```
val a = lazyUnit(42 + 1)
val S = Executors.newFixedThreadPool(1)
println(Par.equal(S)(a, fork(a)))
```

Most implementations of `fork` will result in this code deadlocking. Can you see why? Let's look again at our implementation of `fork`:

```
def fork[A](a: => Par[A]): Par[A] =
  es => es.submit(new Callable[A] {
    def call = a(es).get
  })
```

Waits for the result of one `Callable` inside another `Callable`.

Note that we're submitting the `Callable` first, and *within that* `Callable`, we're 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`. They're waiting on each other and therefore our code deadlocks.

EXERCISE 7.9

Hard: 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 (you could simply stipulate that you 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.

Can we fix `fork` to work on fixed-size thread pools? Let's look at a different implementation:

```
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 `ExecutorService` `es`, would run `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 computation until it's actually needed. Let's give it a name, `delay`:

```
def delay[A](fa: => Par[A]): Par[A] =
  es => fa(es)
```

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 share certain patterns in common. In this chapter, we'll begin identifying these patterns and taking advantage of them.

This chapter will be our first introduction to *purely algebraic* structures. We'll consider a simple structure, the *monoid*,[1] which is defined *only by its algebra*. Other than satisfying the same laws, instances of the monoid interface may have little or nothing to do with one another. Nonetheless, we'll see how this algebraic structure is often all we need to write useful, polymorphic functions.

We choose to start with monoids because they're simple, ubiquitous, and useful. Monoids come up all the time in everyday programming, whether we're aware of them or not. Working with lists, concatenating strings, or accumulating the results of a loop can often be phrased in terms of monoids. We'll see how monoids are useful in two ways: they facilitate parallel computation by giving us the freedom to break our problem into chunks that can be computed in parallel; and they can be composed to assemble complex calculations from simpler pieces.

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

[1] The name *monoid* comes from mathematics. In category theory, it means a category with one object. This mathematical connection isn't important for our purposes in this chapter, but see the chapter notes for more information.

the size of its arguments. For instance, consider the runtime performance of this expression:

```
List("lorem", "ipsum", "dolor", "sit").foldLeft("")(_ + _)
```

At every step of the fold, we're allocating the full intermediate `String` only to discard it and allocate a larger string in the next step. Recall that `String` values are immutable, and that evaluating `a + b` for strings `a` and `b` requires allocating a fresh character array and copying both `a` and `b` into this new array. It takes time proportional to `a.length + b.length`.

Here's a trace of the preceding expression being evaluated:

```
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"
```

Note the intermediate strings being created and then immediately discarded. A more efficient strategy would be to combine the sequence by halves, which we call a *balanced fold*—we first construct `"loremipsum"` and `"dolorsit"`, and then add those together.

EXERCISE 10.7

Implement a `foldMap` for `IndexedSeq`.[4] Your implementation should use the strategy of splitting the sequence in two, recursively processing each half, and then adding the answers together with the monoid.

```
def foldMapV[A,B](v: IndexedSeq[A], m: Monoid[B])(f: A => B): B
```

EXERCISE 10.8

Hard: Also implement a *parallel* version of `foldMap` using the library we developed in chapter 7. Hint: Implement `par`, a combinator to promote `Monoid[A]` to a `Monoid[Par[A]]`,[5] and then use this to implement `parFoldMap`.

```
import fpinscala.parallelism.Nonblocking._

def par[A](m: Monoid[A]): Monoid[Par[A]]
def parFoldMap[A,B](v: IndexedSeq[A], m: Monoid[B])(f: A => B): Par[B]
```

[4] Recall that `IndexedSeq` is the interface for immutable data structures supporting efficient random access. It also has efficient `splitAt` and `length` methods.

[5] The ability to "lift" a `Monoid` into the `Par` context is something we'll discuss more generally in chapters 11 and 12.

EXERCISE 11.7

Implement the Kleisli composition function `compose`.

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 11.8

Hard: Implement `flatMap` in terms of `compose`. It seems that we've found another minimal set of monad combinators: `compose` and `unit`.

EXERCISE 11.9

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's an identity element for `compose` in a monad. Indeed, that's exactly what `unit` is, and that's why we chose this name for this operation:[8]

```
def unit[A](a: => A): F[A]
```

This function has the right type to be passed as an argument to `compose`.[9] 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
```

We can also state these laws in terms of `flatMap`, but they're less clear that way:

```
flatMap(x)(unit) == x
flatMap(unit(y))(f) == f(y)
```

[8] The name *unit* is often used in mathematics to mean an identity for some operation.

[9] Not quite, since it takes a non-strict `A` to `F[A]` (it's an `(=> A) => F[A]`), and in Scala this type is different from an ordinary `A => F[A]`. We'll ignore this distinction for now, though.

Listing 12.5 Validating user input in a web form

```
def validName(name: String): Validation[String, String] =
  if (name != "") Success(name)
  else Failure("Name cannot be empty")

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")
  }

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 `map3`:

```
def validWebForm(name: String,
                 birthdate: String,
                 phone: String): Validation[String, WebForm] =
  map3(
    validName(name),
    validBirthdate(birthdate),
    validPhone(phone))(
    WebForm(_,_,_))
```

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

This section walks through the laws for applicative functors.[4] For each of these laws, you may want to verify that they're satisfied by some of the data types we've been working with so far (an easy one to verify is `Option`).

12.5.1 Left and right identity

What sort of laws should we expect applicative functors to obey? Well, we should definitely expect them to obey the functor laws:

```
map(v)(id) == v
map(map(v)(g))(f) == map(v)(f compose g)
```

This implies some other laws for applicative functors because of how we've implemented `map` in terms of `map2` and `unit`. Recall the definition of `map`:

[4] There are various other ways of presenting the laws for `Applicative`. See the chapter notes for more information.

A case class has one primary constructor whose argument list comes after the class name (here, `Charge`). The parameters in this list become public, unmodifiable (immutable) fields of the `class` and can be accessed using the usual object-oriented dot notation, as in `other.cc`.

```
case class Charge(cc: CreditCard, amount: Double) {

  def combine(other: Charge): Charge =

    if (cc == other.cc)

      Charge(cc, amount + other.amount)

    else

      throw new Exception("Can't combine charges to different cards")

}
```

An `if` expression has the same syntax as in Java, but it also returns a value equal to the result of whichever branch is taken. If `cc == other.cc`, then `combine` will return `Charge(..)`; otherwise the exception in the `else` branch will be thrown.

A `case class` can be created without the keyword `new`. We just use the class name followed by the list of arguments for its primary constructor.

The syntax for throwing exceptions is the same as in Java and many other languages. We'll discuss more functional ways of handling error conditions in a later chapter.

Now let's look at `buyCoffees`, to implement the purchase of `n` cups of coffee. Unlike before, this can now be implemented in terms of `buyCoffee`, as we had hoped.

Listing 1.3 Buying multiple cups with `buyCoffees`

```
class Cafe {

  def buyCoffee(cc: CreditCard): (Coffee, Charge) = ...


  def buyCoffees(cc: CreditCard, n: Int): (List[Coffee], Charge) = {
    val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))
    val (coffees, charges) = purchases.unzip
    (coffees, charges.reduce((c1,c2) => c1.combine(c2)))
  }
}
```

`List[Coffee]` is an immutable singly linked list of `Coffee` values. We'll discuss this data type more in chapter 3.

`List.fill(n)(x)` creates a `List` with `n` copies of `x`. We'll explain this funny function call syntax in a later chapter.

`unzip` splits a list of pairs into a pair of lists. Here we're destructuring this pair to declare two values (`coffees` and `charges`) on one line.

`charges.reduce` reduces the entire list of charges to a single charge, using `combine` to combine charges two at a time. `reduce` is an example of a *higher-order function*, which we'll properly introduce in the next chapter.

Overall, this solution is a marked improvement—we're now able to reuse `buyCoffee` directly to define the `buyCoffees` function, and both functions are trivially testable without having to define complicated mock implementations of some `Payments` interface! In fact, the `Cafe` is now completely ignorant of how the `Charge` values will be

`Await`—and there's no way to extend this protocol short of defining a completely new type. In order to make `Process` extensible, we'll parameterize on the protocol used for issuing requests of the driver. This works in much the same way as the `Free` type we covered in chapter 13.

Listing 15.4 An extensible `Process` type

```
trait Process[F[_],O]

object Process {
  case class Await[F[_],A,O](
    req: F[A],
    recv: Either[Throwable, A] => Process[F,O])
  extends Process[F,O]


  case class Emit[F[_],O](
    head: O,
    tail: Process[F,O]) extends Process[F,O]


  case class Halt[F[_],O](err: Throwable) extends Process[F,O]


  case object End extends Exception
  case object Kill extends Exception
}
```

The `recv` function now takes an `Either` so we can handle errors.

Halt due to `err`, which could be an actual error or `End` indicating normal termination.

An `Exception` that indicates normal termination. This allows us to use Scala's exception mechanism for control flow.

An `Exception` that indicates forceful termination. We'll see how this is used later.

Unlike `Free[F,A]`, a `Process[F,O]` represents a *stream* of `O` values (O for output) produced by (possibly) making external requests using the protocol `F` via `Await`. The `F` parameter serves the same role here in `Await` as the `F` parameter used for `Suspend` in `Free` from chapter 13.

The important difference between `Free` and `Process` is that a `Process` can request to `Emit` values multiple times, whereas `Free` always contains one answer in its final `Return`. And instead of terminating with `Return`, the `Process` terminates with `Halt`.

To ensure resource safety when writing processes that close over some resource like a file handle or database connection, the `recv` function of `Await` takes an `Either[Throwable,A]`. This lets `recv` decide what should be done if there's an error while running the request `req`.[5] We'll adopt the convention that the `End` exception indicates that there's no more input, and `Kill` indicates the process is being forcibly terminated and should clean up any resources it's using.[6]

[5] The `recv` function should be trampolined to avoid stack overflows by returning a `TailRec[Process[F,O]]`, but we've omitted this detail here for simplicity.

[6] There are some design decisions here—we're using exceptions, `End` and `Kill`, for control flow, but we could certainly choose to indicate normal termination with `Option`, say with `Halt[F[_],O](err: Option[Throwable])`.

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 we saw `2 + 3` in a program we could simply replace it with the value `5` and it wouldn't change a thing about the meaning of our 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 calling it with RT arguments is also RT. We'll look at some examples next.

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 meaning of `p`. A function `f` is *pure* if the expression `f(x)` is referentially transparent for all referentially transparent `x`.[3]

1.3 Referential transparency, purity, and the substitution model

Let's see how the definition of RT applies to our original `buyCoffee` example:

```
def buyCoffee(cc: CreditCard): Coffee = {
  val cup = new Coffee()
  cc.charge(cup.price)
  cup
}
```

Whatever the return type of `cc.charge(cup.price)` (perhaps it's `Unit`, Scala's equivalent of `void` in other languages), it's discarded by `buyCoffee`. Thus, the result of evaluating `buyCoffee(aliceCreditCard)` will be merely `cup`, which is equivalent to a `new Coffee()`. For `buyCoffee` to be pure, by our definition of RT, it must be the case that `p(buyCoffee(aliceCreditCard))` behaves the same as `p(new Coffee())`, for *any* `p`. This clearly doesn't hold—the program `new Coffee()` doesn't do anything, whereas `buyCoffee(aliceCreditCard)` will contact the credit card company and authorize a charge. Already we have an observable difference between the two programs.

Referential transparency forces the invariant that everything a function *does* is represented by the *value* that it returns, according to the result type of the function. This constraint enables a simple and natural mode of reasoning about program evaluation called the *substitution model*. When expressions are referentially transparent, we can imagine that computation proceeds much like we'd 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

[3] There are some subtleties to this definition, and we'll refine it later in this book. See the chapter notes at our GitHub site (https://github.com/pchiusano/fpinscala; see the preface) for more discussion.

The `math` object contains various standalone mathematical functions including `abs`, `sqrt`, `exp`, and so on. We didn't need to rewrite the `math.abs` function to work with optional values; we just lifted it into the `Option` context after the fact. We can do this for *any* function. Let's look at another example. Suppose we're implementing the logic for a car insurance company's website, which contains a page where users can submit a form to request an instant online quote. We'd like to parse the information from this form and ultimately call our rate function:

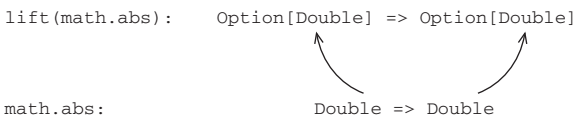
```
/**
 * Top secret formula for computing an annual car
 * insurance premium from two key factors.
 */
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```

We want to be able to call this function, but if the user is submitting their age and number of speeding tickets in a web form, these fields will arrive as simple strings that we have to (try to) parse into integers. This parsing may fail; given a string, `s`, we can attempt to parse it into an `Int` using `s.toInt`, which throws a `NumberFormatException` if the string isn't a valid integer:

```
scala> "112".toInt
res0: Int = 112

scala> "hello".toInt
java.lang.NumberFormatException: For input string: "hello"
  at java.lang.NumberFormatException.forInputString(...)
  ...
```

Lifting functions



`lift(f)` returns a function which maps `None` to `None` and applies `f` to the contents of `Some`. `f` need not be aware of the `Option` type at all.

Let's convert the exception-based API of `toInt` to `Option` and see if we can implement a function `parseInsuranceRateQuote`, which takes the age and number of speeding tickets as strings, and attempts calling the `insuranceRateQuote` function if parsing both values is successful.

Listing 4.3 Using `Option`

```
def parseInsuranceRateQuote(
    age: String,
    numberOfSpeedingTickets: String): Option[Double] = {
  val optAge: Option[Int] = Try(age.toInt)
  val optTickets: Option[Int] = Try(numberOfSpeedingTickets.toInt)
```

The `toInt` method is available on any `String`.

6 Purely functional state

In this chapter, we'll see how to write purely functional programs that manipulate state, using the simple domain of *random number generation* as the example. Although by itself it's not the most compelling use case for the techniques in this chapter, the simplicity of random number generation makes it a good first example. We'll see more compelling use cases in parts 3 and 4 of the book, especially part 4, where we'll say a lot more about dealing with state and effects. The goal here is to give you the basic pattern for how to make *any* stateful API purely functional. As you start writing your own functional APIs, you'll likely run into many of the same questions that we'll explore here.

6.1 Generating random numbers using side effects

If you need to generate random[1] numbers in Scala, there's a class in the standard library, `scala.util.Random`,[2] with a pretty typical imperative API that relies on side effects. Here's an example of its use.

Listing 6.1 Using `scala.util.Random` to generate random numbers

```
scala> val rng = new scala.util.Random

scala> rng.nextDouble
res1: Double = 0.9867076608154569

scala> rng.nextDouble
res2: Double = 0.8455696498024141

scala> rng.nextInt
```

Creates a new random number generator seeded with the current system time

[1] Actually, pseudo-random, but we'll ignore this distinction.
[2] Scala API link: http://mng.bz/3DP7.