives.

8.3.4 Using the library, improving its usability, and future directions

We have converged on what seems like a reasonable API. We could keep tinkering with it, but at this point let's try *using* the library to construct tests and see if we notice any deficiencies, either in what it can express or its general usability. Usability is somewhat subjective, but we generally like to provide convenient *syntax* and appropriate *helper functions* which abstract out common patterns that occur in client usage of the library. We aren't necessarily aiming here to make the library more expressive, we simply want to make it more pleasant to use.

Let's revisit an example that we mentioned at the start of this

chapter—specifying the behavior of the function `max`, available as a method on `List` (API docs link). The maximum of a list should be greater than or equal to every other element in the list. Let's specify this:

```
val smallInt = Gen.choose(-10,10)
val maxProp = forAll(listOf(smallInt)) { l =>
  val max = l.max
  !l.exists(_ > max)                                    (1)
}
```

(1) No value greater than max should exist in l

We can introduce a helper function in `Prop` for actually *running* our `Prop` values and printing their result to the console in a useful format:

```
def run(p: Prop,
        maxSize: Int = 100,                             (1)
        testCases: Int = 100,
        rng: RNG = RNG.simple(System.currentTimeMillis)): Unit = {
  p.run(maxSize, testCases, rng) match {
    case Left(msg) => println("! test failed:\n" + msg)
    case Right((Unfalsified,n)) =>
      println("+ property unfalsified, ran " + n + " tests")
    case Right((Proven,n)) =>
      println("+ property proven, ran " + n + " tests")
    case Right((Exhausted,n)) =>
      println("+ property unfalsified up to max size, ran " +
              n + " tests")
  }
}
```

(1) A default argument of 200

We are using default arguments here to make it more convenient to call in the case that the defaults are fine.

EXERCISE 18: Try running `Prop.run(maxProp)`. Notice that it fails!

Property-based testing has a way of revealing all sorts of hidden assumptions we have about our code, and forcing us to be much more explicit about these assumptions. The Scala standard library implementation of `max` crashes when given the empty list (rather than returning an `Option`).

EXERCISE 19: Define `listOf1`, for generating nonempty lists, then update your specification of `max` to use this generator.

Let's try a few more examples.

EXERCISE 20: Write a property to verify the behavior of `List.sorted` (API docs link), which you can use to sort (among other things) a `List[Int]`.[13] For instance, `List(2,1,3).sorted == List(1,2,3)`.

Footnote 13 `sorted` takes an *implicit* `Ordering` for the elements of the list, to control the sorting strategy.

```
val sortedProp = forAll(listOf(smallInt)) { l =>
  val ls = l.sorted
  l.isEmpty || !l.zip(l.tail).exists { case (a,b) => a > b }
}
```

Recall that in the previous chapter we looked at laws we expected to hold for our parallel computations. Can we express these laws with our library? The first "law" we looked at was actually a particular test case:

```
map(unit(1))(_ + 1) == unit(2)
```

We certainly can express this, but the result is somewhat ugly.[14]

Footnote 14 Recall that `type Par[A] = ExecutorService => Future[A]`.

```
val ES: ExecutorService = Executors.newCachedThreadPool
val p1 = Prop.forAll(Gen.unit(Par.unit(1)))(i =>
  Par.map(i)(_ + 1)(ES).get == Par.unit(2)(ES).get)
```

We've expressed the test, but it's verbose, cluttered, and the "idea" of the test is obscured by details that aren't really relevant here. Notice that this isn't a question of the API being expressive enough—yes we can express what we want, but a combination of missing helper functions and poor syntax obscures the intent.

Let's improve on this. Our first observation is that `forAll` is a bit too general for this test case. We aren't varying the input to this test, we just have a hardcoded example. Hardcoded examples should be just as convenient to write as in a traditional unit testing library. Let's introduce a combinator for it:

```
def check(p: => Boolean): Prop =                    (1)
  forAll(unit(()))(_ => p)
```

(1) Note that we are non-strict here

Is this cheating? Not at all. We provide the unit generator, which, of course, only generates a single value. The value will be ignored in this case, simply used to drive the evaluation of the given `Boolean`. Notice that this combinator is general-purpose, having nothing to do with `Par`—we can go ahead and move it into the `Prop` companion object. Updating our test case to use it gives us:

```
val p2 = check {
  val p = Par.map(Par.unit(1))(_ + 1)
  val p2 = Par.unit(2)
  p(ES).get == p2(ES).get
}
```

Better. Can we do something about the `p(ES).get` and `p2(ES).get` noise? There's something rather unsatisfying about it. For one, we're forcing this code to be aware of the internal implementation details of `Par`, simply to compare two `Par` values for equality. One improvement is to move the equality comparison into `Par`, and into a helper function, which means we only have to run a single `Par` at the end to get our result:

```
def equal[A](p: Par[A], p2: Par[A]): Par[Boolean] =
  Par.map2(p,p2)(_ == _)

val p3 = check {
  equal (
    Par.map(Par.unit(1))(_ + 1),
    Par.unit(2)
  ) (ES) get
}
```

So, we are *lifting* equality to operate in `Par`, which is a bit nicer than having to run each side separately. But while we're at it, why don't we move the *running* of `Par` out into a separate function, `forAllPar`, and the analogous `checkPar`. This also gives us a good place to insert variation *across* different parallel strategies, without it cluttering up the property we are specifying:

```
val S = weighted(
  choose(1,4).map(Executors.newFixedThreadPool) -> .75,
  unit(Executors.newCachedThreadPool) -> .25)                    ❶

def forAllPar[A](g: Gen[A])(f: A => Par[Boolean]): Prop =
  forAll(S.map2(g)((_,_))) { case (s,a) => f(a)(s).get }
```

```
def checkPar(p: Par[Boolean]): Prop =
  forAllPar(Gen.unit(()))(_ => p)
```

❶ a -> b is syntax sugar for (a,b)

`S.map2(g)((_,_))` is a rather noisy way of combining two generators to produce a pair of their outputs. Let's quickly introduce a combinator to clean that up:[15]

Footnote 15 Hints and standalone answers

```
def **[B](g: Gen[B]): Gen[(A,B)] =
  (this map2 g)((_,_))
```

Much nicer:

```
def forAllPar2[A](g: Gen[A])(f: A => Par[Boolean]): Prop =
  forAll(S ** g) { case (s,a) => f(a)(s).get }
```

We can even introduce `**` as a pattern using custom extractors, which lets us write:

```
def forAllPar3[A](g: Gen[A])(f: A => Par[Boolean]): Prop =
  forAll(S ** g) { case s ** a => f(a)(s).get }
```

This syntax works nicely when tupling up multiple generators—when pattern matching, we don't have to nest parentheses like would be required when using the tuple pattern directly. To enable `**` as a pattern, we define an object called `**` with an `unapply` function:

```
object ** {
  def unapply[A,B](p: (A,B)) = Some(p)
}
```

See the custom extractors documentation for more details.

So, `S` is a `Gen[ExecutorService]` that will vary over fixed sized thread pools from 1-4 threads, and also consider an unbounded thread pool. And now our property looks a lot cleaner:[16]

Footnote 16 We cannot use the standard Java/Scala `equals` method, or the `==` method in Scala (which delegates to the `equals` method), since that method returns a `Boolean` directly, and we need to return a `Par[Boolean]`. Some infix syntax for `equal` might be nice. See the chapter code for the previous chapter on purely functional parallelism for an example of how to do this.

```
val p2 = checkPar {
  equal (
    Par.map(Par.unit(1))(_ + 1),
    Par.unit(2)
  )
}
```

These might seem like minor changes, but this sort of factoring and cleanup can greatly improve the usability of your library, and the helper functions we've written make the properties easier to read and more pleasant to write. You may want to add versions of `forAllPar` and `checkPar` for sized generators as well.

Let's look at some other properties from the previous chapter. Recall that we generalized our test case:

```
map(unit(x))(f) == unit(f(x))
```

We then simplified it to the law:

```
map(y)(id) == y
```

Can we express this? Not exactly. This property implicitly states that the equality holds *for all* choices of `y`, for all types. We are forced to pick particular values for `y`:

```
val pint = Gen.choose(0,10) map (Par.unit(_))
val p4 =
  forAllPar(pint)(n => equal(Par.map(n)(y => y), n))
```

We can certainly range over more choices of `y`, but what we have here is probably good enough. The implementation of `map` cannot care about the values of our parallel computation, so there isn't much point in constructing the same test for `Double`, `String`, and so on. What `map` *can* be affected by is the *structure* of the parallel computation. If we wanted greater assurance that our property held, we

could provide richer generators for the structure. Here, we are only supplying `Par` expressions with one level of nesting.

EXERCISE 21 (hard): Writer a richer generator for `Par[Int]`, which builds more deeply nested parallel computations than the simple ones we gave above.

EXERCISE 22: Express the property about `fork` from last chapter, that `fork(x) == x`.

So far, our library seems quite expressive, but there's one area where it's lacking: we don't currently have a good way to test higher-order functions. While we have lots of ways of generating *data*, using our generators, we don't really have a good way of generating *functions*.

For instance, let's consider the `takeWhile` function defined for `List` and `Stream`. Recall that this function returns the longest prefix of its input whose elements all satisfy a predicate. For instance, `List(1,2,3).takeWhile(_ < 3)` results in `List(1,2)`. A simple property we'd like to check is that for any stream, `s: List[A]`, and any `f: A => Boolean`, `s.takeWhile(f).forall(f)`, that is, every element in the returned stream satisfies the predicate.[17]

Footnote 17 In the Scala standard library, `forall` is a method on `List` and `Stream` with the signature `def forall[A](f: A => Boolean): Boolean`.

EXERCISE 23: Come up with some other properties that `takeWhile` should satisfy. Can you think of a good property expressing the relationship between `takeWhile` and `dropWhile`?

We could certainly take the approach of only examining *particular* arguments when testing HOFs like `takeWhile`. For instance, here's a more specific property for `takeWhile`:

```
val isEven = (i: Int) => i%2 == 0
val takeWhileProp =
  Prop.forAll(Gen.listOf(int))(l => l.takeWhile(isEven).forall(isEven))
```

This works, but is there a way we could let the testing framework handle generating functions to use with `takeWhile`?[18] Let's consider our options. To make this concrete, let's suppose we have a `Gen[Int]` and would like to produce

a `Gen[String => Int]`. What are some ways we could do that? Well, we could produce `String => Int` functions that simply ignore their input string and delegate to the underlying `Gen[Int]`.

Footnote 18 Recall that in the previous chapter we introduced the idea of *free theorems* and discussed how parametricity frees us somewhat from having to inspect the behavior of a function for every possible type argument. Still, there are many situations where being able to generate functions for testing is useful.

EXERCISE 24: Implement a function to do this conversion.

```
def genStringIntFn(g: Gen[Int]): Gen[String => Int]
```

This approach isn't sufficient though. We are simply generating *constant* functions that ignore their input. In the case of `takeWhile`, where we need a function that returns a `Boolean`, this will be a function that always returns `true` or always returns `false`—clearly not very interesting for testing the behavior of our function.

EXERCISE 25 (hard, optional): We clearly want to generate a function that *uses its argument* in some way to select which `Int` to return. Can you think of a good way of expressing this? This is a very open-ended and challenging design exercise. See what you can discover about this problem and if there is a nice general solution that you can incorporate into the library we've developed so far. This exercise is optional, but you may find it interesting to work on.

EXERCISE 26: You are strongly encouraged to venture out and try using the library we've developed! See what else you can test with it, and see if you discover any new idioms for its use or perhaps ways it could be extended further or made more convenient. Here are a few ideas to get you started:

- Try writing properties to specify the behavior of some of the other functions we wrote for `List` and `Stream`, for instance `take`, `drop`, `filter`, and `unfold`.
- Try writing a sized generator for producing the `Tree` data type we defined in chapter 3, then use this to specify the behavior of the `fold` function we defined for `Tree`. Can you think of ways to improve the API to make this easier?
- Try writing properties to specify the behavior of the `sequence` function we defined for `Option` and `Either`.

8.3.5 The laws of generators

Isn't it interesting that many of the functions we've implemented here, for our `Gen` type, look quite similar to other functions we've defined on `Par`, `List`, `Stream`, and `Option`? As an example, for `Par` we defined:

```
def map[A,B](a: Par[A])(f: A => B): Par[B]
```

And in this chapter we defined `map` for `Gen` (as a method on `Gen[A]`):

```
def map[B](f: A => B): Gen[B]
```

And we've defined very similar-looking functions for other data types. We have to wonder, is it merely that our functions share similar-looking signatures, or do they satisfy the same *laws* as well? Let's look at a law we introduced for `Par` in the previous chapter:

```
map(x)(id) == x
```

EXERCISE 27: Does this law hold for our implementation of `Gen.map`? What about for `Stream`, `List`, `Option` and `State`?

Fascinating! Not only do these functions share similar-looking signatures, they also in some sense have analogous meanings in their respective domains.

EXERCISE 28 (hard, optional): Spend a little while thinking up laws for some of the functions with similar signatures you've written for `List`, `Option`, and `State`. For each law, see if an analogous law holds for `Gen`.

It appears there are deeper forces at work! We are uncovering some rather fundamental patterns that cut across domains. In part 3, we'll learn the names for these patterns, discover the laws that govern them, and understand what it all means.[19]

Footnote 19 If curiosity is really getting the better of you, feel free to peek ahead at Part 3.

8.4 Conclusion

In this chapter, we worked through another extended exercise in functional library design, using the domain of property-based testing as inspiration. Once again, we reiterate that our goal was not necessarily to learn about property-based testing *per se*, but to give a window into the process of functional design. We hope these chapters are giving you ideas about how to approach functional library design in your own way and preparing you for the sorts of issues and questions you'll encounter. Developing an understanding of the overall process is much more important than following absolutely every small design decision we made as we explored the space of this particular domain.

In the next chapter, we'll look at another domain, *parsing*, with its own set of challenges and design questions. As we'll see, similar patterns will emerge.

Index Terms

9 Parser combinators

9.1 Introduction

In this chapter, we will work through the design of a combinator library for creating *parsers*, using JSON parsing as a motivating use case. As in the past two chapters, we will use this opportunity to provide insight into the process of functional design and notice common patterns that we'll discuss more in part 3.

This chapter will introduce a design approach called *algebraic design*. This is just a natural evolution of what we've already been doing to different degrees in past chapters—designing our interface and associated laws first and letting this guide our choice of data type representations. Here, we take this idea to its logical limit to see what it buys us.

At a few key points during this chapter, we will be giving more open-ended exercises, intended to mimic the scenarios you might encounter when designing and implementing your own libraries from scratch. You'll get the most out of this chapter if you use these opportunities to put the book down and spend some time investigating possible approaches. When you design your own libraries, you won't be handed a nicely chosen sequence of type signatures to fill in with implementations. You will have to make the decisions about what types and combinators should even exist and a goal in part 2 of this book has been to prepare you for doing this on your own. As always, in this chapter, if you get stuck on one of the exercises or want some more ideas, you can keep reading or consult the answers.

SIDEBAR **Parser combinators vs. parser generators**

You might be familiar with *parser generator* libraries like Yacc or similar libraries in other languages (for instance, ANTLR in Java). These libraries *generate* code for a parser based on a specification of the grammar. This approach works fine and can be quite efficient, but comes with all the usual problems of code generation—they produce as as their output a monolithic chunk of code that is difficult to debug. It's also quite difficult to reuse fragments of logic since we cannot introduce new combinators or helper functions to abstract over common patterns in our parsers.

In a parser combinator library, parsers are ordinary values that can be created and manipulated in a first-class way within the language. Reusing parsing logic is trivial (we simply introduce a new combinator), and we don't have to delegate to any sort of separate external tool.

9.2 Designing an algebra, first

Recall that we defined an algebra to mean a collection of functions operating over some data type(s), *along with a set of laws* specifying relationships between these functions. In past chapters, we moved rather fluidly between inventing functions in our algebra, refining the set of functions, and tweaking our data type representations. Laws were somewhat of an afterthought—we worked out the laws only after we had a representation and an API fleshed out. There's absolutely nothing wrong with this style of design[1], but here we are going to take a different approach. We will *start* with the algebra (including its laws) and decide on a representation later. This is an approach we'll call *algebraic design*. This approach can be used for any design problem but works particularly well for parsing, because it's easy to imagine what combinators are required to be able to parse different inputs.[2] Overall, you might find this approach very natural, or you might find it extremely disconcerting to do so much work without committing to any concrete representations. Either way, try to keep an open mind.

Footnote 1 Hints and standalone answers

Footnote 2 Full source files

There are a lot of different parsing libraries.[3] Ours will be designed for expressiveness (we'd like to be able to parse arbitrary grammars), speed, and good error reporting. This last point is important—if there are parse errors, we want to be able to indicate exactly where the error is and accurately indicate its cause.

Error reporting is often somewhat of a second-class citizen in parsing libraries, but we are going to make sure we give careful thought to it.

Footnote 3 Including a parser combinator library in Scala's standard library. As in the previous chapter, we are deriving our own library from first principles partially for pedagogical purposes, and to further encourage the idea that no library is authoritative. Scala's parser combinators don't really satisfy our goals of providing speed and good error reporting (see the chapter notes for some additional discussion).

OK, let's begin. For simplicity and for speed, our library will create parsers that operate on strings as input.[4] We need to pick some parsing tasks to help us discover a good algebra for our parsers. What should we look at first? Something practical like parsing an email address, JSON, or HTML? No! These can come later. For now we are content to focus on a pure, simple domain of parsing various combinations of repeated letters and jibberish words like `"abracadabra"` and `"abba"`. As silly as this sounds, we've seen before how simple examples like this help us ignore extraneous details and focus on the essence of the problem.

Footnote 4 This is certainly a simplifying design choice. We can make the parsing library more generic, at some cost. See the chapter notes for more discussion.

So let's start with the simplest of parsers, one that recognizes the single character input `"a"`. As we've done in past chapters, we can just *invent* a combinator for the task, `char`:

```
def char(c: Char): Parser[Char]
```

What have we done here? We have conjured up a type, `Parser`, which is parameterized on a single parameter indicating the *result type* of the `Parser`. That is, running a parser should not simply yield a yes/no response to the input—if it succeeds, we want to get back a *result* of some useful type, and if it fails we expect *information about the failure*. The `char('a')` parser will succeed only if the input is the string `"a"` and we have chosen (somewhat arbitrarily) to have it return that same character `'a'` as its result.

This talk of "running a parser" makes it clear our algebra needs to be extended somehow to support that. Let's invent another function for it:

```
def run[A](p: Parser[A])(input: String): Either[ParseError,A]
```

Wait a minute, what is `ParseError`? It's another type we've just conjured

into existence! At this point, we don't care about the representation of `ParseError` or `Parser` for that matter. We are in the process of specifying an *interface* that happens to make use of two types whose representation or implementation details we choose to remain ignorant of for now. Let's make this explicit with a `trait`:

```
trait Parsers[ParseError, Parser[_]] {                    (1)

  def run[A](p: Parser[A])(input: String): Either[ParseError,A]
  def char(c: Char): Parser[Char]
}
```

(1) A type constructor type argument

What's with the funny `Parser[_]` type argument? It's not too important for right now, but that is Scala's syntax for a type parameter that is itself a type constructor.[5] Just like making `ParseError` a type argument lets the `Parsers` interface work for multiple representations of `ParseError`, making `Parser[_]` a type parameter means the interface works for multiple representations of `Parser`, which itself can be applied to one type argument.[6] This code will compile as is, without us having to pick a representation for `ParseError` or `Parser`, and you can continue placing additional combinators in the body of this trait.

Footnote 5 We will say much more about this in the next chapter. We can indicate that the `Parser[_]` type parameter should be covariant in its argument with the syntax `Parser[+_]`.

Footnote 6 We will say much more about this in the next chapter. We can indicate that the `Parser[_]` type parameter should be covariant in its argument with the syntax `Parser[+_]`.

Let's continue. We can recognize the single character `'a'`, but what if we want to recognize the string `"abracadabra"`? We don't have a way of recognizing entire strings right now, so let's add that:

```
def string(s: String): Parser[String]
```

What if we want to recognize either the string `"abra"` *or* the string `"cadabra"`? We could certainly add a very specialized combinator for it:

```
def orString(s1: String, s2: String): Parser[String]
```

But choosing between two parsers seems like something that would be more generally useful, regardless of their result type, so let's go ahead and make this polymorphic:

```
def or[A](s1: Parser[A], s2: Parser[A]): Parser[A]
```

Incidentally, we can give this nice infix syntax like `s1 | s2` or alternately `s1 or s2`, using implicits like we did in chapter 7:

Listing 9.1 Adding infix syntax to parsers

```
trait Parsers[ParseError, Parser[+_]] {
  ...
  def or[A](s1: Parser[A], s2: Parser[A]): Parser[A]
  implicit def string(s: String): Parser[String]
  implicit def operators[A](p: Parser[A]) = ParserOps[A](p)
  implicit def asStringParser[A](a: A)(implicit f: A => Parser[String]):
    ParserOps[String] = ParserOps(f(a))

  case class ParserOps[A](p: Parser[A]) {
        def |[B>:A](p2: Parser[B]): Parser[B] = self.or(p,p2) (1)
    def or[B>:A](p2: => Parser[B]): Parser[B] = self.or(p,p2)
  }
}
```

(1) use self to explicitly disambiguate reference to the or method on the trait

We have also made `string` an implicit conversion and added another implicit `asStringParser`. With these two functions, Scala will automatically promote a `String` to a `Parser`, and we get infix operators for any type, `A` that can be converted to a `Parser[String]`. So given a `val P: Parsers`, we can then `import P._` to let us write expressions like `"abra" or "cadabra"` or `"a" | "bbb"` to create parsers. This will work for *all* implementations of `Parsers`.[7] Other binary operators or methods can be added to the body of `ParserOps`. We are going to follow the discipline of keeping the primary definition directly in `Parsers` and delegating in `ParserOps` to this primary definition. See the code for this chapter for more examples. We'll be using the `a | b` syntax liberally throughout the rest of this chapter to mean `or(a,b)`.

Footnote 7 See the appendix *Scalaz, implicits, and large library organization* for more discussion of these issues.

We can now recognize various strings, but we don't have any way of talking about repetition. For instance, how would we recognize three repetitions of our `"abra" | "cadabra"` parser? Once again, let's add a combinator for it:[8]

Footnote 8 This should remind you of a very similar function we wrote in the previous chapter.

```
def listOfN[A](n: Int, p: Parser[A]): Parser[List[A]]
```

We made `listOfN` parametric in the choice of `A`, since it doesn't seem like it should care whether we have a `Parser[String]`, a `Parser[Char]`, or some other type of parser.

At this point, we have just been collecting up required combinators, but we haven't tried to refine our algebra into a minimal set of primitives, and we haven't talked about laws at all. We are going to start doing this next, but rather than give away the "answer", we are going to ask you to examine a few more simple use cases yourself and try to design a minimal algebra with associated laws. This should be a challenging exercise, but enjoy struggling with it and see what you can come up with.

Here are additional parsing tasks to consider, along with some guiding questions:

- A `Parser[Int]` that recognizes zero or more `'a'` characters, and whose result value is the number of `'a'` characters it has seen. For instance, given `"aa"`, the parser results in `2`, given `""` or `"b123"` (a string not starting with `'a'`), it results in `0`, and so on.
- A `Parser[Int]` that recognizes *one* or more `'a'` characters, and whose result value is the number of `'a'` characters it has seen. (Is this defined somehow in terms of the same combinators as the parser for `'a'` repeated zero or more times?) The parser should fail when given a string without a starting `'a'`. How would you like to handle error reporting in this case? Could the API support giving an explicit message like `"Expected one or more 'a'"` in the case of failure?
- A parser that recognizes zero or more `'a'`, followed by one or more `'b'`, and which results in the pair of counts of characters seen. For instance, given `"bbb"`, we get `(0,3)`, given `"aaaab"`, we get `(4,1)`, and so on.

And additional considerations:

- If we are trying to parse a sequence of zero or more `"a"` and are only interested in the number of characters seen, it seems inefficient to have to build up, say, a `List[Char]` only to throw it away and extract the length. Could something be done about this?
- Are the various forms of repetition primitive in your algebra, or could they be defined in terms of something simpler?
- We introduced a type `ParseError` earlier, but so far we haven't chosen any functions for

the API of `ParseError` and our algebra doesn't have any ways of letting the programmer control what errors are reported. This seems like a limitation given that we'd like meaningful error messages from our parsers. Can you do something about it?

- Does `a | b` mean the same thing as `b | a`? This is a choice you get to make. What are the consequences if the answer is yes? What about if the answer is no?
- Does `a | (b | c)` mean the same thing as `(a | b) | c`? If yes, is this a primitive law for your algebra, or is it implied by something simpler?
- Try to come up with a set of laws to specify your algebra. You don't necessarily need the laws to be complete, just write down some laws that you expect should hold for any `Parsers` implementation.

Spend some time coming up with combinators and possible laws based on this guidance. When you feel stuck or at a good stopping point, then continue reading to the next section, which walks through a possible design.

SIDEBAR **The advantages of algebraic design**

When you design the algebra of a library first, representations for the data types of the algebra don't matter as much. As long as they support the required laws and functions, you do not even need to make your representations public. This, as we'll see later, makes it easy for primitive combinators to use cheap tricks internally that might otherwise break referential transparency.

There is a powerful idea here, namely, that a type is given meaning based on its relationship to other types (which are specified by the set of functions and their laws), rather than its internal representation.[9] This is a viewpoint often associated with category theory, a branch of mathematics we've mentioned before. See the chapter notes for more on this connection if you're interested.

Footnote 9 This sort of viewpoint might also be associated with object-oriented design, although OO has not traditionally placed much emphasis on algebraic laws. Furthermore, a big reason for encapsulation in OO is that objects often have some mutable state and making this public would allow client code to violate invariants, a concern that is not as relevant in FP.

9.2.1 A possible algebra

We are going to walk through discovering a set of combinators for the parsing tasks mentioned above. If you worked through this design task yourself, you will likely have taken a different path and may have ended up with a different set of combinators, which is absolutely fine.

First, let's consider the parser that recognizes zero or more repetitions of the character `'a'`, and returns the number of characters it has seen. We can start by adding a primitive combinator for it, let's call it `many`:

```
def many[A](p: Parser[A]): Parser[List[A]]
```

This isn't quite right, though—we need a `Parser[Int]` that counts the number of elements. We could change the `many` combinator to return a `Parser[Int]`, but that feels a little too specific—undoubtedly there will be occasions where we do care about more than just the list length. Better to introduce another combinator that should be familiar by now, `map`:

```
def map[A,B](a: Parser[A])(f: A => B): Parser[B]
```

We can now define our parser as `map(many(char('a')))(_.length)`. And let's go ahead and add `map` and `many` as methods in `ParserOps`, so we can write this as: `char('a').many.map(_.length)`.

We have a strong expectation for the behavior of `map`—it should merely transform the result value if the `Parser` was successful. No additional input characters should be examined by a `map`, a failing parser cannot become a successful one via a `map` and vice versa, and in general, we expect `map` to be *structure preserving* much like we required for `Par` and `Gen`. Let's go ahead and formalize this by stipulating the now-familiar law:

```
map(p)(id) == p
```

How should we document this law that we've just added? We could put it in a documentation comment, but in the preceding chapter we developed a way to make our laws *executable*. Let's make use of that library here:

```
def equal[A](p1: Parser[A], p2: Parser[A])(in: Gen[String]): Prop =
  forAll(in)(s => run(p1)(s) == run(p2)(s))

def mapLaw[A](p: Parser[A])(in: Gen[String]): Prop =
  equal(p, p.map(a => a))(in)
```

```
import fpinscala.testing._

trait Parsers {
  ...
  object Laws {
    def equal[A](p1: Parser[A], p2: Parser[A])(in: Gen[String]): Prop =
      forAll(in)(s => run(p1)(s) == run(p2)(s))

    def mapLaw[A](p: Parser[A])(in: Gen[String]): Prop =
      equal(p, p.map(a => a))(in)
  }
}
```

This will come in handy later when we go to test that our implementation of `Parsers` behaves as we expect. As we discuss other laws, you are encouraged to write them out as actual properties inside the `Laws` object.[10]

Footnote 10 Again, see the chapter code for more examples. In the interest of keeping this chapter shorter, we won't be giving `Prop` implementations of all the laws, but that doesn't mean you shouldn't try writing them out yourself!

Incidentally, now that we have `map`, we can actually implement `char` in terms of `string`:

```
def char(c: Char): Parser[Char] =
  string(c.toString) map (_.charAt(0))
```

And similarly, there's another combinator, `succeed`, that can be defined in terms of `string` and `map`:

```
def succeed[A](a: A): Parser[A] =
  string("") map (_ => a)
```

This parser always succeeds with the value `a`, regardless of the input string (since `string("")` will always succeed, even if the input is empty). Does this combinator seem familiar to you? We can specify its behavior with a law:

```
run(succeed(a))(s) == Right(a)
```

SLICING AND NONEMPTY REPETITION

The combination of `many` and `map` certainly lets us express the parsing task of counting the number of `'a'` characters, but it seems inefficient to be constructing a `List[Char]` only to discard its values and extract its length. It would be nice if we could run a `Parser` purely to see what portion of the input string it examines. Let's conjure up a combinator for that purpose:

```
def slice[A](p: Parser[A]): Parser[String]
```

We call it `slice` since it returns the portion of the input string examined by the parser if successful. As an example, `run(slice(or('a','b').many))("aaba")` results in `Right("aaba")`—that is, we ignore the list accumulated by `many` and simply return the portion of the input string matched by the parser.

With `slice`, our parser can now be written as `char('a').many.slice.map(_.length)` (again, assuming we add an alias for `slice` to `ParserOps`). The `_.length` function here is now referencing the `length` method on `String`, rather than the `length` method on `List`.

Let's consider the next use case. What if we want to recognize *one* or more `'a'` characters? First, we introduce a new combinator for it, `many1`:

```
def many1[A](p: Parser[A]): Parser[List[A]]
```

It feels like `many1` should not have to be primitive, but should be defined somehow in terms of `many`. Really, `many1(p)` is just `p` *followed by* `many(p)`. So it seems we need some way of running one parser, followed by another, assuming the first is successful. Let's add that:

```
def product[A,B](p: Parser[A], p2: Parser[B]): Parser[(A,B)]
```

We can add `**` and `product` as methods on `ParserOps`, where `a ** b` and `a product b` both delegate to `product(a,b)`.

EXERCISE 1: Using `product`, implement the now-familiar combinator `map2`

and then use this to implement `many1` in terms of `many`. (Note that we could have chosen to make `map2` primitive and defined `product` in terms of `map2` as we've done in the previous chapters)

```
def map2[A,B,C](p: Parser[A], p2: Parser[B])(f: (A,B) => C): Parser[C]
```

```
def map2[A,B,C](p: Parser[A], p2: Parser[B])(
                f: (A,B) => C): Parser[C] =
  map(product(p, p2))(f.tupled)
def many1[A](p: Parser[A]): Parser[List[A]] =
  map2(p, many(p))(_ :: _)
```

With `many1`, we can now implement the parser for zero or more `'a'` followed by one or more `'b'` as:

```
char('a').many.slice.map(_.length) **
char('b').many1.slice.map(_.length)
```

EXERCISE 2 (hard): Try coming up with properties to specify the behavior of `product`.

Now that we have `map2`, is `many` really primitive? Let's think about what `many(p)` will do. It tries running `p`, *followed by* another `p`, and another, and so on until attempting to parse `p` fails, at which point the parser returns the empty `List` and combines the results of the successful parses.

EXERCISE 3 (hard): Before continuing, see if you can define `many` in terms of `or`, `map2`, and `succeed`.

EXERCISE 4 (hard, optional): Using `map2` and `succeed`, implement the `listOfN` combinator from earlier.

Now let's try writing `many`:

```
def many[A](p: Parser[A]): Parser[List[A]] =
  map2(p, many(p))(_ :: _) or succeed(List())
```

This looks pretty, but there's a problem with it. Can you spot what it is? We are calling `many` recursively in the second argument to `map2`, which is *strict* in

evaluating its second argument. Consider a simplified program trace of the evaluation of `many(p)` for some parser `p`. We are only showing the expansion of the left side of the `or` here:

```
many(p)
map2(p, many(p))(_ :: _)
map2(p, map2(p, many(p))(_ :: _))(_ :: _)
map2(p, map2(p, map2(p, many(p))(_ :: _))(_ :: _))(_ :: _)
...
```

Because a call to `map2` always evaluates its second argument, our `many` function will never terminate! That's no good. Let's go ahead and make `product` and `map2` non-strict in their second argument:

```
def product[A,B](p: Parser[A], p2: => Parser[B]): Parser[(A,B)]
```

```
def map2[A,B,C](p: Parser[A], p2: => Parser[B])(
                f: (A,B) => C): Parser[C] =
  product(p, p2) map (f.tupled)
```

EXERCISE 5 (optional): We could also deal with non-strictness with a separate combinator like we did in chapter 7. Try this here and make the necessary changes to your existing combinators. What do you think of that approach in this instance?

Now our implementation of `many` should work fine. Conceptually, `product` feels like it should have been non-strict in its second argument anyway, since if the first `Parser` fails, the second will not even be consulted.

Now that we're considering whether combinators should be non-strict, let's revisit `or`:

```
def or[A](p1: Parser[A], p2: Parser[A]): Parser[A]
```

We will assume that `or` is left-biased, meaning it tries running `p1` on the input then tries `p2` only if `p1` fails.[11] In this case, we ought to make it non-strict in its second argument, which may never even be consulted:

Footnote 11 This is a design choice. You may wish to think about the consequences of having a version of `or` that always runs both `p1` and `p2`.

```
def or[A](p1: Parser[A], p2: => Parser[A]): Parser[A]
```

EXERCISE 6: Given this choice of meaning for `or`, is it associative? That is, should `a or (b or c)` equal `(a or b) or c` for all choices of `a`, `b`, and `c`? We'll come back to this question when refining the laws for our algebra.

9.2.2 Handling context-sensitivity

Let's take a step back and look at the primitives we have so far:

- `string(s)`: Recognize and return a single `String`
- `slice(p)`: Return the portion of input inspected by `p` if successful
- `succeed(a)`: Always succeed with the value `a`
- `map(p)(f)`: Apply the function `f` to the result of `p`, if successful
- `product(p1,p2)`: Sequence two parsers, running `p1`, then `p2` and return the pair of their results if both succeed
- `or(p1,p2)`: Choose between two parsers, first attempting `p1`, then `p2` if `p1` fails

Using these primitives, we can express repetition and nonempty repetition (`many`, `listOfN` and `many1`) as well as combinators like `char` and `map2`. What else could we express?

Surprisingly, these primitives are sufficient for parsing *any* context-free grammar, including JSON! We'll get to writing that JSON parser soon.

These combinators are not without limitations, though. Suppose we want to parse a single digit, like `'4'`, followed by *that many* `'a'` characters (this sort of problem should feel familiar). Examples of valid input are `"0"`, `"1a"`, `"2aa"`, `"4aaaa"`, and so on. This is an example of a context-sensitive grammar. It can't be expressed with `product` because our choice of the second parser *depends on* the result of the first (the second parser depends on its context). We want to run the first parser, then do a `listOfN` using the number extracted from the first parser's result. Can you see why `product` cannot express this?

EXERCISE 7: Before continuing, think of a primitive that makes it possible to parse this grammar, and revisit your existing primitives as needed.

This progression might feel familiar to you. In past chapters, we encountered similar expressiveness limitations and dealt with it by introducing a new primitive, `flatMap`. Let's introduce that here (and we'll add an alias to `ParserOps` so we can writer parsers using `for`-comprehensions):

```
def flatMap[A,B](p: Parser[A])(f: A => Parser[B]): Parser[B]
```

Can you see how this signature implies an ability to sequence parsers?

EXERCISE 8: Using `flatMap` and any other combinators, write the context-sensitive parser we could not express above. To parse the digits, you can make use of a new primitive, `regex`, which promotes a regular expression to a `Parser`:[12] In Scala, given a string, `s`, it can be promoted to a `Regex` object (which has methods for performing matching) using `s.r`, for instance: `"[a-zA-Z_][a-zA-Z0-9_]*".r`

Footnote 12 In theory this isn't necessary, we could write out `"0" | "1" | ... "9"` to recognize a single digit, but this isn't likely to be very efficient.

```
implicit def regex(r: Regex): Parser[String]
```

EXERCISE 9: Implement `product` and `map2` in terms of `flatMap`.

EXERCISE 10: `map` is no longer primitive. Express it in terms of `flatMap` and/or other combinators.

So it appears we have a new primitive, `flatMap`, which enables context-sensitive parsing and lets us implement `map` and `map2`. This is not the first time `flatMap` has made an appearance, but up until now we have not really tried to pin down any laws for it. We'll be working through that in this chapter in a later section.

9.2.3 Error reporting

So far we have not discussed error reporting at all. We've been focused exclusively on discovering a set of primitives that let us express parsers for different grammars. But besides just being able to parse a grammar, we want to be able to determine how the parser should respond when given unexpected input.

Even without knowing what an implementation of `Parsers` will look like, we can reason very abstractly about *what information is being specified* by a set of combinators. None of the combinators we have introduced so far say anything about *what error message* should be reported in the event of failure or what other information a `ParseError` should contain. Our existing combinators only specify what the grammar is and what to do with the result if successful. If we were to declare ourselves done and move to implementation at this point, we would have to make some arbitrary decisions about error reporting and error messages that are unlikely to be universally appropriate.

EXERCISE 11 (hard): If you have not already done so, spend some time discovering a nice set of combinators for expressing what errors get reported by a `Parser`. For each combinator, try to come up with laws specifying what its behavior should be. This is a very open-ended design task. Here are some guiding questions:

- Given the parser `"abra" ** " ".many ** "cadabra"`, what sort of error would you like to report given the input `"abra cAdabra"` (note the capital `'A'`)? Only something like `Expected 'a'`? Or `Expected "cadabra"`? What if we wanted to choose a different error message like `"Expected The Magic Word (TM)"`?
- Given `a or b`, if `a` fails on the input, do we *always* want to run `b`, or are the cases where we might not want to? If there are cases, can you think of additional combinators that would allow the programmer to specify when `or` should consider the second parser?
- Given `a or b`, if `a` and `b` both fail on the input, might we want to support reporting both errors? And do we *always* want to report both errors, or do we want to give the programmer a way to specify which of the two errors are reported?
- How do you want to handle reporting the *location* of errors?

Once you are satisfied with your design, you can continue reading. The next section works through a possible design in detail.

SIDEBAR **Combinators specify information**

In a typical library design scenario, where one has a concrete representation at least in mind, we often think of functions in terms of how they will affect this representation. By starting with the algebra first, we are forced to think differently—we must think of functions in terms of *what information they specify* to a possible implementation. The signatures specify what information is given to the implementation, and the implementation is free to use this information however it wants as long as it respects the specified laws.

A POSSIBLE DESIGN

Now that you've spent some time coming up with some good error reporting combinators, we are now going to work through one possible design. Again, you may have arrived at a different design, which is absolutely fine. This is just another opportunity to see a worked design process.

We are going to progressively introduce our error reporting combinators. To start, let's introduce an obvious one. None of the primitives so far let us assign an error message to a parser. We can introduce a primitive combinator for this, `label`:

```
def label[A](msg: String)(p: Parser[A]): Parser[A]
```

The intended meaning of `label` is that if `p` fails, its `ParseError` will "somehow incorporate" `msg`. What does this mean exactly? Well, we could just assume `type ParseError = String` and that the returned `ParseError` will *equal* the label. The problem with this is that we'd like our parse error to also tell us *where* the problem occurred. Let's tentatively add this to our algebra:

```
case class Location(input: String, offset: Int = 0) {
  lazy val line = input.slice(0,offset+1).count(_ == '\n') + 1
  lazy val col = input.slice(0,offset+1).reverse.indexOf('\n')
}
```

```
def errorLocation(e: ParseError): Location
def errorMessage(e: ParseError): String
```

We've picked a concrete representation for `Location` here that includes the full input, an offset into this input, and the line and column numbers (which can be computed, lazily, derived from the full input and offset). We can now say more precisely what we expect from `label`—in the event of failure with `Left(e)`, `errorMessage(e)` will equal the message set by `label`. This can be specified with a `Prop` if we like:

```
def labelLaw[A](p: Parser[A], inputs: SGen[String]): Prop =
  forAll(inputs ** Gen.string) { case (input, msg) =>
    run(label(msg)(p))(input) match {
      case Left(e) => errorMessage(e) == msg
```

```
      case _ => true
    }
  }
```

What about the `Location`? We would like for this to be filled in by the `Parsers` implementation with the location where the error occurred. This notion is still a bit fuzzy—if we have `a or b` and both parsers fail on the input, which location is reported, and which label(s)? We will discuss this in the next section.

ERROR NESTING

Is the `label` combinator sufficient for all our error reporting needs? Not quite. Let's look at an example:

```
val p = label("first magic word")("abra") **
        " ".many **
        label("second magic word")("cadabra")
```

❶

❶ Skip whitespace

What sort of `ParseError` would we like to get back from `run(p)("abra cAdabra")`? (Note the capital `A` in `cAdabra`) The immediate cause is that capital `'A'` instead of the expected lowercase `'a'`. That error will have a location, and it might be nice to report it somehow. But reporting *only* that low-level error would not be very informative, especially if this were part of a large grammar and the parser were being run on a larger input. We have some more context that would be useful to know—the immediate error occurred while in the `Parser` labeled `"second magic word"`. This is certainly helpful information. Ideally, we could be told that while parsing `"second magic word"`, we encountered an error due to an unexpected capital `'A'`, which pinpoints the error and gives us the context needed to understand it. And we can imagine that perhaps the top-level parser (`p` in this case) might be able to provide an even higher-level description of what the parser was doing when it failed (`"parsing magic spell"`, say), which could also be informative.

So, it seems wrong to assume that one level of error reporting will always be sufficient. Let's therefore provide a way to *nest* labels:

```
def scope[A](msg: String)(p: Parser[A]): Parser[A]
```

Unlike `label`, `scope` does not throw away the label(s) attached to `p`—it merely adds additional information in the event that `p` fails. Let's specify what this means exactly. First, we modify the functions that pull information out of a `ParseError`. Rather than containing just a single `Location` and `String` message, we should get a `List[(Location,String)]`:

```
def errorStack(e: ParseError): List[(Location,String)]
```

This is a stack of error messages indicating what the `Parser` was doing when it failed. We can now specify what `scope` does—if `run(p)(s)` is `Left(e)`, then `run(scope(msg)(p))` is `Left(e2)`, where `errorStack(e2)` will have at the top of the stack the message `msg`, followed by any messages added by `p` itself.

We can take this one step further. A stack does not fully capture what the parser was doing at the time it failed. Consider the parser `scope("abc")(a or b or c)`. If `a`, `b`, and `c` all fail, which error goes at the top of the stack? We could adopt some global convention, like always reporting the last parser's error (in this case `c`) or perhaps reporting whichever parser examined more of the input, but it might be nice to allow the implementation to return all the errors if it chooses:

```
case class ParseError(stack: List[(Location,String)] = List(),
                      otherFailures: List[ParseError] = List())
```

This is a somewhat unusual data structure—we have `stack`, the current stack, but also a list of other failures (`otherFailures`) that occurred previously in a chain of `or` combinators.[13] This is potentially a lot of information, capturing not only the current path in the grammar, but also all the previous failing paths. We can write helper functions later to make constructing and manipulating `ParseError` values more convenient and to deal with formatting them nicely for human consumption. For now, our concern is just making sure it contains all the potentially relevant information for error reporting, and it seems like `ParseError` will be more than sufficient. Let's go ahead and pick this as our concrete representation. We can remove the type parameter from `Parsers`:

Footnote 13 We could also represent `ParseError` as a trie in which shared prefixes of the error stack are not duplicated, at a cost of having more expensive inserts. It is easier to recover this sharing information during formatting of errors, which happens only once.

```
trait Parsers[Parser[+_]] {
  def run[A](p: Parser[A])(input: String): Either[ParseError,A]
  ...
}
```

Now we are giving the `Parsers` implementation all the information it needs to construct nice, hierarchical errors if it chooses. As a user of `Parsers`, we will judiciously sprinkle our grammar with `label` and `scope` calls which the `Parsers` implementation can use when constructing parse errors. Note that it would be perfectly reasonable for implementations of `Parsers` to not use the full power of `ParseError` and retain only basic information about the cause and location of errors.[14]

Footnote 14 We may want to explicitly acknowledge this by relaxing the laws specified for `Parsers` implementations, or making certain laws optional.

CONTROLLING BRANCHING AND BACKTRACKING

There is one last concern regarding error reporting that we need to address. As we just discussed, when we have an error that occurs inside an `or` combinator, we need some way of determining which error(s) to report. We don't want to *only* have a global convention for this, we sometimes want to allow the programmer to control this choice. Let's look at a more concrete motivating example:

```
val spaces = " ".many
val p1 = scope("magic spell") {
  "abra" ** spaces ** "cadabra"
}
val p2 = scope("jibberish") {
  "abba" ** spaces ** "babba"
}
val p = p1 or p2
```

What `ParseError` would we like to get back from `run(p)("abra cAdabra")`? (Again, note the capital `A` in `cAdabra`.) Both branches of the `or` will produce errors on the input. The `"jibberish"`-labeled parser will report an error due to expecting the first word to be `"abba"`, and the `"magic spell"`

parser will report an error due to the accidental capitalization in `"cAdabra"`. Which of these errors do we want to report back to the user?

In this instance, we happen to want the `"magic spell"` parse error—after successfully parsing the `"abra"` word, we are *committed* to the `"magic spell"` branch of the `or`, which means if we encounter a parse error we do not examine the next branch of the `or`. In other instances, we may want to allow the parser to consider the next branch of the `or`.

So, it appears we need a primitive for letting the programmer indicate when to commit to a particular parsing branch. Recall that we loosely assigned `p1 or p2` to mean *try running p1 on the input, then try running p2 on the same input if p1 fails*. We can change its meaning to: *try running p1 on the input, and if it fails in an uncommitted state, try running p2 on the same input, otherwise report the failure*. This is useful for more than just providing good error messages—it also improves efficiency by letting the implementation avoid lots of possible parsing branches.

One common solution to this problem is to have all parsers *commit by default* if they examine at least one character to produce a result.[15] We then introduce a combinator, `attempt`, which delays committing to a parse:

Footnote 15 Though see the chapter notes for more discussion of this.

```
def attempt[A](p: Parser[A]): Parser[A]
```

It should satisfy something like:[16]

Footnote 16 This is not quite an equality. Even though we want to run `p2` if the attempted parser fails, we may want `p2` to somehow incorporate the errors from both branches if it fails.

```
attempt(p flatMap (_ => fail)) or p2 == p2
```

Where `fail` is a parser that always fails (we could introduce this as a primitive combinator if we like). That is, even if `p` fails midway through examining the input, `attempt` reverts the commit to that parse and allows `p2` to be run. The `attempt` combinator can be used whenever there is ambiguity in the grammar and multiple tokens may have to be examined before the ambiguity can be resolved and parsing can commit to a single branch. As an example, we might write:

```
(attempt("abra" ** spaces ** "abra") ** "cadabra") or (
 "abra" ** spaces "cadabra!")
```

Suppose this parser is run on `"abra cadabra!"`—after parsing the first `"abra"`, we don't know whether to expect another `"abra"` (the first branch) or `"cadabra!"` (the second branch). By wrapping an `attempt` around `"abra" ** spaces ** "abra"`, we allow the second branch to be considered up until we have finished parsing the second `"abra"`, at which point we commit to that branch.

EXERCISE 12: Can you think of any other primitives that might be useful for letting the programmer specify what error(s) in an `or` chain get reported?

Note that we still haven't written an implementation of our algebra! But this exercise has been more about making sure our combinators provide a way for users of our library to convey the right information to the implementation. It is up to the implementation to figure out how to use this information in a way that satisfies the laws we've stipulated.

9.3 Writing a JSON parser

Let's write that JSON parser now, shall we? We don't have an implementation of our algebra yet, but that doesn't actually matter! Our JSON parser doesn't need to know the internal details of how parsers are represented. It will be constructing a parser purely using the set of primitives we've defined and any derived combinators. Of course, we will not actually be able to run our parser until we have a concrete implementation of the `Parsers` interface.

Recall that we have built up the following set of primitives:

- `string(s)`: Recognizes and returns a single `String`
- `slice(p)`: Returns the portion of input inspected by `p` if successful
- `succeed(a)`: Always succeed with the value `a`
- `label(e)(p)`: In the event of failure, replace the assigned message with `e`
- `scope(e)(p)`: In the event of failure, add `e` to the error stack returned by `p`
- `seq(p)(f)`: Run a parser, then use its result to select a second parser to run in sequence
- `attempt(p)`: Delays committing to `p` until after it succeeds
- `or(p1,p2)`: Chooses between two parsers, first attempting `p1`, then `p2` if `p1` fails in an uncommitted state on the input

We've used these primitives to define a number of combinators like `map`, `map2`, `flatMap`, `many`, and `many1`.

Again, we will be asking *you* to drive the process of writing this parser. After a

brief introduction to JSON and the data type we'll use for the parse result, it's up to you to go off on your own to write the parser.

9.3.1 The JSON format

If you aren't already familiar with the JSON format, you may want to read the Wikipedia page description and the grammar specification. Here's an example JSON document:

```
{
  "Company name" : "Microsoft Corporation",
  "Ticker"  : "MSFT",
  "Active"  : true,
  "Price"   : 30.66,
  "Shares outstanding" : 8.38e9,
  "Related companies" :
    [ "HPQ", "IBM", "YHOO", "DELL", "GOOG" ]
}
```

A *value* in JSON can be one of several types. An *object* in JSON is a `{}` -wrapped, comma separated sequence of key-value pairs. The keys must be strings like `"Ticker"` or `"Price"`, and the values can be either another object, an *array* like `["HPQ", "IBM" ... ]` which contains further values, or a *literal* like `"MSFT"`, `true`, `null`, or `30.66`.

We are going to write a rather dumb parser that simply parses a syntax tree from the document without doing any further processing.[17] We'll need a representation for a parsed JSON document. Let's introduce a data type for this:

Footnote 17 See the chapter notes for discussion of alternate approaches.

```
trait JSON
object JSON {
  case object JNull extends JSON
  case class JNumber(get: Double) extends JSON
  case class JString(get: String) extends JSON
  case class JBool(get: Boolean) extends JSON
  case class JArray(get: IndexedSeq[JSON]) extends JSON
  case class JObject(get: Map[String, JSON]) extends JSON
}
```

The task is two write a `Parser[JSON]`. We want this to work for whatever the chosen `Parsers` implementation. To allow this, we can place the implementation in a regular function that takes a `Parsers` value as its argument:

18

Footnote 18 Unfortunately, we have to mention the covariance of the `Parser` type constructor again, rather than inferring the variance from the definition of `Parsers`.

```
def jsonParser[Parser[+_]](P: Parsers[Parser]): Parser[JSON] = {
  import P._                                                    (1)
  val spaces = char(' ').many.slice
  ...
}
```

(1) gives access to all the combinators

EXERCISE 13 (hard): At this point, *you* are going to take over the process. You will be creating a `Parser[JSON]` from scratch using the primitives we have defined. You don't need to worry (yet) about the representation of `Parser`. As you go, you will undoubtedly discover additional combinators and idioms, notice and factor out common patterns, and so on. Use the skills and knowledge you've been developing throughout this book, and have fun! If you get stuck, you can always consult the answers.

Here is some minimal guidance:

- Any general purpose combinators you discover can be added to the `Parsers trait` directly.
- You will probably want to introduce combinators that make it easier to parse the tokens of the JSON format (like string literals and numbers). For this you could use the `regex` primitive we introduced earlier. You could also add a few primitives like `letter`, `digit`, `whitespace`, and so on, for building up your token parsers.

Consult the hints if you'd like a bit more guidance.

EXERCISE 14 (hard): You are now (hopefully) finished with your `Parser[JSON]`. Aren't you curious to try testing it? Now it's finally time to come up with a representation for `Parser` and implement the `Parsers` interface using this representation. This is a very open-ended design task, but the algebra we have designed places very strong constraints on possible representations. You should be able to come up with a simple, purely functional representation of `Parser` that can be used to implement the `Parsers` interface.[19]

Footnote 19 Note that if you try running your JSON parser once you have an implementation of `Parsers`, you may get a stack overflow error. See the end of the next section for a discussion of this.

Your code will likely look something like this:

```
class MyParser[+A](...) { ... }

object MyParsers extends Parsers[MyParser] {
  // implementations of primitives go here
}
```

Replace `MyParser` with whatever data type you use for representing your parsers. When you have something you are satisfied with, get stuck, or want some more ideas, keep reading.

9.4 Refining the laws

In the next section, we will work through an implementation of our `Parsers` interface. Here, we are going to spend some time refining the laws for just two of our combinators, `flatMap` and `succeed`. We've introduced these combinators several times before (though `succeed` was previously called `unit`), for different data types. This is more than a coincidence: besides having similar type signatures, these combinators give rise to laws that are *ubiquitous in programming*, something we will explore further in part 3 of this book when abstracting over the combinator libraries we have written. This section can be safely skipped but we are hoping to pique your curiosity a bit more in preparation for part 3. Don't worry if you don't follow everything here or it feels very abstract—we will be discussing this much more extensively in part 3.

Earlier, as an exercise, we asked the question: is `or` associative? That is, do we expect that `(a or b) or c` is equal to `a or (b or c)`? As designers of the API, we get to choose whatever laws we wish—let's choose to make the answer *yes*. Associativity is a nice property to expect of operations in an algebra. If `or` is associative, it means we do not need to have knowledge of a parser's context (what parser it is grouped with) to understand its behavior.

The associativity of `or` makes us wonder: is there a kind of associativity law for `flatMap`? Yes, there is. The formulation is usually not given directly in terms of `flatMap`. The problem with `flatMap` is the "shape" of the two sides is different:

```
def flatMap[A,B](p: Parser[A])(f: A => Parser[B]): Parser[B]
```

On the left side we have a `Parser[A]`, but on the right side we have a function. The expression `flatMap(a,flatMap(b,c))` would not even be

well-typed. However, there is a combinator, let's call it `seq`, that is equivalent in expressiveness to `flatMap`, which has a function on *both* sides:

```
def seq[U,A,B](f: U => Parser[A])(g: A => Parser[B]): U => Parser[B]
```

This might seem like a rather odd-looking combinator, but it can be used to express `flatMap`, if we substitute `Unit` in for `U`:[20]

Footnote 20 Recall that the type `Unit` has just a single value, written `()`. That is, `val u: Unit = ()`.

```
def flatMap[A,B](p: Parser[A])(f: A => Parser[B]): Parser[B] =
  seq((u: Unit) => p)(f)(())
```

The key insight here was that `Unit => Parser[A]` could be converted to `Parser[A]` and vice versa.

EXERCISE 15: To complete the demonstration that `seq` and `flatMap` are equivalent in expressiveness, define `seq` in terms of `flatMap`.

Now that we have an operation with the same shape on both sides, we can ask whether it should be associative as well. That is, do we expect the following:

```
seq(seq(f, g), h) == seq(f, seq(g, h))
```

EXERCISE 16 (hard): That answer is again, *yes*, but can you see why? Think about this law and write down an explanation in your own words of why it should hold, and what it means for a `Parser`. We will discuss this more in chapter 11.

EXERCISE 17 (hard): Come up with a law to specify the relationship between `seq` and `succeed`.

EXERCISE 18: The `seq` and `succeed` laws specified are quite common and arise in many different guises. Choose a data type, such as `Option` or `List`. Define functions analogous to `seq` and `succeed` for this type and show that the implementations satisfy the same laws. We will discuss this connection further in chapter 11.

EXERCISE 19 (hard, optional): We can define a function, `kor`, analogous to `seq` but using `or` to combine the results of the two functions:

```
def kor[A,B](f: A => Parser[B], g: A => Parser[B]): A => Parser[B] =
```

```
  a => f(a) or g(a)
```

Can you think of any laws to specify the relationship between `seq` and `kor`?

9.5 Implementing the algebra

We are now going to discuss an implementation of `Parsers`. Our parsing algebra supports a lot of features. Rather than jumping right to the final representation of `Parser`, we will build it up gradually by inspecting the primitives of the algebra and reasoning about the information that will be required to support each one.

Let's begin with the `string` combinator:

```
def string(s: String): Parser[A]
```

We know we need to support the function `run`:

```
def run[A](p: Parser[A])(input: String): Either[ParseError,A]
```

As a first guess, we can assume that our `Parser` *is* simply the implementation of the `run` function:

```
type Parser[+A] = String => Either[ParseError,A]
```

We could certainly use this to implement `string`:

```
def string(s: String): Parser[A] =
  (input: String) =>
    if (input.startsWith(s))
      Right(s)
    else
      Left(Location(input).toError("Expected: " + s))
```

The `else` branch has to build up a `ParseError`, which are a little inconvenient to construct right now, so we've introduced a helper function on `Location` for it, `toError`:

```
def toError(msg: String): ParseError =
  ParseError(List((this, msg)))
```

So far so good. We have a representation for `Parser` that at least supports `string`. Let's move on to sequencing of parsers. Unfortunately, to represent a parser like `"abra" ** "cadabra"`, our existing representation is not going to be sufficient. If the parse of `"abra"` is successful, then we want to consider those characters *consumed* and run the `"cadabra"` parser on the remaining characters. So in order to support sequencing, we require a way of letting a `Parser` indicate how many characters it consumed. Let's try capturing this:[21]

Footnote 21 Recall that `Location` contains the full input string and an offset into this string.

```
type Parser[+A] = Location => Result[A]

trait Result[+A]
case class Success[+A](get: A, charsConsumed: Int) extends Result[A]
case class Failure(get: ParseError) extends Result[Nothing]
```

We introduced a new type here, `Result`, rather than just using `Either`. In the event of success, we return a value of type `A` as well as the number of characters of input consumed, which can be used by the caller to update the `Location` state.[22]. This type is starting to get at the essence of what a `Parser` is—it is a kind of state action that can fail, similar to what we built in chapter 6.[23] It receives an input state, and if successful returns a value as well enough information to control how the state should be updated.

Footnote 22 Note that returning an `(A,Location)` would give `Parser` the ability to set the `input`, which is granting it too much power.

Footnote 23 The precise relationship between these two types will be further explored in part 3 when we discuss what are called *monad transformers*.

This understanding gives us a way of framing how to build a fleshed out representation supporting all the fancy combinators and laws we have stipulated. We simply consider what each primitive requires that we track in our state type (just a `Location` may not be sufficient), and work through the details of how each combinator transforms this state.

Let's again recall our set of primitives:

- `string(s)`: Recognizes and returns a single `String`
- `slice(p)`: Returns the portion of input inspected by `p` if successful
- `label(e)(p)`: In the event of failure, replace the assigned message with `e`
- `scope(e)(p)`: In the event of failure, add `e` to the error stack returned by `p`

- `flatMap(p)(f)`: Run a parser, then use its result to select a second parser to run in sequence
- `attempt(p)`: Delays committing to `p` until after it succeeds
- `or(p1,p2)`: Chooses between two parsers, first attempting `p1`, then `p2` if `p1` fails in an uncommitted state on the input

EXERCISE 20: Implement `string`, `succeed`, `slice`, and `flatMap` for this initial representation of `Parser`. Notice that `slice` is a bit less efficient than it could be, since it must still construct a value only to discard it. We will return to this later.

Let's look at `scope` next. In the event of failure, we want to push a new message onto the `ParseError` stack. Let's introduce a helper function for this on `ParseError`, we'll call it `push`:[24]

Footnote 24 The `copy` method comes for free with a `case class`. It returns an updated copy of the object with one or more arguments replaced, using the usual default argument mechanism in Scala. If no new value is specified for a field, it will have the same value as in the original object.

```
def push(loc: Location, msg: String): ParseError =
  copy(stack = (loc,msg) :: stack)
```

With this we can implement `scope`:

```
def scope[A](msg: String)(p: Parser[A]): Parser[A] =
  s => p(s).mapError(_.push(s.loc,msg))
```

The function `mapError` is defined on `Result`—it just applies a function to the failing case:

```
def mapError(f: ParseError => ParseError): Result[A] = this match {
  case Failure(e,c) => Failure(f(e),c)
  case _ => this
}
```

Because we push onto the stack after the inner parser has returned, the bottom of the stack will have more detailed messages that occurred later in parsing. (Consider the `ParseError` that will result if `scope(msg1)(a ** scope(msg2)(b))` fails while parsing `b`.)

And we can implement `label` similarly. In the event of failure, we want to

replace the failure message. We can write this again using `mapError`:

```
def label[A](msg: String)(p: Parser[A]): Parser[A] =
  s => p(s).mapError(_.label(msg))
```

We added a helper function to `ParseError`, also called `label`. We will make a design decision that `label` trims the error stack, cutting off more detailed messages from inner scopes, using only the most recent location from the bottom of the stack:

```
def label[A](s: String): ParseError =
  ParseError(latestLoc.map((_,s)).toList,
             otherFailures map (_.label(s)))
```

```
def latest: Option[(Location,String)] =
  stack.lastOption

def latestLoc: Option[Location] =
  latest map (_._1)
```

The `latest` combinator can be used to implement `furthest`, which prunes a `ParseError` to the error that occurred after consuming the most characters. We can use this to implement the `furthest` combinator for parsers:

```
def furthest: ParseError =
  copy(otherFailures = List()) ::
  otherFailures maxBy (_.latest.map(_._1.offset))
```

```
def furthest[A](p: Parser[A]): Parser[A] =
  s => p(s).mapError(_.furthest)
```

EXERCISE 21: Revise your implementation of `string` and `flatMap` to work with the new representation of `Parser`.

Let's now consider `or` and `attempt`. Recall what we specified for the expected behavior of `or`: it should run the first parser, and if it fails *in an uncommitted state*, it should run the second parser on the same input. We said that consuming at least one character should result in a committed parse, and that `attempt(p)` converts committed failures of `p` to uncommitted failures.

We can support the behavior we want by adding one more piece of information to the `Failure` case of `Result`—a `Boolean` value indicating whether the parser failed in a committed state:

```
case class Failure(get: ParseError,
                   isAttempted: Boolean) extends Result[Nothing]
```

The implementation of `attempt` just 'uncommits' any failures that occur. It uses a helper function, `uncommit`, which we can define on `Result`:

```
def attempt[A](p: Parser[A]): Parser[A] =
  s => p(s).uncommit
```

```
def uncommit: Result[A] = this match {
  case Failure(e,true) => Failure(e,false)
  case _ => this
}
```

And the implementation of `or` simply checks the `isCommitted` flag before running the second parser—if the first parser fails in a committed state, then we skip running the second parser. Again we first introduce a helper function on `ParseError` to accumulate a failure into the `otherFailures` list:

```
def addFailure(e: ParseError): ParseError =
  this.copy(otherFailures = e :: this.otherFailures)
```

```
def or[A](p: Parser[A], p2: => Parser[A]): Parser[A] =
  s => p(s) match {
    case r@Failure(e,committed) if committed =>
      p2(s).mapError(_.addFailure(e))
    case r => r                                          (1)
  }
```

(1) committed failure or success skips running p2

What about `flatMap`? The implementation is simple, we just advance the location before calling the second parser. Again we use a helper function,

`advanceBy`, on `Location`. There is one subtlety—if the first parser consumes any characters, we ensure that the second parser is committed, using a helper function, `addCommit` on `ParseError`:

```
def flatMap[A,B](f: Parser[A])(g: A => Parser[B]): Parser[B] =
  s => f(s) match {
    case Success(a,n) => g(a)(s.advanceBy(n)).addCommit(n == 0)
    case f@Failure(_,_) => f
  }
```

`advanceBy` has the obvious implementation:

```
def advanceBy(n: Int): Location =
  copy(offset = offset+n)
```

Likewise, `addCommit`, defined on `ParseError`, is straightforward:

```
def addCommit(isCommitted: Boolean): Result[A] = this match {
  case Failure(e,false) if isCommitted => Failure(e, true)
  case _ => this
}
```

EXERCISE 22: Implement the rest of the primitives, including `run` using this representation of `Parser` and try running your JSON parser on various inputs.[25]

Footnote 25 You will find, unfortunately, that it stack overflows for large inputs (for instance, `[1,2,3,...10000]`). One simple solution to this is to provide a specialized implementation of `many` that avoids using a stack frame for each element of the list being built up. So long as any combinators that do repetition are defined in terms of `many` (which they all can be), this solves the problem. See the answers for discussion of more general approaches.

EXERCISE 23: Come up with a nice way of formatting a `ParseError` for human consumption. There are a lot of choices to make, but a key insight is that we typically want to combine or group labels attached to the same location when presenting the error as a `String` for display.

EXERCISE 24 (hard, optional): The `slice` combinator is still less efficient than it could be. For instance, `many(char('a')).slice` will still build up a `List[Char]` only to discard it. Can you think of a way of modifying the `Parser` representation to make slicing more efficient?

9.6 Conclusion

In this chapter, we introduced an approach to writing combinator libraries called algebraic design, and used it to design a parser combinator library and implement a JSON parser. Along the way, we discovered a number of very similar combinators to previous chapters, which again were related by familiar laws. In part 3, we will finally understand the nature of the connection between these libraries and learn how to abstract over their common structure.

This is the final chapter in part 2. We hope you have come away from these chapters with a basic sense for how functional design can proceed, and more importantly, we hope these chapters have motivated you to try your hand at *designing your own functional libraries*, for whatever domains interest *you*. Functional design is not something reserved only for FP experts—it should be part of the day-to-day work done by functional programmers at all experience levels. Before starting on part 3, we encourage you to venture beyond this book and try writing some more functional code and designing some of your own libraries. Have fun, enjoy struggling with design problems that come up, and see what you discover. Next we will begin to explore the universe of patterns and abstractions which the chapters so far have only hinted at.

Index Terms

algebra
algebraic design
backtracking
combinator parsing
commit
laws
parser combinators

10 Monoids

By the end of Part 2, we were getting comfortable with considering data types in terms of their *algebras*—that is, the operations they support and the laws that govern those operations. Hopefully you will have noticed that the algebras of very different data types tend to follow certain patterns that they have in common. In this chapter, we are going to begin identifying these patterns and taking advantage of them. We will consider algebras in the abstract, by writing code that doesn't just operate on one data type or another but on *all* data types that share a common algebra.

The first such abstraction that we will introduce is the *monoid*[1]. We choose to start with monoids because they are very simple and because they are ubiquitous. Monoids come up all the time in everyday programming, whether we're aware of them or not. Whenever you are working with a list or concatenating strings or accumulating the result of a loop, you are almost certainly using a monoid.

Footnote 1 The name "monoid" comes from mathematics. The prefix "mon-" means "one", and in category theory a monoid is a category with one object. See the chapter notes for more information.

10.1 What is a monoid?

Let's consider the algebra of string concatenation. We can add `"foo" + "bar"` to get `"foobar"`, and the empty string is an *identity element* for that operation. That is, if we say `(s + "")` or `("" + s)`, the result is always `s`. Furthermore, if we combine three strings by saying `(r + s + t)`, the operation is *associative*—it doesn't matter whether we parenthesize it `((r + s) + t)` or `(r + (s + t))`.

The exact same rules govern integer addition. It's associative and it has an identity element, `0`, which "does nothing" when added to another integer. Ditto for

multiplication, whose identity element is `1`.

The Boolean operators, `&&` and `||` are likewise associative, and they have identity elements `true` and `false`, respectively.

These are just a few simple examples, but algebras like this are virtually everywhere. The term for this kind of algebra is *"monoid"*. The laws of associativity and identity are collectively called the *monoid laws*. A monoid consists of:

- Some type `A`
- A binary associative operation that takes two values of type `A` and combines them into one.
- A value of type `A` that is an identity for that operation.

We can express this with a Scala trait:

```
trait Monoid[A] {
  def op(a1: A, a2: A): A
  def zero: A
}
```

An example instance of this trait is the `String` monoid:

```
val stringMonoid = new Monoid[String] {
  def op(a1: String, a2: String) = a1 + a2
  def zero = ""
}
```

List concatenation also forms a monoid:

```
def listMonoid[A] = new Monoid[List[A]] {
  def op(a1: List[A], a2: List[A]) = a1 ++ a2
  def zero = Nil
}
```

EXERCISE 1: Give `Monoid` instances for integer addition and multiplication as well as the Boolean operators.

```
val intAddition: Monoid[Int]
val intMultiplication: Monoid[Int]
val booleanOr: Monoid[Boolean]
val booleanAnd: Monoid[Boolean]
```

EXERCISE 2: Give a `Monoid` instance for combining `Option`s:

```
def optionMonoid[A]: Monoid[Option[A]]
```

EXERCISE 3: A function having the same argument and return type is called an *endofunction*[2]. Write a monoid for endofunctions:

```
def EndoMonoid[A]: Monoid[A => A]
```

Footnote 2 The Greek prefix "endo-" means "within", in the sense that an endofunction's codomain is within its domain.

EXERCISE 4: Use the property-based testing framework we developed in Part 2 to implement a property for the monoid laws. Use your property to test the monoids we have written.

```
val monoidLaws[A](m: Monoid[A]): Prop
```

SIDEBAR **Having vs. being a monoid**

There is a slight terminology mismatch between programmers and mathematicians, when they talk about a type *being* a monoid as against *having* a monoid instance. As a programmer, it's natural to think of the instance of type `Monoid[A]` as being *a monoid*. But that is not accurate terminology. The monoid is actually both things—the type together with the instance. When we say that a method accepts a value of type `Monoid[A]`, we don't say that it takes a monoid, but that it takes *evidence* that the type `A` is a monoid.

Just what *is* a monoid, then? It is simply an implementation of an interface governed by some laws. Stated tersely, *a monoid is a type together with an associative binary operation (`op`) which has an identity element (`zero`)*.

What does this buy us? Can we write any interesting programs over *any* data type, knowing nothing about that type other than that it's a monoid? Absolutely! Let's look at an example.

10.2 Folding lists with monoids

Monoids have an intimate connection with lists. If you look at the signatures for `foldLeft` and `foldRight` on `List`, you might notice something about the argument types.

```
def foldRight[B](z: B)(f: (A, B) => B): B
def foldLeft[B](z: B)(f: (B, A) => B): B
```

What happens when `A` and `B` are the same type?

```
def foldRight(z: A)(f: (A, A) => A): A
def foldLeft(z: A)(f: (A, A) => A): A
```

The components of a monoid fit these argument types like a glove. So if we had a list of `String`s, we could simply pass the `op` and `zero` of the `stringMonoid` in order to reduce the list with the monoid.

```
scala> val words = List("Hic", "Est", "Index")
words: List[String] = List(Hic, Est, Index)

scala> val s = words.foldRight(stringMonoid.zero)(stringMonoid.op)
s: String = "HicEstIndex"

scala> val t = words.foldLeft(stringMonoid.zero)(stringMonoid.op)
t: String = "HicEstIndex"
```

Notice that it doesn't matter if we choose `foldLeft` or `foldRight` when folding with a monoid[3]—we should get the same result. This is precisely because the laws of associativity and identity hold. A left fold associates operations to the left while a right fold associates to the right, with the identity element on the left and right respectively:

Footnote 3 As of this writing, with Scala at version 2.9.2, the implementation of `foldRight` in the standard library is not tail-recursive, so for large lists it matters operationally which we choose since they have different memory usage characteristics.

```
words.foldLeft("")(_ + _)  == (("" + "Hic") + "Est") + "Index"

words.foldRight("")(_ + _) == "Hic" + ("Est" + ("Index" + ""))
```

EXERCISE 5: Write a monoid instance for `String` that inserts spaces between words unless there already is one, and trims spaces off the ends of the result. For example:

```
op("Hic", op("est ", "chorda ")) == "Hic est chorda"
op("Hic ", op(" est"), "chorda") == "Hic est chorda"
```

```
def wordsMonoid(s: String): Monoid[String]
```

EXERCISE 6: Implement `concatenate`, a function that folds a list with a monoid:

```
def concatenate[A](as: List[A], m: Monoid[A]): A
```

But what if our list has an element type that doesn't have a `Monoid` instance? Well, we can always `map` over the list to turn it into a type that does.

```
def foldMap[A,B](as: List[A], m: Monoid[B])(f: A => B): B
```

EXERCISE 7: Write this function.

EXERCISE 8 (hard): The `foldMap` function can be implemented using either `foldLeft` or `foldRight`. But you can also write `foldLeft` and `foldRight` using `foldMap`! Try it.

10.3 Associativity and parallelism

The fact that a monoid's operation is associative means that we have a great deal of flexibility in how we fold a data structure like a list. We have already seen that operations can be associated to the left or right to reduce a list sequentially with `foldLeft` and `foldRight`. But we could instead split the data into chunks, fold them *in parallel*, and then combine the chunks with the monoid. Folding to the right, the combination of chunks `a`, `b`, `c`, and `d` would look like this:

```
op(a, op(b, op(c, d)))
```

Folding to the left would look like this:

```
op(op(op(a, b), c), d)
```

But folding in parallel looks like this:

```
op(op(a, b), op(c, d))
```

This parallelism might give us some efficiency gains, because the two inner `op`s could be run simultaneously in separate threads.

10.3.1 Example: Parallel parsing

As a nontrivial use case, let's say that we wanted to count the number of words in a `String`. This is a fairly simple parsing problem. We could scan the string character by character, looking for whitespace and counting up the number of runs of consecutive non-whitespace characters. Parsing sequentially like that, the parser state could be as simple as tracking whether the last character seen was a whitespace.

But imagine doing this not for just a short string, but an enormous text file. It would be nice if we could work with chunks of the file in parallel. The strategy would be to split the file into manageable chunks, process several chunks in parallel, and then combine the results. In that case, the parser state needs to be slightly more complicated, and we need to be able to combine intermediate results regardless of whether the section we're looking at is at the beginning, end, or middle of the file. In other words, we want that combination to be associative.

To keep things simple and concrete, let's consider a short string and pretend it's a large file:

```
"lorem ipsum dolor sit amet, "
```

If we split this string roughly in half, we might split it in the middle of a word. In the case of our string above, that would yield `"lorem ipsum do"` and `"lor sit amet, "`. When we add up the results of counting the words in these strings, we want to avoid double-counting the word `dolor`. Clearly, just counting the words as an `Int` is not sufficient. We need to find a data structure that can handle partial results like the half words `do` and `lor`, and can track the complete words seen so far, like `ipsum`, `sit`, and `amet`.

The partial result of the word count could be represented by an algebraic data

type:

```
sealed trait WC
case class Stub(chars: String) extends WC
case class Part(lStub: String, words: Int, rStub: String) extends WC
```

A `Stub` is the simplest case, where we have not seen any complete words yet. But a `Part` keeps the number of complete words we have seen so far, in `words`. The value `lStub` holds any partial word we have seen to the left of those words, and `rStub` holds the ones on the right.

For example, counting over the string `"lorem ipsum do"` would result in `Part("lorem", 1, "do")` since there is one complete word. And since there is no whitespace to the left of `lorem` or right of `do`, we can't be sure if they are complete words, so we can't count them yet. Counting over `"lor sit amet, "` would result in `Part("lor", 2, "")`.

EXERCISE 9: Write a monoid instance for `WC` and make sure that it meets the monoid laws.

```
val wcMonoid: Monoid[WC]
```

EXERCISE 10: Use the `WC` monoid to implement a function that counts words in a `String` by recursively splitting it into substrings and counting the words in those substrings.

SIDEBAR **Monoid homomorphisms**

If you have your law-discovering cap on while reading this chapter, you may notice that there is a law that holds for some functions *between* monoids. Take the `String` concatenation monoid and the integer addition monoid. If you take the length of two strings and add them up, it's the same as taking the length of the concatenation of those two strings:

```
"foo".length + "bar".length == ("foo" + "bar").length
```

Here, `length` is a function from `String` to `Int` that *preserves the monoid structure*. Such a function is called a *monoid homomorphism*[4]. A monoid homomorphism `f` between monoids `M` and `N` obeys the following general law for all values `x` and `y`:

Footnote 4 This word comes from Greek, "homo" meaning "same" and "morphe" meaning "shape".

```
M.op(f(x), f(y)) == f(N.op(x, y))
```

The same law holds for the homomorphism from `String` to `WC` in the current example.

This property can be very useful when designing your own libraries. If two types that your library uses are monoids, and there exist functions between them, it's a good idea to think about whether those functions are expected to preserve the monoid structure and to check the monoid homomorphism law with automated tests.

There is a higher-order function that can take any function of type `A => B`, where `B` is a monoid, and turn it in to a monoid homomorphism from `List[A]` to `B`.

Sometimes there will be a homomorphism in both directions between two monoids. Such a relationship is called a *monoid isomorphism* ("iso-" meaning "equal") and we say that the two monoids are isomorphic.

Associativity can also be exploited to gain efficiency. For example, if we want to concatenate a list of strings, doing so with a left or right fold can be less efficient than we would like. Consider the memory usage of this expression:

```
List("lorem", "ipsum", "dolor", "sit").foldLeft("")(_ + _)
```

At every step of the fold, we are allocating the full intermediate `String` only to discard it and allocate a larger string in the next step:

```
List("lorem", ipsum", "dolor", "sit").foldLeft("")(_ + _)
List("ipsum", "dolor", "sit").foldLeft("lorem")(_ + _)
List("dolor", "sit").foldLeft("loremipsum")(_ + _)
List("sit").foldLeft("loremipsumdolor")(_ + _)
List().foldLeft("loremipsumdolorsit")(_ + _)
"loremipsumdolorsit"
```

A more efficient strategy would be to combine the list by halves. That is, to first construct `"loremipsum"` and `"dolorsit"`, then add those together. But a `List` is inherently sequential, so there is not an efficient way of splitting it in half. Fortunately, there are other structures that allow much more efficient random access, such as the standard Scala library's `Vector` data type which provides very efficient `length` and `splitAt` methods.

EXERCISE 11: Implement an efficient `foldMap` for `IndexedSeq`, a common supertype for various data structures that provide efficient random access.

```
def foldMapV[A,B](v: IndexedSeq[A], m: Monoid[B])(f: A => B): B
```

EXERCISE 12: Use `foldMap` to detect whether a given `IndexedSeq[Int]` is ordered. You will need to come up with a creative `Monoid` instance.

10.4 Foldable data structures

In chapter 3, we implemented the data structures `List` and `Tree`, both of which could be folded. In chapter 5, we wrote `Stream`, a lazy structure that also can be folded much like a `List` can, and now we have just written a fold for `IndexedSeq`.

When we're writing code that needs to process data contained in one of these structures, we often don't care about the shape of the structure (whether it's a tree or a list), or whether it's lazy or not, or provides efficient random access, etc.

For example, if we have a structure full of integers and want to calculate their sum, we can use foldRight:

```
ints.foldRight(0)(_ + _)
```

Looking at just this code snippet, we don't really know the type of `ints`. It could be a `Vector`, a `Stream`, or a `List`, or anything at all with a `foldRight` method. We can capture this commonality in a trait:

```
trait Foldable[F[_]] {
  def foldRight[A, B](as: F[A])(f: (A, B) => B): B
  def foldLeft[A, B](as: F[A])(f: (B, A) => B): B
  def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]): B
  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    as.foldLeft(m.zero)(m.op)
}
```

Here we are abstracting over a type constructor `F`, much like we did with the `Parser` type in the previous chapter. We write it as `F[_]`, where the underscore indicates that `F` is not a type but a *type constructor* that takes one type argument. Just like functions that take other functions as arguments are called higher-order functions, something like `Foldable` is a *higher-order type constructor* or a *higher-kinded type*[5].

Footnote 5 Just like values and functions have types, types and type constructors have *kinds*. Scala uses kinds to track how many type arguments a type constructor takes, whether it is co- or contravariant in those arguments, and what their kinds are.

EXERCISE 13: Implement `Foldable[List]`, `Foldable[IndexedSeq]`, and `Foldable[Stream]`. Rembember that `foldRight`, `foldLeft` and `foldMap` can all be implemented in terms of each other, but that might not be the most efficient implementation.

EXERCISE 14: Recall the `Tree` data type from Chapter 3. Implement a `Foldable` instance for it.

```
sealed trait Tree[+A]
case object Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

EXERCISE 15: Write a `Foldable[Option]` instance.

EXERCISE 16: Any foldable structure can be turned into a `List`. Write this conversion in a generic way:

```
def toList[A](fa: F[A]): List[A]
```

10.5 Monoids compose

The `Monoid` abstraction by itself is not all that compelling, and with the generalized `foldMap` it's only slightly more interesting. The real power of monoids comes from the fact that they *compose*.

This means, for example, that if types `A` and `B` are monoids, then the tuple type `(A, B)` is also a monoid (called their *product*).

EXERCISE 17: Prove it.

```
def productMonoid[A,B](A: Monoid[A], B: Monoid[B]): Monoid[(A,B)]
```

EXERCISE 18: Do the same with `Either`. This is called a monoid *coproduct*.

```
def coproductMonoid[A,B](A: Monoid[A],
                         B: Monoid[B]): Monoid[Either[A,B]]
```

10.5.1 Assembling more complex monoids

Some data structures also have interesting monoids as long as their value types are monoids. For instance, there is a monoid for merging key-value `Map`s, as long as the value type is a monoid.

```
def mapMergeMonoid[K,V](V: Monoid[V]): Monoid[Map[K, V]] =
  new Monoid[Map[K, V]] {
    def zero = Map()
    def op(a: Map[K, V], b: Map[K, V]) =
      a.map {
        case (k, v) => (k, V.op(v, b.get(k) getOrElse V.zero))
      }
  }
```

Using these simple combinators, we can assemble more complex monoids fairly easily:

```
scala> val M: Monoid[Map[String, Map[String, Int]]] =
     | mapMergeMonoid(mapMergeMonoid(intAddition))
M: Monoid[Map[String, Map[String, Int]]] = $anon$1@21dfac82
```

This allows us to combine nested expressions using the monoid, with no

additional programming:

```
scala> val m1 = Map("o1" -> Map("i1" -> 1, "i2" -> 2))
m1: Map[String,Map[String,Int]] = Map(o1 -> Map(i1 -> 1, i2 -> 2))

scala> val m2 = Map("o1" -> Map("i2" -> 3))
m2: Map[String,Map[String,Int]] = Map(o1 -> Map(i2 -> 3))

scala> val m3 = M.op(m1, m2)
m3: Map[String,Map[String,Int]] = Map(o1 -> Map(i1 -> 1, i2 -> 5))
```

EXERCISE 19: Write a monoid instance for functions whose results are monoids.

```
def functionMonoid[A,B](B: Monoid[B]): Monoid[A => B]
```

EXERCISE 20: Use monoids to compute a frequency map of words in an `IndexedSeq` of `String`s.

```
def frequencyMap(strings: IndexedSeq[String]): Map[String, Int]
```

A frequency map contains one entry per word, with that word as the key, and the number of times that word appears as the value under that key. For example:

```
scala> frequencyMap(Vector("a rose", "is a", "rose is", "a rose"))
res0: Map[String,Int] = Map(a -> 3, rose -> 3, is -> 2)
```

10.5.2 Using composed monoids to fuse traversals

The fact that multiple monoids can be composed into one means that we can perform multiple calculations simultaneously when folding a data structure. For example, we can take the length and sum of a list at the same time in order to calculate the mean:

```
scala> val m = productMonoid(intAddition, intAddition)
m: Monoid[(Int, Int)] = $anon$1@8ff557a

scala> val p = listFoldable.foldMap(List(1,2,3,4))(a => (1, a))(m)
p: (Int, Int) = (10, 4)

scala> val mean = p._1 / p._2.toDouble
mean: Double = 2.5
```

10.5.3 Implicit monoids

The ability to compose monoids really comes into its own when you have complicated nested structures. But it can be somewhat tedious to assemble them by hand in each case. Fortunately, Scala has a facility that can make monoids a lot more pleasant to work with by making instances implicit:

```
implicit val stringMonoid: Monoid[String] = ...
```

Such implicit instances can then be used to implicitly construct more complicated instances:

```
implicit def productMonoid[A](implicit A: Monoid[A]): Monoid[List[A]] =
  ...
```

If we make all of our monoid instances implicit, a simple function definition with an implicit argument can automatically assemble arbitrarily complex monoids for us.

```
def monoid[A](implicit A: Monoid[A]): Monoid[A] = A
```

All we have to do is specify the type of the monoid we want. As long as that type has an implicit `Monoid` instance, it will be returned to us.

```
scala> monoid[Map[List[String], Int => Map[String, List[Boolean]]]]
res0: Monoid[Map[List[String], Int => Map[String, List[Boolean]]]] =
$anon$1@356380e8
```

But what about types that have more than once `Monoid` instance? For example, a valid monoid for `Int` could be either addition (with 0 as the identity) or multiplication (with 1 as the identity). A common solution is to put values in a simple wrapper type and make that type the monoid.

```
case class Product(value: Int)
implicit val productMonoid: Monoid[Product] = new Monoid[Product] {
  def op(a: Product, b: Product) = Product(a.value * b.value)
  def zero = 1
```

```
}

case class Sum(value: Int)
implicit val sumMonoid: Monoid[Sum] = new Monoid[Sum] {
  def op(a: Sum, b: Sum) = Sum(a.value + b.value)
  def zero = 0
}
```

This plays nicely with `foldMap` on `Foldable`. For example if we have a list `ints` of integers and we want to sum them:

```
listFoldable.foldMap(ints)(Sum(_))
```

Or to take their product:

```
listFoldable.foldMap(ints)(Product(_))
```

10.6 Summary

In this chapter we introduced the concept of a monoid, a simple and common type of abstract algebra. When you start looking for it, you will find ample opportunity to exploit the monoidal structure of your own libraries. The associative property lets you fold any `Foldable` data type and gives you the flexibility of doing so in parallel. Monoids are also compositional, and therefore they let you write folds in a declarative and reusable way.

`Monoid` has been our first totally abstract trait, defined only in terms of its abstract operations and the laws that govern them. This gave us the ability to write useful functions that know nothing about their arguments except that their type happens to be a monoid.

Index Terms

associativity
higher-kinded types
identity element
implicit instances
monoid
monoid coproduct
monoid homomorphism
monoid laws
monoid product
type constructor polymorphism

11 Monads

In Part 2, we implemented several different combinator libraries. In each case, we proceeded by writing a small set of primitives and then a number of derived combinators defined purely in terms of existing combinators. You probably noticed some similarities between implementations of different derived combinators *across* the combinator libraries we wrote. For instance, we implemented a `map` function for each data type, to lift a function taking one argument. For `Gen`, `Parser`, and `Option`, the type signatures were as follows:

```
def map[A,B](ga: Gen[A])(f: A => B): Gen[B]

def map[A,B](pa: Parser[A])(f: A => B): Parser[B]

def map[A,B](oa: Option[A])(f: A => B): Option[A]
```

These type signatures are very similar. The only difference is the concrete type involved. We can capture, as a Scala trait, the idea of "a data type that implements `map`".

11.1 Functors: Generalizing the map function

If a data type `F` implements `map` with a signature like above, we say that `F` is a *functor*. Which is to say that we can provide an implementation of the following trait:

```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

Much like we did with `Foldable` in the previous chapter, we introduce a trait `Functor` that is parameterized on a type constructor `F[_]`. Here is an instance for `List`:

```
val listFunctor extends Functor[List] {
  def map[A,B](as: List[A])(f: A => B): List[B] = as map f
}
```

What can we do with this abstraction? There happens to be a small but useful library of functions that we can write using just `map`. For example, if we have `F[(A, B)]` where `F` is a functor, we can "distribute" the `F` over the pair to get `(F[A], F[B])`:

```
def distribute[A,B](fab: F[(A, B)]): (F[A], F[B]) =
  (map(fab)(_._1), map(fab)(_._2))
```

It's all well and good to introduce a new combinator like this in the abstract, but we should think about what it *means* for concrete data types like `Gen`, `Option`, etc. For example, if we `distribute` a `List[(A, B)]`, we get two lists of the same length, one with all the `A`s and the other with all the `B`s. That operation is sometimes called "unzip". So we just wrote a generic unzip that works not just for lists, but for any functor!

Whenever we create an abstraction like this, we should consider not only what abstract methods it should have, but which *laws* we expect should hold for the implementations. If you remember, back in chapter 7 (on parallelism) we found a general law for `map`:

```
map(v)(x => x) == v
```

Later in Part 2, we found that this law is not specific to `Par`. In fact, we would expect it to hold for all implementations of `map` that are "structure-preserving". This law (and its corollaries given by parametricity) is part of the specification of what it means to "map". It would be very strange if it didn't hold. It's part of what a functor *is*.

11.2 Generalizing the flatMap and unit functions

Not only did we implement `map` for many data types, we also implemented a `map2` to lift a function taking two arguments. For `Gen`, `Parser`, and `Option`, the `map2` function could be implemented as follows.

```
def map2[A,B,C](                                          ❶
      ga: Gen[A], gb: Gen[B])(f: (A,B) => C): Gen[C] =
  ga flatMap (a => fb map (b => f(a,b)))

def map2[A,B,C](                                          ❷
      pa: Parser[A], pb: Parser[B])(f: (A,B) => C): Parser[C] =
  pa flatMap (a => pb map (b => f(a,b)))

def map2[A,B,C](                                          ❸
      pa: Option[A], pb: Option[B])(f: (A,B) => C): Option[C] =
  pa flatMap (a => pb map (b => f(a,b)))
```

❶ Makes a generator of a random C that runs random generators ga and gb, combining their results with the function f.
❷ Makes a parser that produces C by combining the results of parsers pa and pb with the function f.
❸ Combines two Options with the function f, if both have a value. Otherwise returns None.

These functions have more in common than just the name. In spite of operating on data types that seemingly have nothing to do with one another, the implementations are exactly identical! And again the only thing that differs in the type signatures is the particular data type being operated on. This confirms what we have been suspecting all along—that these are particular instances of some more general pattern. We should be able to exploit that fact to avoid repeating ourselves. For example, we should be able to write `map2` once and for all in such a way that it can be reused for all of these data types.

We've made the code duplication particularly obvious here by choosing uniform names for our functions, taking the arguments in the same order, and so on. It may be more difficult to spot in your everyday work. But the more combinator libraries you write, the better you will get at identifying patterns that you can factor out into a common abstraction. In this chapter, we will look at an abstraction that unites `Parser`, `Gen`, `Par`, `Option`, and some other data types we have already looked at: They are all *monads*. We will explain in a moment exactly what that means.

11.2.1 The Monad trait

In both parts 1 and 2 of this book, we concerned ourselves with finding a minimal set of primitive operations for various data types. We implemented `map2` for many data types, but as we know by now, `map2` can be implemented in terms of `map` and `flatMap`. Is that a minimal set? Well, the data types in question all had a `unit`, and we know that `map` can be implemented in terms of `flatMap` and `unit`. For example, on `Gen`:

```
def map[A,B](f: A => B): Gen[B] =
  flatMap(a => unit(f(a)))
```

So let's pick `unit` and `flatMap` as our minimal set. We will unify under a single concept all data types that have these combinators defined. `Monad` has `flatMap` and `unit` abstract, and provides default implementations for `map` and `map2`.

```
trait Monad[M[_]] extends Functor[M] {
  def unit[A](a: => A): M[A]
  def flatMap[A,B](ma: M[A])(f: A => M[B]): M[B]

  def map[A,B](ma: M[A])(f: A => B): M[B] =
    flatMap(ma)(a => unit(f(a)))
  def map2[A,B,C](ma: M[A], mb: M[B])(f: (A, B) => C): M[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))
}
```

1. Since Monad provides a default implementation of map, it can extend Functor. All monads are functors, but not all functors are monads.

SIDEBAR **The name "monad"**

We could have called `Monad` anything at all, like `FlatMappable` or whatever. But "monad" is already a perfectly good name in common use. The name comes from category theory, a branch of mathematics that has inspired a lot of functional programming concepts. The name "monad" is intentionally similar to "monoid", and the two concepts are related in a deep way. See the chapter notes for more information.

To tie this back to a concrete data type, we can implement the `Monad` instance for `Gen`:

```
object Monad {
  val genMonad = new Monad[Gen] {
    def unit[A](a: => A): Gen[A] = Gen.unit(a)
    def flatMap[A,B](ma: Gen[A])(f: A => Gen[B]): Gen[B] =
      ma flatMap f
  }
}
```

We only need to implement `unit` and `flatMap` and we get `map` and `map2` at no additional cost. We have implemented them once and for all, for any data type for which it is possible to supply an instance of `Monad`! But we're just getting started. There are many more combinators that we can implement once and for all in this manner.

EXERCISE 1: Write monad instances for `Par`, `Parser`, `Option`, `Stream`, and `List`.

EXERCISE 2 (optional, hard): `State` looks like it would be a monad too, but it takes two type arguments and you need a type constructor of one argument to implement `Monad`. Try to implement a `State` monad, see what issues you run into, and think about possible solutions. We will discuss the solution later in this chapter.

11.3 Monadic combinators

Now that we have our primitive combinators for monads, we can look back at previous chapters and see if there were some other combinators that we implemented for each of our monadic data types. Many of of them can be implemented once for all monads, so let's do that now.

EXERCISE 3: The `sequence` and `traverse` combinators should be pretty familiar to you by now, and your implementations of them from various prior chapters are probably all very similar. Implement them once and for all on `Monad[M]`:

```
def sequence[A](lma: List[M[A]]): M[List[A]]
def traverse[A,B](la: List[A])(f: A => M[B]): M[List[B]]
```

One combinator we saw for e.g. `Gen` and `Parser` was `listOfN`, which allowed us to replicate a parser or generator `n` times to get a parser or generator of

lists of that length. We can implement this combinator for all monads `M` by adding it to our `Monad` trait. We should also give it a more generic name such as `replicateM`.

EXERCISE 4: Implement `replicateM`:

```
def replicateM[A](n: Int, ma: M[A]): M[List[A]]
```

EXERCISE 5: Think about how `replicateM` will behave for various choices of `M`. For example, how does it behave in the `List` monad? What about `Option`? Describe in your own words the general meaning of `replicateM`.

There was also a combinator `product` for our `Gen` data type to take two generators and turn them into a generator of pairs, and we did the same thing for `Par` computations. In both cases, we implemented `product` in terms of `map2`. So we can definitely write it generically for any monad `M`. This combinator might more appropriately be called `factor`, since we are "factoring out" `M`, or "pushing `M` to the outer layer":

```
def factor[A,B](ma: M[A], mb: M[B]): M[(A, B)] = map2(ma, mb)((_, _))
```

We don't have to restrict ourselves to combinators that we have seen already. It's important to *play around* and see what we find. Now that we're thinking in the abstract about monads, we can take a higher perspective on things.

We know that `factor` and `map2` let us combine pairs, or when we otherwise have *both* of two things. But what about when we have *either* of two things (sometimes called their *coproduct*)? What happens then? We would have something like this:

```
def cofactor[A,B](e: Either[M[A], M[B]]): M[Either[A, B]]
```

EXERCISE 6: Implement the function `cofactor`. Explain to yourself what it does.

The combinators we have seen here are only a small sample of the full library that `Monad` lets us implement once and for all.

11.4 Monad laws

In chapters past, we found that the `map` function, for all of the data types that support it, obeys the functor law. It would be very strange if it didn't hold. It's part of the meaning we ascribe to `map`. It's what a `Functor` *is*.

But what about `Monad`? Certainly we would expect the functor laws to also hold for `Monad`, but what else would we expect? What laws should govern `flatMap` and `unit`?

11.4.1 The associative law

For example, if we wanted to combine three monadic values into one, which two would we expect to be combined first? Would it matter? To answer this question, let's for a moment take a step down from the abstract level and look at a simple concrete example of using the `Gen` monad.

Say we are testing a product order system and we need to mock up some orders. We might have an `Order` case class and a generator for that class.

```
case class Order(item: Item, quantity: Int)
case class Item(name: String, price: Double)

val genOrder: Gen[Order] = for {
  name <- Gen.nextString
  price <- Gen.nextDouble
  quantity <- Gen.nextInt
} yield Order(Item(name, price), quantity)
```

Above, we are generating the `Item` inline, but there might be places where we want to generate an `Item` separately. So we could pull that into its own generator:

```
val genItem: Gen[Item] = for {
  name <- Gen.nextString
  price <- Gen.nextDouble
} yield Item(name, price)
```

Then we can use that in `genOrder`:

```
val genOrder: Gen[Order] = for {
  item <- genItem
  quantity <- Gen.nextInt
} yield Order(item, quantity)
```

And that should do exactly the same thing, right? It seems safe to assume that. But not so fast. How can we be sure? It's not exactly the same code.

EXERCISE 7 (optional): Expand both implementations of `genOrder` to `map` and `flatMap` calls to see the true difference.

Once you expand them out, those two implementations are clearly not identical. And yet when we look at the `for`-comprehension, it seems perfectly reasonable to assume that the two implementations do exactly the same thing. In fact, it would be surprising and weird if they didn't. It's because we are assuming that `flatMap` obeys an *associative law*:

```
x.flatMap(f).flatMap(g) == x.flatMap(a => f(a).flatMap(g))
```

And this law should hold for all values `x`, `f`, and `g` of the appropriate types.

EXERCISE 8: *Prove* that this law holds for `Option`.

EXERCISE 9 (hard): Show that the equivalence of the two `genOrder` implementations above follows from this law.

11.4.2 Kleisli composition

It's not so easy to see that the law we have discovered is an *associative* law. Remember the associative law for monoids? That was very clear:

```
append(append(a, b), c) == append(a, append(b, c))
```

But our associative law for monads doesn't look anything like that! Fortunately, there's a way we can make the law clearer if we consider not the monadic values of types like `M[A]`, but monadic *functions* of types like `A => M[B]`. Functions like that are called *Kleisli arrows*[1], and they can be composed with one another:

Footnote 1 This name comes from category theory and is after the Swiss mathematician Heinrich Kleisli.

```
def compose[A,B,C](f: A => M[B], g: B => M[C]): A => M[C]
```

EXERCISE 10: Implement this function.

We can now state the associative law for monads in a much more symmetric way:

```
compose(compose(f, g), h) == compose(f, compose(g, h))
```

EXERCISE 11: Implement `flatMap` in terms of `compose`. It seems that we have found another minimal set of monad combinators: `compose` and `unit`.

EXERCISE 12 (optional): Show that the two formulations of the associative law, the one in terms of `flatMap` and the one in terms of `compose`, are equivalent.

11.4.3 The identity laws

The other monad law is now pretty easy to see. Just like `zero` was an *identity element* for `append` in a monoid, there is an identity element for `compose` in a monad. Indeed, that is exactly what `unit` is, and that is why we chose its name the way we did:

```
def unit[A](a: => A): M[A]
```

This function has exactly the right type to be passed to `compose`. The effect should be that anything composed with `unit` is that same thing. This usually takes the form of two laws, *left identity* and *right identity*:

```
compose(f, unit) == f

compose(unit, f) == f
```

EXERCISE 13: Rewrite these monad identity laws in terms of `flatMap`.

EXERCISE 14: There is a third minimal set of monadic combinators: `map`, `unit`, and `join`. Implement `join`.

```
def join[A](mma: M[M[A]]): M[A]
```

EXERCISE 15: Implement either `flatMap` or `compose` in terms of `join`.

EXERCISE 16: Use `join` to restate the monad laws.

EXERCISE 17 (optional): Write down an explanation, in your own words, of what the associative law means for `Par` and `Parser`.

EXERCISE 18 (optional): Explain in your own words what the identity laws are stating in concrete terms for `Gen` and `List`.

11.5 Just what is a monad?

Let us now take a wider perspective. There is something unusual about the `Monad` interface. The data types for which we've given monad instances don't seem to have very much to do with each other. Yes, `Monad` factors out code duplication among them, but what *is* a monad exactly? What does "monad" *mean*?

You may be used to thinking of interfaces as providing a relatively complete API for an abstract data type, merely abstracting over the specific representation. After all, a singly-linked list and an array-based list may be implemented differently behind the scenes, but they will share a common interface in terms of which a lot of useful and concrete application code can be written. `Monad` is much more abstract than that. The `Monad` combinators are often just a small fragment of the full API for a given data type that happens to be a monad. So `Monad` doesn't generalize one type or another; rather, many vastly different data types can satisfy the `Monad` interface.

We have discovered that there are at least three minimal sets of primitive `Monad` combinators, and instances of `Monad` will have to provide implementations of one of these sets:

- `unit` and `flatMap`
- `unit` and `compose`
- `unit`, `map`, and `join`

And we know that there are two monad laws to be satisfied, associativity and identity, that can be formulated in various ways. So we can state quite plainly what a monad *is*:

A monad is an implementation of one of the minimal sets of monadic combinators, satisfying the laws of associativity and identity.

That's a perfectly respectable, precise, and terse definition. But it's a little dissatisfying. It doesn't say very much about what it implies—what a monad *means.* The problem is that it's a *self-contained* definition. Even if you're a beginning programmer, you have by now obtained a vast amount of knowledge about programming, and this definition integrates with none of that. In order to really *understand* what's going on with monads (or with anything for that matter), we need to think about them in terms of things we already understand. We need to connect this new knowledge into a wider context.

To say "this data type is a monad" is to say something very specific about how

it behaves. But what exactly? To begin to answer the question of what monads *mean*, let's look at another couple of monads and compare their behavior.

11.5.1 The identity monad

To really distill monads to their essentials, we should look at the simplest interesting specimen, the identity monad, given by the following type:

```
case class Id[A](value: A)
```

EXERCISE 19: Implement `map` and `flatMap` as methods on this class, and give an implementation for `Monad[Id]`.

Now, `Id` is just a simple wrapper. It doesn't really add anything. Applying `Id` to `A` is an identity since the wrapped type and the unwrapped type are totally isomorphic (we can go from one to the other and back again without any loss of information). But what is the meaning of the identity *monad*? Let's try using it in the REPL:

```
scala> Id("Hello, ") flatMap (a =>
     |   Id("monad!") flatMap (b =>
     |     Id(a + b)))
res0: Id[java.lang.String] = Id(Hello, monad!)
```

When we write the exact same thing with a `for`-comprehension, it might be clearer:

```
scala> for {
     |   a <- Id("Hello, ")
     |   b <- Id("monad!")
     | } yield a + b
res1: Id[java.lang.String] = Id(Hello, monad!)
```

So what is the *action* of `flatMap` for the identity monad? It's simply variable substitution. The variables `a` and `b` get bound to `"Hello, "` and `"monad!"`, respectively, and then they get substituted into the expression `a + b`. In fact, we could have written the same thing without the `Id` wrapper, just using Scala's own variables:

```
scala> val a = "Hello, "
a: java.lang.String = "Hello, "
```

```
scala> val b = "monad!"
b: java.lang.String = monad!

scala> a + b
res2: java.lang.String = Hello, monad!
```

Besides the `Id` wrapper, there is no difference. So now we have at least a partial answer to the question of what monads mean. We could say that monads provide a scheme for binding variables. Stated more dramatically, *a monad is a kind of programming language that supports variable substitution.*

Let's see if we can get the rest of the answer.

11.5.2 The State monad and partial type application

Look back at the chapter on the `State` data type. Recall that we implemented some combinators for `State`, including `map` and `flatMap`:

```
case class State[S, A](run: S => (A, S)) {
  def map[B](f: A => B): State[S, B] =
    State(s => {
      val (a, s1) = run(s)
      (f(a), s1)
    })
  def flatMap[B](f: A => State[S, B]): State[S, B] =
    State(s => {
      val (a, s1) = run(s)
      f(a).run(s1)
    })
}
```

It looks like `State` definitely fits the profile for being a monad. But its type constructor takes two type arguments and `Monad` requires a type constructor of one argument, so we can't just say `Monad[State]`. But if we choose some particular `S` then we have something like `State[S, _]`, which is the kind of thing expected by `Monad`. So `State` doesn't just have one monad instance but a whole family of them, one for each choice of `S`. We would like to be able to partially apply `State` to where the `S` type argument is fixed to be some concrete type.

This is very much like how you would partially apply a function, except at the type level. For example, we can create an `IntState` type constructor, which is an alias for `State` with its first type argument fixed to be `Int`:

```
```

```
type IntState[A] = State[Int, A]
```

And `IntState` is exactly the kind of thing that we can build a `Monad` for:

```
object IntStateMonad extends Monad[IntState] {
  def unit[A](a: => A): IntState[A] = State(s => (a, s))
  def flatMap[A,B](st: IntState[A])(f: A => IntState[B]): IntState[B] =
    st flatMap f
}
```

Of course, it would be really repetitive if we had to manually write a separate `Monad` instance for each specific state type. Unfortunately, Scala does not allow us to use underscore syntax to simply say `State[Int, _]` to create an anonymous type constructor like we create anonymous functions with the underscore syntax. But instead we can use something similar to lambda syntax at the type level. For example, we could have declared `IntState` directly inline like this:

```
object IntStateMonad extends
  Monad[({type IntState[A] = State[Int, A]})#IntState] {
  ...
}
```

This syntax can be a little jarring when you first see it. But all we are doing is declaring an anonymous type within parentheses. This anonymous type has, as one of its members, the type alias `IntState`, which looks just like before. Outside the parentheses we are then accessing its `IntState` member with the # syntax. Just like we can use a "dot" (`.`) to access a member of an object at the value level, we can use the # symbol to access a type member (See the "Type Member" section of the Scala Language Specification).

A type constructor declared inline like this is often called a *type lambda* in Scala. We can use this trick to partially apply the `State` type constructor and declare a `StateMonad` trait. An instance of `StateMonad[S]` is then a monad instance for the given state type `S`.

```
def stateMonad[S] = new Monad[({type lambda[x] = State[S,x]})#lambda] {
  def unit[A](a: => A): State[S,A] = State(s => (a, s))
  def flatMap[A,B](st: State[S,A])(f: A => State[S,B]): State[S,B] =
    st flatMap f
}
```

Again, just by giving implementations of `unit` and `flatMap`, we get implementations of all the other monadic combinators for free.

EXERCISE 20: Now that we have a `State` monad, you should try it out to see how it behaves. What is the meaning of `replicateM` in the `State` monad? How does `map2` behave? What about `sequence`?

Let's now look at the difference between the `Id` monad and the `State` monad. Remember that the primitive operations on `State` (besides the monadic operations `unit` and `flatMap`) are that we can read the current state with `getState` and we can set a new state with `setState`.

```
def getState[S]: State[S, S]
def setState[S](s: => S): State[S, Unit]
```

Remember that we also discovered that these combinators constitute a minimal set of primitive operations for `State`. So together with the monadic primitives (`unit` and `flatMap`), they *completely specify* everything that we can do with the `State` data type.

EXERCISE 21: What laws do you expect to mutually hold for `getState`, `setState`, `unit` and `flatMap`?

What does this tell us about the meaning of the `State` *monad*? Let's study a simple example:

```
val M = new StateMonad[Int]

def zipWithIndex[A](as: List[A]): List[(Int,A)] =
  as.foldLeft(M.unit(List[(Int, A)]()))((acc,a) => for {
    n  <- getState
    xs <- acc
    _  <- setState(n + 1)
} yield (n, a) :: xs).run(0)._1.reverse
```

This function numbers all the elements in a list using a `State` action. It keeps a state that is an `Int` which is incremented at each step. The whole composite state action is run starting from 0. We are then reversing the result since we constructed it in reverse order[2].

Footnote 2 This is asymptotically faster than appending to the list in the loop.

The details of this code are not really important. What is important is what's

going on with `getState` and `setState` in the `for`-comprehension. We are obviously getting variable binding just like in the `Id` monad—we are binding the value of each successive state action (`getState`, `acc`, and then `setState`) to variables. But there is more going on, literally *between the lines.* At each line in the `for`-comprehension, the implementation of `flatMap` is making sure that the current state is available to `getState`, and that the new state gets propagated to all actions that follow a `setState`.

What does the difference between the action of `Id` and the action of `State` tell us about monads in general? We can see that a chain of `flatMap`s (or an equivalent `for`-comprehension) is like an imperative program with statements that assign to variables, and *the monad specifies what occurs at statement boundaries.* For example, with `Id`, nothing at all occurs except unwrapping and re-wrapping in the `Id` constructor. With `State`, the most current state gets passed from one statement to the next. With the `Option` monad, a statement may return `None` and terminate the program. With the `List` monad a statement may return many results, which causes statements that follow it to potentially run multiple times, once for each result.

EXERCISE 22: To cement your understanding of monads, give a monad instance for the following type, and explain what it means. What are its primitive operations? What is the action of `flatMap`? What meaning does it give to monadic functions like `sequence`, `join`, and `replicateM`? What meaning does it give to the monad laws?

```
case class Reader[R, A](run: R => A)

object Reader {
  def readerMonad[R] = new Monad[({type f[x] = Reader[R,x]})#f] {
    def unit[A](a: => A): Reader[R,A]
    def flatMap[A,B](st: Reader[R,A])(f: A => Reader[R,B]): Reader[R,B]
  }
}
```

11.6 Conclusion

In this chapter, we took a pattern that we had seen repeated throughout the book and we unified it under a single concept: monad. This allowed us to write a number of combinators once and for all, for many different data types that at first glance don't seem to have anything in common. We discussed laws that they all satisfy, the monad laws, from various perspectives, and we tried to gain some insight into what it all means.

An abstract topic like this cannot be fully understood all at once. It requires an iterative approach where you keep revisiting the topic from new perspectives. When you discover new monads, new applications of them, or see them appear in a new context, you will inevitably gain new insight. And each time it happens you might think to yourself: "OK, I thought I understood monads before, but now I *really* get it."

In the next chapter, we will explore a slight variation on the theme of monads, and develop your ability to discover new abstractions on your own.

Index Terms

functor
functor law
identity element
Kleisli arrow
left identity law for monads
listOfN
monad
monadic join
monad identity law
monad laws
replicateM
right identity law for monads
type lambda
unit law

12 Applicative and traversable functors

In the previous chapter on monads we saw how a lot of the functions we have been writing for different combinator libraries can be expressed in terms of a single interface, `Monad`.

Monads provide a very powerful interface, as evidenced by the fact that we can use `flatMap` to essentially write imperative programs in a purely functional way. But sometimes this is more power than we need, and such power comes at a price. As we will see in this chapter, the price is that we lose some compositionality. We can reclaim it by instead using *applicative functors*, which are simpler and more general than monads.

12.1 Generalizing monads

By now you have seen `sequence` and `traverse` many times for different monads, implemented them in terms of each other, and generalized them to *any* monad `M`:

```
def sequence[A](lma: List[M[A]]): M[List[A]]
  traverse(mas, ma => ma)

def traverse[A,B](as: List[A])(f: A => M[B]): M[List[B]]
  as.foldRight(unit(List[B]()))((a, mbs) => map2(f(a), mbs)(_ :: _))
```

Here, the implementation of `traverse` is using `map2` and `unit`, and we have seen that `map2` can be implemented in terms of `flatMap`:

```
def map2[A,B,C](ma: M[A], mb: M[B])(f: (A,B) => C): M[C] =
  flatMap(ma)(a => map(mb)(b => f(a,b)))
```

What you may not have noticed is that almost all of the useful combinators we wrote in the previous chapter can be implemented just in terms of `map2` and `unit`. What's more, for all the data types we have discussed, `map2` can be written *without* resorting to `flatMap`.

Is `map2` with `unit` then just another minimal set of operations for monads? No it's not, because there are monadic combinators such as `join` and `flatMap` that cannot be implemented with just `map2` and `unit`. This suggests that `map2` and `unit` is a less powerful subset of the `Monad` interface, that nonetheless is very useful on its own.

12.2 The Applicative trait

Since so many combinators can be written in terms of just these two functions, there is a name for data types that can implement `map2` and `unit`. They are called *applicative functors*:

```
trait Applicative[F[_]] extends Functor[F] {
  def map2[A,B](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]
  def apply[A,B](fab: F[A => B])(fa: F[A]): F[B]
  def unit[A](a: A): F[A]
}
```

There is an additional combinator, `apply`, that we haven't yet discussed. The `map2` and `apply` combinators can be implemented in terms of each other, so a minimal implementation of `Applicative` must provide one or the other.

EXERCISE 1: Implement `map2` in terms of `apply` and `unit`. Then implement `apply` in terms of `map2` and `unit`.

`Applicative` also extends `Functor`. This implies that `map` can be written in terms of `map2`, `apply`, and `unit`. Here it is in terms of `apply` and `unit`:

```
def map[A,B](a: F[A])(f: A => B): F[B] =
  apply(unit(f))(a)
```

And here in terms of `map2` and `unit`:

```
def map[A,B](a: F[A])(f: A => B): F[B] =
  map2(a, unit(f))(_(_))
```

You should already have a good sense of what `map2` means. But what is the

meaning of `apply`? At this point it's a little bit of a floating abstraction, and we need an example. Let's drop down to `Option` and look at a concrete implementation:

```
def apply[A,B](oab: Option[A => B])(oa: Option[A]): Option[B] =
  (oab, oa) match {
    case (Some(f), Some(a)) => Some(f(a))
    case _ => None
  }
```

You can see that this method combines two `Option`s. But one of them contains a *function* (unless it is `None` of course). The action of `apply` is to apply the function inside one argument to the value inside the other. This is the origin of the name "applicative". This operation is sometimes called *idiomatic function application* since it occurs within some idiom or context. In the case of the `Identity` idiom from the previous chapter, the implementation is literally just function application. In the example above, the idiom is `Option` so the method additionally encodes the rule for `None`, which handles the case where either the function or the value are absent.

The action of `apply` is similar to the familiar `map`. Both are a kind of function application in a context, but there's a very specific difference. In the case of `apply`, the function being applied might be affected by the context. For example, if the second argument to `apply` is `None` in the case of `Option` then there is no function at all. But in the case of `map`, the function must exist independently of the context. This is easy to see if we rearrange the type signature of `map` a little and compare it to `apply`:

```
def   map[A,B](f:   A => B )(a: F[A]): F[B]
def apply[A,B](f: F[A => B])(a: F[A]): F[B]
```

The only difference is the `F` around the function argument type. The `apply` function is strictly more powerful, since it can have that added `F` effect. It makes sense that we can implement `map` in terms of `apply`, but not the other way around.

We have also seen that we can implement `map2` as well in terms of `apply`. We can extrapolate that pattern and implement `map3`, `map4`, etc. In fact, `apply` can be seen as a general function lifting combinator. We can use it to lift a function

of any arity into our applicative functor. Let's look at `map3` on `Option` as an example.

We start out with a ternary function like `(_ + _ + _)`, which just adds 3 integers:

```
val f: (Int, Int, Int) => Int = (_ + _ + _)
```

If we Curry this function, we get a slightly different type. It's the same function, but one that can be gradually applied to arguments one at a time.

```
val g: Int => Int => Int => Int = f.curried
```

Pass that to `unit` (which is just `Some` in the case of `Option`) and we have:

```
val h: Option[Int => Int => Int => Int] = unit(g)
```

We can now use idiomatic function application three times to apply this to three `Option` values. To complete the picture, here is the fully general implementation of `map3` in terms of `apply`.

```
def map3[A,B,C,D](fa: F[A],
                  fb: F[B],
                  fc: F[C])(f: (A, B, C) => D): F[D] =
  apply(apply(apply(unit(f.curried))(fa))(fb))(fc)
```

The pattern is simple. We just Curry the the function we want to lift, pass the result to `unit`, and then `apply` as many times as there are arguments. Each call to `apply` is a partial application of the function[1].

Footnote 1 Notice that in this sense `unit` can be seen as `map0`, since it's the case of "lifting" where `apply` is called zero times.

Another interesting fact is that we have just discovered a new minimal definition of `Monad`! We already know that a monad can be given by implementations of `map`, `unit`, and `join`. Well, since we can implement `map` in terms of `apply`, we now know that a `Monad` can be given by `unit`, `join`, and either `map2` or `apply`. This gives us further insight into what a monad is! A monad is just an applicative functor with an additional combinator, `join`.

12.3 The difference between monads and applicative functors

We know that we can implement `map2` using `flatMap`. And following the reasoning above, this means that *all monads are applicative functors*. Since we have already implemented `Monad` instances for a great many data types, we don't have to write `Applicative` instances for them. We can simply change `Monad` so that it extends `Applicative` and overrides `apply`:

```
trait Monad[M[_]] extends Applicative[M] {
  def flatMap[A,B](ma: M[A])(f: A => M[B]): M[B] = join(map(ma)(f))

  def join[A](mma: M[M[A]]): M[A] = flatMap(mma)(ma => ma)

  def compose[A,B,C](f: A => M[B], g: B => M[C]): A => M[C] =
    a => flatMap(f(a))(g)

  override def apply(mf: M[A => B])(ma: M[A]): M[B] =
    flatMap(mf)(f => flatMap(ma)(a => f(a))
}
```

But the converse is not true—not all applicative functors are monads. Using `unit` together with either `map2` or `apply`, we cannot implement `flatMap` or `join`. If we try it, we get stuck pretty quickly.

Let's look at the signatures of `flatMap` and `apply` side by side, rearranging the signature of `flatMap` slightly from what we're used to in order to make things clearer:

```
def apply[A,B](fab: F[A => B])(fa: F[A]): F[B]
def flatMap[A,B](f: A => F[B])(fa: F[A]): F[B]
```

The difference is in the type of the function argument. In `apply`, that argument is fully contained inside `F`. So the only way to pass the `A` from the second argument to the function of type `A => B` in the first argument is to somehow consider both `F` contexts. For example, if `F` is `Option` we pattern-match on both arguments to determine if they are `Some` or `None`:

```
def apply[A,B](oa: Option[A], oab: Option[A => B]): Option[B] =
  (oa, oab) match {
    case (Some(a), Some(f)) => Some(f(a))
    case _ => None
  }
```

Likewise with `map2` the function argument `f` is only invoked if both of the `Option` arguments are `Some`. The important thing to note is that whether the answer is `Some` or `None` is entirely determined by whether the inputs are both `Some`.

```
def map2[A,B](x: Option[A], y: Option[B])(f: (A, B) => C): Option[C] =
  (x, y) match {
    case (Some(a), Some(b)) => Some(f(a, b))
    case _ => None
  }
```

But in `flatMap`, the second argument is a function that produces an `F`, and its structure *depends* on the value of the `A` from the first argument. So in the `Option` case, we would match on the first argument, and if it's `Some` we apply the function `f` to get a value of type `Option[B]`. And whether *that* result is `Some` or `None` actually depends on the value inside the first `Some`:

```
def flatMap[A,B](oa: Option[A])(f: A => Option[B]): Option[B] =
  oa match {
    case Some(a) => f(a)
    case _ => None
  }
```

One way to put this difference into words is to say that *applicative operations preserve structure* while monadic operations may alter a structure. What does this mean in concrete terms? Well, for example, if you `map` over a `List` with three elements, the result will always also have three elements. But if you `flatMap` over a list with three elements, you may get many more since each function application can introduce a whole `List` of new values.

EXERCISE 2: Transplant the implementations of as many combinators as you can from `Monad` to `Applicative`, using only `map2`, `apply`, and `unit`, or methods implemented in terms of them.

```
def sequence[A](fas: List[F[A]]): F[List[A]]
def traverse[A](as: List[A])(f: A => F[B]): F[List[B]]
def replicateM[A](n: Int, fa: F[A]): F[List[A]]
def factor[A,B](fa: F[A], fb: F[A]): F[(A,B)]
```

12.4 Not all applicative functors are monads

To further illustrate the difference between applicative functors and monads, getting a better understanding of both in the process, we should look at an example of an applicative functor that is not a monad.

In chapter 4, we looked at the `Either` data type and considered the question of how such a data type would have to be modified to allow us to report multiple errors. To make this concrete, think of validating a web form submission. You would want to give an error to the user if some of the information was entered incorrectly, and you would want to tell the user about all such errors at the same time. It would be terribly inefficient if we only reported the first error. The user would have to repeatedly submit the form and fix one error at a time.

This is the situation with `Either` if we use it monadically. First, let's actually write the monad for the partially applied `Either` type.

EXERCISE 3: Write a monad instance for `Either`:

```
def eitherMonad[E]: Monad[({type f[x] = Either[E, x]})#f]
```

Now consider what happens in a sequence of `flatMap`s like this, where each of the functions `validateEmail`, `validPhone`, and `validatePostcode` has type `Either[String, T]` for a given type `T`:

```
validName(field1) flatMap (f1 =>
validBirthdate(field2) flatMap (f2 =>
validPhone(field3) map (WebForm(_, _, _))
```

If `validName` fails with an error, then `validBirthdate` and `validPhone` won't even run. The computation with `flatMap` is inherently sequential and causal. The variable `f1` will never be bound to anything unless `validName` succeeds.

Now think of doing the same thing with `apply`:

```
apply(apply(apply((WebForm(_, _, _)).curried)(
  validName(field1)))(
  validBirthdate(field2)))(
  validPhone(field3))
```

Here, there is no causality or sequence implied. We could run the validation functions in any order, and there's no reason we couldn't run them all first and then combine any errors we get. Perhaps by collecting them in a `List` or `Vector`. But the `Either` monad we have is not sufficient for this. Let's invent a new data type, `Validation`, that is very much like `Either` except that it can explicitly handle more than one error.

```
sealed trait Validation[+E, +A]

case class Failure[E](head: E, tail: Vector[E])
  extends Validation[E, Nothing]

case class Success[A](a: A) extends Validation[Nothing, A]
```

EXERCISE 4: Write an `Applicative` instance for `Validation` that accumulates errors in `Failure`. Notice that in the case of `Failure` there is always at least one error, stored in `head`. The rest of the errors accumulate in the `tail`.

To continue the example above, consider a web form that requires an email address, a phone number, and a post code.

```
case class WebForm(name: String, birthdate: Date, phoneNumber: String)
```

This data will likely be collected from the user as strings, and we must make sure that the data meets a certain specification, or give a list of errors to the user indicating how to fix the problem. The specification might say that `name` cannot be empty, that `birthdate` must be in the form `"yyyy-MM-dd"`, and that `phoneNumber` must contain exactly 10 digits:

```
def validName(name: String): Validation[String, String] =
  if (name != "")
       Success(name)
  else Falure("Name cannot be empty", List())

def validBirthdate(birthdate: String): Validation[String, Date] =
  try {
    import java.text._
    Success((new SimpleDateFormat("yyyy-MM-dd")).parse(birthdate))
  } catch {
    Failure("Birthdate must be in the form yyyy-MM-dd", List())
  }
```

```
def validPhone(phoneNumber: String): Validation[String, String] =
  if (phoneNumber.matches("[0-9]{10}"))
       Success(phoneNumber)
  else Failure("Phone number must be 10 digits")
```

And to validate an entire web form, we can simply lift the `WebForm` constructor with `apply`:

```
def validWebForm(name: String,
                 birthdate: String,
                 phone: String): Validation[String, WebForm] =
  apply(apply(apply((WebForm(_, _, _)).curried)(
    validName(name)))(
    validBirthdate(birthdate)))(
    validPhone(phone))
```

If any or all of the functions produce `Failure`, the whole `validWebForm` method will return all of those failures combined.

12.5 The applicative laws

What sort of laws should we expect applicative functors to obey? Well, we should definitely expect them to obey the functor law:

```
map(v)(x => x) == v
```

What does this mean for applicative functors? Let's remind ourselves of our implementation of `map`:

```
def map[A,B](a: F[A])(f: A => B): F[B] =
  apply(unit(f))(a)
```

This definition of `map` simply shows that an applicative functor *is a functor* in a specific way. And this definition, together with the functor law, imply the first applicative law. This is the *law of identity* for applicative functors:

```
apply(unit(x => x))(v) == v
```

This law demands that `unit` preserve identities. Putting the identity function through `unit` and then `apply` results in the identity function itself. But this really

is just the functor law with an applicative accent, since by our definition above, `map` is the same as `unit` followed by `apply`.

The second thing that the laws demand of applicative functors is that they *preserve function composition*. This is stated as the *composition law*:

```
   apply(apply(apply(unit(a => b => c => a(b(c)))))(g))(x)
== apply(f)(apply(g)(x))
```

Here, `f` and `g` have types like `F[A => B]` and `F[B => C]`, and `x` is a value of type `F[A]`. Both sides of the `==` sign are applying `g` to `x` and then applying `f` to the result of that. In other words they are applying the composition of `f` and `g` to `x`. All that this law is saying is that these two ways of *lifting function composition* should be equivalent. In fact, we can write this more tersely using `map2`:

```
apply(map2(f, g)(_compose _))(x) == apply(f)(apply(g)(x))
```

Or even more tersely than that, using `map3`:

```
map3(f, g, x)(_(_(_)))
```

This makes it much more obvious what's going on. We're lifting the higher-order function `_(_(_))` which, given the arguments `f`, `g`, and `x` will return `f(g(x))`. And the composition law essentially says that even though there might be different ways to implement `map3` for an applicative functor, they should all be equivalent.

The two remaining applicative laws establish the relationship between `unit` and `apply`. These are the laws of *homomorphism* and *interchange*. The homomorphism law states that passing a function and a value through `unit`, followed by idiomatic application, is the same as passing the result of regular application through `unit`:

```
apply(unit(f))(unit(x)) == unit(f(x))
```

We know that passing a function to `unit` and then `apply` is the same as simply calling `map`, so we can restate the homomorphism law:

```
map(unit(x))(f) == unit(f(x))
```

The *interchange law* completes the picture by saying that `unit` should have the same effect whether applied to the first or the second argument of `apply`:

```
apply(u)(unit(y)) == apply(unit(_(y)))(u)
```

The interchange law can be stated a different way, showing a relationship between `map2` and `map` with regard to lifted function application:

```
map2(u, unit(y))(_(_)) == map(u)(_(y))
```

The applicative laws are not surprising or profound. Just like the monad laws, these are simple sanity checks that the applicative functor works in the way that we would expect. They ensure that `apply`, `unit`, `map`, and `map2` behave in a consistent manner.

EXERCISE 5 (optional, hard): Prove that all monads are applicative functors, by showing that the applicative laws are implied by the monad laws.

EXERCISE 6: Just like we can take the product of two monoids `A` and `B` to give the monoid `(A, B)`, we can take the product of two applicative functors. Implement this function:

```
def product[G[_]](G: Applicative[G]):
  Applicative[({type f[x] = (F[x], G[x])})#f]
```

EXERCISE 7 (hard): Applicative functors also compose another way! If `F[_]` and `G[_]` are applicative functors, then so is `F[G[_]]`. Prove this.

```
def compose[G[_]](G: Applicative[G]):
  Applicative[({type f[x] = F[G[x]]})#f]
```

EXERCISE 8 (optional): Try to write `compose` on `Monad`. It is not possible, but it is instructive to attempt it and understand why this is the case.

```
def compose[N[_]](N: Monad[N]): Monad[({type f[x] = M[N[x]]})#f]
```

12.6 Traversable functors

Look again at the signatures of `traverse` and `sequence`:

```
def traverse[F[_],A,B](as: List[A], f: A => F[B]): F[List[B]]
def sequence[F[_],A](fas: List[F[A]]): F[List[A]]
```

Can we generalize this further? Recall that there were a number of data types other than `List` that were `Foldable` in chapter 10. Are there data types other than `List` that are *traversable*? Of course!

EXERCISE 9: On the `Applicative` trait, implement `sequence` over a `Map` rather than a `List`:

```
def sequenceMap[K,V](ofa: Map[K,F[V]]): F[Map[K,V]]
```

But traversable data types are too numerous for us to write specialized `sequence` and `traverse` methods for each of them. What we need is a new interface. We will call it `Traverse`[2]:

Footnote 2 The name `Traversable` is already taken by an unrelated trait in the Scala standard library.

```
trait Traverse[F[_]] {
  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]] =
    sequence(map(fa)(f))
  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

The really interesting operation here is `sequence`. Look at its signature closely. It takes `F[M[A]]` and swaps the order of `F` and `M`. In other words it *commutes* `F` and `M` past each other to give `M[F[A]]`. This is as general as a type signature gets. What `Traverse[F]` is saying is that `F` can be swapped with an `M` inside of it, as long as that `M` is an applicative functor. We have seen lots of examples of this in past chapters, but now we are seeing the general principle.

The `traverse` method, given an `F` full of `A`s, will turn each `A` into an `M[B]` and then use the applicative functor to combine the `M[B]`s into an `F` full of `B`s.

EXERCISE 10: Write `Traverse` instances for `List`, `Option`, and `Tree`:

```

```

```
case class Tree[+A](head: A, tail: List[Tree[A]])
```

At this point you might be asking yourself what the difference is between a traversal and a fold. Both of them take some data structure and apply a function to the data within in order to produce a result. The difference is that `traverse` *preserves the original structure*, while `foldMap` actually throws the structure away and replaces it with the operations of a monoid.

For example, when traversing a `List` with an `Option`-valued function, we would expect the result to always either be `None` or to contain a list of the same length as the input list. This sounds like it might be a law! But we can choose a simpler applicative functor than `Option` for our law. Let's choose the simplest possible one, the identity functor:

```
type Id[A] = A
```

We already know that `Id` is a monad so it's also an applicative functor:

```
val idMonad = new Monad[Id] {
  def unit[A](a: => A) = a
  override def flatMap[A,B](a: A)(f: A => B): B = f(a)
}
```

Then our law, where `xs` is an `F[A]` for some `Traverse[F]`, can be written like this:

```
traverse[Id, A, A](xs)(x => x) == xs
```

If we replace `traverse` with `map` here, then this is just the functor identity law! This means that in the context of the `Id` applicative functor, `map` and `traverse` are the same operation. This implies that `Traverse` can extend `Functor` and we can write a default implementation of `map` in terms of `traverse` through `Id`, and `traverse` itself can be given a default implementation in terms of `sequence` and `map`:

```
trait Traverse[F[_]] extends Functor[F] {
  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]] =
    sequence(map(fa)(f))
```

```
  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
  def map[A,B](fa: F[A])(f: A => B): F[B] =
    traverse[Id, A, B](fa)(f)(idMonad)
}
```

A valid instance of `Traverse` must then override at least either `sequence` or `traverse`.

But what is the relationship between `Traverse` and `Foldable`? The answer involves a connection between `Applicative` and `Monoid`.

12.6.1 From monoids to applicative functors

Take another look at the signature of `traverse`:

```
def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
```

Suppose that our `M` were a type constructor `ConstInt` that takes any type to `Int`, so that `ConstInt[A]` throws away its type argument `A` and just gives us `Int`:

```
type ConstInt[A] = Int
```

Then in the type signature for `traverse`, if we instantiate `M` to be `ConstInt`, it becomes:

```
def traverse[A,B](fa: F[A])(f: A => Int): Int
```

This looks a lot like `foldMap` from `Foldable`. Indeed, if `F` is something like `List` then what we need to implement this signature is a way of combining the `Int` values returned by `f` for each element of the list, and a "starting" value for handling the empty list. In other words, we only need a `Monoid[Int]`. And that's easy to come by.

Indeed, given a constant functor like above, we can turn any `Monoid` into an `Applicative`:

```
type Const[A, B] = A ❶

implicit def monoidApplicative[M](M: Monoid[M]) =
```

```
  new Applicative[({ type f[x] = Const[M, x] })#f] {
    def unit[A](a: => A): M = M.zero
    override def apply[A,B](m1: M)(m2: M): M = M.op(m1, m2)
  }
```

1. This is ConstInt generalized to any A, not just Int.

For this reason, applicative functors are sometimes called "monoidal functors". The operations of a monoid map directly onto the operations of an applicative.

This means that `Traverse` can extend `Foldable` and we can give a default implementation of `foldMap` in terms of `traverse`:

```
override def foldMap[A,B](as: F[A])(f: A => B)(mb: Monoid[B]): B =
  traverse[({type f[x] = Const[B,x]})#f,A,Nothing](
    as)(f)(monoidApplicative(mb))
```

Note that `Traverse` now extends both `Foldable` *and* `Functor`! Importantly, `Foldable` itself cannot extend `Functor`. Even though it's possible to write `map` in terms of a fold for most foldable data structures like `List`, it is not possible *in general*.

EXERCISE 11: Answer, to your own satisfaction, the question of why it's not possible for `Foldable` to extend `Functor`. Can you think of a `Foldable` that is not a functor?

12.7 Uses of Traverse

So what is `Traverse` really for? We have already seen practical applications of particular instances, such as turning a list of parsers into a parser that produces a list. But in what kinds of cases do we want the *generalization*? What sort of generalized library does `Traverse` allow us to write?

12.7.1 Traversals with State

The `State` applicative functor is a particularly powerful one. Using a `State` action to `traverse` a collection, we can implement complex traversals that keep some kind of internal state.

There's an unfortunate amount of type annotation necessary in order to partially apply `State` in the proper way, but traversing with `State` is common enough that we can just have a special method for it and write those type annotations once and for all:

```
```

```
def traverseS[S,A,B](fa: F[A])(f: A => State[S, B]): State[S, F[B]] =
  traverse[({type f[x] = State[S,x]})#f,A,B](fa)(f)(Monad.stateMonad)
```

To demonstrate this, here is a `State` traversal that labels every element with its position. We keep an integer state, starting with 0, and add 1 at each step:

```
def zipWithIndex[A](ta: F[A]): F[(A,Int)] =
  traverseS(ta)((a: A) => (for {
    i <- get[Int]
    _ <- set(i + 1)
  } yield (a, i))).run(0)._1
```

By the same token, we can keep a state of type `List[A]`, to turn any traversable functor into a List:

```
def toList[A](fa: F[A]): List[A] =
  traverseS(fa)((a: A) => (for {
    as <- get[List[A]] ❶
    _  <- set(a :: as) ❷
  } yield ()).run(Nil)._2.reverse
```

❶ Get the current state, the accumulated list.
❷ Add the current element and set the new list as the new state.

It begins with the empty list `Nil` as the initial state, and at every element in the traversal it adds it to the front of the accumulated list. This will of course construct the list in the reverse order of the traversal, so we end by reversing the list we get from running the completed state action. Note that we `yield ()` because in this instance we don't want to return any value other than the state.

Of course, the code for `toList` and `zipWithIndex` is nearly identical. And in fact most traversals with `State` will follow this exact pattern: We get the current state, compute the next state, set it, and yield some value. We should capture that in a function:

```
def mapAccum[S,A,B](fa: F[A], s: S)(f: (A, S) => (B, S)): (F[B], S) =
  traverseS(fa)((a: A) => (for {
    s1 <- get[S]
    (b, s2) = f(a, s)
    _  <- set(s2)
  } yield b)).run(s)

override def toList[A](fa: F[A]): List[A] =
```

```
  mapAccum(fa, List[A]())((a, s) => ((), a :: s))._2.reverse

def zipWithIndex[A](fa: F[A]): F[(A, Int)] =
  mapAccum(fa, 0)((a, s) => ((a, s), s + 1))._1
```

EXERCISE 12: An interesting consequence of being able to turn any traversable functor into a *reversed* list is that we can write, once and for all, a function to reverse any traversable functor! Write this function.

```
def reverse[A](fa: F[A]): F[A]
```

It should obey the following law, for all `x` and `y` of the appropriate types:

```
toList(reverse(x)) ++ toList(reverse(y)) ==
 reverse(toList(y) ++ toList(x))
```

EXERCISE 13: Use `mapAccum` to give a default implementation of `foldLeft` for the `Traverse` trait.

12.7.2 Combining traversable structures

It is in the nature of a traversal that it must preserve the shape of its argument. This is both its strength and its weakness. This is well demonstrated when we try to combine two structures into one.

Given `Traverse[F]` can we combine a value of some type `F[A]` and another of some type `F[B]` into an `F[C]`? We could try using `mapAccum` to write a generic version of `zip`:

```
def zip[A,B](fa: F[A], fb: F[B]): F[(A, B)] =
  (mapAccum(fa, toList(fb)) {
    case (a, Nil) => sys.error("zip: Incompatible shapes.")
    case (a, b :: bs) => ((a, b), bs)
  })._1
```

Notice that this version of `zip` is not able to handle arguments of different "shapes". For example if `F` is `List` then it can't handle lists of different lengths. In this implementation, the list `fb` must be at least as long as `fa`. If `F` is `Tree`, then `fb` must have at least the same number of branches as `fa` at every level.

We can change the generic `zip` slightly and provide two versions so that the shape of one side or the other is dominant:

```
def zipL[A,B](fa: F[A], fb: F[B]): F[(A, Option[B])] =
  (mapAccum(fa, toList(fb)) {
    case (a, Nil) => ((a, None), Nil)
    case (a, b :: bs) => ((a, Some(b)), bs)
  })._1

def zipR[A,B](fa: F[A], fb: F[B]): F[(Option[A], B)] =
  (mapAccum(fb, toList(fa)) {
    case (b, Nil) => ((None, b), Nil)
    case (b, a :: as) => ((Some(a), b), as)
  })._1
```

In the case of `List` for example, the result of `zipR` will have the shape of the `fb` argument, and it will be padded with `None` on the left if `fb` is longer than `fa`. In the case of `Tree`, the result of `zipR` will have the shape of the `fb` tree, and it will have `Some(a)` on the `A` side only where the shapes of the two trees intersect.

12.7.3 Traversal fusion

In the chapter on streams and laziness we talked about how multiple passes over a structure can be fused into one. In the chapter on monoids, we looked at how we can use monoid products to carry out multiple computations over a foldable structure in a single pass. Using products of applicative functors, we can likewise fuse multiple traversals of a traversable structure.

EXERCISE 14: Use applicative functor products to write the fusion of two traversals. This function will, given two functions `f` and `g`, traverse `fa` a single time, collecting the results of both functions at once.

```
def fuse[M[_],N[_],A,B](fa: F[A])(f: A => M[B], g: A => N[B])
                       (M: Applicative[M], N: Applicative[N]):
                       (M[F[B]], N[F[B]])
```

12.7.4 Nested traversals

Not only can we use composed applicative functors to fuse traversals, traversable functors themselves compose. If we have a nested structure like `Map[K,Option[List[V]]]` then we can traverse the map, the option, and the list at the same time and easily get to the `V` value inside, because `Map`, `Option`, and `List` are all traversable.

EXERCISE 15: Implement the composition of two `Traverse` instances.

```

```

```
def compose[G[_]](implicit G: Traverse[G]):
    Traverse[({type f[x] = F[G[x]]})#f]
```

12.7.5 Monad composition

Let's now return to the issue of composing monads. As we saw earlier in this chapter, `Applicative` instances always compose, but `Monad` instances do not. If you tried before to implement general monad composition, then you would have found that in order to implement `join` for nested monads `M` and `N`, you would have to write something of a type like `M[N[M[N[A]]]] => M[N[A]]`. And that can't be written generically. But if `N` also happens to have a `Traverse` instance, we can `sequence` to turn `N[M[_]]` into `M[N[_]]`, leading to `M[M[N[N[A]]]]`. Then we can join the adjacent `M` layers as well as the adjacent `N` layers using their respective `Monad` instances.

EXERCISE 16: Implement the composition of two monads where one of them is traversable:

```
def composeM[M[_],N[_]](M: Monad[M], N: Monad[N], T: Traverse[N]):
    Monad[({type f[x] = M[N[x]]})#f]
```

Expressivity and power often comes at the price of compositionality and modularity. The issue of composing monads is often addressed with a custom-written version of each monad that is specifically constructed for composition. This kind of thing is called a *monad transformer*. For example, the `OptionT` monad transformer composes `Option` with any other monad:

```
case class OptionT[M[_],A](value: M[Option[A]])(implicit M: Monad[M]) {
  def flatMap[B](f: A => OptionT[M, A]): OptionT[M, B] =
    OptionT(value flatMap {
      case None => M.unit(None)
      case Some(a) => f(a).value
    })
}
```

The `flatMap` definition here maps over both the `M` and the `Option`, and flattens structures like `M[Option[M[Option[A]]]]` to just `M[Option[A]]`. But this particular implementation is specific to `Option`. And the general strategy of taking advantage of `Traverse` works only with traversable functors. To compose with `State` for example (which cannot be traversed), a specialized

`StateT` monad transformer has to be written. There is no generic composition strategy that works for every monad.

See the chapter notes for more information about monad transformers.

12.8 Summary

Applicative functors are a very useful abstraction that is highly modular and compositional. The functions `unit` and `map` allow us to lift functions and values, while `apply` and `map2` give us the power to lift functions of higher arities. This in turn enables `traverse` and `sequence` to be generalized to traversable functors. Together, `Applicative` and `Traverse` let us construct complex nested and parallel traversals out of simple elements that need only be written once.

This is in stark contrast to the situation with monads, where each monad's composition with others becomes a special case. It's a wonder that monads have historically received so much attention and applicative functors have received so little.

Index Terms

applicative functor laws
composition law
homomorphism law
identity law
interchange law
law of composition
law of homomorphism
law of identity
law of interchange
laws of applicative functors
monad transformer

13 External effects and I/O

13.1 Introduction

In this chapter, we will introduce the I/O monad (usually written 'the `IO` monad'), which extends what we've learned so far to handle *external effects*, like writing to a file, reading from a database, etc, in a purely functional way. The `IO` monad will be important for two reasons:

- It provides the most straightforward way of embedding imperative programming into FP, while preserving referential transparency and keeping pure code separate from what we'll call *effectful* code. We will be making an important distinction here in this chapter between *effects* and *side effects*.
- It illustrates a key technique for dealing with external effects—using pure functions to compute a *description* of an imperative computation, which is then executed by a separate *interpreter*. Essentially, we are crafting an embedded domain specific language (EDSL) for imperative programming. This is a powerful technique we'll be using throughout part 4; part of our goal is to equip you with the skills needed to begin crafting your own descriptions for interesting effectful programs you are faced with.

13.2 Factoring effects

We are going to work our way up to the `IO` monad by first considering a very simple example, one we discussed in chapter 1 of this book:

```
case class Player(name: String, score: Int)

def printWinner(p: Player): Unit =
  println(p.name + " is the winner!")

def declareWinner(p1: Player, p2: Player): Unit =
  if (p1.score > p2.score) printWinner(p1)
  else printWinner(p2)
```

In chapter 1, we factored out the logic for computing the winner into a function separate from displaying the winner:

```
def printWinner(p: Player): Unit =
  println(p.name + " is the winner!")

def winner(p1: Player, p2: Player): Player =
  if (p1.score > p2.score) p1 else p2

def declareWinner(p1: Player, p2: Player): Unit =
  printWinner(winner(p1, p2))
```

We commented in chapter 1 that it was *always* possible to factor an impure function into a pure 'core' and two functions with side effects, (possibly) one that produces the pure function's input and (possibly) one that accepts the pure function's output. In this case, we factored the pure function `winner` out of `declareWinner`—conceptually, `declareWinner` was doing *two things*, it was *computing* the winner, and it was *displaying* the winner that was computed.

Interestingly, we can continue this sort of factoring—the `printWinner` function is also doing two things—it is computing a message, and then printing that message to the console. We could factor out a pure function here as well, which might be beneficial if later on we decide we want to display the winner message in some sort of UI or write it to a file instead:

```
def winnerMsg(p: Player): String =
  p.name + " is the winner!"

def printWinner(p: Player): Unit =
  println(winnerMsg(p))
```

These might seem like silly examples, but the same principles apply in larger, more complex programs and we hope you can see how this sort of factoring is quite natural. We aren't changing what our program does, just the internal details of how it is factored into smaller functions. The insight here is that *inside every function with side effects is a pure function waiting to get out*. We can even formalize this a bit. Given an impure function of type `A => B`, we can often split this into two functions:[1]

Footnote 1 We will see many more examples of this in this chapter and in the rest of part 4.

- A *pure* function `A => D`, where `D` is some *description*
- An *impure* function `D => B`, which can be thought of as an *interpreter* of these descriptions

We will extend this to handle 'input' side effects shortly. For now, though, consider applying this strategy repeatedly to a program. Each time we apply it, we make more functions pure and push side effects to the outer layers of the program. We sometimes call these impure functions the 'imperative shell' around the pure core of the program. Eventually, we reach functions that seem to *necessitate* side effects like the built-in `println`, which has type `String => Unit`. What do we do then?

13.3 A simple IO type

It turns out that even functions like `println` can be factored into a pure core, by introducing a new data type we'll call `IO`:

```
trait IO { def run: Unit }

def PrintLine(msg: String): IO =
  new IO { def run = println(msg) }

def printWinner(p: Player): IO =
  PrintLine(winnerMsg(p))
```

Our `printWinner` function is now pure—it returns an `IO` value, which simply describes an action that is to take place without actually *executing* it. We say that `printWinner` has or produces an *effect* or is *effectful*, but it is only the interpreter of `IO` (its `run` function) that actually has *side* effects.

Other than technically satisfying the requirements of FP, has the `IO` type actually bought us anything? This is a subjective question, but as with any other data type, we can access the merits of `IO` by considering what sort of algebra it provides—is it something interesting, from which we can define a large number of useful operations and assemble useful programs, with nice laws that give us the ability to reason about what these larger programs will do? Not really. Let's look at the operations we can define:

```
trait IO { self =>
  def run: Unit
  def ++(io: IO): IO = new IO {
    def run = { self.run; io.run }
```

```
  }
}
object IO {
  def empty: IO = new IO { def run = () }
}
```

The only thing we can say about `IO` as it stands right now is that it forms a `Monoid` (`empty` is the identity, and `++` is the associative operation). So if we have for instance a `List[IO]`, we can reduce that to an `IO`, and the associativity of `++` means we can do this by folding left or folding right. On its own, this isn't very interesting. All it seems to have given us is the ability to delay when a side effect gets 'paid for'.

Now we will let you in on a secret—you, as the programmer, get to invent whatever API you wish to represent your computations, including those that interact with the universe external to your program. This process of crafting pleasing, useful, and composable descriptions of what you want your programs to do is at its core *language design*. You are crafting a little language and an associated *interpreter* that will allow you to express various programs. If you don't like something about this language you have created, change it! You should approach this task just like any other combinator library design task, and by now you've had plenty of experience designing and writing such libraries.

13.3.1 Handling input effects

As we have seen many times before, sometimes, when building up your little language, you will encounter a program that cannot be expressed. So far, our `IO` type can represent only 'output' effects. There is no way to express `IO` computations that must, at various points, wait for input from some external source. Suppose we would like to write a program that prompts the user for a temperature in degrees fahrenheit, then converts this value to celsius and echoes it to the user. A typical imperative program might look something like:[2]

Footnote 2 We are not doing any sort of error handling here. This is just meant to be an illustrative example.

```
def farenheitToCelsius(f: Double): Double =
  (f - 32) * 5.0/9.0

def converter: Unit = {
  println("Enter a temperature in degrees fahrenheit: ")
  val d = readLine.toDouble
  println(fahrenheitToCelsius(d))
}
```

Unfortunately, we run into problems if we want to make `converter` into a pure function that returns an `IO`:

```
def fahrenheitToCelsius(f: Double): Double =
  (f - 32) * 5.0/9.0

def converter: IO = {
  val prompt: IO = PrintLine(
    "Enter a temperature in degrees fahrenheit: ")
  // now what ???
}
```

In Scala, `readLine` is a `def` with a side effect of capturing a line of input from the console. It returns a `String`. We could wrap a call to this in an `IO`, but we have nowhere to put the result! We don't yet have a way of representing this sort of effect with our current `IO` type. The problem is we cannot express `IO` computations that *yield a value* of some meaningful type—our interpreter of `IO` just produces `Unit` as its output. Should we give up on our `IO` type and resort to using side effects? Of course not! We extend our `IO` type to allow *input*, by adding a type parameter to `IO`:

```
trait IO[+A] { self =>
  def run: A
  def map[B](f: A => B): IO[B] =
    new IO[B] { def run = f(self.run) }
  def flatMap[B](f: A => IO[B]): IO[B] =
    new IO[B] { def run = f(self.run).run }
}
```

An `IO` computation can now return a meaningful value. Notice we've added `map` and `flatMap` functions so `IO` can be used in for-comprehensions. And `IO` now forms a `Monad`:

```
object IO extends Monad[IO] {
  def unit[A](a: => A): IO[A] = new IO[A] { def run = a }
  def flatMap[A,B](fa: IO[A])(f: A => IO[B]) = fa flatMap f
  def apply[A](a: => A): IO[A] = unit(a) ❶
}
```

❶ syntax for IO ..

We can now write our `converter` example:

```
def ReadLine: IO[String] = IO { readLine }
def PrintLine(msg: String): IO[Unit] = IO { println(msg) }

def converter: IO[Unit] = for {
  _ <- PrintLine("Enter a temperature in degrees fahrenheit: ")
  d <- ReadLine.map(_.toDouble)
  _ <- PrintLine(fahrenheitToCelsius(d).toString)
} yield ()
```

Our `converter` definition no longer has side effects—it is a RT *description* of a computation that will have side effects when interpreted via `converter.run`. And because it forms a `Monad`, we can use all the combinators we've written previously. Here are some other example usages of `IO`:

- `val echo = ReadLine.flatMap(PrintLine)`: An `IO[Unit]` that reads a line from the console and echoes it back.
- `val readInt = ReadLine.map(_.toInt)`: An `IO[Int]` that parses an `Int` by reading a line from the console.
- `val readInts = readInt ** readInt`: An `IO[(Int,Int)]` that parses an `(Int,Int)` by reading two lines from the console.
- `replicateM_(5)(converter)`: An `IO[Unit]` that will repeat `converter` 5 times, discarding the results (which are just `Unit`).[3] We can replace `converter` here with any `IO` action we wished to repeat 5 times (for instance, `echo` or `readInts`).

  Footnote 3 Recall that `replicateM(3)(fa)` is the same as `sequence(List(fa,fa,fa))`.

- `replicateM(10)(ReadLine)`: An `IO[List[String]]` that will read 10 lines from the console and return the list of results.

Here's a larger example, an interactive program which prompts the user for input in a loop, then computes the factorial of the input. Here's an example run:

```
The Amazing Factorial REPL, v2.0
q - quit
<number> - compute the factorial of the given number
<anything else> - bomb with horrible error
3
factorial: 6
7
factorial: 5040
q
```

This code uses a few `Monad` functions we haven't seen yet, `when`, `foreachM`,

and `sequence_`, discussed in the sidebar below. For the full listing, see the associated chapter code. The details of this code aren't too important; the point here is just to demonstrate how we can embed an imperative programming language into the purely functional subset of Scala. All the usual imperative programming tools are here—we can write loops, perform I/O, and so on. If you squint, it looks a bit like normal imperative code.

```
def factorial(n: Int): IO[Int] = for { ❶
  acc <- ref(1) ❷
  _ <- foreachM (1 to n toStream) (i => acc.modify(_ * i).skip) ❸
  result <- acc.get ❹
} yield result

val factorialREPL: IO[Unit] = sequence_(
  IO { println(helpstring) },
  doWhile { IO { readLine } } { line => ❺
    val ok = line != "q"
    when (ok) { for {
      n <- factorial(line.toInt)
      _ <- IO { println("factorial: " + n) }
    } yield () }
  })
```

❶ Imperative factorial using a mutable IO reference
❷ Allocation a mutable reference
❸ Modify reference in a loop
❹ Dereference to obtain the value inside a reference
❺ See sidebar

SIDEBAR **Additional monad combinators**

The above example makes use of some monad combinators we haven't seen before, although they can be defined for any `Monad`. You may want to think about what these combinators mean for types other than `IO`. Notice that not all these combinators make sense for every monadic type. (For instance, what does `forever` mean for `Option`? For `Stream`?)

```
def when[A](b: Boolean)(fa: => F[A]): F[Boolean] =
  if (b) as(fa)(true) else unit(false)
def doWhile[A](a: F[A])(cond: A => F[Boolean]): F[Unit] = for {
  a1 <- a
  ok <- cond(a1)
  _ <- if (ok) doWhile(a)(cond) else unit(())
} yield ()
def forever[A,B](a: F[A]): F[B] = {
  lazy val t: F[B] = forever(a)
  a flatMap (_ => t)
}
def foreachM[A](l: Stream[A])(f: A => F[Unit]): F[Unit] =
  foldM_(l)(())((u,a) => skip(f(a)))
def foldM_[A,B](l: Stream[A])(z: B)(f: (B,A) => F[B]): F[Unit] =
  skip { foldM(l)(z)(f) }
def foldM[A,B](l: Stream[A])(z: B)(f: (B,A) => F[B]): F[B] =
  l match {
    case h #:: t => f(z,h) flatMap (z2 => foldM(t)(z2)(f))
    case _ => unit(z)
  }
```

We don't necessarily endorse writing code this way.[4] What this does demonstrate, however, is that FP is not in any way limited in its expressiveness—any program that can be expressed can be expressed in FP, even if that functional program is a straightforward embedding of the imperative program into the `IO` monad.

Footnote 4 If you have a monolithic block of impure code like this, you can always just write a definition which performs actual side effects then wrap it in `IO`—this will be more efficient, and the syntax is nicer than what is provided using a combination of for-comprehension syntax and the various `Monad` combinators.

13.3.2 Has this bought us anything?

The `IO` monad is a kind of least common denominator for expressing programs with external effects. Its usage is important mainly because it *clearly separates pure code from impure code*, forcing us to be honest about where interactions with the outside world are occurring and also encouraging the beneficial factoring of effects that we discussed earlier. When programming *within* the `IO` monad, we have many of the same problems and difficulties as we would in ordinary imperative programming, which has motivated functional programmers to search for more composable ways of describing effectful programs.[5] Nonetheless, the `IO` monad does provide some real benefits:

Footnote 5 As we'll see in chapter 15, we don't need to give up on all the nice compositionality we've come to expect from FP just to interact with the outside world.

- `IO` computations are ordinary *values*. We can store them in lists, pass them to functions, create them dynamically, etc. Any common pattern we notice can be wrapped up in a function and reused. This is one reason why it is sometimes argued that functional languages provide *more powerful tools for imperative programming*, as compared to languages like C or Java.
- Reifying `IO` computations as values means we can craft a more interesting interpreter than the simple `run`-based "interpreter" baked into the `IO` type itself. Later on in this chapter, we will build a more refined `IO` type and sketch out an interpreter that uses nonblocking I/O in its implementation. Interestingly, client code like our `converter` example remains identical—we do not expose callbacks to the programmer at all! They are entirely an implementation detail of our `IO` interpreter.

SIDEBAR **IO computation reuse in practice**

In practice, the amount of reuse we get by factoring out common patterns of `IO` usage is limited (you will notice this yourself if you start writing programs with `IO`). An `IO[A]` is a completely opaque description of a computation that yields a single `A`. Most of the general purpose combinators for `IO` are functions that can be defined for any monad—`IO` itself is not contributing anything new, which means we only have the monad laws to reason with.

What this means in practice is that we generally need to know something more about an `IO[A]` than just its type to compose it with other computations (contrast this with, say, `Stream[A]`). Reuse is extremely limited as a result.

13.4 The meaning of the IO type

In this section, we are going to explore what an `IO[A]` *means* and introduce a more nuanced `IO` type that clarifies what is actually going on in a functional program that performs I/O. A primary purpose here is to get you thinking, and to make it clear how it is possible to craft more interesting interpreters of `IO` than the simple one we baked into the `run` function. Don't worry too much about following absolutely everything in this section.

With that said, let's see what we can learn about the meaning of `IO`. You may have noticed that our `IO` type is roughly equivalent to our `Id` type, introduced in chapter 13 as the simplest possible type that gives rise to a monad. Let's compare the two types side-by-side:

```
case class Id[+A](value: A)
trait IO[+A] { def run: A }
```

Aside from the fact that `IO` is non-strict, and that retrieving the value is done using `run` instead of `value`, the types are identical! Our "IO" type is just a non-strict value. This is rather unsatisfying. What's going on here?

We have actually cheated a bit with our `IO` type. We are relying on the fact that Scala allows unrestricted side effects at any point in our programs. But let's think about what happens when we evaluate the `run` function of an `IO`. During evaluation of `run`, the pure part of our program will occasionally make requests of the outside world (like when it invokes `readLine`), wait for a result, and then pass this result to some further pure computation (which may subsequently make some further requests of the outside world). Our current `IO` type is completely inexplicit about where these interactions are occurring. But we can model these interactions more explicitly if we choose:

```
trait IO[+A]
case class Pure[+A](a: A) extends IO[A]
case class Request[I,+A](expr: External[I],
                         receive: I => IO[A]) extends IO[A]
```

This type separates the pure and effectful parts of an `IO` computation. We'll get to writing its `Monad` instance shortly. An `IO[A]` can be a pure value, or it can be a request of the external computation. The type `External` defines the *protocol*

—it encodes what possible external requests our program can make. We can think of `External[I]` much like an expression of type `I`, but it is an expression that is "external" that must be evaluated by whatever program is running this `IO` action. The `receive` function defines what to do when the result of the request becomes available, it is sometimes called the *continuation*. We'll see a bit later how this can be exploited to write an interpreter for `IO` that uses nonblocking I/O internally.

The simplest possible representation of `External[I]` would be simply a nonstrict value:[6]

Footnote 6 Similar to our old `IO` type!

```
trait Runnable[A] { def run: A }
object Delay { def apply[A](a: => A) = new Runnable[A] { def run = a }}
```

This implies that our `IO` type can call absolutely any impure Scala function, since we can wrap any expression at all in `Delay` (for instance, `Delay { println("Side effect!") }`). If we want to restrict access to only certain functions, we can parameterize our `IO` type on the choice of `External`.[7]

Footnote 7 Of course, Scala will not technically prevent us from invoking a function with side effects at any point in our program. This discussion assumes we are following the discipline of not allowing side effects unless this information is tracked in the type.

```
trait IO[F[_], +A]
case class Pure[F[_], +A](get: A) extends IO[F,A]
case class Request[F[_], I, +A](
    expr: F[I],
    receive: I => IO[F,A]) extends IO[F,A]
```

We have renamed `External` to just `F` here. With this representation, we can define an `F` type that grants access to exactly the effects we want:

```
trait Console[A]
case object ReadLine extends Console[Option[String]]
case class PrintLine(s: String) extends Console[Unit]
```

Now an `IO[Console,A]` is an `IO` computation that can only read from and write to the console.[8] We can introduce other types for different I/O capabilities—a file system `F`, granting read/write access (or even just read access) to the file

system, a network `F` granting the ability to open network connections and read from them, and so on. Notice, interestingly, that nothing about `Console` implies that any side effects must actually occur! That is a property of the *interpreter* of `F` values now required to actually run an `IO[F,A]`:

Footnote 8

```
def printWinner(p: Player): Unit =
  println(p.name + " is the winner!")

def winner(p1: Player, p2: Player): Player =
  if (p1.score > p2.score) p1 else p2

def declareWinner(p1: Player, p2: Player): Unit =
  printWinner(winner(p1, p2))
```

```
trait Run[F[_]] {
  def apply[A](expr: F[A]): (A, Run[F])
}

object IO {
  @annotation.tailrec
  def run[F[_],A](R: Run[F])(io: IO[F,A]): A = io match {
    case Pure(a) => a
    case Request(expr,recv) =>
      R(expr) match { case (e,r2) => run(r2)(recv(e)) }
  }
}
```

A completely valid `Run[Console]` could ignore `PrintLine` requests and always return the string `"Hello world!"` in response to `ReadLine` requests:

```
object RunConsoleMock extends Run[Console] {
  def apply[A](c: Console[A]) = c match {
    case ReadLine => (Some("Hello world!"), RunConsoleMock)
    case PrintLine(_) => ((), RunConsoleMock) ❶
  }
}
```

❶ Ignored!

While the real `Run[Console]` could actually execute the effects:

```
object RunConsole extends Run[Console] {
```

```
  def apply[A](c: Console[A]) = c match {
    case ReadLine =>
      val r = try Some(readLine) catch { case _ => None }
      (r, RunConsole)
    case PrintLine(s) => (println(s), RunConsole)
  }
}
```

EXERCISE 1: Give the `Monad` instance for `IO[F,_]`.[9] You may want to override the default implementations of various functions with more efficient versions.

Footnote 9 Note we must use the same trick discussed in chapter 10, to partially apply the `IO` type constructor.

```
def monad[F[_]] = new Monad[({ type f[a] = IO[F,a]})#f] {
  ...
}
```

EXERCISE 2: Implement a `Run[Console]` which uses elements from a `List[String]` to generate results from calls to `ReadLine` (it can ignore `PrintLine` calls):

```
def console(lines: List[String]): Run[Console]
```

EXERCISE 3: Our definition of `Run` is overly specific. We only need a `Monad[F]` to implement `run`! Implement `run` given an arbitrary `Monad[F]`.[10] With this definition, the interpreter is no longer tail recursive, and it is up to the `Monad[F]` to ensure that the chains of `flatMap` calls built up during interpretation do not result in stack overflows. We will discuss this further in the next section.

Footnote 10 Note on this signature: when passing a `Monad[F]` or `Monad[G]` to a function, we typically give the variable the same name as its type parameter (`F` or `G` in this case). We can then write `F.map(expr)(f)` almost as if `F` were the companion object of a type `F`.

```
def run[F[_],A](F: Monad[F])(io: IO[F,A]): F[A]
```

These last two exercises demonstrate something rather amazing—there is nothing about our `IO` type nor our `Run[F]` or `F` which require side effects of any

kind. From the perspective of our `IO` programs, we cannot even *distinguish* an interpreter that uses 'real' side effects from one like `console` that doesn't. `IO` values are simply pure computations that may occasionally make requests of some interpreter that exists outside the universe established by the program.

With our `IO` type, the behavior of this external interpreter is modeled only in a very opaque way—we say that each evaluation we request from the interpreter results in a new interpreter state and beyond that make no assumptions about how the sequence of requests made will affect future behavior.[11] As we will learn in chapter 15, creating more composable abstractions for dealing with I/O and other effects is all about designing types that codify additional assumptions about the nature of the interactions between the 'internal' and 'external' computations.

Footnote 11 Our definition for an arbitrary `Monad[F]` is telling us the same thing, since a `Monad[F]` provides us with a strategy for sequencing programs, but makes no additional assumptions about the nature of this sequencing.

13.5 A realistic I/O data type and interpreter

In this section we will develop an `IO` type and associated interpreter that uses nonblocking I/O internally. We will also demonstrate a general technique, called *trampolining*, for avoiding stack overflow errors in monadic code.[12] Trampolining is particularly important for `IO` since `IO` is often used for defining top-level program loops and other large or long-running computations that are susceptible to excessive stack usage.

Footnote 12 See chapter notes for more discussion and links to further reading on this.

13.5.1 A trampolined IO type and a tail-recursive interpreter

Our current `IO` type will stack overflow for some programs. In this section, we will develop an `IO` type and a `run` function that uses *constant stack space*. By this we mean that the stack usage of `run` is not dependent on the input in any way. To start, let's look at a few examples of `IO` programs that can be problematic. We'll make use of a helper function, `IO.apply`, which constructs an `IO[Runnable,A]` from a nonstrict `A`:

```
object IO {
  ...
  def apply[A](a: => A): IO[Runnable,A] =
    Request(Delay(a), (a:A) => Pure(a))
}
```

Here are some perfectly reasonable `IO` programs that can present problems:

```
val F = monad[Runnable]; import F._ ❶
val ex1 = sequence_(Stream.fill(100000)(IO { math.random })) ❷
val ex2 = foreachM(Stream.range(1,100000))(i => IO { println(i) }) ❸
val ex3 = forever(IO { println("hi") }) ❹
val ex4 = Stream.fill(100000)("hi").foldLeft(IO { () })( ❺
  (acc,s) => acc *> IO { println(s) })
```

❶ Bring Monad combinators into scope
❷ Generates a large list of random values
❸ Prints the numbers from 1 to 100,000
❹ Prints "hi" in an infinite loop
❺ Prints out "hi" 100,000 times, but the IO action is built up using a left fold so that calls to flatMap are associated to the left

EXERCISE 4 (hard): Try running these examples for your current implementation of `IO`, and also (optional) for the simple `IO` type we defined first in this chapter. For those that fail, can you explain why? It can be a little difficult to tell what's going on just by looking at the stack trace. You may want to instead trace on paper the evaluation of these expressions until a pattern becomes clear.

To fix the problem, we are going to build what is called a *trampolined* version of `IO`. To illustrate the idea, let's first look at the type `Trampoline`.[13]

Footnote 13 We are not going to work through derivation of this type, but if you are interested, follow the references in the chapter notes for more information.

```
trait Trampoline[+A]
case class Done[+A](get: A) extends Trampoline[A]
case class More[+A](force: () => Trampoline[A]) extends Trampoline[A]
case class Bind[A,+B](force: () => Trampoline[A],
                      f: A => Trampoline[B]) extends Trampoline[B]
```

What is `Trampoline`? First, note that we can extract an `A` from a `Trampoline[A]` using a tail-recursive function, `run`:

EXERCISE 5: Write the tail-recursive function, `run`, to convert a `Trampoline[A]` to an `A`. Just follow the types—there is only one implementation that typechecks!

```
@annotation.tailrec
def run[A](t: Trampoline[A]): A
```

So a `Trampoline[A]` can be used in place of `A`. But even more interestingly, `Trampoline` forms a `Monad`, and if implemented correctly, arbitrary monadic expressions can be guaranteed not to stack overflow.

EXERCISE 6 (hard): Implement `Monad[Trampoline]`. (You may want to add `map` and `flatMap` methods to `Trampoline`.) A valid implementation will never evaluate `force()` or call `run` in the implementation of `flatMap` or `map`.

Can you see how it works? `More` lets us return control to the `run` function, and `Bind` represents a call to `flatMap` as an ordinary value. If implemented correctly, `flatMap` will always return control to the `run` function after a constant number of steps, and the `run` function is guaranteed to make progress.

EXERCISE 7 (hard, optional): Show that `flatMap` always returns after doing a constant amount of work, and that `run` will always call itself after a single call to `force()`.

Can you see why this is called trampolining? The name comes from the fact that interpretation of a trampolined program bounces back and forth between the central `run` loop and the functions being called. `Trampoline` is essentially providing an interpreter for function application that uses the heap instead of the usual call stack. The key to making it work is that we reify `flatMap` calls as data, but associate these calls to the right, which lets us implement the `run` function as a tail-recursive function. There is some overhead to using it, but its advantage is that we gain predictable stack usage.[14]

Footnote 14 There are some interesting optimizations to this scheme—it isn't necessary to return to the central loop after every function call, only periodically to avoid stack overflows. See the chapter notes for more information.

We can now add the same trampolining behavior directly to our `IO` type:

```
trait IO[F[_], +A]
case class Pure[F[_], +A](get: A) extends IO[F,A]
case class Request[F[_],I,+A](
    expr: F[I],
    receive: I => IO[F,A]) extends IO[F,A]
case class BindMore[F[_],A,+B](
    force: () => IO[F,A],
    f: A => IO[F,B]) extends IO[F,B]
case class BindRequest[F[_],I,A,+B](
    expr: F[I], receive: I => IO[F,A],
    f: A => IO[F,B]) extends IO[F,B]
case class More[F[_],A](force: () => IO[F,A]) extends IO[F,A]
```

We have a `More` constructor, as before, but we now have two additional `Bind` constructors.

EXERCISE 8: Implement both versions of `run` for this new version of `IO`.

```
def run[F[_],A](R: Run[F])(io: IO[F,A]): A
def run[F[_],A](F: Monad[F])(io: IO[F,A]): F[A]
```

EXERCISE 9 (hard): Implement `Monad` for this new version of `IO`. Once again, a correct implementation of `flatMap` will not invoke `run` or `force()`. Test your implementation using the examples we gave above. Can you see why it works? Again, you may want to try tracing their execution until the pattern becomes clear, or even construct a proof that stack usage is guaranteed to be bounded.

```
def monad[F[_]]: Monad[({ type f[x] = IO[F,x]})#f]
```

SIDEBAR **Enabling infix monadic syntax for IO**

When compiling an expression like `a.map(f)`, if `a` does not have a function `map`, Scala will look in the companion object of the `a` type for an implicit conversion to some type with a `map` function and call that if such a conversion exists. We can use this to expose infix syntax for all the various `Monad` combinators—we just add the following function to the `IO` companion object:

```
implicit def toMonadic[F[_],A](a: IO[F,A]) = monad[F].toMonadic(a)
```

With this definition in the companion object, we we can write expressions like `a replicateM 10`, `a as "hello!"`, `(a map2 b)(_ + _)` for any `IO` values `a` and `b`.

13.5.2 A nonblocking I/O interpreter

The `F` type used for `IO` frequently includes operations that can take a long time to complete and which do not occupy the CPU, like accepting a network connection from a server socket, reading a chunk of bytes from an input stream, writing a large number of bytes to a file, and so on. Our current `IO` `run` function will wait idly for these operations to complete:

```
case Request(expr, recv) =>
  val (expensive, r2) = R(expr) ❶
  run(r2)(recv(expensive))
```

❶ Might take a long time

We can do better. There are I/O libraries that support *nonblocking* I/O. The details of these libraries vary, but to give the general idea, a nonblocking *source* of bytes might have an interface like this:

```
trait Source {
  def requestBytes(
    nBytes: Int,
    callback: Either[Throwable, Array[Byte]] => Unit): Unit
}
```

Here, it is assumed that `requestBytes` returns immediately. We give `requestBytes` a callback function indicating what to do when the result becomes available or an error is encountered. Using this sort of library directly is quite painful, for obvious reasons.[15]

Footnote 15 Even this API is rather nicer than what is offered directly by the `nio` package in Java (API link), which supports nonblocking I/O.

One of the nice things about making I/O computations into values is that we can build more interesting interpreters for these values than the default 'interpreter' which performs I/O actions directly. Here we will sketch out an interpreter that uses nonblocking I/O internally. The ugly details of this interpreter are completely hidden from code that uses the `IO` type, and code that uses `IO` can be written in a much more natural style without explicit callbacks.

Recall that earlier, we implemented a function with the following signature:

```
def run[F[_],A](F: Monad[F])(io: IO[F,A]): F[A]
```

We are going to simply choose `Future` for our `F`, though we will be using a different definition than the `java.util.concurrent.Future` introduced in chapter 7. The version we use here can be backed by a nonblocking I/O primitive. It is trampolined, as we've seen before.[16]

Footnote 16 This definition can be extended to handle timeouts, cancellation, and deregistering callbacks. We won't be discussing these extensions here, but you may be interested to explore this on your own. See the `Task.scala` for an extension to `Future` supporting proper error handling.

```
trait Future[+A]
object Future {
  case class Now[+A](get: A) extends Future[A]
  case class Later[+A](listen: (A => Unit) => Unit) extends Future[A]
  case class More[+A](force: () => Future[A]) extends Future[A]
  case class BindLater[A,+B](listen: (A => Unit) => Unit,
                             f: A => Future[B]) extends Future[B]
  case class BindMore[A,+B](force: () => Future[A],
                            f: A => Future[B]) extends Future[B]
}
```

`Future` looks almost identical to `IO`, except we have replaced the `Request` and `BindRequest` cases with `Later` and `BindLater` constructors. The `listen` function of `Later` is presumed to have some side effect—perhaps adding `callback` to a mutable list of listeners to notify when the result becomes available. We will only be using it here as an implementation detail of the `run` function for `IO` (this could be enforced using access modifiers).

EXERCISE 10 (hard, optional): Implement `Monad[Future]`. We will need it to implement our nonblocking `IO` interpreter. Also implement `runAsync`, for an asynchronous evaluator for `Future`, and `run`, the synchronous evaluator:

```
def runAsync[A](a: Future[A])(onFinish: A => Unit): Future[Unit]
def run[A](a: Future[A]): A
```

With this in place, we can asynchronously evaluate any `IO[Future,A]` to a `Future[A]`. But what do we do if we have an `IO[Console,A]`, say, and we wish to evaluate it asynchronously to a `Future[A]`? We need a way to translate between `IO[Console,A]` and `IO[Future,A]`, which we can think of as a form of compilation—we are replacing the abstract requests for `Console` with concrete requests that will actually read from and write to the standard input and output streams. We will use the following `trait`:

```
trait Trans[F[_], G[_]] {
  def apply[A](f: F[A]): G[A]
}
```

EXERCISE 11: Implement a version of `run` which translates from `F` to `G` as it interprets the `IO` action:

```
def run[F[_],G[_],A](T: Trans[F,G])(G: Monad[G])(io: IO[F,A]): G[A]
```

With this in place, we can now write the following to asynchronously evaluate an `IO[Console,A]`:

```
val RunConsole: Trans[Console,Future] = ...
val prog: IO[Console,A] = ...
val result: Future[A] = run(RunConsole)(Future)(prog)
```

That's it! In general, for any `F` type, we just require a way to translate that `F` to a `Future` and we can then evaluate `IO[F,A]` programs asynchronously. Let's look at some possible definitions of `RunConsole`:

First, if we wish, we can always implement `RunConsole` using ordinary blocking I/O calls:[17]

Footnote 17 Recall that `Future.unit` doesn't do anything special with its argument—the argument will still be evaluated in the main thread using ordinary function calls.

```
object RunConsole extends Trans[Console,Future] {
  def apply[A](c: Console[A]): Future[A] =
    c match {
      case ReadLine =>
        Future.unit {
          try Some(readLine)
          catch { case _: Exception => None }
        }
      case PrintLine(a) =>
        Future.unit { println(a) }
    }
}
```

But because we are returning a `Future`, we have more flexibility. Here, we delegate `ReadLine` interpretation to a shared thread pool, to avoid blocking the main computation thread while waiting for input. Notice that nothing requires us to implement nonblocking calls everywhere in our interpreter—`PrintLine` uses the ordinary blocking `println` call directly (with the assumption that submitting this as a task to a thread pool is not worth the overhead).

```
object RunConsole extends Trans[Console,Future] {
  def apply[A](c: Console[A]): Future[A] =
    c match {
      case ReadLine =>
        Future {
          try Some(readLine)
          catch { case _: Exception => None }
        }
      case PrintLine(a) =>
        Future.unit { println(a) }
    }
}
```

In the `Future` companion object, we define an `apply` function, which evaluates its argument in a thread pool. It returns a `Later` which notifies any listeners once the result is available. The implementation is somewhat involved; see the answer code for this chapter if you are interested.

```
object Future {
  ...
  def apply[A](a: => A): Future[A] = { ... }
}
```

EXERCISE 12 (hard, optional): Going one step further, we can use an API that directly supports nonblocking I/O. We are not going to work through such an implementation here, but you may be interested to explore this on your own by building off the `java.nio` library (API link. As a start, try implementing an asynchronous read from an `AsynchronousFileChannel` (API link).[18]

Footnote 18 This requires Java 7.

```
def read(file: AsynchronousFileChannel,
         fromPosition: Long,
         nBytes: Int): Future[Either[Throwable, Array[Byte]]]
```

What is remarkable is that regardless of the implementation, we get to retain the same straightforward style in code that uses `IO`, rather than being forced to program with callbacks directly.

13.6 Why the IO type is insufficient for streaming I/O

Despite the fancy implementation of our `IO` interpreter and the advantage of having first-class `IO` values, the `IO` type fundamentally present us the same level of abstraction as ordinary imperative programming. This means that writing efficient, streaming I/O will generally involve monolithic loops.

Let's look at an example. Suppose we wanted to write a program to convert a file, `fahrenheit.txt`, containing a sequence of temperatures in degrees fahrenheit, separated by blank lines, to a new file, `celsius.txt`, containing the same temperatures in degrees celsius. An `F` type for this might look something like:[19]

Footnote 19 We are ignoring exception handling in this API.

```
trait Files[A]
case class ReadLines(file: String) extends Files[List[String]]
case class WriteLines(file: String, lines: List[String])
  extends Files[Unit]
```

```
for {
  lines <- request(ReadLines("fahrenheit.txt"))
  cs = lines.map(s => fahrenheitToCelsius(s.toDouble).toString)
  _ <- request(WriteLines("celsius.txt", cs))
} yield ()
```

This works, although it requires loading the contents of `fahrenheit.txt` entirely into memory to work on it, which could be problematic if the file is very large. We would prefer to perform this task using roughly constant memory—read a line or a fixed size buffer full of lines from `farenheit.txt`, convert to celsius, dump to `celsius.txt`, and repeat. To achieve this efficiency we could expose a lower-level file API that gives access to I/O handles:

```
trait Files[A]
case class OpenRead(file: String) extends Files[HandleR]
case class OpenWrite(file: String) extends Files[HandleW]
case class ReadLine(h: HandleR) extends Files[Option[String]]
case class WriteLine(h: HandleW, line: String) extends Files[Unit]

trait HandleR
trait HandleW
```

The only problem with this is we now need to write a monolithic loop:

```
def loop(f: HandleR, c: HandleW): IO[Files,Boolean] = for {
  line <- request(ReadLine(f))
  b <- line.map(s =>
    request(WriteLine(c,
            fahrenheitToCelsius(s.toDouble).toString)) as true) .
    getOrElse (IO.unit(false))
} yield b

def convertFiles = for {
  f <- request(OpenRead("fahrenheit.txt"))
  c <- request(OpenWrite("celsius.txt"))
  _ <- while_(loop(f,c))
} yield ()
```

There's nothing inherently wrong with writing a monolithic loop like this, but it's not composable. Suppose we decide later that we'd like to compute a 5-element moving average of the temperatures. Modifying our `loop` function to do this would be somewhat painful. Compare that to the equivalent change we'd make to our `List`-based code, where we could define a `movingAvg` function and just stick it before or after our conversion to celsius:

```
def movingAvg(n: Int)(l: List[Double]): List[Double]
```

```
cs = movingAvg(5)(lines.map(s => fahrenheitToCelsius(s.toDouble))).
     map(_.toString)
```

Even `movingAvg` could be composed from smaller pieces—we could build it using a generic combinator, `windowed`:[20]

Footnote 20 This can actually be implemented for any `Foldable`. Also, if we have a *group* instead of just a monoid it can be implemented more efficiently. A group defines an additional operation, negation, for which `op(a,neg(a)) == zero`.

```
def windowed[A](n: Int, l: List[A])(f: A => B)(m: Monoid[B]): List[B]
```

The point to all this is that programming with a composable abstraction like `List` is much nicer than programming directly with the primitive I/O operations. Lists are not really special—they are just one instance of a composable API that is pleasant to use. We should not have to give up all the nice compositionality we've

come to expect from FP just to write programs that make use of efficient, streaming I/O.[21] Luckily we don't have to. As we will see in chapter 15, we get to build *whatever* abstractions we want for creating computations that perform I/O. If we like the metaphor of lists or streams, we can find a way to encode a list-like API for expressing I/O computations. If we invent or discover some other composable abstraction, we can often find some way of using that.

Footnote 21 One might ask—could we just have various `Files` operations return the `Stream` type we defined in chapter 5? This is called *lazy I/O*, and it is problematic for several reasons we'll discuss more in chapter 15.

13.7 Conclusion

This chapter introduced the simplest model for how external effects and I/O can be handled in a purely functional way. We began with a discussion of effect-factoring and demonstrated how effects can be moved to the outer layers of a program. From there, we defined two `IO` data types, one with a simple interpreter built into the type, and another which made interactions with the external universe more explicit. This second representation allowed us to implement a more interesting interpreter of `IO` values that used nonblocking I/O internally.

The `IO` monad is not the final word in writing effectful programs. It is important because it represents a kind of lowest common denominator. We don't normally want to program with `IO` directly, and in chapter 15 we will discuss how to build nicer, more composable abstractions.

Before getting to that, in our next chapter, we will apply what we've learned so far to fill in the other missing piece of the puzzle: *local effects*. At various places throughout this book, we've made use of local mutation rather casually, with the assumption that these effects were not *observable*. Next chapter we explore what this means in more detail, show more example usages of local effects, and show how *effect scoping* can be enforced by the type system.

Index Terms

..
continuation
effect scoping
trampolining
unobservable

14 Local effects and mutable state

14.1 Introduction

In the first chapter of this book we introduced the concept of referential transparency, setting the premise for purely functional programming. We declared that pure functions cannot mutate data in place or interact with the external world. In the previous chapter on I/O, we learned that this is not exactly true. We *can* write purely functional and compositional programs that describe interactions with the outside world. These programs are unaware that they can be interpreted with an evaluator that has side-effects.

In this chapter, we will develop a more mature concept of referential transparency. We will consider the idea that effects can occur *locally* inside an expression, and that we can guarantee that no other part of the larger program can observe these effects occurring.

We will also introduce the idea that expressions can be referentially transparent *with regard to* some programs and not others.

14.2 Purely functional mutable state

Up until this point, you may have had the impression that in purely functional programming we're not allowed to use mutable state. But if we look carefully, there is nothing about the definitions of referential transparency and purity that disallows mutation of *local* state. Let's remind ourselves of our definitions from chapter 1:

SIDEBAR **Definition of referential transparency and purity**

An expression `e` is referentially transparent if for all programs `p`, every occurrence of `e` in `p` can be replaced with the result of evaluating `e` without changing the result of evaluating `p`.

A function `f` is *pure* if the expression `f(x)` is referentially transparent for all referentially transparent inputs `x`.

By that definition, the following function is pure, even though it uses a `while` loop, an updatable `var`, and a mutable array:

```
def quicksort(xs: List[Int]): List[Int] = if (xs.isEmpty) xs else {
  val arr = xs.toArray
  def swap(x: Int, y: Int) = {
    val tmp = arr(x)
    arr(x) = arr(y)
    arr(y) = tmp
  }
  def partition(l: Int, r: Int, pivot: Int) = {
    val pivotVal = arr(pivot)
    swap(pivot, r)
    var j = l
    for (i <- l until r) if (arr(i) < pivotVal) {
      swap(i, j)
      j += 1
    }
    swap(j, r)
    j
  }
  def qs(l: Int, r: Int): Unit = if (l < r) {
    val pi = partition(l, r, l + (l - r) / 2)
    qs(l, pi - 1)
    qs(pi + 1, r)
  }
  qs(0, arr.length - 1)
  arr.toList
}
```

The `quicksort` function sorts a list by turning it into a mutable array, sorting the array in place using the well-known Quicksort algorithm, and then turning the array back into a list. It's not possible for any caller to know that the individual subexpressions inside the body of `quicksort` are not referentially transparent or that the local methods `swap`, `partition`, and `qs` are not pure, because at no point does any code outside the `quicksort` function hold a reference to the mutable array. Since all of the mutation is locally scoped, the overall function is

pure. That is, for any referentially transparent expression `xs` of type `List[Int]`, the expression `quicksort(xs)` is also referentially transparent.

Some algorithms, like Quicksort, need to mutate data in place in order to work correctly or efficiently. Fortunately for us, we can always safely mutate data that is created locally. Any function can use side-effecting components internally and still present a pure external interface to its callers.

14.2.1 Composable mutations

But working with mutable data can force us to take a monolithic approach. For example, the constituent parts of `quicksort` above would have direct side-effects if used on their own, which makes it impossible to compose them in the same way that we would compose pure functions. Of course, we could just make them work in `IO`, but that's really not appropriate for local mutable state. If `quicksort` returned `IO[List[Int]]` then it would be an `IO` action that is perfectly safe to run and would have no side-effects, which is not the case in general for `IO` actions. We want to be able to distinguish between effects that are safe to run (like locally mutable state) and external effects like I/O. So a new data type is in order.

The most natural approach is to make a little language for talking about mutable state. Writing and reading a state is something we can already do with the `State[S,A]` monad, which you will recall is just a function of type `S => (A, S)` that takes an input state and produces a result and an output state. But when we're talking about mutating the state *in place*, we're not really passing it from one action to the next. What we'll pass instead is a kind of token marked with the type `S`. A function called with the token then has the authority to mutate data that is tagged with the same type `S`.

This new data type will employ Scala's type system to gain two static guarantees. That is, we want code that violates these invariants to *not complile*:

- A mutable object can never be observed outside of the scope in which it was created.
- If we hold a reference to a mutable object, then nothing can observe us mutating it.

We will call this new local effects monad `ST`, which could stand for "State Thread", "State Transition", "State Token", or "State Tag". It's different from the `State` monad in that its `run` method is protected, but otherwise its structure is exactly the same.

```
sealed trait ST[S,A] { self =>
  protected def run(s: S): (A,S)
  def map[B](f: A => B): ST[S,B] = new ST[S,B] {
    def run(s: S) = {
      val (a, s1) = self.run(s)
      (f(a), s1)
    }
  }
  def flatMap[B](f: A => ST[S,B]): ST[S,B] = new ST[S,B] {
    def run(s: S) = {
      val (a, s1) = self.run(s)
      f(a).run(s1)
    }
  }
}
object ST {
  def apply[S,A](a: => A) = {
    lazy val memo = a ❶
    new ST[S,A] {
      def run(s: S) = (memo, s)
    }
  }
}
```

❶ Cache the value in case run is called more than once.

The reason the `run` method is protected is that an `S` represents the ability to *mutate* state, and we don't want the mutation to escape. So how do we then run an `ST` action, giving it an initial state? These are really two questions. We will start by answering the question of how we specify the initial state.

As always, don't feel compelled to understand every detail of the implementation of `ST`. What matters is the idea that we can use the type system to constrain the scope of mutable state.

14.2.2 An algebra of mutable references

Our first example of an application for the `ST` monad is a little language for talking about mutable references. This takes the form of a combinator library with some primitive combinators. The language for talking about mutable memory cells should have these primitive commands:

- Allocate a new mutable cell
- Write to a mutable cell
- Read from a mutable cell

The data structure we'll use for mutable references is just a wrapper around a protected `var`:

```
sealed trait STRef[S,A] {
  protected var cell: A
  def read: ST[S,A] = ST(cell)
  def write(a: => A): ST[S,Unit] = new ST[S,Unit] {
    def run(s: S) = {
      cell = a
      ((), s)
    }
  }
}

object STRef {
  def apply[S,A](a: A): ST[S, STRef[S,A]] = ST(new STRef[S,A] {
    var cell = a
  })
}
```

The methods on `STRef` to read and write the cell are pure since they just return `ST` actions. Notice that the type `S` is *not* the type of the cell that's being mutated, and we never actually use the value of type `S`. Nevertheless, in order to call `apply` and actually run one of these `ST` actions, you do need to have a value of type `S`. That value therefore serves as a kind of token—an authorization to mutate or access the cell, but it serves no other purpose.

The question of how to give an initial state is answered by the `apply` method on the `STRef` companion object. The `STRef` is constructed with an initial value for the cell, of type `A`. But what is returned is not a naked `STRef`, but an `ST` action that constructs the `STRef` when run. That is, when given the token of type `S`.

At this point, let's try writing a trivial `ST` program. It's a little awkward right now because we have to choose a type `S` arbitrarily. Here, we arbitrarily choose `Nothing`:

```
for {
  r1 <- STRef[Nothing,Int](1)
  r2 <- STRef[Nothing,Int](1)
  x  <- r1.read
  y  <- r2.read
  _  <- r1.write(y+1)
  _  <- r2.write(x+1)
  a  <- r1.read
  b  <- r2.read
} yield (a,b)
```

This little program allocates two mutable `Int` cells, swaps their contents, adds

one to both, and then reads their new values. But we can't yet *run* this program because `run` is still protected (and we could never actually pass it a value of type `Nothing` anyway). Let's work on that.

14.2.3 Running mutable state actions

By now you will have figured out the plot with the `ST` monad. The plan is to use `ST` to build up a computation that, when run, allocates some local mutable state, proceeds to mutate it to accomplish some task, and then discards the mutable state. The whole computation is referentially transparent because all the mutable state is private and locally scoped. But we want to be able to *guarantee* that. For example, an `STRef` contains a mutable `var` and we want Scala's type system to guarantee that we can never extract an `STRef` out of an `ST` action because that would violate the invariant that the mutable reference is local to the `ST` action, violating referential transparency in the process.

So how do we safely run `ST` actions? First we must differentiate between actions that are safe to run and ones that aren't. Spot the difference between these types:

- `ST[S, STRef[S, Int]]` (not safe to run)
- `ST[S, Int]` (completely safe to run)

The former is an `ST` action that contains a mutable reference. But the latter is quite different. A value of type `ST[S,Int]` is quite literally just an `Int`, even though computing the `Int` may involve some local mutable state. Fortunately for us, there's an exploitable difference between these two types. The `STRef` involves the type `S`, but `Int` does not.

We want to disallow running an action of type `ST[S, STRef[S,A]]` because that would expose the `STRef`. And in general we want to disallow running any `ST[S,T]` where `T` involves the type `S`. On the other hand, it's easy to see that it should always be safe to run an `ST` action that doesn't expose a mutable object. If we have such a pure action of a type like `ST[S,Int]`, it should be safe to pass it an `S` to get the `Int` out of it. Furthermore, *we don't care what `S` actually is* in that case because we are going to throw it away. The value might as well be polymorphic in `S`.

In order to represent this, we will introduce a new trait that represents `ST` actions that are safe to run. In other words, actions that are polymorphic in `S`:

```
trait RunnableST[A] {
  def apply[S]: ST[S,A]
}
```

This is very similar to the idea behind the `Trans` trait from the previous chapter. A value of type `RunnableST[A]` is a function that takes a *type* `S` and produces a *value* of type `ST[S,A]`.

In the section above we arbitrarily chose `Nothing` as our `S` type. Let's instead wrap it in `RunnableST` making it polymorphic in `S`. Then we do not have to choose the type `S` at all. It will be supplied by whatever calls `apply`.

```
val p = new RunnableST[(Int, Int)] {
  def apply[S] = for {
    r1 <- STRef(1)
    r2 <- STRef(2)
    x  <- r1.read
    y  <- r2.read
    _  <- r1.write(y+1)
    _  <- r2.write(x+1)
    a  <- r1.read
    b  <- r2.read
  } yield (a,b)
}
```

We are now ready to write the `runST` function that will call `apply` on any polymorphic `RunnableST` by arbitrarily choosing a type for `S`. Since the `RunnableST` action is polymorphic in `S`, it's guaranteed to not make use of the value that gets passed in. So it's actually completely safe to pass `null`!

The `runST` function must go on the `ST` companion object. Since `run` is protected on the `ST` trait, it's accessible from the companion object but nowhere else:

```
object ST {
  def apply[S,A](a: => A) = {
    lazy val memo = a
    new ST[S,A] {
      def run(s: S) = (memo, s)
    }
  }
  def runST[A](st: RunnableST[A]): A =
    st[Null].run(null)._1
}
```

We can now run our trivial program `p` from above:

```
scala> val p = new RunnableST[(Int, Int)] {
     |   def apply[S] = for {
     |     r1 <- STRef(1)
     |     r2 <- STRef(2)
     |     x  <- r1.read
     |     y  <- r2.read
     |     _  <- r1.write(y+1)
     |     _  <- r2.write(x+1)
     |     a  <- r1.read
     |     b  <- r2.read
     |   } yield (a,b)
     | }
p: RunnableST[(Int, Int)] = $anon$1@e3a7d65

scala> val r = ST.runST(p)
r: (Int, Int) = (3,2)
```

The expression `runST(p)` uses mutable state internally but it does not have any side-effects. As far as any other expression is concerned, it's just a pair of integers like any other. It will always return the same pair of integers and it will do nothing else.

But this is not the most important part. Most importantly, we *cannot* run a program that tries to return a mutable reference. It's not possible to create a `RunnableST` that returns a naked `STRef`.

```
scala> new RunnableST[STRef[Nothing,Int]] {
     |   def apply[S] = STRef(1)
     | }
<console>:17: error: type mismatch;
 found   : ST[S,STRef[S,Int]]
 required: ST[S,STRef[Nothing,Int]]
               def apply[S] = STRef(1)
```

In this example, we arbitrarily choose `Nothing` just to illustrate the point. The point is that the type `S` is bound in the `apply` method, so when we say `new RunnableST`, that type is not accessible.

Because an `STRef` is always tagged with the type `S` of the `ST` action that it lives in, it can never escape. And this is guaranteed by Scala's type system! As a corollary, the fact that you cannot get an `STRef` out of an `ST` action guarantees that if you have an `STRef` then you are inside of the `ST` action that created it, so it's always safe to mutate the reference.

SIDEBAR **A note on the wildcard type**

It is possible to bypass the type system in `runST` by using the *wildcard type*. If we pass it a `RunnableST[STRef[_,Int]]`, this will allow an `STRef` to escape:

```
scala> val ref = ST.runST(new RunnableST[STRef[_, Int]] {
     |   def apply[S] = for {
     |     r1 <- STRef(1)
     |   } yield r1
     | })
ref: STRef[_, Int] = STRef$$anonfun$apply$1$$anon$6@20e88a41
```

The wildcard type is an artifact of Scala's interoperability with Java's type system. Fortunately, when you have an `STRef[_,Int]`, using it will cause a type error:

```
scala> new RunnableST[Int] {
     |   def apply[R] = for { x <- ref.read } yield x }
 error   : type mismatch;
 found   : ST[_$1,Int]
 required: ST[R,Int]
             def apply[R] = for { x <- ref.read } yield x }
                                          ^
```

This type error is caused by the fact that the wildcard type in `ref` represents some concrete type that only `ref` knows about. In this case it's the `S` type that was bound in the `apply` method of the `RunnableST` where it was created. Scala is unable to prove that this is the same type as `R`. Therefore, even though it's possible to abuse the wildcard type to get the naked `STRef` out, this is still safe since we can't use it to mutate or access the state.

14.2.4 Mutable arrays

Mutable references on their own are not all that useful. A more useful application of mutable state is arrays. In this section we will define an algebra for manipulating mutable arrays in the `ST` monad and then write an in-place QuickSort algorithm compositionally. We will need primitive combinators to allocate, read, and write mutable arrays:

```
sealed abstract class STArray[S,A](implicit manifest: Manifest[A]) {
```

```
  protected def value: Array[A]
  def size: ST[S,Int] = ST(value.size)

  def write(i: Int, a: A): ST[S,Unit] = new ST[S,Unit] { ❷
    def run(s: S) = {
      value(i) = a
      ((), s)
    }
  }

  def read(i: Int): ST[S,A] = ST(value(i)) ❸

  def freeze: ST[S,List[A]] = ST(value.toList) ❹
}

object STArray {
  def apply[S,A:Manifest](sz: Int, v: A): ST[S, STArray[S,A]] = ❺
    new STArray[S,A] {
      lazy val value = Array.fill(sz)(v)
    }
}
```

❶ Scala requires an implicit Manifest for constructing arrays.
❷ Write a value at the give index of the array
❸ Read the value at the given index of the array
❹ Turn the array into an immutable list
❺ Construct an array of the given size filled with the value v

A thing to note is that Scala cannot create arrays for every type. It requires that there exist a `Manifest` for the type in implicit scope. Scala's standard library provides manifests for most types that you would in practice want to put in an array. The `implicitly` function simply gets that manifest out of implicit scope.

Just like with `STRef`s, we always return `STArray`s packaged in an `ST` action with a corresponding `S` type, and any manipulation of the array (even reading it), is an `ST` action tagged with the same type `S`. It's therefore impossible to observe a naked `STArray` outside of the `ST` monad (except in the Scala source file in which the `STArray` data type itself is declared).

Using these primitives, we can write more complex functions on arrays.

EXERCISE 1: Add a combinator on `STArray` to fill the array from a `Map` where each key in the map represents an index into the array, and the value under that key is written to the array at that index. For example, `fill(Map(0->"a", 2->"b"))` should write the value `"a"` at index `0` in the array and `"b"` at index `2`. Use the existing combinators to write your implementation.

```
def fill(xs: Map[Int,A]): ST[S,Unit]
```

Not everything can be done efficiently using these existing combinators. For example, the Scala library already has an efficient way of turning a list into an array. Let's make that primitive as well:

```
def fromList[S,A:Manifest](xs: List[A]): ST[S, STArray[S,A]] =
  ST(new STArray[S,A] {
    lazy val value = xs.toArray
  })
```

14.2.5 A purely functional in-place quicksort

The components for `quicksort` are now easy to write in `ST`. For example, the `swap` function that swaps two elements of the array:

```
def swap[S](i: Int, j: Int): ST[S,Unit] = for {
  x <- read(i)
  y <- read(j)
  _ <- write(i, y)
  _ <- write(j, x)
} yield ()
```

EXERCISE 2: Write the purely functional versions of `partition` and `qs`.

```
def partition[S](arr: STArray[S,Int],
                 l: Int, r: Int, pivot: Int): ST[S,Int]
def qs[S](a: STArray[S,Int], l: Int, r: Int): ST[S,Unit]
```

With those components written, quicksort can now be assembled out of them in the `ST` monad:

```
def quicksort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) xs else ST.runST(new RunnableST[List[Int]] {
    def apply[S] = for {
      arr    <- STArray.fromList(xs)
      size   <- arr.size
      _      <- qs(arr, 0, size - 1)
      sorted <- arr.freeze
    } yield sorted
})
```

As you can see, the `ST` monad allows us to write pure functions that nevertheless mutate the data they receive. Scala's type system ensures that we don't combine things in an unsafe way.

EXERCISE 3: Give the same treatment to `scala.collection.mutable.HashMap` as we have given here to references and arrays. Come up with a minimal set of primitive combinators for creating and manipulating hash maps.

14.3 Purity is contextual

In the preceding section, we talked about effects that are not observable because they are entirely local to some scope. There are no programs that can observe the mutation of data to which it doesn't hold a reference.

But there are other effects that may be non-observable, depending on who is looking. As a simple example let's take a kind of side-effect that occurs all the time in ordinary Scala programs, even ones that we would usually consider purely functional.

```
scala> case class Foo(s: String)

cala> val b = Foo("hello") == Foo("hello")
b: Boolean = true

scala> val c = Foo("hello") eq Foo("hello")
c: Boolean = false
```

Here, `Foo("hello")` looks pretty innocent. We could be forgiven if we assumed that it was a completely referentially transparent expression. But each time it appears, it produces a *different* `Foo` in a certain sense. If we test `Foo("hello")` for equality using the `==` function, we get `true` as we would expect. But testing for *reference equality* (a notion inherited from the Java language) with `eq`, we get `false`. The two appearances of `Foo("hello")` are not references to the "same object" if we look under the hood.

Notice that if we evaluate `Foo("hello")` and store the result as `x`, then substitute `x` to get the expression `x eq x`, it has a different result.

```
scala> val x = Foo("hello")
x: Foo = Foo(hello)

scala> val d = x eq x
d: Boolean = true
```

Therefore, by our original definition of referential transparency, *every data constructor in Scala has a side-effect*. The effect is that a new and unique object is created in memory, and the data constructor returns a reference to that new object.

For most programs this makes no difference because most programs do not check for reference equality. It is only the `eq` method that allows our programs to observe this side-effect occurring. We could therefore say that it's not a side-effect at all in the context of the vast majority of programs.

Our definition of referential transparency doesn't take this into account. It seems like we need the definition to be more general:

SIDEBAR **More general definition of referential transparency**

An expression e is referentially transparent with regard to program p if every occurrence of e in p can be replaced with the result of evaluating e without changing the result of evaluating p.

This definition is only slightly modified to reflect the fact that not all programs can observe the same effects. We say that an effect of `e` is *non-observable* by `p` if it doesn't affect the referential transparency of `e` with regard to `p`.

We should also note that this definition makes some assumptions. What is meant by "evaluating"? And what is the standard by which we determine whether the results of two evaluations are the same?

In Scala, there is a kind of standard answer to these questions. Generally, evaluation means *reduction to some normal form*. Since Scala is a strictly evaluated language, we can force the evaluation of an expression `e` to normal form in Scala by assigning it to a `val`:

```
val v = e
```

And referential transparency of `e` with regard to a program `p` means that we can rewrite `p` replacing every appearance of `e` with `v`.

But what do we mean by "changing the result"? We mean that the two results, before and after rewriting, are in some sense equivalent. And what it means for two expressions to be equal is a little more nuanced than it might at first appear. Do we

mean equality by the `==` operator? Or do we mean reference equality by the `eq` operator? And what about the case where `e` is a function? When are two functions equal?

Again, there is a standard answer. In Scala we usually refer to *extensional equality* when talking about whether two functions are equivalent. We say that two functions, `f` and `g` are extensionally equal when `f(x)` equals `g(x)` for all inputs `x`. We could just as well take the position of requiring *intensional* equality which means that `f` and `g` would have to have the same implementation in order to be considered equal. But whatever context we choose, the point is that we must choose *some* context.

14.3.1 What counts as a side-effect?

Above, we talked about how the `eq` method is able to *observe* the side-effect of object creation. Let's look more closely at this idea of observable behavior. It requires that we delimit what we consider observable and what we don't. Take for example this method that has a definite side-effect:

```
def timesTwo(x: Int) = {
  if (x < 0) println("Got a negative number")
  x * 2
}
```

If you replace `timesTwo(1)` with `2` in your program, you do not have the same program in every respect. It may compute the same result, but we can say that the observable behavior of the program has changed. But this is not true for all programs that call `timesTwo`, nor for all notions of program equivalence.

We need to decide up front whether changes in standard output are something we care to observe—whether it's part of the changes in behavior that *matter* in our context. In this case it's exceedingly unlikely that any other part of the program will be able to observe that `println` side-effect occurring inside `timesTwo`.

Of course, `timesTwo` has a *hidden dependency* on the I/O subsystem. It requires access to the standard output stream. But as we have seen above, most programs that we would consider purely functional also require access to some of the underlying machinery of Scala's environment, like being able to construct objects in memory and discard them. At the end of the day, we have to decide for ourselves which effects are important enough to track. We could use the `IO` monad to track `println` calls, but maybe we don't want to bother. If we're just using the

console to do some temporary debug logging, it seems like a waste of time to track that. But if the program's correct behavior depends in some way on the what it prints to the console (like if it's a UNIX utility), then we definitely want to track it.

This brings us to an essential point: Keeping track of effects is a *choice* we make as programmers. It's a value judgement, and there are trade-offs associated with how we choose. We can take it as far as we want. But as with the context of referential transparency, in Scala there is a kind of standard choice. For example it would be completely valid and possible to track memory allocations in the type system if that really mattered to us. But in Scala we have the benefit of automatic memory management so the cost of explicit tracking is usually higher than the benefit.

The policy we should adopt is to *track those effects that program correctness depends on*. If a program is fundamentally about reading and writing files, then file I/O should be tracked in the type system to the extent feasible. If a program relies on object reference equality, it would be nice to know that statically as well. Static type information lets us know what kinds of effects are involved, and thereby lets us make educated decisions about whether they matter to us in a given context.

The `ST` type in this chapter and the `IO` monad in the previous chapter should have given you a taste for what it's like to track effects in the type system. But this is not the end of the road. You're limited only by your imagination and the expressiveness of Scala's types.

14.4 Summary

In this chapter, we discussed two different implications of referential transparency.

We saw that we can get away with mutating data that never escapes a local scope. At first blush it may seem that mutating state can't be compatible with pure functions. But as we have seen, we can write components that have a pure interface and mutate local state behind the scenes, using Scala's type system to guarantee purity.

We also discussed that what counts as a side-effect is actually a choice made by the programmer or language designer. When we talk about functions being pure, we should have already chosen a context that establishes what it means for two things to be equal, what it means to execute a program, and which effects we care to take into account when observing the program's behavior.

Index Terms

extensional equality
extensionality
intensional equality
intensionality

15 Stream processing and incremental I/O

15.1 Introduction

We said in the introduction to part 4 that functional programming is a *complete* paradigm. Any program that can be imagined can be expressed functionally, including those that interact with the external world. But it would be disappointing if the `IO` type were our only way of constructing such programs. `IO` (and `ST`) work by simply *embedding* an imperative programming language into the purely functional subset of Scala. While programming *within* the `IO` monad, we have to reason about our programs much like we would in ordinary imperative programming.

We can do better. Not only can functional programs embed arbitrary imperative programs; in this chapter we show how to recover the high-level, compositional style developed in parts 1-3 of this book, even for programs that interact with the outside world. The design space in this area is enormous, and our goal here is more to convey ideas and give a sense of what is possible.[1]

Footnote 1 As always, there is more discussion and links to further reading in the chapter notes.

15.2 Problems with imperative I/O: an example

Rather than simply giving the 'answer' up front, we will build up a library for streaming, composable I/O incrementally. We are going to start by considering a very simple, concrete usage scenario, which we'll use to highlight some of the problems with imperative I/O embedded in the `IO` monad:

Check whether the number of lines in a file is greater than 40,000.

This is a rather simplistic task, intended to be illustrative and help us get at the essence of the problem we are trying to solve with our library. We could certainly

accomplish this task with ordinary imperative code, inside the `IO` monad. Let's look at that first:[2]

Footnote 2 For simplicity, in this chapter we are not going to parameterize our `IO` type on the `F` language used. That is, let's assume that `type IO[A] = fpinscala.iomonad.IO[Task,A]`, where `Task[A]` just wraps a `Future[Either[Throwable,A]]` with some functions for error handling. This should be taken to mean that within an `IO[A]` we can make use of any impure Scala function. See the chapter code for details.

```
def linesGt40k(filename: String): IO[Boolean] = IO {
  val src = io.Source.fromFile(filename) ❶
  try {
    var count = 0
    val lines: Iterator[String] = src.getLines ❷
    while (count <= 40000 && lines.hasNext) {
      lines.next ❸
      count += 1
    }
    count > 40000
  }
  finally src.close
}
```

❶ There are a number of convenience functions in scala.io.Source for reading from external sources like files.
❷ Obtain a stateful Iterator from the Source
❸ Has side effect of advancing to next element

Although this code is rather low-level, there are a number of *good* things about it. First, it is *incremental*—the entire file is not loaded into memory up front. Instead, lines are fetched from the file only when needed. If we didn't buffer our input, we could keep as little as a single line of the file in memory at a time. It also terminates early, as soon as the answer is known, rather than reading the entire file and then returning an answer.

There are some bad things about this code, too. For one, we have to remember to close the file when we're done. This might seem obvious, but if we forget to do this or (more commonly) if we close the file outside of a `finally` block and an exception occurs first, the file will remain open.[3] This is called a *resource leak*. A file handle is an example of a scarce *resource*—the operating system can only have a limited number of files open at any given time. If this task were part of a larger

program, say we were scanning an entire directory recursively, building up a list of all files with more than 40,000 lines, our larger program could easily fail because too many files were left open.

Footnote 3 The JVM will actually close an `InputStream` (which is what backs a `scala.io.Source`) when it is garbage collected, but there is no way to guarantee this will occur in a timely manner, or at all! This is especially true in generational garbage collectors that perform 'full' collections infrequently.

We want to write programs that are *resource safe*—that is, they should close file handles as soon as they are finished with them (whether because of normal termination or an exception), and they should not attempt to read from a closed file. Likewise for other resources like network connections, database connections, and so on. Using `IO` directly can be problematic because it means our programs are *entirely responsible for ensuring resource safety*, and we get no help from the compiler in making sure of this. It would be nice if our library would ensure resource safety by construction.

SIDEBAR

The bracket combinator

A commonly used combinator that helps with ensuring resource safety in `IO` code is `bracket`:

```
def bracket[A,B](acquire: IO[A], release: A => IO[Unit])(
                 using: A => IO[B]): IO[B]
```

The exact implementation of this combinator depends on the representation of `IO`, but the implementation should ensure that the resource is released, either just after the `using` action finishes successfully or immediately if an exception occurs. As an exercise, you may wish to implement `bracket` for our existing `IO` type.

But even aside from the problems with resource safety, there is something rather low-level and unsatisfying about this code. We should be able to express the *algorithm*—of counting elements and stopping with a response as soon as we hit 40,000, independent of *how* we are to obtain these elements. Opening and closing files and catching exceptions is a separate concern from the fundamental algorithm being expressed, but this code intertwines these concerns. This isn't just ugly, it's not *compositional*, and our code will be difficult to extend later. For instance, consider a few variations of this scenario:

- Check whether the number of *nonempty* lines in the file exceeds 40,000

- Find a line index before 40,000 where the first letter of consecutive lines spells out `"abracadabra"`.

For this first case, we could imagine passing a `String => Boolean` into our `linesGt40k` function. But for the second case, we would need to modify our loop to keep track of some further state, and besides being uglier, the resulting code will likely be tricky to get right. In general, writing efficient code in the `IO` monad generally means writing monolithic loops, and monolithic loops are not composable.

Let's compare this to the case where we have a `Stream[String]` for the lines being analyzed.

```
lines.zipWithIndex.exists(_._2 + 1 >= 40000)
```

Much nicer! With a `Stream`, we get to assemble our program from preexisting combinators, `zipWithIndex` and `exists`. If we want to filter these lines, we can do so easily:

```
lines.filter(!_.trim.isEmpty).zipWithIndex.exists(_._2 + 1 >= 40000)
```

And for the second scenario, we can use the `indexOfSlice` function defined on `Stream`,[4] in conjunction with `take` (to terminate the search after 40,000 lines) and `map` (to pull out the first character of each line):

Footnote 4 If the argument to `indexOfSlice` does not exist as a subsequence of the input, `-1` is returned. See the API docs for details, or experiment with this function in the REPL.

```
lines.take(40000).map(_.head).indexOfSlice("abracadabra".toList)
```

A natural question to ask is, could we just write the above programs if reading from an actual file? Not quite. The problem is we don't have a `Stream[String]`, we have a file from which we can read a line at a time. We could cheat by writing a function, `lines`, which returns an `IO[Stream[String]]`:

```
def lines(filename: String): IO[Stream[String]] = IO {
  val src = io.Source.fromFile(filename)
```

```
  src.getLines.toStream append { src.close; Stream.empty }
}
```

This is called *lazy I/O*. We are cheating because the `Stream[String]` inside the `IO` is not actually a pure value. As elements of the stream are forced, it will execute side effects of reading from the file, and only if we examine the entire stream and reach its end will we close the file. Although it is appealing that lazy I/O lets us recover the compositional style to some extent, it is problematic for several reasons:

- It isn't resource safe. The resource (in this case, a file) will be released only if we traverse to the end of the stream. But we will frequently want to terminate traversal early (here, `exists` will stop traversing the `Stream` as soon as it finds a match) and we certainly don't want to leak resources every time we do this.

- Nothing stops us from traversing that same `Stream` again, after the file has been closed, resulting in either excessive memory usage (if the `Stream` is one that caches or *memoizes* its values once forced) or an error if the `Stream` is unmemoized and this causes a read from a closed file handle. Also, having two threads traverse an unmemoized `Stream` at the same time can result in unpredictable behavior.
- In more realistic scenarios, we won't necessarily have full knowledge of what is happening with the `Stream[String]` we created. It could be passed on to some other function we don't control, which might store it in a data structure for a long period of time before ever examining it, etc. Proper usage now requires some out-of-band knowledge—we cannot necessarily just manipulate this `Stream[String]` like a typical pure value, we have to know something about its origin. This is bad for the compositional style typically used in FP, where most of our code won't know anything about a value other than its type.

Lazy I/O is problematic, but it would be nice to recover the high-level style we are accustomed to from our usage of `Stream` and `List`. In the next section, we'll introduce the notion of *stream transducers* or *stream processors*, which is our first step toward achieving this.

15.3 Simple stream transducers

A stream transducer specifies a transformation from one stream to another. We are using the term *stream* more generally here, to refer to a sequence, possibly lazily generated or supplied by an external source (for instance, a stream of lines from a file, a stream of HTTP requests, a stream of mouse click positions, etc). Let's consider a simple data type, `Process`, that lets us express stream transformations:
5

Footnote 5 We have chosen to omit variance annotations in this chapter for simplicity, but it is possible to write this as `Process[-I,+O]`.

```
trait Process[I,O]
case class Emit[I,O](
    head: Seq[O],
    tail: Process[I,O] = Halt[I,O]())
  extends Process[I,O]
case class Await[I,O](
    recv: I => Process[I,O],
    finalizer: Process[I,O] = Halt[I,O]())
  extends Process[I,O]
case class Halt[I,O]() extends Process[I,O]
```

A `Process[I,O]` can be used to transform a stream containing `I` values to a stream of `O` values (this is sometimes called a stream transducer). The viewpoint of `Process` is somewhat inverted from how we might be used to thinking of things. `Process[I,O]` is not a typical function `Stream[I] => Stream[O]`, which could pattern match on the input and construct the output itself. Instead, we have a state machine which must be interpreted by a *driver*. There are three possible states a `Process` can be in, each of which signals something to the driver:

- `Emit(head,tail)` indicates to the driver that the `head` values should be emitted to the output stream, and that `tail` should be the next state following that.[6]

  Footnote 6 We could choose to have `Emit` produce just a single value. The use of `Seq` avoids stack overflows for certain `Process` definitions.

- `Await(recv,finalizer)` requests a value from the input stream, indicating that `recv` should be used by the driver to produce the next state, and that `finalizer` should be consulted if the input has no more elements available.
- `Halt` indicates to the driver that no more elements should be read from the input stream or emitted to the output.

Let's look at a sample driver that will actually interpret these requests. Here is one that actually transforms a `Stream`. We can implement this as a function on `Process`:

```
def apply(s: Stream[I]): Stream[O] = this match {
  case Halt() => Stream()
  case Await(recv, finalizer) => s match {
    case h #:: t => recv(h)(t)
    case _ => finalizer(s) // Stream is empty
  }
```

```
  case Emit(h,t) => h.toStream append t(s)
}
```

Thus, given `p: Process[I,O]` and `in: Stream[I]`, `p(in)` produces a `Stream[O]`. What is interesting is that `Process` is agnostic to how it is fed input. We have written a driver that feeds a `Process` from a `Stream`, but we can also write drivers that perform `IO`. We'll get to writing such a driver a bit later, but first, we are going to explore the large number of operations expressible with `Process`.

15.3.1 Operations on Process

We can think about `Process[I,O]`, on the one hand, as a sequence of `O` values, and many of the operations defined for sequences are defined for `Process` as well. Let's start with a familiar one, `map`:

```
def map[O2](f: O => O2): Process[I,O2] = this match {
  case Halt() => Halt()
  case Emit(h, t) => Emit(h map f, t map f)
  case Await(recv,fb) => Await(recv andThen (_ map f), fb map f)
}
```

The implementation simply calls `map` on any values produced by the `Process`. As with lists, we can also *append* processes. Given two processes, `x` and `y`, `x ++ y` runs `x` to completion, then runs `y` to completion on whatever input remains after the first has halted. For the implementation, we simply replace the `Halt` of `x` with `y` (much like how `++` on `List` replaces the `Nil` of the first list with the second list):

```
def ++(p: => Process[I,O]): Process[I,O] = this match {
  case Halt() => p
  case Emit(h, t) => emitAll(h, t ++ p)  (1)
  case Await(recv,fb) => Await(recv andThen (_ ++ p), fb ++ p)
}
```

(1) Helper function described below

This uses a helper function, `emitAll`, which behaves just like the `Emit` constructor but combines adjacent emit states into a single `Emit`. For instance, `emitAll(h1, Emit(h2, t))` becomes `Emit(h1 ++ h2, t)`. (A

function like this that just calls some constructor of a data type but enforces some addition invariant is often called a *smart constructor*.)

```
def emitAll[I,O](head: Seq[O],
                 tail: Process[I,O] = Halt[I,O]()): Process[I,O] =
  tail match {
    case Emit(h2, tl) => Emit(head ++ h2, tl)
    case _ => Emit(head, tail)
  }

def emit[I,O](head: O,
              tail: Process[I,O] = Halt[I,O]()): Process[I,O] =
    emitAll(Stream(head), tail)
```

Consistent use of `emitAll` lets us assume that an `Emit` will always be followed by an `Await` or a `Halt`, which avoids stack overflow errors in certain `Process` definitions.

With the help of `++` on `Process`, we can define `flatMap`:

```
def flatMap[O2](f: O => Process[I,O2]): Process[I,O2] = this match {
  case Halt() => Halt()
  case Emit(h, t) =>
    if (h.isEmpty) t flatMap f
    else f(h.head) ++ emitAll(h.tail, t).flatMap(f)
  case Await(recv,fb) =>
    Await(recv andThen (_ flatMap f), fb flatMap f)
}
```

Incidentally, `Process` forms a `Monad`. The `unit` function just emits a single value, then halts:

```
def unit[O](o: => O): Process[I,O] = emit(o)
```

To write the `Monad` instance, we have to partially apply the `I` parameter of `Process`, which we've seen before:

```
def monad[I]: Monad[({ type f[x] = Process[I,x]})#f] =
  new Monad[({ type f[x] = Process[I,x]})#f] {
    def unit[O](o: => O): Process[I,O] = Emit(o)
    def flatMap[O,O2](p: Process[I,O])(
                      f: O => Process[I,O2]): Process[I,O2] =
      p flatMap f
  }
```

We use the same trick introduced in chapter 13 of placing a `toMonadic` implicit conversion in the companion object to give us infix syntax for the `Monad` combinators:

```
implicit def toMonadic[I,O](a: Process[I,O]) = monad[I].toMonadic(a)
```

This lets us write, for instance: `emit(42) as "hello!"`.[7]

Footnote 7 Recall that `a as b` is equal to `a map (_ => b)`.

The `Monad` instance is exactly the same 'idea' as the `Monad` for `List`. What makes `Process` more interesting than just `List` is it can accept *input*. And it can transform that input through mapping, filtering, folding, grouping, and so on. It turns out that `Process` can express almost any stream transformation, all while remaining agnostic to how exactly it is obtaining its input or what should happen with its output.

PROCESS COMPOSITION, LIFTING, AND REPETITION

The way we will build up complex stream transformations is by *composing* `Process` values. Given two `Process` values, `f` and `g`, we can feed the output `f` into the input of `g`. We'll call this operation `|>` (pronounced 'pipe' or 'compose') and implement it as a function on `Process`.[8] It has the nice property that `f |> g` *fuses* transformations of `f` and `g`. As soon as values are emitted by `f`, they are transformed by `g`.

Footnote 8 This operation might remind you of function composition, which feeds the (single) output of a function in as the (single) input to another function. Both `Process` and functions form *categories*. We won't be discussing that much here, but see the chapter notes.

EXERCISE 1 (hard): Implement `|>`. Let the types guide your implementation.

```
def |>[O2](p2: Process[O,O2]): Process[I,O2]
```

We can convert any function `f: I => O` to a `Process[I,O]`. We repeatedly `Await`, then `Emit` the value received, transformed by `f`.

```
def lift[I,O](f: I => O): Process[I,O] =
  Await((i: I) => emit(f(i), lift(f)))
```

This pattern is quite common—we often have some `Process` whose steps we wish to repeat forever. We can write a combinator for it, `repeat`:

```
def repeat: Process[I,O] = {
  def go(p: Process[I,O]): Process[I,O] = p match {
    case Halt() => go(this)
    case Await(recv,fb) => Await(recv andThen go, fb)
    case Emit(h, t) => Emit(h, go(t))
  }
  go(this)
}
```

This is very typical of `Process` definitions. We define a recursive internal function (often called `go` or `loop`) whose parameters are the *state* used for the transformation. (In the case of `repeat`, the only piece of state is the current `Process`; for other transformations the state may be more complicated.) We then call this internal function with some initial state. Let's use `repeat` to write `filter`, which constructs a `Process` that filters its input:

```
def filter[I](f: I => Boolean): Process[I,I] =
  Await[I,I](i => if (f(i)) emit(i) else Halt()) repeat
```

We can now write expressions like `filter(_ % 2 == 0) |> lift(_ + 1)` to filter and map in a single transformation. We'll sometimes call a sequence of transformations like this a *pipeline*.

There are a huge number of other combinators we can write for `Process`. Let's look at another one, `sum`, which outputs a running total of the values seen so far:

```
def sum: Process[Double,Double] = {
  def go(acc: Double): Process[Double,Double] =
    Await((d: Double) => emit(d+acc, go(d+acc)))
  go(0.0)
}
```

Again, we use the same pattern of an inner function which tracks the current state (in this case, the total so far). Here's an example of its use in the REPL:

```
scala> sum(Stream(1.0, 2.0, 3.0, 4.0)).toList
val res0: List[Double] = List(1.0, 3.0, 6.0, 10.0)
```

Let's get write some more `Process` combinators to get accustomed to this style of programming. Try to work through implementations of at least some of these exercises until you get the hang of it.

EXERCISE 2: Implement `take`, which halts the `Process` after it encounters the given number of elements, and `drop`, which ignores the given number of arguments, then emits the rest. Optional: implement `takeWhile` and `dropWhile`.

```
def take[I](n: Int): Process[I,I]
```

```
def drop[I](n: Int): Process[I,I]
```

```
def takeWhile[I](f: I => Boolean): Process[I,I]
```

```
def dropWhile[I](f: I => Boolean): Process[I,I]
```

EXERCISE 3: Implement `count`. It should emit the number of elements seen so far, for instance, `count(Stream("a", "b", "c"))` should yield `Stream(1, 2, 3)` (or `Stream(0, 1, 2, 3)`, your choice). Feel free to use existing combinators.

```
def count[I]: Process[I,Int]
```

EXERCISE 4: Implement `mean`. It should emit a running average of the values seen so far.

```
def mean: Process[Double,Double]
```

Just as we have seen many times before throughout this book, when we notice common patterns when defining a series of functions, we can factor these patterns out into generic combinators. The functions `sum`, `count` and `mean` all share a

common pattern. Each has a single piece of state, and a state transition function that updates this state in response to input and produces a single output. We can generalize this to a combinator, `loop`:

```
def loop[S,I,O](z: S)(f: (I,S) => (O,S)): Process[I,O] =
  Await((i: I) => f(i,z) match {
    case (o,s2) => emit(o, loop(s2)(f))
  })
```

Using `loop`, we can, for instance, express `sum` as `loop(0)((n:Double,acc) => (acc,n+acc))`.

EXERCISE 5 (optional): Write `sum` and `count` in terms of `loop`.

EXERCISE 6 (hard, optional): Come up with a generic combinator that lets us express `mean` in terms of `sum` and `count`. Define this combinator and implement `mean` in terms of it.

EXERCISE 7 (optional): Implement `zipWithIndex`. It emits a running count of values emitted along with each value. For example: `Process("a","b").zipWithIndex` yields `Process(("a",0), ("b",1))`.

EXERCISE 8 (optional): Implement `exists`. There are multiple ways to implement it. Given `exists(_ % 2 == 0)(Stream(1,3,5,6,7))` could produce `Stream(true)` (halting, and only yielding the final result), `Stream(false,false,false,true)` (halting, and yielding all intermediate results), or `Stream(false,false,false,true,true)` (*not* halting, and yielding all the intermediate results). Note that because `|>` fuses, there is no penalty to implementing the 'trimming' of this last form with a separate combinator.

```
def exists[I](f: I => Boolean): Process[I,Boolean]
```

We can now express the core stream transducer for our line-counting problem as `count |> exists(_ > 40000)`. Of course, it's easy to attach filters and other transformations to our pipeline.

15.3.2 External sources

We can use an external source to drive a `Process`. We'll look first at a simplistic approach in which sources are a completely separate type from `Process`; later, we will consider a generalized `Process` type which can represent sources as well as single-input stream transducers.

```
trait Source[O] {
  def |>[O2](p: Process[O,O2]): Source[O2]
  def filter(f: O => Boolean) = this |> Process.filter(f)
  def map[O2](f: O => O2) = this |> Process.lift(f)
}
case class ResourceR[R,I,O]( // A resource from which we can read values
  acquire: IO[R],
  release: R => IO[Unit],
  step: R => IO[Option[I]],
  trans: Process[I,O]) extends Source[O] {
  def |>[O2](p: Process[O,O2]) =
    ResourceR(acquire, release, step, trans |> p)
}
```

As the definitions of `filter` and `map` demonstrate, we can implement various operations with helper functions that simply attach the appropriate `Process` onto the output of the `Source`. Using this approach we can implement `take`, `takeWhile`, and lots of other typical list processing functions, almost as if `Source` were an actual `List`. We only need to provide an *interpreter* for `Source` that actually performs the `IO` actions and feeds them to the transducer, but this is absolutely straightforward:

```
def collect: IO[IndexedSeq[O]] = {

  def tryOr[A](a: => A)(cleanup: IO[Unit]) = ❶
    try a catch { case e: Exception => cleanup.run; throw e }

  @annotation.tailrec
  def go(acc: IndexedSeq[O],
         step: IO[Option[I]],
         p: Process[I,O],
         release: IO[Unit]): IndexedSeq[O] =
    p match {
      case Halt() => release.run; acc
      case Emit(h, t) =>
        go(tryOr(acc ++ h)(release), step, t, release) ❷
      case Await(recv, fb) => tryOr(step.run)(release) match { ❸
        case None => go(acc, IO(None), fb, release)
```

```
        case Some(i) => go(acc, step, recv(i), release)
      }
    }
  acquire map (res =>
    go(IndexedSeq(), step(res), trans, release(res)))
}
```

1. Helper function: evaluates a, and runs cleanup if an exception occurs.
2. We tryOr(acc ++ h) since h may be a non-strict Seq like Stream which forces some computations that can fail.
3. Our IO computation can of course fail during evaluation.

Notice we are guaranteed to run the `release` action, whether we terminate normally or if an exception occurs during processing.[9] This is important since we will often construct `Source` values backed by some resource like a file handle we want to ensure gets closed. Here is an example of a primitive `Source`, created from the lines of a file:

Footnote 9 One might reasonably ask—if we are eliminating usage of exceptions by using `Either` and `Option` throughout our code, is this really necessary? Yes. For one, not all functions in our programs are defined for all inputs and we typically still use exceptions to signal unrecoverable errors. We may also be using some third-party API which may throw exceptions or errors. And lastly, exceptions may be triggered *asynchronously*, through no fault of the program—for instance, when the thread a program runs in is killed, this generates a `ThreadInterruped` exception to give the program the opportunity to clean up.

```
def lines(filename: String): Source[String] =
  ResourceR(
    IO(io.Source.fromFile(filename)),
    (src: io.Source) => IO(src.close),
    (src: io.Source) => {
      lazy val iter = src.getLines
      IO { if (iter.hasNext) Some(iter.next) else None }
    },
    Process.id[String])
```

Our code for checking whether the number of lines in a file exceeds 40,0000 now looks like `Source.lines("input.txt").count.exists(_ > 40000)`. This is nicely compositional, and we are assured that calling `collect` on this `Source` will open the file and guarantee it is closed, regardless of whether exceptions occur. We deal with resource safety in just two places, the `collect` function we wrote earlier, and the definition of `lines`—the knowledge of how to allocate and release a resource is encapsulated in a single type, `Source`, and `collect` is the sole driver that must take care to use this information to ensure

resource safety. This is in contrast to ordinary imperative I/O (in the `IO` monad or otherwise) where any code that reads from files must repeat the same (error-prone) patterns to ensure resource safety.

Although we can get quite far with `Process` and `Source`, and the simple way we have combined them here is resource safe, these data types are too simple to express a number of interesting and important use cases. Let's look at one of those next:

15.3.3 External sinks

Transform `fahrenheit.txt`, a file containing temperatures in degrees fahrenheit, to `celsius.txt`, a file containing the same temperatures in degrees celsius.

Here's a hypothetical `fahrenheit.txt`:

```
# a comment - any line starting with #
# temperatures in fahrenheit
85.2
83.1
80.0
71.9
...
```

We'd like to write a program that reads this and produces `celsius.txt`:

```
29.5556
28.38889
26.6667
22.16667
...
```

Our program should work in a streaming fashion, emitting to the output file as lines are read from the input file, while staying resource safe. With the library we have so far, we can certainly produce a `Source[Double]` containing the temperatures we need to output to `celsius.txt`:

```
val tempsC: Source[Double] =
  Source.lines("fahrenheit.txt").
         filter(!_.startsWith("#")).
         map(s => fahrenheitToCelsius(s.toDouble))
```

Unfortunately, `Source` lacks the ability to actually write these lines to the output file. In general, one way we can handle these expressiveness problems is by adding extra cases to `Source`. Here, we could try solving our immediate problem by first introducing a new type, `Sink`, analogous to `Source`:

```
trait Sink[I] {
  def <|[I0](p: Process[I0,I]): Sink[I0]
  def filter(f: I => Boolean) = this <| Process.filter(f)
  ...
}
case class ResourceW[R,I,I2](
    acquire: IO[R],
    release: R => IO[Unit],
    recv: R => (I2 => IO[Unit]),
    trans: Process[I,I2]) extends Sink[I] {

  def <|[I0](p: Process[I0,I]) =
    ResourceW(acquire, release, recv, p |> trans)
}
```

Here's a simple `Sink` combinator, for writing to a file:

```
def file(filename: String, append: Boolean = false): Sink[String] =
  ResourceW(
    IO(new FileWriter(filename, append)),
    (w: FileWriter) => IO(w.close),
    (w: FileWriter) => (s: String) => IO(w.write(s)),
    Process.id[String]
  )
```

How might we integrate this into our `Source` API? Let's imagine a new combinator, `observe`:

```
def observe(snk: Sink[O]): Source[O]
```

Implementing this combinator will likely require an additional `Source` constructor and updates to our `collect` implementation (taking care to ensure resource safety in our usage of the `Sink`). Assuming we do this, our complete scenario now looks something like:

```
val convert: IO[Unit] =
  Source.lines("fahrenheit.txt").
         filter(!_.startsWith("#")).
```

```
        map(s => fahrenheitToCelsius(s.toDouble)).
        map(d => d.toString + "\n"). (1)
        observe(Sink.file("celsius.txt")).
        run
```

(1) add line separators back in

This uses the helper function `run`, which ignores the output of a `Source`, evaluating it only for its effects. See the chapter code for its implementation.

Ultimately, this approach of adding special cases to `Source` starts getting rather ugly. Let's take a step back and consider some additional scenarios for which our existing API is insufficient. These are just informal descriptions.

- *Multi-source input/zipping:* 'Zip' together two files, `f1.txt` and `f2.txt`, each containing temperatures in degrees fahrenheit, one per line. Add corresponding temperatures together, convert the result to celsius, apply a 5-element moving average, and output to `celsius.txt`.
- *Concatenation*: Concatenate two files, `fahrenheit1.txt`, and `fahrenheit2.txt`, into a single logical stream, apply the same transformation as above and output to `celsius.txt` .
- *Dynamic resource allocation*: Read a file, `fahrenheits.txt`, containing a list of filenames. Concatenate these files into a single logical stream, convert this stream to celsius, and output the joined stream to `celsius.txt`.
- *Multi-sink output:* As above, but rather than producing a single output file, produce an output file for each input file in `fahrenheits.txt`. Name the output file by appending `.celsius` onto the input file name.
- *Internal effects:* Given a stream of HTTP requests, parse each into some object and use it to construct a database query. Execute this query, generating a stream of rows, which are further processed using other stream transformations before being assembled into an HTTP response. Here, the effect is no longer just a sink—we need to get back a result and continue processing.

These scenarios can't be expressed with our existing API without dropping down into normal, low-level `IO` monad programming (can you see why?). Although we can try just adding more special cases to `Source` (perhaps a `Zip` constructor, then an `Append` constructor, etc.), we can see this getting ugly, especially if done naively.[10] It seems we need a more principled way of extending `Process`. This is what we will consider next.

Footnote 10 Still, you may be interested to explore this approach. It is a challenging design exercise—the problem is coming up with a nice, small set of primitive combinators that lets us express all the programs we wish to write. We don't want to have 50 special cases which, in addition to being ugly, makes writing the `collect` function extremely complicated. If you decide to experiment with this approach, think about what combinators are needed to express each of these scenarios and any others you can think of. Can the combinator be expressed using existing primitives in a resource safe way? If not, you can try adding another primitive case for it, refining your primitives as we did throughout part 2, and updating your `collect` function to handle additional cases in a resource-safe way.

15.4 An extensible process type

Our existing `Process` type implicitly assumes an *environment* or *context* containing a single stream of values. Furthermore, the *protocol* for communicating with the driver is also fixed—a `Process` can only issue three instructions to the driver, `Halt`, `Emit`, and `Await`, and there is no way to extend this protocol short of defining a completely new type. In order to make `Process` extensible, we are going to parameterize on the protocol used for issuing requests of the driver. This works in much the same way as the `IO` type we covered in chapter 13:

```
trait Process[F[_],O]

object Process {
  case class Await[F[_],A,O](
    req: F[A], recv: A => Process[F,O],
    finalizer: Process[F,O]) extends Process[F,O]

  case class Emit[F[_],O](
    head: Stream[O],
    tail: Process[F,O]) extends Process[F,O]

  case class Halt[F[_],O]() extends Process[F,O]
}
```

Unlike `IO`, a `Process[F,O]` represents a *stream* of `O` values ('O' for 'output'), produced by (possibly) making external requests using the protocol `F`. Otherwise, the `F` parameter serves the same role here as the `F` type constructor we used for `IO`.

This type is more general than the previous `Process` (which we'll refer to from now on as a 'single-input `Process`' or a `Process1`), and we can represent single-input `Process` as a special instance of this generalized `Process` type. We'll see how this works in a later section.

First, let's note that a number of operations are defined for `Process`

regardless of the choice of `F`. We can still define `++` ('append'), `flatMap`, `map` and `filter` for `Process`, and the definitions are almost identical to before. Here's `++` and `flatMap` (see chapter code for other functions, including `repeat`, `map`, and `filter`):

```
def ++(p: => Process[F,O]): Process[F,O] = this match {
  case Halt() => p
  case Emit(h, t) => emitAll(h, t ++ p)
  case Await(req,recv,fb) =>
    Await(req, recv andThen (_ ++ p), fb ++ p)
}
```

We use the same smart constructors as before, `emitAll` and `emit`, with similar definitions:

```
def emitAll[F[_],O](head: Seq[O], tail: Process[F,O] = Halt[F,O]()) =
  tail match {
    case Emit(h2,t) => Emit(head ++ h2, t)
    case _ => Emit(head, tail)
  }
def emit[F[_],O](head: O, tail: Process[F,O] = Halt[F,O]()) =
  emitAll(Stream(head), tail)
```

We will also introduce the helper function, `await`, which just curries the `Await` constructor for better type inference:

```
def await[F[_],A,O](req: F[A])(
    recv: A => Process[F,O] = (a: A) => Halt[F,O](),
    fallback: Process[F,O] = Halt[F,O](),
    cleanup: Process[F,O] = Halt[F,O]()): Process[F,O] =
  Await(req, recv, fallback, cleanup)
```

Again, using `++`, we define `flatMap`:

```
def flatMap[O2](f: O => Process[F,O2]): Process[F,O2] =
  this match {
    case Halt() => Halt()
    case Emit(o, t) =>
      if (o.isEmpty) t.flatMap(f)
      else f(o.head) ++ emitAll(o.tail, t).flatMap(f)
    case Await(req,recv,fb,c) =>
      Await(req, recv andThen (_ flatMap f),
            fb flatMap f, c flatMap f)
  }
```

Let's see what else we can express with this new `Process` type. The `F` parameter gives us a lot of flexibility.

15.4.1 Sources

Before, we were forced to introduce a separate type to represent sources. Now, we can represent an effectful source using a `Process[IO,O]`.[11]

Footnote 11 There are some issues with making this representation resource-safe that we'll discuss shortly.

Whereas before, `Source` was a completely separate type from `Process`, now it is merely a particular instance of it! To see how `Process[IO,O]` is indeed a source of `O` values, consider what the `Await` constructor looks like when we substitute `IO` for `F`:

```
case class Await[A,O](
  req: IO[A], recv: A => Process[F,O],
  fallback: Process[IO,O],
  cleanup: Process[IO,O]) extends Process[Step,O]
```

Thus, any requests of the 'external' world can be satisfied, just by running the `IO` action. If this action returns an `A` successfully, we invoke the `recv` function with this result. If the action throws a special exception (perhaps called `End`) it indicates normal termination and we switch to the `fallback` state. And if the action throws any other exception, we switch to the `cleanup` state. Below is simple interpreter of `Source` which collects up all the values emitted:

Here is the exception type `End` that we use for signaling normal termination.[12]

Footnote 12 There are some design decisions here—we are using an exception, `End`, for control flow, but we could choose to indicate normal termination with `Option`, say with `type Step[A] = IO[Option[A]]`, then having `Process[Step,O]` represent sources. We could also choose to pass the exception along to the `recv` function, requiring the `recv` function to take an `Either[Throwable,A]`. We are adopting the convention that any exceptions that bubble all the way up to `collect` are by definition unrecoverable. Programs can certainly choose to throw and catch exceptions internally if they wish.

```
case object End extends Exception
```

With that we define `collect`:

```
def collect[O](src: Process[IO,O]): IndexedSeq[O] = {
  @annotation.tailrec
```

```
  def go(cur: Process[IO,O], acc: IndexedSeq[O]): IndexedSeq[O] =
    cur match {
      case Emit(h,t) => go(t, acc ++ h)
      case Halt() => acc
      case Await(req,recv,fb,err) =>
        val next =
          try recv(req.run)
          catch {
            case End => fb                          // (1)
            case e: Exception => err ++ failIO(e)   // (2)
          }
        go(next, acc)
    }
  go(src, IndexedSeq())
}
```

(1) Normal termination
(2) Helper function, defined below

This uses a helper function, `failIO`:

```
def failIO[O](e: Throwable): Process[IO,O] =
  await[IO,O,O](IO(throw e))()
```

Importantly, we are guaranteed to run either `fallback` or `cleanup` before halting the `Process`, regardless of whether exceptions occur. We'll see later how this allows us to define a `Process` backed by some resource like a file handle that we want to close in a prompt, deterministic fashion.

Notice how in the `Await` case, we run `req` and block waiting for its result before continuing the interpretation. It turns out that we don't require `IO` in particular, any `F` will do, so long as we have a `Monad[F]` and as long as `F` supports catching (and throwing) exceptions. We'll introduce a new interface for this, `Partial`:

```
trait Partial[F[_]] {
  def attempt[A](a: F[A]): F[Either[Throwable,A]]
  def fail[A](t: Throwable): F[A]
}
```

Rather than invoking `run` on our `IO` values, we can simply `flatMap` into the `req` to obtain the result. We define a function on `Process[F,O]` to produce an `F[IndexedSeq[O]]` given a `Monad[F]` and a `Partial[F]`:

```
def collect(implicit F: Monad[F], P: Partial[F]): F[IndexedSeq[O]] = {
  def go(cur: Process[F,O], acc: IndexedSeq[O]): F[IndexedSeq[O]] =
    cur match {
      case Emit(h,t) => go(t, acc ++ h)
      case Halt() => F.unit(acc)
      case Await(req,recv,fb,c) =>
         F.flatMap (P.attempt(req)) {
           case Left(End) => go(fb, acc)
           case Left(err) =>
             go(c ++ await[F,Nothing,O](P.fail(err))(), acc)
           case Right(o) => go(recv(o), acc)
         }
    }
  go(this, IndexedSeq())
}
```

Unlike the simple tail recursive `collect` function above, this implementation is no longer tail recursive, which means our `Monad` instance is now responsible for ensuring constant stack usage. Luckily, the `IO` type we developed in chapter 13 is already suitable for this, and as an added bonus, it supports the use of asynchronous I/O primitives as well.

15.4.2 Ensuring resource safety

`Source` can be used for talking to external resources like files and database connections, but care must be taken to ensure resource safety—we want all file handles to be closed, database connections released, and so on, even (especially!) if exceptions occur. Let's look at what's needed to make this happen.

To make the discussion concrete, suppose we have `lines: Process[IO,String]` representing the lines of some large file. This implicitly references a resource (a file handle) that we want to ensure is closed. When should we close the file handle? At the very end of our program? No, ideally we would close the file once we know we are done reading from `lines`. We are certainly done if we reach the last line of the file—at that point there are no more values to produce and it is certainly safe to close the file. So this gives us our first simple rule to follow: *a resource should close itself immediately after emitting its final value.*

How do we do this? We do this by placing the file-closing action in the `fallback` argument of any `Await`, and the `collect` function(s) above will ensure this gets called before halting (via catching of the `End` exception). But this is not sufficient—we also want to ensure that the file-closing action is run in the event of an uncaught exception. Thus we place the same file-closing action in the

`cleanup` argument to the `Await`, which again the `collect` function will ensure gets called should errors occur.

As an example, let's use this policy to create a `Process[IO,O]` backed by the lines of a file. We define it terms of the more general combinator, `resource`, the `Process` analogue of the `bracket` function we introduced earlier for `IO`:

```
def resource[R,O](acquire: IO[R])(
                  release: R => IO[Unit])(
                  step: R => IO[O]): Process[IO,O] = {
  def go(step: IO[O], onExit: IO[Unit]): Process[IO,O] =
    await[IO,O,O](step) (
      o => emit(o, go(step, onExit)) ❶
    , await[IO,Unit,O](onExit)() ❷
    , await[IO,Unit,O](onExit)()) ❸
  await(acquire) ( r => go(step(r), release(r)), Halt(), Halt() )
}
```

❶ Emit the value and repeat the step action
❷ Release resource when exhausted
❸ Also release in event of error

We can now write `lines` in terms of `resource`:

```
def lines(filename: String): Process[IO,String] =
  resource(IO(io.Source.fromFile(filename)))(
           src => IO(src.close)) { src =>
    lazy val lines = src.getLines // A stateful iterator
    IO { if (lines.hasNext) lines.next else throw End }
  }
```

So far so good. However, we cannot *only* make sure that `lines` keeps its `fallback` and `cleanup` parameters up to date whenever it produces an `Await` —we need to make sure they actually get called. To see a potential problem, consider `collect(lines("names.txt") |> take(5))`. The `take(5)` process will halt early after only 5 elements have been received, possibly before the file has been exhausted. It must therefore make sure before halting that `cleanup` of `lines` is run. Note that `collect` cannot be responsible for this, since `collect` has no idea that the `Process` it is interpreting is internally composed of two other `Process` values, one of which requires finalization.

Thus, we have our second simple rule to follow: *any process, `d`, which pulls*

values from another process, `p`, must ensure the cleanup action of `p` is run before `d` halts.

This sounds rather error prone, but luckily, we get to deal with this concern in just a single place, the `|>` combinator. We'll show how that works shortly in the next section, when we show how to encode single-input processes using our general `Process` type.

15.4.3 Single-input processes

We now have nice, resource-safe sources, but we don't yet have any way to apply transformations to them. Fortunately, our `Process` type can also represent the single-input processes we introduced earlier in this chapter. To represent `Process1[I,O]`, we craft an appropriate `F` that only allows the `Process` to make requests for elements of type `I`. Lets look at how this works—the encoding is a bit unusual in Scala, but there's nothing fundamentally new here:

```
case class Is[I]() {
  sealed trait f[X] { def is: Eq[X,I] }
  val Get = new f[I] { def is = Eq.refl }
}

case class Eq[A,B] private(to: A => B, from: B => A) ❶
object Eq { def refl[A]: Eq[A,A] = Eq(identity, identity) }
```

❶ Evidence that types A and B are equal

It is a bit strange to define the type `f` inside of `One`. Let's unpack what's going on. Notice that `f` takes one parameters, `X`, but we have just one instance, `Get`, which fixes `X` to be the `I` in the outer `One[I]`. Therefore, the type `One[I]#f`[13] can only ever be a request for a value of type `I`! Moreover, we get *evidence* that `X` is equal to `I` in the form of the `Eq[X,I]` which comes equipped with a pair of functions to convert between the two types.[14] We'll see how the `Eq` value gets used a bit later during pattern matching. But now that we have all this, we can define `Process1` as just a type alias:

Footnote 13 Note on syntax: recall that if `x` is a type, `x#foo` references the type `foo` defined inside `x`.

Footnote 14 We are prevented from instantiating an `Eq[Int,String]`, say, because there is only one public constructor, `Eq.refl[A]`, which takes just a single type parameter and uses the identity function for both `to` and `from`.

```
type Process1[I,O] = Process[Is[I]#f, O]
```

To see what's going on, it helps to substitute the definition of `Is[I]#f` into a call to `Await`:

```
case class Await[A,O](
  req: Is[I]#f[A], recv: A => Process[F,O],
  fallback: Process[Is[I]#f,O] = Halt[F,O](),
  cleanup: Process[Is[I]#f,O] = Halt[F,O]()) extends Process[Is[I]#f,R]
```

From the definition of `One[I]#f`, we can see that `req` has just one possible value, `Get: f[I]`. Therefore, `recv` must accept an `I` as its argument, which means that `Await` can only be used to request `I` values. This is important to understand—if this explanation didn't make sense, try working through these definitions on paper, substituting the type definitions.

Our `Process1` alias supports all the same operations as our old single-input `Process`. Let's look at a couple. We first introduce a few helper functions to improve type inference when calling the `Process` constructors:

```
def await1[I,O](recv: I => Process1[I,O],
                fb: Process1[I,O] = Halt()): Process1[I,O] =
  await(Get[I])(recv, fb)

def emit1[I,O](h: O,
               tl: Process1[I,O] = halt1[I,O]): Process1[I,O] =
  emit(h, tl)

def emitAll1[I,O](h: Seq[O],
                  tl: Process1[I,O] = halt1[I,O]): Process1[I,O] =
  emitAll(h, tl)

def halt1[I,O]: Process1[I,O] = Halt[Is[I]#f, O]()
```

Using these, our definitions of, for instance, `lift` and `filter` look almost identical to before, except they return a `Process1`:

```
def lift[I,O](f: I => O): Process1[I,O] =
  await1[I,O](i => emit(f(i))) repeat

def filter[I](f: I => Boolean): Process1[I,I] =
  await1[I,I](i => if (f(i)) emit(i) else halt1) repeat
```

Let's look at process composition next. The implementation looks very similar to before, but we make sure to run the `finalizer` of the left process before the right process halts. (Recall that we are using the `finalizer` argument of `Await` to finalize resources—see the implementation of the `resource` combinator from earlier.)

```
def |>[O2](p2: Process1[O,O2]): Process[F,O2] = {
  // if this is emitting values and p2 is consuming values,
  // we feed p2 in a loop to avoid using stack space
  @annotation.tailrec
  def feed(emit: Seq[O], tail: Process[F,O], recv: O => Process1[O,O2],
           fb: Process1[O,O2], cleanup: Process1[O,O2]): Process[F,O2] =
    if (emit isEmpty) tail |> await1(recv, fb)
    else recv(emit.head) match {
      case Await(_, recv2, fb2, c2) =>
        feed(emit.tail, tail, recv2, fb2, c2)
      case p => Emit(emit.tail, tail) |> p
    }
  p2 match {
    case Halt() => this.kill ++ Halt()
    case Emit(h,t) => emitAll(h, this |> t)
    case Await(req,recv,fb,c) => this match {
      case Emit(h,t) => feed(h, t, recv, fb, c)
      case Halt() => Halt() |> fb
      case Await(req0,recv0,fb0,c0) =>
        await(req0)(i => recv0(i) |> p2, fb0 |> fb, c0 |> c)
    }
  }
}
```

We use a helper function, `kill`—it runs the `cleanup` of a `Process` but ignores any of its remaining output:

```
@annotation.tailrec
final def kill[O2]: Process[F,O2] = this match {
  case Await(req,recv,fb,c) => c.drain
  case Halt() => Halt()
  case Emit(h, t) => t.kill
}

def drain[O2]: Process[F,O2] = this match {
  case Halt() => Halt()
  case Emit(h, t) => t.drain
  case Await(req,recv,fb,c) => Await(
    req, recv andThen (_.drain),
    fb.drain, c.drain)
}
```

Note that `|>` is defined for any `Process[F,O]` type, so this operation works for transforming a `Process1` value, an effectful `Process[IO,O]`, and the two-input `Process` type we will discuss next.

With `|>`, we can add convenience functions on `Process` for attaching various `Process1` transformations to the output. For instance, here's `filter`, defined for any `Process[F,O]`:

```
def filter(f: O => Boolean): Process[F,O] =
  this |> Process.filter(f)
```

We can add similar convenience functions for `take`, `takeWhile`, and so on. See the chapter code for more examples.

15.4.4 Multiple input streams

One of the scenarios we mentioned earlier was zipping or merging of input streams:

'Zip' together two files, `f1.txt` and `f2.txt`, add corresponding temperatures together, convert the result to celsius, apply a 5-element moving average, and output to `celsius.txt`.

We can address these sorts of scenarios with `Process` as well. Much like sources and `Process1` were just a specific instance of our general `Process` type, a `Tee`, which combines two input streams in some way,[15] can also be expressed by our `Process` type. Once again, we simply craft an appropriate choice of `F`:

Footnote 15 The name 'Tee' comes from the letter 'T', which approximates a diagram merging two inputs (the top of the 'T') into a single output.

```
case class T[I,I2]() {
  sealed trait f[X] { def get: Either[Eq[X,I], Eq[X,I2]] }
  val L = new f[I] { def get = Left(Eq.refl) }
  val R = new f[I2] { def get = Right(Eq.refl) }
}
def L[I,I2] = T[I,I2]().L
def R[I,I2] = T[I,I2]().R
```

This looks quite similar to our `Is` type from earlier, except that we now have two possible values, `L` and `R`, and we get an `Either[Eq[X,I], Eq[X,I2]]` for pattern matching. With `T`, we can now define a type alias, `Tee`, which accepts

two inputs:

```
type Tee[I,I2,O] = Process[T[I,I2]#f, O]
```

Once again, we define a few convenience functions for building these particular types of `Process`:

```
def awaitL[I,I2,O](
    recv: I => Tee[I,I2,O],
    fallback: Tee[I,I2,O] = haltT[I,I2,O]): Tee[I,I2,O] =
  await[T[I,I2]#f,I,O](L)(recv, fallback)

def awaitR[I,I2,O](
    recv: I2 => Tee[I,I2,O],
    fallback: Tee[I,I2,O] = haltT[I,I2,O]): Tee[I,I2,O] =
  await[T[I,I2]#f,I2,O](R)(recv, fallback)

def haltT[I,I2,O]: Tee[I,I2,O] =
  Halt[T[I,I2]#f,O]()

def emitT[I,I2,O](h: O, tl: Tee[I,I2,O] = haltT[I,I2,O]): Tee[I,I2,O] =
  emit(h, tl)
```

Let's define some `Tee` combinators. Zipping is a special case of `Tee`—we read from the left, then the right (or vice versa), then emit the pair. Notice we get to be explicit about the order we read from the inputs, a capability that can be important when a `Tee` is talking to streams with external effects.[16]

Footnote 16 We may also wish to be *inexplicit* about the order of the effects, allowing the driver to choose nondeterministically and allowing for the possibility that the driver will execute both effects concurrently. See the chapter notes and chapter code for some additional discussion of this.

```
def zipWith[I,I2,O](f: (I,I2) => O): Tee[I,I2,O] =
  awaitL[I,I2,O](i  =>
  awaitR        (i2 => emitT(f(i,i2)))) repeat

def zip[I,I2]: Tee[I,I2,(I,I2)] = zipWith((_,_))
```

This transducer will halt as soon as either input is exhausted, just like the `zip` funtion on `List`. Let's define a `zipWithAll` which continues as long as either input has elements. We accept a value to 'pad' each side with when its elements run out:

```
```

```
def zipWithAll[I,I2,O](padI: I, padI2: I2)(
                       f: (I,I2) => O): Tee[I,I2,O] = {
  val fbR = passR[I,I2] map (f(padI, _    ))
  val fbL = passL[I,I2] map (f(_   , padI2))
  awaitLOr(fbR)(i =>
  awaitROr(fbL)(i2 => emitT(f(i,i2)))) repeat
}
```

This uses a few helper functions—`passR` and `passL` ignore one of the inputs to a `Tee` and echo the other branch.

```
def passR[I,I2]: Tee[I,I2,I2] = awaitR(emitT(_, passR))
def passL[I,I2]: Tee[I,I2,I] = awaitL(emitT(_, passL))
```

`awaitLOr` and `awaitROr` just call `await` with the `fallback` argument as the first argument, which is a bit more readable here.

```
def awaitLOr[I,I2,O](fallback: Tee[I,I2,O])(
                     recvL: I => Tee[I,I2,O]): Tee[I,I2,O] =
  awaitL(recvL, fallback)

def awaitROr[I,I2,O](fallback: Tee[I,I2,O])(
                     recvR: I2 => Tee[I,I2,O]): Tee[I,I2,O] =
  awaitR(recvR, fallback)
```

There are a lot of other `Tee` combinators we could write. Nothing requires that we read values from each input in lockstep. We could read from one input until some condition is met, then switch to the other; we could read five values from the left, then ten values from the right, read a value from the left then use it to determine how many values to read from the right, and so on.

We will typically want to feed a `Tee` by connecting it to two processes. We can define a function on `Process` that combines *two* processes using a `Tee`. This function works for any `Process` type:

```
def tee[O2,O3](p2: Process[F,O2])(t: Tee[O,O2,O3]): Process[F,O3] =
  t match {
    case Halt() => this.kill ++ p2.kill ++ Halt()
    case Emit(h,t) => Emit(h, (this tee p2)(t))
    case Await(side, recv, fb, c) => side.get match {
      case Left(isO) => this match {
        case Halt() => p2.kill ++ Halt()
        case Emit(o,ot) =>
          feedL(o, ot, p2, isO.to andThen recv, fb, c)
        case Await(reqL, recvL, fbL, cL) =>
```

```
          Await(reqL, recvL andThen (this2 => (this2 tee p2)(t)),
                (fbL tee p2)(t), (cL tee p2)(t))
      }
      case Right(isO2) => p2 match {
        case Halt() => this.kill ++ Halt()
        case Emit(o,ot) =>
          feedR(o, ot, this, isO2.to andThen recv, fb, c)
        case Await(reqR, recvR, fbR, cR) =>
          Await(reqR, recvR andThen (p3 => (this tee p3)(t)),
                (this tee fbR)(t), (this tee cR)(t))
      }
    }
  }
```

This uses two helper functions, `feedL` and `feedR`, which serve the same purpose as before—to feed the `Tee` in a loop as long as it expects values from either side. See the chapter code for the full definition.

The one subtlety in this definition is we make sure to run cleanup for *both* inputs before halting. What is nice about this overall approach is that we have exactly four places in the library where we must do anything to ensure resource safety: `tee`, `|>`, `resource` and the `collect` interpreter. All the other client code that uses these and other combinators is guaranteed to be resource safe.

15.4.5 Sinks

How do we perform output using our `Process` type? We will often want to send the output of a `Source[O]` to some `Sink` (perhaps sending a `Source[String]` to an output file). Somewhat surprisingly, we can represent sinks in terms of sources!

```
type Sink[F[_],O] = Process[F[_], O => F[Unit]]
```

This makes a certain kind of sense. A `Sink[F[_], O]` provides a sequence of functions to call with the input type `O`. The function returns `F[Unit]` (later, we'll see how to get back values from sinks). Let's look at a file `Sink` that writes strings to a file:

```
def fileW(file: String, append: Boolean = false): Sink[IO,String] =
  resource(IO { new java.io.FileWriter(file, append) })(
           w => IO(w.close)) {
    w => IO { (s: String) => IO(w.write(s)) }
  }
```

That was easy. And notice what *isn't* included—there is no exception handling code here—the combinators we are using guarantee that the `FileWriter` will be closed if exceptions occur or when whatever is feeding the `Sink` signals it is done.

We can use `tee` to implement a combinator `to`, which pipes the output of a `Process` to a `Sink`:

```
def to[O2](snk: Sink[F,O]): Process[F,Unit] =
  eval { (this zipWith p2)((o,f) => f(o) }
```

EXERCISE 9: The definition of `to` uses a new combinator, `eval`, defined for any `Process`, which just runs all the actions emitted. Implement `eval`.

```
def eval[F[_],O](p: Process[F, F[O]]): Process[F,O]
```

Using `to`, we can now write programs like:

```
val converter: Process[IO,Unit] =
  lines("fahrenheit.txt").
  filter(!_.startsWith("#")).
  map(line => fahrenheitToCelsius(line.toDouble).toString).
  to(fileW("celsius.txt")).
  drain
```

When run via `collect`, this will open the input file and the output file and incrementally transform the input stream, ignoring commented lines.

15.4.6 Effectful channels

We can generalize `to` to allow responses other than `Unit`. The implementation is identical! The operation had a more general type than we gave it before. Let's call this operation `through`:

```
def through[O2](p2: Process[F, O => F[O2]]): Process[F,O2] =
  eval { (this zipWith p2)((o,f) => f(o)) }
```

Let's introduce a type alias for this pattern:

```
type Channel[F[_],I,O] = Process[F, I => F[O]]
```

`Channel` is useful when a pure pipeline must execute some I/O action as one of its stages. A typical example might be an application that needs to execute database queries. It would be nice if our database queries could return a `Source[Row]`, where `Row` is some representation of a database row. This would allow the program to process the result set of a query using all the fancy stream transducers we've built up so far.

Here's a very simple query executor, which uses `Map[String,Any]` as the (untyped) row representation:

```
import java.sql.{Connection, PreparedStatement, ResultSet}

def query(conn: IO[Connection]):
    Channel[IO, Connection => PreparedStatement,
                Process[IO,Map[String,Any]]] =
  resource(conn)(c => IO(c.close)) { conn => IO {
    (q: Connection => PreparedStatement) => {
      IO { resource ( IO {
        val rs = q(conn).executeQuery
        val ncols = rs.getMetaData.getColumnCount
        val colnames = (1 to ncols).map(rs.getMetaData.getColumnName)
        (rs, colnames)
      }) ( p => IO { p._1.close } ) { // close the ResultSet
        case (rs, cols) => IO {
          if (!rs.next) throw End
          else cols.map(c => (c, rs.getObject(c): Any)).toMap
        }
      }}
    }
  }}
```

We could certainly write a `Channel[PreparedStatement, Source[Map[String,Any]]]`, why don't we do that? Because we don't want code that uses our `Channel` to have to worry about how to obtain a `Connection` (which is needed to build a `PreparedStatement`). That dependency is managed entirely by the `Channel` itself, which also takes care of closing the connection when it is finished executing queries (this is guaranteed by the implementation of `resource`).

This implementation is is directly closing the connection when finished. A real application may obtain the connections from some sort of connection pool and release the connection back to the pool when finished. This can be done just by passing different arguments to the `resource` combinator.

15.4.7 Dynamic resource allocation

Realistic programs may need to allocate resources dynamically, while transforming some input stream. Recall the scenarios we mentioned earlier:

- *Dynamic resource allocation*: Read a file, `fahrenheits.txt`, containing a list of filenames. Concatenate these files into a single logical stream, convert this stream to celsius, and output the joined stream to `celsius.txt`.
- *Multi-sink output:* As above, but rather than producing a single output file, produce an output file for each input file in `fahrenheits.txt`. Name the output file by appending `.celsius` onto the input file name.

Can these capabilities be incorporated into our definition of `Process`, in a way that preserves resource safety? Yes, they can! We actually already have the power to do these things, using the `flatMap` combinator that we have already defined for an arbitrary `Process` type.

For instance, `flatMap` plus our existing combinators let us write this first scenario as:

```
val convertAll: Process[IO,Unit] = (for {
  out <- fileW("celsius.txt").once (1)
  file <- lines("fahrenheits.txt")
  _ <- lines(file).
       map(line => fahrenheitToCelsius(line.toDouble)).
       map(celsius => out(celsius.toString)).
       eval (2)
} yield ()) drain
```

(1) Trim the stream to at most a single element; see chapter code
(2) We can give eval infix syntax using implicits; see chapter code for details

This code is completely resource-safe—all file handles will be closed automatically by the runner as soon as they are finished, even in the presence of exceptions. Any exceptions encountered will be thrown to the `collect` function when invoked.

We can write to multiple files just by switching the order of the calls to `flatMap`:

```
val convertMultisink: Process[IO,Unit] = (for {
  file <- lines("fahrenheits.txt")
  _ <- lines(file).
       map(line => fahrenheitToCelsius(line.toDouble)).
```

```
      map(_ toString).
      to(fileW(file + ".celsius"))
} yield ()) drain
```

And of course, we can attach transformations, mapping, filtering and so on at any point in the process:

```
val convertMultisink2: Process[IO,Unit] = (for {
  file <- lines("fahrenheits.txt")
  _ <- lines(file).
       filter(!_.startsWith("#")).
       map(line => fahrenheitToCelsius(line.toDouble)).
       filter(_ > 0). // ignore below zero temperatures
       map(_ toString).
       to(fileW(file + ".celsius"))
} yield ()) drain
```

There are additional examples using this library in the chapter code.

15.5 Applications

The ideas presented in this chapter are extremely widely applicable. A surprising number of programs can be cast in terms of stream processing—once you are aware of the abstraction, you begin seeing it everywhere. Let's look at some domains where it is applicable:

- *File I/O*: We've already demonstrated how to use stream processing for file I/O. Although we have focused on line-by-line reading and writing for the examples here, we can also use the library for processing binary files.
- *Message processing, state machines, and actors*: Large systems are often organized as a system of loosely-coupled components that communicate via message passing. These systems are often expressed in terms of *actors*, which communicate via explicit message sends and receives. We can express components in these architectures as stream processors, which lets us describe extremely complex state machines and behaviors while retaining a high-level, compositional API.
- *Servers, web applications*: A web application can be thought of as converting a stream of HTTP requests to a stream HTTP responses.
- *UI programming*: We can view individual UI events such as mouseclicks as streams, and the UI as one large network of stream processors determining how the UI responds to user interaction.
- *Big data, distributed systems*: Stream processing libraries can be *distributed* and *parallelized* for processing large amounts of data. The key insight here is that `Process` values being composed need not all live on the same machine.

If you're curious to learn more about these applications (and others), see the chapter notes for additional discussion and links to further reading. The chapter

notes and code also discuss some extensions to the `Process` type we discussed here, including the introduction of *nondeterministic choice* which allows for concurrent evaluation in the execution of a `Process`.

15.6 Conclusion

We began this book with the introduction of a simple premise: that we assemble our programs using only pure functions. From this sole premise and its consequences we were led to develop a new approach to programming, one with its own ideas, techniques, and abstractions. In this final chapter, we constructed a library for stream processing and incremental I/O, demonstrating that we can retain the compositional style developed throughout this book even for programs that interact with the outside world. *Our story is now complete* of how to use FP to architect programs both large and small.

While good design is always hard, over time, *expressing code functionally* becomes effortless. By this point, you have all the tools needed to *start functional programming*, no matter the programming task. FP is a deep subject, and as you apply it to more problems, new ideas and techniques will emerge. Enjoy the journey, keep learning, and good luck!

Index Terms