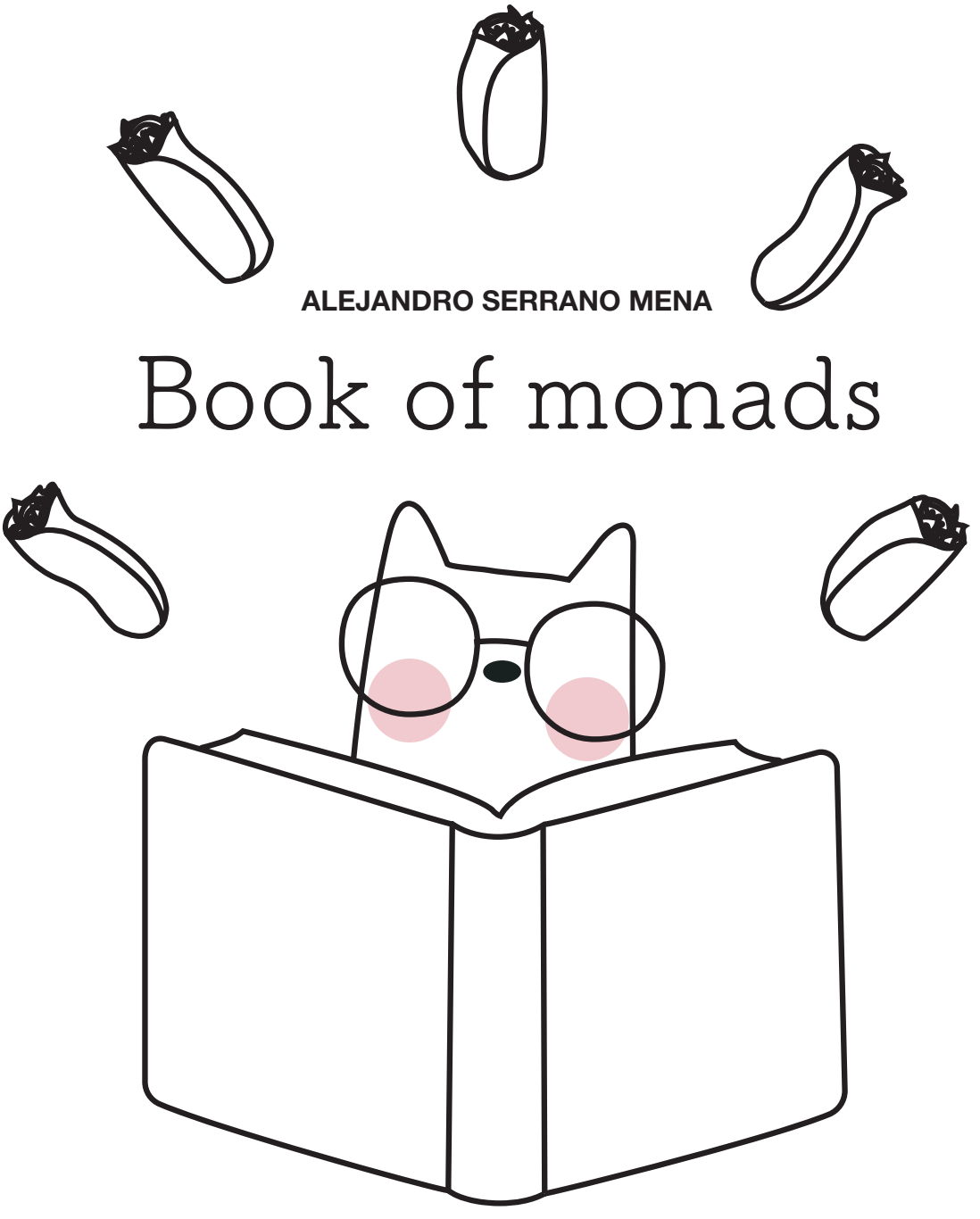
ALEJANDRO SERRANO MENA

Book of monads



Cover page
Blanca Vielva Gómez

Reviewers (1st edition)
Nicolas Biri
Harold Carr
John A. De Goes
Oli Makhasoeva
Steven Syrek

Reviewers (2nd edition)
Coming soon

Contents

0 Introduction

Welcome to the Book of Monads! My aim with this book is to guide you through a number of topics related to one of the core — and at the same time one of the most misunderstood — concepts in modern, functional programming. The choice of topics is guided by pragmatic considerations, in particular what works and is used by the community, with an occasional detour into theory. Other programming concepts such as functors, applicatives, and continuations are introduced wherever their relation to monads is interesting or leads to further insight.

Roadmap. As you have already seen, this book is divided into five sections.

Part I is concerned with generalities about monads — in other words, with those elements that all monads and monadic computations share. Apart from these core concepts, we discuss the special monadic notation that many functional languages provide as well as generic functions that work on every monad.

Part II goes to the opposite end of the spectrum. Each of the chapters in this part describes one specific monad that is frequently encountered in the wild: `Reader`, `Writer`, `State`, `Maybe` or `Option`, `Either`, `List` (`[]` for Haskellers), `IO`, and `Resource`, among others. Knowing which monads you have at hand is as important as understanding the generic functionality that all monads share.

Part III describes one of the primary approaches for combining the functionality of several monads, namely, monad transformers, along with their advantages and shortcomings. One of the concerns with monad transformers is how lifting — in short, injecting an operation from an individual into a combined, or higher-order, monad — quickly becomes difficult.

Part IV will help you "transform" from a passive consumer of monads that others have defined into an active producer of your own monadic types. We review all the approaches — final, initial, and operational style — and how the common pattern in each of them can be abstracted away to give rise to *free* and *freer* monads. This

is the part of the book in which we explore some new developments that compete with monads, such as effects.

Part V has a more concrete goal: to give meaning to the phrase, "a monad is a monoid in the category of endofunctors," which has become a well-known meme for functional programmers. To do so, we turn to the mathematical foundation of monads, category theory. Finally, we look at the relation between monads and another important categorical concept: adjunctions.

Conventions. Throughout this book, we show many snippets of code. Most of them feature both Haskell and Scala code and occasionally other functional programming languages. We follow a color convention to distinguish among the different languages:

```haskell
data Bool = True | False -- a simple data type
-- Define conjunction
or :: Bool -> Bool -> Bool
or True  _ = True
or False x = x
```

```scala
sealed abstract class Boolean  // a simple case class
case object True extends Boolean
case object False extends Boolean
// Define conjunction
def or(x: Boolean, y: Boolean) = x match {
  case True => True
  case _ => y
}
```

Sometimes, the code blocks are too long. In those cases, we usually show only the Haskell version, except when we want to highlight something specific in Scala or another language.

> **Exercise 0.1.** Think about the precise way in which the beginning of all exercises in this book are labeled *Exercise*.

Prerequisites. In order to follow along with this book, you only need to have a basic command of statically-typed functional programming, as found in languages such as Haskell, Scala, F#, or OCaml. In particular, we assume that you know about algebraic data types, pattern matching, and higher-order functions. If you need to refresh your memory, there is a wide range of beginner books such as *Practical Haskell* [Serrano Mena, 2019] (from the same author of this book), *Programming in Haskell* [Hutton, 2016], *Learn You a Haskell for Great Good!* [Lipovača, 2011], *Get*

Programming with Haskell [Kurt, 2018], *Haskell Programming from First Principles* [Allen and Moronuki, 2015], and *Essential Scala* [Gurnell and Welsh, 2015]. In the rest of this chapter, we discuss some intermediate topics that are important for understanding the contents of this book.

0.1 Type Classes

Almost every programming language provides a feature for declaring that a certain operation exists for a given type. The archetypal example is stating that a type has a notion of equality, in other words, that we can compare two values of that type to check whether they are equal. Numbers and strings usually provide that functionality, for example, whereas functions cannot be compared, in most cases.

Object-oriented languages use inheritance to declare such operations. Haskell takes another approach, using *type classes* and *instances*.* A type class declaration introduces a name and a set of functions (sometimes called methods) that a member of that class must provide. Our example of equality looks like this:

```
class Eq a where
  (==) :: a -> a -> Bool
```

One difference between this style and that of other languages is that a variable — `a` in this case — is introduced in the header of the class to refer to any possible instance of that variable in the methods. Another important difference is that a function such as `(==)` not only requires its arguments to support equality, but those arguments must also be of the same type, since the same `a` is used in both positions.

If you now use `(==)` in a function, it reflects the constraint that some of the types used in that function must be members of the `Eq` type class. Take, for example, the function that compares two lists for equality:

```
eqList :: Eq a => [a] -> [a] -> Bool
eqList []     []     = True
eqList (x:xs) (y:ys) = x == y && eqList xs ys
eqList _      _      = False
```

This function works over two lists of the same type — since both arguments are of type `[a]` – where the type of their elements supports equality. It returns a true value only when the two lists contain exactly the same elements in the same order.

We can now define functions that work generically over any `Eq` type. But how do we declare that any given type provides that functionality? We define an *instance*, as follows:

*Confusingly, these terms have a different meaning in Haskell than they do in common object-oriented languages. In Scala, *classes* and *type classes* are unrelated.

```haskell
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

The header of the instance looks similar to that of the class. The difference is that the type variable is replaced by the actual type we are dealing with — `Bool` in this case. Instead of the type signature of the methods, we provide the corresponding *implementation*. Once we do this, we may use `eqList` to compare lists of Booleans.

One of the cool features of type classes is that instances may depend on the availability of other instances. We have just defined a way to compare two lists, but it only works if the elements themselves are comparable. We can turn this idea into an actual instance for lists, where the constraint on elements is also present:

```haskell
instance Eq a => Eq [a] where
  (==) = eqList
```

The link between the instance we are declaring — `Eq` for lists — and the prerequisites is given by means of shared type variables, `a` in this case.

> **Exercise 0.2.** Define the `Eq` instance for tuples `(a, b)`. In case you need more than one prerequisite in the declaration, the syntax is:
>
> ```haskell
> instance (Prereq1, Prereq2, ...) => Eq (a, b)
> ```

0.1.1 Type Classes in Scala: Implicits and Givens

Scala, in contrast to Haskell, provides many different ways to abstract common functionality. One common way is using object-oriented features such as classes and traits. In Scala 2 we can use *implicits* to model type classes within the language; this is usually called the *type class pattern*. This pattern has "graduated" in Scala 3, in which *contextual abstractions* provide a similar functionality.

When using the type class pattern, the declaration of which functionality each type should provide is given in the form of a trait. Such a trait always takes a type parameter, which represents the type we will be working with. The reason is that we want to stay away from inheritance. By using implicits, we can mimic the instance resolution of Haskell instead:*

```scala
trait Eq[A] {
  def eq(x: A, y: A): Boolean
}
```

*Braces are optional in Scala 3.

Similarly to the Haskell version, by sharing a single type parameter, we require the types of the two arguments to `eq` to coincide.

Now, if you require an implementation of a trait in a given function, you just include it as an additional argument. In the body of the function, you can use the trait argument to access every operation on it:*

```
def eqList[A](xs: List[A], ys: List[A], eq: Eq[A])
    : Boolean = ???
```

Just doing this would result in an explosion of code, however. Every time you call `eqList`, you would need to provide that `eq` argument. Luckily, the Scala compiler provides either implicits (in Scala 2) or using clauses (in Scala 3) to solve that exact problem. If we mark one or more arguments using the aforementioned features, then whenever the function is called, the compiler searches the current scope for the right `eq` implicit value:

```
// Scala 2 uses implicits
def eqList[A](xs: List[A], ys: List[A])(implicit eq: Eq[A])
    : Boolean = ???

// Scala 3 uses contextual abstractions
def eqList[A](xs: List[A], ys: List[A])(using eq: Eq[A])
    : Boolean = ???
```

Note that not every value will be included in the search, only those explicitly marked. This is done by annotating the value with the `implicit` or `using` keyword, depending on the language version:

```
// Scala 2 uses 'implicit'
implicit val eqBoolean: Eq[Boolean] = new Eq[Boolean] {
  def eq(x: Boolean, y: Boolean): Boolean = (x, y) match {
    case (True, True) => True
    case (False, False) => True
    case _ => False
  }
}
// Scala 3 uses 'given'
given eqBoolean: Eq[Boolean] with {
  def eq(x: Boolean, y: Boolean): Boolean = (x, y) match {
    case (True, True) => True
    case (False, False) => True
    case _ => False
  }
}
```

*It is customary in Scala to use the value `???` of type `Nothing` to mark code that has yet to be written.

Furthermore, if the declaration itself has implicit or contextual parameters, those are searched for recursively. In this way, we can make membership to a class depend on some further constraints, as we did for lists in Haskell:

```
// Scala 2 uses the same keyword
implicit def eqList[A]
  (implicit eqElement: Eq[A]): Eq[List[A]] = ???

// Scala 3 uses two different keywords
given eqList[A](using eqElement: Eq[A]): Eq[List[A]] = ???
```

Exercise 0.3. Define the function `notEq`, which returns `False` if the given arguments are not equal. You should use the `Eq` trait defined above.

Exercise 0.4. Write the implicit or given required to create `Eq` for tuples.

This description only touches the tip of the iceberg of defining type class-like hierarchies in Scala; when using Scala 2, you would also define some companion objects to make working with them much easier. Scala 3 greatly improves the ergonomics of type classes, but even in Scala 2 the excellent Simulacrum library provides a `@typeclass` macro that generates most of the boilerplate for you. In the rest of the book, in order to make our descriptions more concise, we will assume that the concrete monad types implement the core operations as part of the class, so we do not need to pass implicit parameters all the time.

0.2 Higher-kinded Abstraction

Containers are also useful abstractions that we can form using a wide variety of types. For example, lists, sets, queues, and search trees are all generally defined with the ability both to insert an element into an existing data structure of the corresponding type and to create a new, empty structure:*

```
[]    ::      [a]
empty :: Set    a
empty :: Tree   a
empty :: Queue  a
```

```
(:)    :: a ->      [a] ->      [a]
insert :: a -> Set    a -> Set    a
insert :: a -> Tree   a -> Tree   a
push   :: a -> Queue  a -> Queue  a
```

One peculiarity of these containers is that the same set of operations is available *irrespective* of the types of the elements they contain. The only restriction is that if a container starts its life with a given type of element, only more elements of that type can be inserted, as the types of `insert` and `push` show.

*Here is our first example of the obsessive behavior of Haskellers to *line things up* in order to see the commonalities among types.

If we want to create a type class encompassing all of these types, the abstraction is not in the elements. The moving parts here are `[]`, `Set`, `Tree`, and `Queue`. One property that they share is that they require a type argument to turn them into real types. We say that they are *type constructors*. In other words, we cannot have a value or parameter of type `Set` — we need to write `Set Int` or `Set[Int]`.

The definition of a `Container` type class in Haskell does not obviously reflect that we are abstracting over a type constructor. The only way we can notice this fact is by observing how the variable `c` is used — in this case, `c` is applied to yet another type `a`, which means that `c` must be a type constructor:

```
class Container c where
  empty  :: c a
  insert :: a -> c a -> c a
```

Scala is more explicit, in this respect. If a trait is parametrized by a type constructor, it has to be marked as such. Given that type application in Scala is done with square brackets, as in `Set[Int]`, the need for a type argument is declared by one or more underscores between square brackets:

```
trait Container[C[_]] {
  def empty[A]: C[A]
  def insert[A](x: A, xs: C[A]): C[A]
}
```

In Scala, in contrast to Haskell, every type argument must be explicitly introduced. This makes it easier to identify where each type variable is coming from. In the previous code, `C` refers to the container type we are abstracting over, whereas `A` refers to the element type we use in each function.

> **Exercise 0.5.** Write the `List` instance, implicit value, or given for the `Container` type class.

Abstraction over type constructors is also known as *higher-kinded abstraction*. This form of abstraction is fundamental to the rest of this book. Almost every new structure we introduce, including monads, is going to abstract some commonality over several type constructors. Haskell (and its derivatives such as PureScript) and Scala are the only mainstream languages with built-in support for higher-kinded abstraction, although you can encode it via different tricks in languages such as F# and Kotlin — this is the reason our code blocks are mostly written in Haskell and Scala.

0.3 Haskell's Newtype

The most common way to write a `Container` instance for lists is to use them as stacks, that is, to insert new elements at the beginning of a list:

```
instance Container [] where
  empty       = []
  insert x xs = x:xs
  -- or insert = (:)
```

But there are many other ways to abide by this interface. For example, the inserted elements could go at the end of the list, simulating a queue:

```
instance Container [] where
  empty       = []
  insert x xs = xs ++ [x]  -- (++) is concatenation
```

If you try to include (or even import) both definitions in the same piece of code, the compiler complains about *overlapping instances*. The problem is that Haskell uses the type of the data to decide which instance to use. But consider the following code:

```
insertTwice :: a -> [a] -> [x]
insertTwice x xs = insert x (insert x xs)
```

It is not possible to know which of the previous instances the programmer is referring to, since *both* apply to a list of values `xs`. Whereas Scala offers more control when searching for implicits, Haskell takes the simpler path and rejects programs with multiple instances that might be applicable for the same type (unless you enable the `OverlappingInstances` language extension, which is usually a bad idea).

The solution to this problem is to turn one of the instances into its own data type. For this purpose, we only need one constructor with one field, which holds the data:

```
data Queue a = Queue { unQueue :: [a] }
```

In the eyes of the compiler, `Queue` is *completely different* from `[]`. Thus, we can write an instance for it without fear of overlapping with the previous one. The downside is that now we have to pattern match and apply the constructor every time we want to access or build the contents of the `Queue` type. For example, the implementation of the instance reads:

```
instance Container Queue where
  empty = Queue []
  insert x (Queue xs) = Queue (xs ++ [x])
  -- or using the field accessor unQueue
  insert x xs = Queue (unQueue xs ++ [x])
```

This pattern is so common in Haskell code that the compiler includes a specific construct for data types like `Queue`, above: one constructor with one field. Instead of using `data`, we can use the `newtype` keyword:

```
newtype Queue a = Queue { unQueue :: [a] }
```

The compiler then ensures that no extra memory is allocated other than that which is used for the single, wrapped value. Furthermore, all uses of the constructor for pattern matching are completely erased in the compiled code. A `newtype` is merely a way to direct the compiler to choose the correct instance.

0.4 Language Extensions in Haskell

The Haskell programming language is defined in a standard, the current incarnation of which is Haskell 2010 (the previous one was Haskell 98). This standard defines the minimum set of constructions that ought to be recognized for a compiler to call itself "a Haskell compiler." In practice, though, just about everybody uses the Glasgow Haskell Compiler — GHC for short — and we follow that practice in this book.

GHC extends the language in many different directions. By default, however, it only allows constructions defined in the standard. To use the rest of them, you need to use different *language extensions*. Each extension provides additional syntax, new elements for the language, or richer types.

Let us assume you want to enable the `MultiParamTypeClasses` extension (which provides the ability for type classes to have more than one parameter). How to enable an extension depends on whether you want to do it in a file or in an interactive session. In the former case, you need to add the following line *at the top* of your file, even before the `module` declaration:

```
{-# LANGUAGE MultiParamTypeClasses #-}
```

If you need to enable more than one extension, you can either add additional `LANGUAGE` lines or put all of the extensions on one line, separating them with commas. The other case is enabling an extension in the GHC interpreter. To do that, you need to enter the following at the REPL prompt:

```
> :set -XMultiParamTypeClasses
```

Note that the name of the extension to enable is preceded by `-X`. You can also turn off an extension by using `-XNo` before its name. Be aware that not all extensions can be disabled without restarting the interpreter.

Required extensions for GHC 8. The following extensions are required for different parts of this book, although not all of them will be required at the same time. Note

that the name and restrictions of each extension may differ depending on your version of GHC, so if you are following the book with an older or more recent compiler, you may need to adjust them:

- Extensions related to type classes: `MultiParamTypeClasses`, `FunctionalDependencies`, `TypeSynonymInstances`, `FlexibleContexts`, `FlexibleInstances`, `InstanceSigs`, `UndecidableInstances`.
- Extensions to the type system: `GADTs`, `RankNTypes`, `PolyKinds`, `ScopedTypeVariables`, `DataKinds`, `KindSignatures`, `TypeApplications`.
- Extensions to the language syntax: `LambdaCase`, `TypeOperators`.
- Extensions related to monads: `MonadComprehensions`, `ApplicativeDo`.
- Extensions to the `deriving` functionality: `DeriveFunctor`, `GeneralizedNewtypeDeriving`.
- Only for part V: `ConstraintKinds`, `TypeInType` (not required anymore since GHC 8.6).

It's time. The world of monads awaits after a mere turn of the page. Get ready!

Part I

What is a Monad?

1 Discovering Monads

Monad is an abstract concept, which we programmers discovered after many attempts to refactor and generalize code. One way to introduce monads is to state the definition and derive information from it — a mathematical, deductive style.

In this chapter, however, we will try to arrive at the concept of a monad by means of a series of examples. In order to help you cultivate an intuition for it, words such as "context" or "box" are used throughout the chapter. Feel free to ignore those parts that do not make sense to you in order to come to your own understanding of what a monad is.

1.1 State Contexts

Take a simple type representing binary trees that stores information in its leaves. This can be represented as an *algebraic data type* in Haskell as follows:

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

In Scala, the declaration of binary trees is similar:

```scala
sealed abstract class Tree[A]  // binary trees
case class Leaf[A](value: A) extends Tree[A]
case class Node[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

As a brief reminder of what each part of this declaration represents:

- The first step is declaring the data type itself and how many type parameters it takes. In this case, a `Tree` is *parametrized* by yet another type, which means that `Tree` cannot stand on its own as the type of an argument to a function. Rather, we need to specify the type of its elements as `Tree Int` in Haskell or `Tree[String]` in Scala.

In the case of Haskell, this information appears right after the `data` keyword, and each subsequent type parameter starts with a lowercase letter. In Scala, the declaration is decoupled as a set of classes. The parent class is `Tree[A]`, and it declares one type parameter between square brackets.

- In Haskell, after the equals sign, we find two *constructors* separated by a vertical bar. Those are the functions that are used to build new elements of the data type. They are also the basic patterns you use to match on a `Tree` in a function definition.

  The first constructor is called `Leaf` and holds one piece of information of the type given as the argument to `Tree`. For example, `Leaf True` is a value of type `Tree Bool`, since `True` is of type `Bool`. The second constructor, `Node`, represents an internal node. This constructor is recursive, since it holds two other trees inside of it.

  The same information is represented in Scala by a series of *case classes* that extend the previously defined `Tree[A]`. These classes are treated in a special way by the compiler — in particular, they enable the use of pattern matching. In contrast to Haskell, fields in a case class must be given a name.

As a first example, let us write a function that counts the number of leaves in a binary tree:

```
numberOfLeaves :: Tree a -> Integer
numberOfLeaves (Leaf _)   = 1
numberOfLeaves (Node l r) = numberOfLeaves l + numberOfLeaves r
```

```
def numberOfLeaves[A](t: Tree[A]): Int = t match {
  case Leaf(_) => 1
  case Node(l, r) => numberOfLeaves(l) + numberOfLeaves(r)
}
```

This function works in a recursive fashion: the output of the function `numberOfLeaves` applied to the subtrees of a node is enough to compute the result for the node itself. To declare how each constructor or case in the data type ought to be handled, we make use of *pattern matching*. In Scala, this functionality needs an explicit keyword, `match`, whereas in Haskell the patterns can be declared as a set of equations.

Things get a bit hairier for our second example: relabeling the leaves of the tree left-to-right. If you start with a tree `t`, the result of `relabel` should contain the same elements, but with each one paired with the index it would receive if the leaves of `t` were flattened into a list starting with the leftmost leaf and ending with the rightmost one, as shown in Figure 1.1. We can spend a long time trying to implement this function in a simple recursive fashion, as we did for `numberOfLeaves`, only to fail. This suggests that we must hold onto some extra information throughout the procedure.

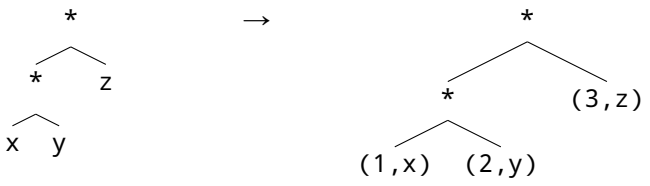


Figure 1.1: Example of left-to-right relabeling

Let us look at the problem more closely, focusing on how we would implement the following:

```
relabel :: Tree a -> Tree (Int, a)
relabel (Leaf x) = (???, x)
```

The missing information that we mark with `???` is the index to be returned. But every `Leaf` should be relabeled with a different index! A straightforward solution is to get this information from the outside, as an argument:

```
relabel :: Tree a -> Int -> Tree (Int, a)
relabel (Leaf x) i = Leaf (i, x)
```

The next step is thinking about the case of `Node`s. In order to relabel the whole tree, we should relabel each of the subtrees. For the left side, we can just reuse the index passed as an argument:

```
relabel (Node l r) i = Node (relabel l i) (relabel r ???)
```

The problem now is that we do not know which index to use in relabeling the right subtree. The naïve answer would be to use `i + 1`. Alas, this does not give a correct result, since we do not know in advance how many `Leaf`s the left subtree has. What we can do is return that information *in addition* to the relabeled subtree:

```
relabel :: Tree a -> Int -> (Tree (Int, a), Int)
relabel (Leaf x)   i = (Leaf (i, x), i+1)
relabel (Node l r) i = let (l', i1) = relabel l i
                           (r', i2) = relabel r i1
                       in (Node l' r', i2)
```

```
def relabel[A](t: Tree[A], i: Int): (Tree[(Int, A)], Int) = t match {
  case Leaf(x) => (Leaf((i, x)), i + 1)
  case Node(l, r) => {
    val (ll, i1) = relabel(l, i)
    val (rr, i2) = relabel(r, i1)
    (Node(ll, rr), i2)
  }
}
```

This `relabel` function works, but it would be rather ugly in the eyes of many functional programmers. The main issue is that you are mixing the *logic* of the program, which is simple and recursive, with its *plumbing*, which handles the index at each stage. Passing and obtaining the current index is quite tedious and, at the same time, has terrible consequences if it's wrong — if we accidentally switch `i1` and `i2`, this function would return the wrong result.

In order to separate out the plumbing, we can introduce a type synonym for functions that depend on a counter:

```
type WithCounter a = Int -> (a, Int)
```

```
type WithCounter[A] = Int => (A, Int)
```

This way, the type of `relabel` becomes `Tree a -> WithCounter (Tree a)`.

Now, let us look at the patterns that we find in the `Node` branch. The first one is the nesting of `let` expressions (or `val` declarations, depending on the language):

```
let (r, newCounter) = ... oldCounter
 in nextAction ... r ... newCounter
```

Using the power of higher-order functions, we can turn this pattern into its own function:

```
next :: WithCounter a -> (a -> WithCounter b) -> WithCounter b
f `next` g = \i -> let (r, i') = f i in g r i'
```

In Scala we prefer to use `next` in infix position. Instead of the following code:

```
next(relabel(l)) { ll => ??? /* the next thing to do */ }
```

We would rather have the the plumbing operation, `next`, moved from the top-level role to a secondary status:

```
relabel(l) next { ll => ??? /* next thing to do */ }
```

There is a problem, though, `WithCounter` is defined as a function type, and we have no control over that type. That means that we cannot add our desired method `next` directly. In Scala 2 we need to define an *implicit class*:

```
implicit class RichWithCounter[A](f: WithCounter[A]) {
  def next[B](g: A => WithCounter[B]): WithCounter[B] = i => {
    val (r, i1) = f(i)
    g(r)(i1)
  }
}
```

Under the hood, the Scala compiler rewrites the code above using `next` to introduce a conversion to `RichWithCounter`:

```
RichWithCounter(relabel(l)).next({ ll => ??? /* next thing to do */ })
```

This (somehow convoluted) pattern is no longer required in Scala 3. In this case you simply use an *extension method*:

```
extension (f: WithCounter[A]) {
  def next[B](g: A => WithCounter[B]): WithCounter[B] = i => {
    val (r, i1) = f(i)
    g(r)(i1)
  }
}
```

We will assume for the rest of the book that such implicit classes or extensions are defined whenever required.

The second pattern that we find is returning a value with the counter unchanged:

```
pure :: a -> WithCounter a
pure x = \i -> (x, i)
```

```
def pure[A](x: A): WithCounter[A] = i => (x, i)
```

This is all the plumbing we need to make `relabel` shine!

```
relabel (Leaf x)   = \i -> (Leaf (i,x), i + 1)
relabel (Node l r) = relabel l `next` \l' ->
                     relabel r `next` \r' ->
                     pure (Node l' r')
```

```
def relabel[A](t: Tree[A]): WithCounter[Tree[(A, Int)]] = t match {
  case Leaf(x) => i => (Leaf((x, i)), i + 1)
  case Node(l, r) => relabel(l) next { ll =>
                       relabel(r) next { rr =>
                         pure(Node(ll, rr)) } }
}
```

Note that even though we now describe the type of `relabel` as `Tree a -> WithCounter (Tree (a, Int))`, once we expand the synonyms we get a plain `Tree a -> Int -> (Tree (a, Int), Int)`. In particular, to call the function we simply use the following without any additional ceremony:

```
relabel myTree a
```

```
relabel(myTree)(0)
```

Throughout this book, we will use this technique of hiding some details of our types in a synonym and only unveil some of those details in particular circumstances.

The previous example uses a concrete type, `WithCounter`, to maintain an index in combination with an actual value. But in the same vein, we can keep other kinds of *state* and thread them through each of a series of computations. The following type synonym has two type variables — the first one is the state itself, and the second is the value type:

```
type State s a = s -> (a, s)
```

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

In fact, our old `WithCounter` is nothing other than `State Int`!

> **Exercise 1.1.** Rewrite the definitions of `pure` and `next` to work with an arbitrary stateful computation `State s a`. Hint: you only need to change the type signatures.

In the functional world, we refer to the pattern that `State` embodies in several ways. We say that `State` adds a *context* to a value. An element of type `State s a` is not a single value but rather a transformation of a given state into a new state and a result value. We also say that `next` makes `State` work in a sequential fashion: the result of one computation is fed to the next computation in the sequence.

1.2 Magical Multiplying Boxes

Binary trees are simple data structures, but in Haskell there is a simpler and more essential container: *lists*. The special syntax for lists is baked into the compiler, but we can think of the type `[a]` or `List[A]` as being defined in the following way:

```
data [a] = [] | a : [a]
```

In Scala, there is no built-in syntax for linked lists, but they are defined similarly:

```
sealed abstract class List[+A]  // your usual lists
case object Nil extends List[Nothing]
case class ::[A](hd: A, tl: List[A]) extends List[A]
```

The `+` before the `A` in `List[+A]` declares the `List` type to be covariant in the element type. This points to the interplay between functional and object-oriented features in Scala: if we have a `List[Cat]`, we want to be able to use it wherever we require a `List[Mammal]`. Subtyping plays no role in this book, however. We will encounter variance annotations only when describing built-in classes.

The first techniques that a functional programmer learns when working with lists are *pattern matching* — defining different branches for the empty list and the

cons, with a head and a tail — and *recursion* — using the result of the same function on the tail of the list to build the result for the whole. As a reminder, here is how you compute the `length` of a list:

```haskell
length :: [a] -> Integer
length []     = 0
length (_:xs) = 1 + length xs
```

```scala
def length[A](lst: List[A]) : Int = lst match {
  case Nil => 0
  case _ :: xs => 1 + length(xs)
}
```

Exercise 1.2. Write a function `(++)` that takes two lists and returns its concatenation. That is, given two lists `l1` and `l2`, `l1 ++ l2` contains the elements of `l1` followed by the elements of `l2`.

This is such a boring way to look at lists! I prefer to think of them as magical devices, incredible boxes with not one but a series of objects of the same type. Clearly, you do not want to open the boxes, because the magic might fade away. Fear not! The device comes with several spells — usually referred to as functions — that allow us to manipulate its contents.

The best-known spell for lists is `map`. This function receives a description of how to treat one element — that is, yet another function — which is then applied to all the elements in the list. Its type shows that information in a clear way:

```haskell
map :: (a -> b) -> [a] -> [b]
```

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

Exercise 1.3. Do you remember how `map` is defined? Try it out!

The second spell is less well-known but much simpler. The `singleton` function takes one value and, using energy coming from black holes in outer space, packs it into a list. Well, I guess I am being too literal — a singleton list is actually just a list with one element!

```haskell
singleton :: a -> [a]
singleton x = [x]  -- or x : []
```

```scala
def singleton[A](x: A): List[A] = ::(x, Nil)
// Using the built-in List class, you can write List(x) instead
```

Note the similarity of the type with that of `pure` as defined for state contexts, `a -> State s a`. In both cases, we get a pure value and "inject" it into either a context or a magical box.

The third spell is my favorite, `concat`. Let me first show its type and definition:

```haskell
concat :: [[a]] -> [a]
concat []     = []
concat (x:xs) = x ++ concat xs
-- alternatively
concat = foldr (++) []
```

```scala
// This operation is called flatten in the built-in List class
def concat[A](lst: List[List[A]]): List[A] = lst match {
  case Nil => Nil
  case ::(x, xs) => x ++ concat(xs)
}
```

What is so magical about this function? Well, it takes a *list of lists* and somehow manages to turn it into *one flattened list*. I picture leprechauns obsessively opening the inner boxes and throwing their contents back into the outer box.

Jokes aside, this is another point of view that is useful for understanding these concepts. A list is a kind of "box" that supports three operations: (1) mapping a function over all the elements contained in it; (2) creating a box from a single element; and (3) flattening nested boxes of boxes into a single layer.

1.3 Both, Maybe? I Don't Think That's an Option

One of the more widely-advertised features of functional languages — but in fact coming from their support for algebraic data types — is how they avoid the "billion dollar mistake," that is, having a `null` value that fits into every type but which raises an error when you try to access it.*

Since creating new data types is so cheap, and it is possible to work with them polymorphically, most functional languages define some notion of an *optional value*. In Haskell, it is called `Maybe`, in Scala it is `Option`, in Swift it is called `Optional`, and even in C# we find `Nullable`. Regardless of the language, the structure of the data type is similar:

```haskell
data Maybe a = Nothing  -- no value
             | Just a   -- holds a value
```

```scala
sealed abstract class Option[+A]                // optional value
case object None extends Option[Nothing]        // no value
case class Some[A](value: A) extends Option[A]  // holds a value
```

*Its inventor, Sir Tony Hoare, apologized in 2009 for the creation of `null` using those same words.

A primary use case for optional values is the validation of user input. For example, let's suppose we have a small record or class representing a person:

```haskell
type Name = String
data Person = Person { name :: Name, age :: Int }
```

```scala
type Name = String
case class Person(name: Name, age: Int)
```

In this case, we might want to check some properties of the name and age before creating a value. Suppose that those checks are factored into two different functions:

```haskell
validateName :: String -> Maybe Name
validateAge  :: Int    -> Maybe Int
```

```scala
def validateName(s: String): Option[Name]
def validateAge(n: Int): Option[Int]
```

How do we compose those functions to create `validatePerson`? This is our first attempt:

```haskell
validatePerson :: String -> Int -> Maybe Person
validatePerson name age
  = case validateName name of
      Nothing    -> Nothing
      Just name' -> case validateAge age of
        Nothing   -> Nothing
        Just age' -> Just (Person name' age')
```

```scala
def validatePerson(s: String, n: Int): Option[Person]
  = validateName(s) match {
      case None => None
      case Some(name) => validateAge(n) match {
        case None => None
        case Some(age) => Some(Person(name, age))
      }
    }
```

This solution clearly does not scale. As we introduce more and more validations, we need to nest the branches more and more, as well. Furthermore, we need to repeat the following code over and over again:

```haskell
Nothing -> Nothing  -- in Haskell
```

```scala
case None => None   // in Scala
```

So, as we did for `State`, let us look at the common pattern in this code:

```
case v of
  Nothing -> Nothing
  Just v' -> nextAction ... v' ...
```

Using the power of higher-order functions, we can turn it into its own function:

```
then_ :: Maybe a -> (a -> Maybe b) -> Maybe b
then_ v g = case v of
              Nothing -> Nothing
              Just v' -> g v'
```

```
sealed class Option[A] {
  def then[B](f: A => Option[B]): Option[B] = this match {
    case None => None
    case Some(x) => f(x)
  }
}
```

And rewrite `validatePerson` without all the nesting:

```
validatePerson name age
  = validateName name `then_` \name' ->
    validateAge  age  `then_` \age'  ->
    Just (Person name' age')
```

```
def validatePerson(s: String, n: Int) = validateName(s) then { name =>
  validateAge(n) then { age  =>
    Some(Person(name, age)) }
}
```

Let us compare the types of the previously defined `next` for state contexts and the new `then_` for optional values. Their similarity is a bit obscured by the fact that `State` takes an additional type argument in the first position, but you should take `State s` as a single block:

```
next  :: State s a -> (a -> State s b) -> State s b
then_ :: Maybe   a -> (a -> Maybe   b) -> Maybe   b
```

```
def next[S, A, B](s: State[S, A], f: A => State[S, B]): State[S, B]
def then[A,    B](o: Option  [A], f: A => Option  [B]): Option  [B]
```

They look fairly similar, right? In some sense, being optional is also a *context*, but instead of adding the ability to consume and modify a state, it allows failure with no value, in addition to returning a value. In a similar fashion, `then_` sequences computations, each of them possibly failing, into one complete computation.

`Maybe` (or `Option`) may also work as a *box* that is either empty or filled with one value. We can modify such a value, if one is present, as follows:

```haskell
map :: (a -> b) -> Maybe a -> Maybe b
map f Nothing  = Nothing
map f (Just x) = Just (f x)
```

```scala
def map[A, B](f: A => B, o: Option[A]): Option[B] = o match {
  case None => None
  case Some(x) => Some(f(x))
}
```

Or we can create a box with a single value:

```haskell
singleton :: a -> Maybe a
singleton = Just
```

```scala
def singleton[A](x: A): Option[A] = Some(x)
```

But what is more interesting, we have an operation to flatten a box that is inside another box. If either the inner box or the outer box is empty, there is actually no value at all. Only if we have a box containing a box containing a value can we wrap it in a single box:

```haskell
flatten :: Maybe (Maybe a) -> Maybe a
flatten (Just (Just x)) = Just x
flatten _               = Nothing
```

```scala
def flatten[A](oo: Option[Option[A]]): Option[A] = oo match {
  case Some(Some(x)) => Some(x)
  case _ => None
}
```

Let us play a game with boxes and contexts.* Is it possible to flatten two layers of boxes by using just `then_`? Following the types points us in the right direction:

```haskell
then_   :: Maybe a -> (a -> Maybe b) -> Maybe b
flatten :: Maybe (Maybe c)           -> Maybe c
```

We need to make `a` equal to `Maybe c` and `b` equal to `c` to make the types match. In turn, this means that we need to provide `then_` with a function of type `Maybe c -> Maybe c`. Wait a minute! We can use the *identity* function!

```haskell
flatten oo = then_ oo id
```

> **Exercise 1.4.** Convince yourself that the two definitions of `flatten` are equivalent by expanding the code of `then_` in the second one.

Now, dear reader, you may be wondering whether we can go in the opposite direction. The closest type to `then_` is `map` with its arguments reversed:

*Spanish-speaking readers might think of *trileros* at this point.

```
then_    :: Maybe a -> (a -> Maybe b) -> Maybe b
flip map :: Maybe c -> (c ->       d) -> Maybe d
```

Imagine that we have a function `f :: a -> Maybe b` supplied for `then_`, but we give it by mistake to `flip map`. In response, the type variable `d` in the type of `flip map` is instantiated to `Maybe b`. This implies that `flip map f o` has type `Maybe (Maybe d)` — not exactly what we aimed for. Is there any way to flatten those two layers? This is exactly what `flatten` does:

```
then_ o f = flatten (fmap f o)
```

1.4 Two for the Price of One

The definition of `flatten` in terms of `then_`, and vice versa, does not depend on any specifics of `Maybe` or `Option`. These are just particular usages of those functions. This suggests that we could do the same for the other two data types introduced in this chapter.

Let us begin with the version of `flatten` for `State`:

```
flatten :: State s (State s a) -> State s a
flatten ss = next ss id
```

To understand what is going on, we can replace `next` with its definition:

```
flatten ss = \i -> let (r, i') = ss i in id r i'
             -- in other words
           = \i -> let (r, i') = ss i in r i'
```

In summary, we use the initial state `i` to unwrap the first layer of `State`. The result of this unwrapping is yet another `State` computation and some change in the state. This second state is what we use to execute the unwrapped computation.

The corresponding operation for lists is a bit better known than `flatten` for `State` computations. The signature of this function is:

```
f :: [a] -> (a -> [b]) -> [b]
```

```
def f[A, B](xs: List[A], g: A => List[B]): List[B]
```

This function is known in Haskell circles as `concatMap` and in the Scala base library as `flatMap`. That name makes sense, since by translating the definition of `then_` in terms of `flatten` (called `concat` in the case of lists) and `map`, we reach:

```
concatMap xs f = concat (map f xs)
```

```
def flatMap[A, B](xs: List[A], g: A => List[B]): List[B]
  = concat(map(g, xs))
```

To understand what `concatMap` does, let us look at a concrete example:

```
> concatMap [1,10] (\x -> [x + 1, x + 2])
[2,3,11,12]
```

The result shows that *for each* value in the first list `[1,10]`, the function is executed, and all the results are concatenated into the final list. This example suggests another view of the list type not as a box but as a *context*: in the same way that `Maybe`/`Option` adds the possibility of not returning a value, `[]`/`List` adds the ability to return *multiple* values.

The fact that you can always define a "box-like" interface from a "context-like" one, and vice versa, shows that those two interfaces are just two sides of the same coin. In a bit fewer than 10 pages, we have been able to find three data types — `State s a`, `[a]`, and `Maybe a` — that share a common abstraction. This common interface is what we call a *monad!*

At this point, Haskell and Scala diverge in the names assigned to the generic functions defining a monad. In fact, the two main categorically-inspired libraries for Scala (Scalaz and Cats) do not agree, either:

- The "singleton" operation (building a monadic value from a pure one) is called `return` or `pure` in Haskell and `point`, `pure`, or `unit` in Scala.
- The "sequence" operation is sometimes called `bind` (this is how Haskellers pronounce their symbolic `(>>=)`), whereas the Scala standard library leans toward `flatMap`.
- The "flatten" operation is known as either `join` or `flatten`.

For the sake of consistency in terminology, the rest of this book assumes that the monadic interface is available through a `Monad` type class (in the case of Haskell) or a `Monad` trait (in the case of Scala), which may be defined in the following ways:

```haskell
-- Monad as a context
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b)
         -> m b
```

```haskell
-- Monad as a box
class Monad m where
  return :: a -> m a
  fmap   :: (a -> b) -> m a -> m b
  join   :: m (m a) -> m a
```

```scala
trait Monad[M[_]] {
  def point[A](x: A): M[A]
  def bind[A, B](x: M[A])
    (f: A => M[B]): M[B]
}
```

```scala
trait Monad[M[_]] {
  def point[A](x: A): M[A]
  def map[A, B](x: M[A])(f: A => B): M[B]
  def join[A](xx: M[M[A]]): M[A]
}
```

The definition of the generic interface of the monad highlights an important point: monads are a *higher-kinded* abstraction. Being a monad is not a property

of a concrete type (like `Int` or `Bool`) but of a type constructor (like `Maybe` or `List`). Haskell's syntax hides this subtle point, but Scala is explicit: we need to write `M[_]` to tell the compiler that members of this interface have a "type hole" that may vary in its different methods.

As an example, here is the implementation of the interface for `Maybe`/`Option`. In the Scala code, we use the *type class* pattern as described in Section 0.1:

```
instance Monad Maybe where
  return = Just
  (>>=)  = then_
```

```
object Option {
  implicit val optionMonad: Monad[Option] = new Monad[Option] {
    def point[A](x: A) = Some(x)
    def bind[A, B](x: Option[A])(f: A => Option[B])
      : Option[B] = then(x, f)
  }
}
```

1.5 Functors

Before closing this first chapter, we should remark that part of our definition of monad comes from a more generic abstraction called a *functor*. Functors encompass the simplest notion of a "magic box" that we can only inspect by means of functions but never unwrap or generate anew. In other words, a functor provides a function `fmap` — think "functor map" — but nothing comparable to the monadic `return`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
trait Functor[F[_]] {
  def map[A, B](x: F[A])(f: A => B): F[B]
}
```

As explained above, for every monad, we can write a `map` operation. This implies that every monad is also a functor, and thus by using monads, we also gain the capabilities of functors.

The type signature of `fmap` leads to another intuition, if we sprinkle in some parentheses:

```
fmap :: Functor f => (a -> b) -> (f a -> f b)
```

By calling `fmap` over a function `g`, you turn it into a new function that operates on elements contained in the functor. This is called *lifting*, since we "lift" `g` to operate at a higher-level of abstraction. For example, `\x -> x + 1` is a humble function that operates on numbers and thus might have the type `Int -> Int`. If we write `fmap (\x -> x + 1)`, this function may now map `[Int] -> [Int]` (by executing the function on every element), or `Maybe Int -> Maybe Int` (by modifying the value, only if there is one), or even `State s Int -> State s Int` (by changing the value to be returned from the computation but keeping the state untouched).

In fact, this way of looking at `Functor` leads us to the discovery of another interesting structure, the *applicative functor*. But for that you need to wait until Chapter 3, dear reader.

2 Better Notation

In Chapter 1, we introduced a few early examples of monads. We also introduced the generic interface that these type constructors support. However, we have not yet rewritten the original examples into this common interface. Let us do that now:

```
relabel (Leaf x)   = \i -> (Leaf (i,x), i + 1)
relabel (Node l r) = relabel l >>= \l' ->
                     relabel r >>= \r' ->
                     return (Node l' r')
```

```
case Leaf(x) => i => (Leaf((x, i)), i + 1)
case Node(l, r) =>
  relabel(l) bind { ll =>
    relabel(r) bind { rr =>
      point(Node(ll, rr)) } }
```

```
validatePerson name age
  = validateName name >>= \name' ->
    validateAge  age  >>= \age'  ->
    return (Person name' age')
```

```
def validatePerson(s: String, n: Int)
  = validateName(s) bind { name =>
      validateAge(n) bind { age  =>
        point(Person(name, age)) } }
```

As discussed in Section 0.1, the Scala version would require additional `implicit` parameters for the `Monad` trait and extension methods that declare `bind` as part of `Option` and `State`. For the sake of conciseness, we drop them in our code blocks.

Even in these simple examples, we can see the usual pattern of monads. We have a bunch of monadic operations — functions with a return type of the form `m`

a or `M[A]` for the corresponding monad — and after each of them, we apply a bind function in order to do something with the value the monad contains in subsequent computations. Eventually, we use `return` or `point` to construct the final value, which is wrapped in — or "returned into" — the monadic context. Note that this use of "return" is different from the keyword of the same name that often appears in imperative languages. In Haskell, `return` is just another function, and it does not imply that control flow escapes from its monadic context in any way. For `relabel`, we will focus only on the `Node` branch, which exhibits this pattern, leaving `Leaf` aside for the moment.

Given the pervasiveness of this pattern, many programming languages have dedicated syntactic sugar to describe monadic computations. In this chapter, we look at the implementation of this syntax in mainstream functional languages such as Haskell, Scala, F#, and other, more academic languages like Idris, each of them having slight variations. The cool thing is that after looking closely enough, you will see that even C# has something like a monadic syntax!

2.1 Block Notation

There are three main problems with writing code using monadic operators directly:

1. We need to repeat the name of the bind function over and over. Haskell tries to "hide" it under a symbolic name, `(>>=)`, but you still need to write it on every line. And sometimes Haskell's solution does not work very well: beginners may feel intimidated by so many strange symbols on their screens!

2. In most programming languages, giving a name to a value takes the form `name = value`, maybe with a slightly different equality or assignment symbol. Monadic code breaks this pattern, since the name we give to the element in the monad — for example, `l'` or `ll` in the preceding code — is written *after* the expression that produces it.

3. In those languages where anonymous functions are delineated by parentheses or braces, such as Scala, we need to keep track of how many to close at the end of the expression. This seems like a small annoyance, but it makes it harder to refactor monadic code. In this respect, the Haskell syntax works a bit better, because anonymous functions always extend until the end of the expression. In practice, this is a good default for monadic code.

In order to combat these problems, several functional programming languages natively support the notion of *monadic block*s. In a monadic block, code may be written in a more pleasing way and is transformed by the compiler into nested calls to the bind operation. Unfortunately, there is no consensus about the extent to which monadic operations should be hidden with syntax, which has led to many different variations of monadic blocks.

2.1.1 Do Notation

Haskell introduced `do` blocks in one of the first revisions of its specification, and it has become a flagship feature of the language. Nowadays, `do` notation is also available in other Haskell-inspired languages, such as PureScript and Idris.

The key idea of `do` notation is simple: fix problem (1) by automatically inserting calls to `(>>=)`, and fix problem (2) by providing specialized syntax for naming in a monadic context. The two code blocks at the beginning of this chapter could be rewritten as follows:

```
relabel (Node l r)
  = do l' <- relabel l
       r' <- relabel r
       return (Node l' r')
```

```
validatePerson name age
  = do name' <- validateName name
       age'  <- validateAge  age
       return (Person name' age')
```

In full generality, this transformation takes the form:

```
do x <- expression
   something #1
   ...
   something #n
```

$\Longrightarrow$

```
expression >>= x ->
  do something #1
     ...
     something #n
```

where the resulting `do` block on the right-hand side can be transformed again. Eventually, we reach a `do` block with a single line, which is translated into itself.

Haskell's `do` notation includes a couple of refinements for other common scenarios. The first common scenario is when you need to use some pure (non-monadic) code inside a larger monadic context. Imagine that our `Person` data type includes an extra piece of data that computes whether a certain country considers a person underage. Instead of writing the expression directly in the constructor, we would prefer to separate it into its own variable. If we are not using `do` notation, we can use the usual `let .. in` construct in the language, as demonstrated in the following code:

```
validateAge age >>= \age' ->
let uAge = age < 18 -- or 21
 in validateName name >>= \name' ->
    return (Person name' age' uAge)
```

Alas, if we want to rewrite this code as a `do` block, we must either (1) introduce a separate `do` block after `in`; or (2) wrap the pure value in a monadic box using `return`, so we can use the monadic operator `<-` to give it a name. These two strategies are shown in the following snippets:

```
do age' <- validateAge age
   let uAge = age < 18
    in do name' <- validateName
          return (Person name' age' uAge)
```

```
do age'  <- validateAge age
   uAge  <- return (age < 18)
   name' <- validateName name
   return (Person name' age' uAge)
```

Both solutions are unsatisfactory. The first option leads to a spurious nesting of `do` blocks, making it harder to see the flow of control. The second option forces us to write `return` more than necessary, and the reuse of `<-` obscures whether or not the code has monadic effects.

The final decision of the Haskell Committee was to introduce yet another syntactic form, `let name = expression`. Note that in contrast to the non-monadic version of `let`, when it appears in `do` blocks, it must *not* be followed by the keyword `in`. This syntactic form is demonstrated in the following code:

```
do age'  <- validateAge age
   let uAge = age < 18
   name' <- validateName name
   return (Person name' age' uAge)
```

The second refinement deals with values that are not used in a later computation. We will talk more about `State` operations later on, but for now let us just introduce the one that updates a state with a new value and the one that obtains the current value of a `State`. These two operations are called `put` and `get`, respectively, and they have the following type signatures:

```
put :: s -> State s ()
get :: State s s
```

Using these two building blocks, we can write a function that increments a counter and returns the next value, as demonstrated in the following code:

```
incrCounter :: State Int Int
incrCounter = do n <- get
                 p <- put (n+1)
                 return (n+1)
```

However, we are not interested at all in the return value of `put`. Since it is of type `()`, we know that the only possible value is `()`. When using monadic functions, it is common to have operations such as `put` that are only useful for their monadic effects.

Haskell provides a few workarounds for this situation. One permits the replacing of `p` with an underscore, `_`, the symbol that we use to say, "I don't care about giving this parameter a name." But we can even go one step further and drop the `_ <-` prefix completely, resulting in:

```
incrCounter :: State Int Int
incrCounter = do n <- get
                 put (n+1)
                 return (n+1)
```

The transformation to the underlying monadic operations then becomes as follows:

```
do expression          do _ <- expression     expression >>= \_ ->
   something #1   ⟹       something #1   ⟹      do something #1
   ...                    ...                       ...
   something #n           something #n              something #n
```

It is a bit silly, though, to construct a function that completely ignores its argument. Instead of doing that, Haskell monads come with an additional operation we can use, which is defined as:

```
(>>) :: Monad m => m a -> m b -> m b
m >> n = m >>= \_ -> n  -- default definition
```

This operation encodes the idea of sequencing two computations, where the result of the first computation is not used at all in the second computation. Using this newly introduced operator, the translation of lines in a `do` block without `<-` becomes as follows:

```
do expression          expression >>
   something #1   ⟹      do something #1
   ...                       ...
   something #n              something #n
```

We have now discussed almost all the features in Haskell's `do` notation. The only thing missing is the generalization of the `x <- expression` syntax to account for patterns — not just plain variables — in the right-hand side. We discuss the implications of supporting this syntax in Section 2.2.

2.1.2 Comprehensions

We saw in Chapter 1 that lists form a monad. Most functional languages include dedicated syntax to work with monadic lists, often termed *list comprehensions*. The designers of Scala decided against adding a new syntactic concept to the language — such as `do` notation in Haskell — and instead generalized list comprehensions to work with any monad.

The examples at the beginning of this chapter are written in more idiomatic Scala, as shown in the following snippets:

```
case Node(l, r) => for {
  ll <- relabel(l)
  rr <- relabel(r)
} yield Node(ll, rr)
```

```
def validatePerson(s: String, n: Int)
  = for {
      name <- validateName(s)
      age  <- validateAge(n)
    } yield Person(name, age)
```

Under the hood, a syntactic translation similar to that in Haskell takes place. The Scala compiler uses the name `flatMap` for the binding operator of monads by default, whereas we prefer the more mnemonic name `bind`. Luckily, most implementations of monads in Scala define both names in order to please everyone.

The transformation process is given by the following rule. As in the case of Haskell, the translation may lead to a nested `for` comprehension, which is then translated recursively:

```
for {                              expression flatMap { x =>
  x <- expression          ⟹        for {
  ??? // something else                ??? // something else
} yield finalExpression              } yield finalExpression
```

In contrast to Haskell, Scala's `for` comprehensions always end with a final pure expression, which is written after the keyword `yield`. A first approach to a translation might use a `point` function to embed that value in a monadic context, as shown in the following example:

```
for { } yield value   ⟹ point(value)
```

However, this is not how it is implemented in the Scala compiler. Instead, Scala takes the value terminating the `for` block and translates it into a `map` operation, as shown in the following example:

```
for {                              expression map { x =>
  x <- expression          ⟹        finalExpression
} yield finalExpression            }
```

At first, it looks like this last transformation using `map` gives rise to different code than the direct combination of `flatMap` for the binding of the inner expresion and the use of `point` for yielding the value. However, in any monad we can write `map` as a combination of `flatMap` and `point`:

```
def map[A, B](value: A => B, expression: M[A]): M[B] =
  expression flatMap { x => point(value(x)) }
```

This definition works for any monad, and both can be shown to be equivalent. However, the Scala compiler prefers using `map` wherever possible, because it usually has a faster implementation than the combination of `flatMap` and `point`. Just for the sake of completeness, the following code defines the same function in Haskell:

```
map f x = x >>= return . f
```

There are scenarios in which we only need a monad for its effects, and we don't need to return any value. In Haskell, we usually return an empty tuple in these cases — this introduces a bit of syntactic noise, but only in the very few cases in which we *really* do not return a value. Scala, on the other hand, is not a completely pure language, which makes this scenario more common. For this

reason, the language also includes `for` comprehensions *without* a final `yield`. In these cases, the compiler uses a different method for iteration, which relies on a function equivalent to the following type:

```
def foreach[A](x: M[A], f: A => Unit): Unit
```

As we can see, the type of the function passed as an argument to `foreach` should not return any value, which means that the reason for calling it is some side effect. The translation proceeds as follows:

```
for {                                    expression foreach { x => for {
  x <- expression                  ⟹      ??? // something else
  ??? // something else                  } }
}
```

There is only one final element of Scala `for` comprehensions, which is related to the use of `if` instructions to filter elements. As in the case of `flatMap`, this operation arises naturally for list comprehensions, but it can be generalized to other monadic contexts. However, dear reader, you will need to wait until Chapter 7 to find out the details — or skip ahead right now!

Looking back in history, Haskell previously supported comprehension syntax for arbitrary monads. However, in its first official version — Haskell 98 — they were dropped, because the error messages they produced were too hard for beginners to understand. Luckily, in 2011, they came back! To use them, include the following pragma line at the top of any source file:

```
{-# LANGUAGE MonadComprehensions #-}
```

Then, any subsequent comprehension in the file will be translated using rules similar to the ones for Scala. The examples at the end of this chapter are written using this feature, as follows:

```
relabel (Node l r)                  | validatePerson name age
  = [ Node l' r'                    |   = [ Person name' age'
      | l' <- relabel l              |       | name' <- validateName name
      , r' <- relabel r ]           |       , age'  <- validateAge  age ]
```

The main difference between Scala and Haskell comprehensions is that in the former case, the expression to return is placed at the *end* of the block, after the `yield` keyword, whereas in Haskell it appears right at the *beginning* of the block.

2.1.3 Bangs in Idris

Special syntax for monadic blocks alleviates much of the boilerplate associated with these structures. But the Haskell and Scala solutions both suffer from a problem: every time we want to bind the result of a monadic expression so we

can use the value later, we need to choose a name. And as every programmer can attest, naming is *hard*.

Idris offers an attractive alternative, the `(!)` operator, which can be thought of as an operator that extracts a value without giving it a name. The previous examples can thus be written more concisely in Idris, as follows:

```
relabel (Node l r)
  = return (Node !(relabel l) !(relabel r))

validatePerson name age
  = return (Person !(validateName name) !(validateAge age))
```

Discussions about supporting this syntax come up periodically in the Haskell community. But in the end, most people seem happy enough with the applicative notation we discuss in Section 3.2.

2.1.4 Computation Expressions in F#

F# includes yet another variation on monadic syntax. As in the previous cases, the language defines a translation from high-level constructs (called *computation expressions* in F#) into monadic building blocks. There are two main differences between these expressions, as found in F#, and do notation and comprehensions.

The first difference between F# and both Scala and Haskell is that the language of computation expressions is much richer. This makes computation expressions useful for other kinds of control structures besides monads. For the sake of conciseness, we will restrict ourselves to the subset of computation expressions that handle monads.

The second difference is that a computation expression is always preceded by the name of a *builder*, which has the task of interpreting the following expression. In our terms, every computation expression declares *explicitly* which monad should handle the code:

```
| Node(l, r) ->
    state {
      let! ll = relabel(l)
      let! rr = relabel(r)
      return Node(ll, rr)
    }
```

```
let validatePerson s n =
  maybe {
    let! name = validateName s
    let! age  = validateAge  n
    return Person(name, age)
  }
```

The `FSharpx.Extras` library* contains builders for the most common monads, all of them discussed at some point in this book.

Why do we need to state the builder at the beginning of the block? The reason lies in the translation baked into the F# compiler: each computation expression is turned into a call of a *method in a builder*. This is demonstrated in the following translation rule:

*`http://fsprojects.github.io/FSharpx.Extras/`

```
builder {                          builder.Bind(expression, fun x ->
  let! x = expression      ⟹          builder {
  something else                       something else
}                                    })
```

In contrast, Haskell and Scala use type inference to deduce which monad to use to bind each of the steps in a computation.

The other important monadic operation is the embedding of pure values, which we call `return` or `point`, depending on the language. F# contains not one, but four variations, namely:

```
builder { return  expression } ⟹ builder.Return(expression)
builder { yield   expression } ⟹ builder.Yield(expression)
builder { return! expression } ⟹ builder.ReturnFrom(expression)
builder { yield!  expression } ⟹ builder.YieldFrom(expression)
```

The first distinction is between `return` and `yield`. Both have the same type, and it is up to the builder to implement only one or both operations and to make them work in a distinct way. In most scenarios, `yield` is used to indicate some kind of delayed computation, which could be restarted later — like a generator for a list — whereas `return` embodies the usual monadic notion.

The second distinction is between the versions with and without exclamation marks, or conversely, between the functions that end in `From` and those that do not. The difference is clear when looking at their types. Note that F#, following the ML tradition, distinguishes type variables by a tick in front of their names:

```
Return     :   'T  -> M<'T>       Yield     :   'T  -> M<'T>
ReturnFrom : M<'T> -> M<'T>       YieldFrom : M<'T> -> M<'T>
```

When you use `return!` or `yield!`, you *already* have a value in a monadic context, and you just want to make this value the end result of the computation.

You can achieve the same result when using `do` notation by simply placing the monadic expression in the last line of the block. However, it is not possible to do so when using comprehensions. F# offers both options but requires you to be explicit about which one you want.

2.1.5 Even C# Has Monads These Days

Modern versions of both C# and F# include *query expressions*, which form a small sub-language (inspired by SQL) to search, filter, and group values from an enumerable source. As with all the other kinds of syntax presented in this section, the compiler translates this notation into a series of method calls. And considering the relative closeness of these method calls to comprehensions, it is not surprising that they can be (ab)used for monadic computations.

Let us focus on two elements of this translation as defined by the C# 5.0 Language Specification. The simpler kind of query expression is as follows:

```
from x in e select v
```

This is not far from the list comprehension `[v | x <- e]`. In this case, we expect a translation to a `map`-like function. Indeed, the translation does just this but using a different name:

```
(e).Select(x => v)
```

The type of `Select` is, in fact, similar to `map` but has the C# type signature that follows. Note that in C#, the function type is written Func<A, B>. To bridge the intuition gap between languages, we also include a version with Scala's `=>` syntax in the comments:

```
IEnumerable<TResult> Select<TSource,TResult>(
  this IEnumerable<TSource> source, Func<TSource,TResult> selector)
                               // TSource => TResult
```

Let us say you have more than one `from` clause, as in the following example:

```
from x1 in e1 from x2 in e2 select v
```

In this case, we need a more powerful operation called `SelectMany`, which results in the following translation:

```
(e1).SelectMany(x1 => e2, (x1, x2) => v)
```

The type of this operation is a bit more complicated and comes with the following two overloads:

```
IEnumerable<TResult> SelectMany<TSource,TResult>(
  this IEnumerable<TSource> source,
  Func<TSource,IEnumerable<TResult>> selector)
  // TSource => IEnumerable<TResult>

IEnumerable<TResult> SelectMany<TSource,TCollection,TResult>(
  this IEnumerable<TSource> source,
  Func<TSource,IEnumerable<TCollection>> collectionSelector,
  // TSource => IEnumerable<TCollection>
  Func<TSource,TCollection,TResult> resultSelector)
  // TSource => TCollection => TResult
```

The first overload follows exactly the type of a `bind` operation for a monad! The second overload can be thought of as a generalization of the first one, where after binding the result you perform an additional map. In fact, if you write a list comprehension such as `[v | x1 <- e1, x2 <- e2]`, you get the same translation, a `bind` followed by a `map`. While technically redundant, the combined operation can often be implemented in a more performant way than merely composing the `bind` and `map` functions.

Looking at query expressions with our new glasses reveals that monads can be found where you do not expect them. On the other hand, you should by now have the feeling that any syntax that seems to manipulate lists or sequences can be reworked as a notation for monadic operations.

2.2 Pattern Matching and Fail

Up until this point, all the examples you have seen so far use the results of monadic actions without further inspection. But, of course, we can pattern match on these results as with any other values. The following code builds an action that asks the user for input until a number is found, building on top of a `getNumber` function that tries and maybe fails to get a number:

```
getNumber :: IO (Maybe Int)  -- get a number from the keyboard
getNumber' :: IO Int         -- retry until we get a number
getNumber' = do n <- getNumber
                case n of
                  Just m  -> return m
                  Nothing -> getNumber'
```

This works, but there's another way to write this code. In Haskell, wherever you may assign a name, you may *also* perform a bit of pattern matching. For example, one could write the code:

```
do Just n <- getNumber
   printLine $ "Your number is " ++ show n
```

This raises a question, though: what happens in the preceding code if the result of `getNumber` is `Nothing` instead of a `Just` value?

Before considering monadic contexts, we should consider how the Haskell compiler handles missing patterns, in general. The following function only works for the value `True`:

```
f True = 1
```

If we then ask the interpreter for the value of `f False`, this is what we get:

```
*** Exception: <interactive>:1:1-10:
    Non-exhaustive patterns in function f
```

This behavior is the same that we would get if we had added the following line to `f`:

```
f False = error "Non-exhaustive patterns in function f"
```

The monadic case is similar. The previous example with `getNumber` is translated into the following code:

```
getNumber >>= \e -> case e of
  Just n  -> printLine $ "Your number is " ++ show n
  Nothing -> fail "Pattern match failure in do expression"
```

The main difference is that we use `fail` instead of `error`. Those who have been programming in Haskell for some time may recognize `fail` as the "ugly duckling" of the `Monad` type class. The type signature is as follows:

```
class Monad m where
  ...
  fail :: String -> m a
```

In reality, `fail` is not part of the definition of a monad. However, we have seen how pattern matching makes this method quite useful in practice.

The problem with this approach is that not all monads provide a notion of failure, as required by non-exhaustive pattern matching. `List` or `Maybe` are positive examples — with the empty list and the `Nothing` value, respectively — whereas `State` provides a `fail` method that just calls `error`. In order to distinguish monads for which pattern matching failure has a sane implementation from those for which it does not, GHC has since version 8.0 moved the `fail` method into its own type class, called `MonadFail`. This type class is defined as follows:

```
class Monad m => MonadFail m where
  fail :: String -> m a
```

From now on, every `do` block with possibly-failing matches may be marked with an additional `MonadFail` constraint. In the future, `MonadFail` will become mandatory in these situations.

The Idris programming language takes a different approach to this problem, by providing pattern matching with alternatives in `do` notation. The first alternative still appears on the right-hand side of the `<-` symbol, and it represents the preferred match. The others appear after a vertical bar, using a syntax that resembles guards in function definitions. The following example shows this syntax at work:

```
getNumber' = do Just m <- getNumber
                  | Nothing => getNumber'
                return m
```

The design of this feature allows us to keep the main path of computation free of superfluous branching, but it also provides visual cues for error cases.

3 Lifting Pure Functions

We have seen what a monad is and what syntactic facilities various programming languages provide for monadic computations. However, because of the strong typing discipline found in, for example, Haskell and Scala, it is *a priori* difficult to mix pure computations in monadic contexts, even with the aforementioned syntactic sugar. This chapter is devoted to the best-known solution to the problem of embedding pure computations into monads: *lifting*. As we will see, this solution led to the discovery of a close relative of monads: applicative functors, or *applicatives*, for short.

3.1 Lift2, Lift3, ..., Ap

We introduced the idea of lifting when we discussed functors in Chapter 1. Remember the type of the only method in that type class or trait:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```scala
trait Functor[F[_]] {
  def map[A, B](x: F[A])(f: A => B): F[B]
}
```

If we reorder the arguments (for Scala's `map`) and then make the parentheses more explicit, we can also write the type as:

```haskell
(a -> b) -> (f a  -> f b)
```

```scala
(A => B) => (F[A] => F[B])
```

This new point of view enables us to read `fmap` in another way, in addition to the intuitive idea of "applying a function to whatever is inside of a container `f`." If we have a *pure* function `g`, we write `fmap g` for the version that consumes and produces values in the context denoted by `f`. We often say that `g` has been *lifted into* that functor.

The power of `Functor` is rather limited, though: the set of functions that can be lifted using `fmap` is restricted to those that take *one* argument. Alas, many interesting functions take several arguments. Some primary examples are arithmetic operations, which are usually binary. Being more concrete, we cannot lift the addition operation `(+)` to work on `Maybe Int` or `Option[Int]` values with the method provided by `Functor`. Writing a lifted version of addition using monads, however, is straightforward:

```
plus :: Maybe Int -> Maybe Int -> Maybe Int
plus x y = do a <- x
              b <- y
              return (a + b)
```

```
def plus(x: Option[Int], y: Option[Int]): Option[Int]
  = for { a <- x ; b <- y } yield (a + b)
```

In essence, all we are doing above is unwrapping the monadic boxes with the `<-` operator and applying the pure computation at the end. We can even abstract this idea into a generic `lift2` function:

```
lift2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
lift2 f x y = do a <- x
                 b <- y
                 return (f a b)
```

```
def lift2[M[_], A, B, C](f: (A, B) => C)
      (implicit m: Monad[M]): (M[A], M[B]) => M[C]
  = (x, y) => for { a <- x ; b <- y } yield f(a,b)
```

Functions similar to `lift2` can be written for any number of arguments: `lift3`, `lift4`, etc. You only need to make the initial unwrapping procedure as long as the series of arguments the pure function receives:

```
liftN :: Monad m =>  (a1 ->   a2 -> ... ->   aN ->   r)
                  -> m a1 -> m a2 -> ... -> m aN -> m r
liftN f x1 ... xN = do a1 <- x1
                       ...
                       aN <- xN
                       return (f a1 ... aN)
```

Obviously, this solution is far from perfect. First of all, somebody has to write all those `lift` functions. So either your base library provides enough variations for any reasonable number of arguments, or the functions are generated by some kind of macro. In any case, you will end up with lots of duplication. The second problem is the maintainability of the lifting code: if you decide that your pure function now takes one more argument, you not only need to add this new argument to all of its call sites, but you also need to update the corresponding `liftN` to the one that takes the additional argument. We would prefer a solution that is not so weak with respect to such changes.

The good news is that we only need *one* function on top of `fmap` to lift any function with *any* number of arguments. Consider a function `g` with two arguments, `g :: a -> b -> c`. By lifting it through `fmap`, we are able to apply it first to an argument of a monadic type. The result, however, is a function wrapped in the functorial or monadic context, so we cannot apply it to further arguments:

```
fmap g   :: m a -> m (b -> c)
       x :: m a
fmap g x :: m (b -> c)
```

```
map(g): M[A] => M[B => C]
x: M[A]
map(g)(x): M[B => C]
```

The scenario is as follows: we have something of type `m (b -> c)`, and we would like to "apply" it to a value of type `m b` to get our final `m c`. In other words, we want the following operation to be available for our monad:

```
ap :: Monad m => m (b -> c) -> m b -> m c
```

```
def ap[M[_], B, C](implicit m: Monad[M]): M[B => C] => M[B] => M[C]
```

> **Exercise 3.1.** Write the implementation of `ap`. Hint: you should start by unwrapping both arguments from their monadic boxing. Then, you just have a function and a value to which it can be applied.

Now, we come to one of those moments in which all the small elements in functional programming fit together in an almost magical way: we need nothing more than `fmap` and `ap` to be able to lift any pure function *regardless* of the number of arguments. Take a function with three arguments, `f :: a -> b -> c -> d`, that we want to apply in a monadic context — that is, we want to apply it to the values `x :: m a`, `y :: m b`, and `z :: m c`.

The first step is `fmap f x`, which has the type `m (b -> c -> d)`. As we discussed above, to apply yet another argument, we need `ap`. The result of `fmap f x` `` `ap` `` `y` is:

```
ap       :: m (u -> v)      -> m u -> m v
fmap f x :: m (b -> c -> d)
       y ::                    m b
-------------------------------------------
fmap f x `ap` y ::                 m (c -> d)
```

In this last step, we need to remember that `b -> c -> d` stands for `b -> (c -> d)`. Thus, our use of `ap` chooses `u` to become `b` and `v` to become `c -> d`. As a result, we obtain `m (c -> d)` as our final value. Here comes the trick: this type is exactly of the form we need to apply `ap` once again:

```
fmap f x `ap` y `ap` z :: m d
```

If you strive for a bit more generality, you can even rewrite each of the `liftN` functions using a similar number of `ap` combinators:

```
liftN f x1 ... xN = fmap f x1 `ap` x2 `ap` ... `ap` xN
```

In summary, we have found a small set of primitive operations that we can use to embed pure computations into the monadic world without too much pain.

3.2 Applicatives

Up to this point, we have seen `ap` as a by-product of lifting a pure function into a monadic context. But actually, this abstraction has a life of its own, independent of monads. *Applicative functors* — or simply *applicatives* — are those contexts, containers, or boxes for which you cannot only lift unary functions — which you get from `Functor` — but functions with any number of arguments, for which you use `ap`. Note that in both Haskell and Scala, the abstraction is defined as extending `Functor`, from which we inherit the mapping operation:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

```
trait Applicative[F[_]] extends Functor[F] {
  def point[A](x: A): F[A]
  def ap[A, B](x: F[A])(f: F[A => B]): F[B]
}
```

Wait a minute, what are `pure` and `point` doing here? They just complete the sequence! By using `pure` and `ap`, we can lift functions that take at least one argument. What about constants — in other words, functions with *no* parameters? Just use `pure`.

> **Exercise 3.2.** You do not need `fmap` at all in `Applicative`. Write `fmap` as a combination of `pure` and `ap`. Hint: if `f` is a function with type `a -> b`, then `pure f` has the type `m (a -> b)`.

The discussion in the first part of this chapter explains why every monad is also an applicative functor. In fact, more recent versions of GHC define `Monad` as being

a subclass of `Applicative`, which is in turn a subclass of `Functor`. In this context, the `pure` function in a monad is "inherited" from `Applicative`. Therefore, one fair question is: are there any applicatives that are *not* monads?

The classic example is the `ZipList` functor. For most operations, a zip-list is merely a list, but zip-lists behave differently when they are combined applicatively. Instead of obtaining all combinations of values in each list, the effect of this functor is to "zip" each value in the first list together with its corresponding value, in the same position, in the second list:

```
> fmap (,) [1,2] <*> [3,4]
[(1,3),(1,4),(2,3),(2,4)]
> fmap (,) (ZipList [1,2]) <*> (ZipList [3,4])
ZipList [(1,3),(2,4)]
```

The `ZipList` type is just a wrapper around the vanilla list type. As explained in Section 0.3, this additional wrapping with `newtype` allows us to define for it a different `Applicative` instance:

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

> **Exercise 3.3.** Write the `Functor` instance for `ZipList`. Hint: it has the same behavior as for normal lists, except for `newtype` wrapping and unwrapping.

The definition of this `Applicative` instance uses the `zipWith` operation, which gathers each function in `fs` with the corresponding one in `xs` and then applies the first to the second:

```
instance Applicative ZipList where
  pure x = ZipList (repeat x)  -- infinite list of x values
  ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

But it turns out that `ZipList` cannot be made an instance of `Monad` unless you only consider infinite lists (for more details, see [McBride, 2011]). It is proven, then: applicatives are more general than monads.

3.3 Applicative Style

Looking at the examples above, a pattern emerges, the one we captured with `liftN`. Each time we want to use a pure function in a monadic (or applicative) context, we use:

```
fmap f x1 `ap` x2 `ap` ... `ap` xN
```

This is rather compact, but by using `fmap` with `ap` interleaved, we obscure the important parts of the expression: the function to be applied and the arguments. Haskellers' solution to this problem is defining synonyms for both operations using only symbols. More concretely, `(<$>) = fmap` and `(<*>) = ap`. As a result, lifting a function looks like this, instead:

```
f <$> x1 <*> x2 <*> ... <*> xN
```

This form of writing code is called *applicative style*. It has become increasingly popular, because it allows us to describe computations that take several lines using a `do` block in only one line:

```
do x <- [1,2,3]
   y <- [4,5,6]
   return (x + y)
```

The above may be written more simply in applicative style:

```
(+) <$> [1,2,3] <*> [4,5,6]
```

On the other hand, not everybody feels comfortable with this style of writing code. Its premise rests on the assumption that programmers will ignore the symbols in between the functions and arguments and just take them as a reminder that "this application is running under an applicative functor." On the other hand, these symbols are obscure for most beginners, so they stand out the most in the expression, making it harder to decipher what is really going on.

As a historical aside, the applicative style made its first appearance in 1996 in a paper called *Deterministic, error-correcting combinator parsers* [Swierstra and Duponcheel, 1996], but it only became popular after the publication in 2007 of *Applicative programming with effects* [McBride and Paterson, 2008]. In the latter, the authors also introduced *idiom brackets* as syntactic sugar to describe computations with applicatives, in a similar fashion to the way `do` blocks help with writing monadic code.* At the time of writing, there is only one language that supports idiom brackets, and that is Idris. In Idris, you write:

```
[| f x1 x2 ... xN |]
```

And it is translated to the general form with `fmap` and `(<*>)` introduced above. Given the similarities between Idris and Haskell syntax, it is possible for Haskell to introduce this feature in the future. In the meantime, you can always try to use the Strathclyde Haskell Enhancement preprocessor or the `applicative-quoters` package.

*Some people refer to applicatives as *idioms*.

3.3.1 Applicative Goodies

In the realm of pure computation, it never makes sense to execute code if you do not want its return value, since that is the only visible effect coming out of that call. For monadic or applicative computations, this is not the case, as you might want to execute one of these actions for its side effects alone, completely disregarding the value returned from the action. Haskell's standard library provides three, special combinators for that case. A trick for remembering the names is that the argument that has no ">" next to it is the one discarded:

```
(<$) :: Functor     f =>   a -> f b -> f a
(<*) :: Applicative f => f a -> f b -> f a
(*>) :: Applicative f => f a -> f b -> f b
```

Imagine that in `validatePerson`, you want to check the requirements of the provided name and age. However, the result value should not be a new `Person` value, but the full name in capital letters — to format it for a boarding pass, for example. You cannot write:

```
map toUpper <$> validateName name <*> validateAge age
```

The problem is that `map toUpper` takes only one argument. One solution is to resort to a monadic block, either fully or using some applicative help:

```
-- Option 1
do validateAge age
   map toUpper <$> validateName name
```

```
-- Option 2
do validateAge age
   m <- validateName name
   return (map toUpper m)
```

But another possibility is to discard the result of validating the age and only use its contextual behavior — if validation fails, the full computation ought to fail. This is made explicit by the combinator `(<*)`, which discards the value following it:

```
map toUpper <$> validateName name <* validateAge age
```

Another possibility is to validate the age before the name. In that case, we need to use `(<$)` instead of `(<$>)`:

```
map toUpper <$ validateAge age <*> validateName name
```

In this case, the choice is not important: if the validation fails, the whole computation fails in the same way — it returns `Nothing` — independently of the placement of `validateAge`. This is not the case for all monads, though. If you are not only failing but also remembering the error that led to the failing computation, and both the name and the age are incorrect with respect to their requirements, the choice affects which part is reported first.

Until now, our only references to `pure` and `point` were their close connection to monadic `return` and how they arise as the lifting of a nullary function, that is, a constant. This second behavior is rather useful when using applicative style to pass some arguments free of side effects.

Going back to our `Person` example, if we want to initialize the age to always be 20 years, we need to introduce an explicit anonymous function:

```
(\n -> Person n 20) <$> validateName name
```

Or we can use the `flip` combinator to swap the order of the arguments:

```
flip Person 20 <$> validateName name
```

Both solutions are far from perfect, since we are polluting the description of the behavior with explicit argument manipulation. Using `pure` and `point`, we can provide another solution, which involves moving the constant `20` from the world of pure values into the `Maybe` universe:

```
Person <$> validateName name <*> pure 20
```

In some sense, `(<$>)` and `(<*>)` state, "Here come monadic effects," to the compiler, and `pure` contradicts them by saying, "Actually, no, this is just a pure computation."

3.3.2 Applicative Do

Starting with version 8.0, GHC includes a language extension called `ApplicativeDo`, which repurposes the `do` notation introduced in Section 2.1.1 to work with applicative combinators. The idea is quite simple. Consider the following piece of code:

```
do x <- action1
   y <- action2
   return (x, y)
```

The usual translation of monadic blocks turns the code into a series of binds:

```
action1 >>= \x -> action2 >>= \y -> return (x, y)
```

The use of the bind operator imposes a `Monad` constraint on this code. But we know that there is a better way to translate the original `do` block:

```
(,) <$> action1 <*> action2
```

The improvements come from two fronts. First of all, we no longer require the `Monad` constraint, instead opting for the weaker but more appropriate `Applicative`. As a result, it might be possible to execute `action1` and `action2` in parallel, if the corresponding applicative allows it.

That is exactly the aim of the `ApplicativeDo` extension: to provide an alternative translation scheme for `do` blocks that does not require a monad in every

case. This translation is not as straightforward as for monads, so we will gloss over the details. The first problem is that one has to detect whether or not it is indeed possible to translate the code using only `Applicative` combinators — some data dependencies can only be expressed using binds. In addition, the algorithm in `ApplicativeDo` never reorders the execution of different actions. For example, suppose that instead of the original code at the beginning of this section, we were to write this:

```
do y <- action2
   x <- action1
   return (x, y)
```

At first sight, it looks like we could also translate this example as follows:

```
(,) <$> action1 <*> action2
```

However, if the result of `action1` depends on the side effects in `action2` — for example, if `action1` changes a state — the translation is wrong. In that case, the right code to produce would be this:

```
\y x -> (x, y) <$> action2 <*> action1
```

If you are wondering why we should work so hard to rewrite `do` blocks into `Applicative` combinators, since other styles of programming are already available, you might want to check out *There is no Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access* [Marlow, Brandy, Coens, and Purdy, 2014]. That paper introduces Haxl, a library for automatic parallelization of data accesses, which leverages applicatives to indicate when several computations may be chunked together. Whether or not a translation uses monads could result in fewer accesses to shared resources.

3.4 Definition Using Tuples

We have approached the concept of the applicative functor from the idea of lifting arbitrary, pure functions. Since in Haskell, functions are curried — they take one argument at a time — a combinator like `ap` does the job perfectly. In other languages like Scala, arguments are given in batches, making this combinator a bit less useful.

What kind of combinator should we use instead to lift those computations? In general, a function in Scala with a single argument block has the following shape:

```
def f(a1: t1, ..., aN: tN): R
```

In contrast with Haskell, such a function takes a *tuple* of values as a *single* argument. Thus, we only need `map` to lift the function:

```
M[(t1, ..., tN)] => M[R]
```

But when lifting a function, you do not have a tuple in a monadic context. Rather, you have each of the arguments in its own monadic container, that is:

```
x1: M[t1]  ...  xN: M[tN]
```

As a result, we need a function that takes a tuple of arguments in a context and puts them into a single context. Once we have such a function, we just need `map` to lift any pure function:

```
(x1: M[t1], ..., xN: M[tN]) => M[(t1, ..., tN)]
```

Both Haskell and Scala have different types for pairs, triples, 4-tuples, and so on. An alternative way to represent them is via nested pairs: a triple `(a,b,c)`, for example, is simply a pair where the second component is again a pair, `(a, (b, c))`. The same holds for a 4-tuple, `(a, b, c, d)`, which can be represented as `(a, (b, (c, d)))`.

Exercise 3.4. Write the functions that perform the conversions between the "normal" implementations of triples and 4-tuples and their implementations as nested pairs.

If we use this approach, we no longer need different functions for lifting tuples of different lengths. It is enough to have a function for lifting a pair of values `M[A]` and `M[B]` to `M[(A, B)]`, which we then apply repeatedly over nested components. A functor supporting that operation is called a *monoidal functor*:

```haskell
class Functor f => Monoidal f where
  unit :: f ()
  (**) :: f a -> f b -> f (a, b)
```

```scala
trait MonoidalFunctor[F[_]] {
  def unit: F[Unit]
  def tupled[A, B](x: F[A], y: F[B]): F[(A, B)]
}
```

Note that `Monoidal` also features a `unit` function, which can be seen as lifting a nullary tuple. The key is observing that a single value `f ()` may also be seen as a function `() -> f ()`, where the lifting is more explicit.

Amazingly, applicative and monoidal functors define the same notion! We can see it by providing an implementation of `Applicative` in terms of `Monoidal` and vice versa:

```haskell
pure :: a -> f a
pure x = fmap (\_ -> x) unit

(<*>) :: f (a -> b) -> f a -> f b
f <*> x = fmap (\(g, y) -> g y) (f ** x)
```

Exercise 3.5. Write the other direction of the conversion. In other words, give definitions of `unit` and `(**)` in terms of `fmap`, `pure`, and `(<*>)`.

Presenting a concept in two different ways might seem at first like a beautiful academic excursion but without much practical value. Not in this case: the idea of an applicative as a structure that is able to lift tuples lies beneath Scala's *applicative builder* operator: `|@|` — also known as the "scream operator." Here's an example:

```
(x1 |@| ... |@| xN)(f)
// If the last argument is a function, parentheses may be omitted
(x1 |@| x2) { _ + _ }
// With the help of Scala's notation for partial application
```

The trick is simple: `|@|` does the same job as Haskell's `(**)`. The resulting tuple then overloads the `apply` method to provide convenient syntax for the last map.

Regardless of your language of choice, the facilities provided by applicatives to manipulate simple computations in contexts make them a great tool for writing monadic code. Even within `do` blocks or comprehensions, it is becoming increasingly popular to sprinkle some `(<*>)` operators in order to obtain more concise code. Needless to say, applicative functors are a whole world of their own. What we have described here is circumscribed by their relationship to monads.

4 Utilities for Monadic Code

In the previous chapters, we learned how to write monadic code using dedicated syntax and how to lift functions from the pure world into monadic contexts. Yet, none of that functionality is higher-order, in the sense of assembling large monadic actions out of simpler monadic blocks. Being more concrete, do we have functions similar to `map` or `fold` but that use monadic `a -> m b` instead of plain functions `a -> b`? The answer is affirmative, and the goal of this chapter is to survey those combinators.

In contrast to other parts of this book, this chapter describes the current status of the Haskell and Scala ecosystems — in the latter case, for the Scalaz and Cats libraries. In other words, the functions shown here are those that are available in the corresponding libraries. But of course, there is always room for more monadic utilities. If you find that a certain function is lacking, write your own. And even better, submit a pull request to the maintainers of these libraries!

For the sake of conciseness, all examples from now on are written exclusively in Haskell, except for those occasions in which Haskell and Scala practices deviate. Translations between both languages should be simple, especially from `do` blocks into `for` comprehensions and vice versa. Note, however, that Haskell provides considerably more utilities than both Scalaz and Cats. On the other hand, utilities for Haskell are spread among multiple packages, whereas Scalaz and Cats come with more of them built-in.

4.1 Lifted Combinators

For most functional programmers, the entry point to higher-order functions is the venerable `map`. A call to `map` comprises a function `f` and a list `xs` and returns another list in which `f` is applied to each element in `xs`:

```
> map (+1) [1, 2, 3]  -- adds 1 to all numbers in the list
[2, 3, 4]
```

Now take a monadic action — we will use printing a string as an example — that you want to execute over all the elements of a list. Is it possible to use `map` in the same way?

```
map (\name -> print ("Hello, " ++ name)) ["Alejandro", "Elena"]
```

The answer is *no*. The problem is that the type of `print` is `String -> IO ()`, and thus the result of that mapping is of type `[IO ()]`. In other words, it is a list of monadic actions, not a monadic action itself. In the latter case, the result type would be `IO T` for some `T` — note the absence of the top-level list constructor.

There are two approaches to solving this problem. The first one is to write a new version of `map` that takes a monadic action as its first argument. In this way, we arrive at `mapM`:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM _ []     = return []
mapM f (x:xs) = do r  <- f x
                   rs <- mapM f xs
                   return (r:rs)
-- Alternatively, the second branch can be written in applicative style
mapM f (x:xs) = (:) <$> f x <*> mapM f xs
```

Haskell's base library defines a synonym for `mapM` called `forM` that reverses the argument order. In some sense, `forM` resembles an imperative loop, where the thing to be iterated over is written before the action to perform on each item:

```
forM ["Alejandro", "Elena"] $ \name -> print ("Hello, " ++ name)
```

The other option is to keep `map` as is but introduce a new function to move the monadic context from inside the list constructor, `[m a]`, to the outside, `m [a]`:

```
sequence :: Monad m => [m a] -> m [a]
sequence []     = return []
sequence (x:xs) = do r  <- x
                     rs <- sequence xs
                     return (r:rs)
-- Once again, we may use applicative style
sequence (x:xs) = (:) <$> x <*> sequence xs
```

Our previous `mapM` is now just a composition of two pieces:

```
mapM f = sequence . map f
```

In addition to `mapM`, many other list functions have monadic counterparts. Just to name a few from different sources:

```
-- In module Control.Monad, package base
filterM    :: Monad m => (a -> m Bool) -> [a] -> m [a]
zipWithM   :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
replicateM :: Monad m => Int -> m a -> m [a]
```

> **Exercise 4.1.** Write implementations for the last two functions introduced above in terms of their pure counterparts — `zipWith` and `replicate` — composed with `sequence`. What goes wrong when you try to do the same with `filterM`?

```
-- foldM performs a left fold, like foldl
foldM      :: Monad m => (b -> a -> m b) -> b -> [a] -> m b

-- In module Control.Monad.Extra, package extra
partitionM :: Monad m => (a -> m Bool) -> [a] -> m ([a], [a])
concatMapM :: Monad m => (a -> m [b]) -> [a] -> m [b]

-- In module Control.Monad.Loops, package monad-loops
--       and Control.Monad.Extra, package extra
andM, orM  :: Monad m => [m Bool] -> m Bool
anyM, allM :: Monad m => (a -> m Bool) -> [a] -> m Bool

-- In module Control.Monad.Loops, package monad-loops
takeWhileM :: Monad m => (a -> m Bool) -> [a] -> m [a]
dropWhileM :: Monad m => (a -> m Bool) -> [a] -> m [a]
firstM     :: Monad m => (a -> m Bool) -> [a] -> m (Maybe a)
```

4.1.1 Actions Without Value

A pure expression has no side effects. It just computes its output value. In contrast, monadic actions are executed for *both* their side effects and their resulting values. In some extreme — but not unusual — cases, a monadic action is apparent only from its side effects. This is the case, for example, when you change the state value contained by the `State` monad:

```
put :: s -> State s ()
```

Or when you print some text to the console:

```
putStrLn :: String -> IO ()
```

In the first case, we could argue about returning the modified state, but this is always equal to the given argument. In the second case, there is no sensible return value, taking into consideration that such a function may be called amidst a huge variety of operating systems and configurations.

The designers of these libraries decided to make explicit the fact that no interesting value comes out of these actions by returning a dummy value. The type `()`, pronounced "unit" or "empty tuple," has only one possible value, which is `()`. Thus, by knowing the type you know the value. Conversely, inspecting the return value gives you no information that you did not already have from its type.*

The reader may also remember that `do` notation, as described in Section 2.1.1, includes a way to indicate that you are not interested in the result of an action. The following three ways to call `put` are equivalent, although the third one is generally preferred:

```
do ...                    |  do ...                    |  do ...
   () <- put newValue     |     _ <- put newValue      |     put newValue
```

If you forget to pattern match on a `()` value, you know that you are not losing any information. But if you ignore a value of a different type, you might have done so inadvertently. For that reason, the compiler warns you about dropped values in a monadic context. This warning can be annoying, though. Imagine once again that we want to print a list of values to the screen:

```
forM ["Alejandro", "Elena"] $ \name -> print ("Hello, " ++ name)
```

The return type is `IO [()]`, not `IO ()`, so we might still be interested in some properties of the return value. In this case, the only other piece of information is the length of the list, since we know that all the elements are going to be empty tuples. In order to silence the warning, we can use the `void` function:†

```
void :: Functor m => m a -> m ()
void = fmap (\_ -> ())
```

This function substitutes any meaningful value contained in the monad with the dummy `()`. Now, we can finally iterate over all the elements in a list solely to perform side effects and without any warnings:

```
void $ forM ["Alejandro", "Elena"]
     $ \name -> print ("Hello, " ++ name)
```

*There is always the possibility that a function with return type `()` will not terminate. However, for most practical purposes, we assume that this scenario should not arise.

†As you may have guessed, Haskell's `return` function is not your imperative `return`, and neither Haskell's `void` nor `Void` have anything to do with the `void` type in procedural languages. Even worse, `void` is a function and `Void` is a type, but the former does not return a value of the latter. In fact, there are no values of type `Void`.

This idiom is common in monadic code. For that reason, Haskell libraries follow the convention of defining two variants of their combinators, the common one — usually ending in `M` — and the one that ignores the return value — which ends with the _ symbol. For example:

```
mapM_ :: Monad m => (a -> m b) -> m a -> m ()
mapM_ f = void . fmap f
-- We can also define mapM_ f = sequence_ . map f
sequence_ :: Monad m => [m a] -> m ()
sequence_ = void . sequence
-- And so forth
zipWithM_   :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
replicateM_ :: Monad m => Int -> m a -> m ()
foldM_      :: Monad m => (b -> a -> m b) -> b -> [a] -> m ()
```

One advantage of the empty tuple is the ease of constructing one. If you want to create a monadic action `m ()`, you can just use `return ()`, regardless of the monad. In imperative languages, you usually have conditionals where one of the branches may not be present. We can provide a similar abstraction for monadic contexts by using the empty tuple as a dummy value:

```
when :: Monad m => Bool -> m () -> m ()
when cond action = if cond
                     then action
                     else return ()
-- Alternative definition
when True  action = action
when False _      = return ()

unless :: Monad m => Bool -> m () -> m ()
unless cond = when (not cond)
```

In my experience, it is also helpful to define a version of the conditional where the check is done in a monadic context. You can get it from the module `Control.Monad.Extra` in the package `extra` or just define it yourself:

```
ifM :: Monad m => m Bool -> m a -> m a -> m a
ifM cond th el = do c <- cond
                    if c then th else el
-- Alternative definition
ifM = liftM3 (\c t e -> if c then t else e)
```

In the monad world, there is a related function, `guard :: Bool -> m ()`, that can be used to short circuit computations that do not satisfy a condition. Think of a list comprehension that checks a property of its elements. However, the `Monad` type class does not define such an operation — for that, we need `MonadPlus`, which will be introduced in Chapter 7.

4.1.2 Loops

Our walk around monadic libraries now leads us to consider how monadic code interacts with loops. Whereas a `while` loop does not make much sense in a pure context, since the condition is always going to be the same and thus result in either an infinite loop or one that is never executed, monads are powerful enough to define loops where the condition actually changes in each iteration.

The standard library bundled with GHC only defines a single looping function:

```
forever :: Monad m => m a -> m b
forever action = do action
                    forever action
```

As you can read in the code, `forever f` repeats `action` infinitely. The type is a bit surprising, though: how can it return a value of *any* type `b` you want? The trick is that `forever f` *never* returns, so it is free to promise whatever it wants — it will never fulfill that promise anyway.

Many other looping constructs are defined in one of my favorite packages in Hackage,* `monad-loops`. The simplest of these combinators is `whileM`, which executes an action until a condition is satisfied:

```
whileM :: Monad m => m Bool -> m a -> m [a]
whileM cond action = do
  c <- cond
  if c
     then (:) <$> action <*> whileM cond action
     else return []
```

This version of the function keeps the value obtained in each iteration, but it has a counterpart, `whileM_`, that throws it away.

The function `iterateWhile` also checks a condition until it fails, but it does so on the result of the previous iteration. For that reason, you do not get a list of values, as with `whileM`, just the one that makes the iteration stop:

```
iterateWhile :: Monad m => (a -> Bool) -> m a -> m a
iterateWhile cond action = do
  r <- action
  if cond r
    then iterateWhile cond action
    else return r
```

As in most functions using predicates, we have versions where the predicate is negated. In this case, the corresponding version of `whileM` is `untilM`, and for `iterateWhile`, we get `iterateUntil`. This last one has an interesting generalization, such that a new value is computed on each run of the loop using the value from the previous iteration as input:

*A community-maintained package repository for Haskell.

```
iterateUntilM :: Monad m => (a -> Bool) -> (a -> m a) -> a -> m a
iterateUntilM cond producer value
  | cond value = return value
  | otherwise  = do next <- producer value
                    iterateUntilM cond producer next
```

A close relative of this function is `loopM` from the `extra` package, in which the decision about stopping or continuing is not given by a Boolean predicate but is determined instead by the constructor of an `Either` value. If the result for a run of the loop is `Right`, we keep going. If it is `Left`, we stop and return that value:

```
loopM :: Monad m => (a -> m (Either a b)) -> a -> m b
loopM cond x = do r <- cond x
                  case r of
                    Left next -> loopM cond next
                    Right end -> return end
```

The fact that the stop condition comes with an associated payload allows us to return a different type than the one we use to iterate. This can be helpful if we are more interested in knowing what made the iteration come to an end than in the values we dealt with.

4.2 Traversables

The list constructor supports two ways of mapping a function over all of its elements:

- Purely, via `map :: (a -> b) -> [a] -> [b]`.
- Monadically, via `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`.

In Section 1.5, we discussed how the first notion could be extended to other type constructors, giving rise to the `Functor` type class. Can we do the same with `mapM`? The answer is affirmative, as we will see in this section, and the corresponding notion is `Traversable`.

Let us first compare the implementation of `map` and `mapM`:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

```
-- The usual implementation
mapM f [] = pure []
mapM f (x:xs)
  = (:) <$> f x <*> mapM f xs
-- Equivalent way to write it
mapM f [] = pure []
mapM f (x:xs)
  = liftM2 (:) (f x) (mapM f xs)
```

As we can see, the shape of the function is roughly the same. The core difference is that in `mapM`, the constructors are lifted — `pure` is a function that lifts a constant, a sort of `liftM0`. Using this idea, we can generalize `fmap` for `Maybe` in a similar fashion:

```
fmap f Nothing  = Nothing
fmap f (Just x) = Just (f x)
```

```
-- Using lifting
mapM f Nothing  = pure Nothing
mapM f (Just x) = fmap Just (f x)
-- Applicative style
mapM f Nothing  = pure Nothing
mapM f (Just x) = Just <$> f x
```

The type class that gives a name to this monadic mapping functionality is `Traversable`. Its members are sometimes called *traversable functors*:

```
class Functor f => Traversable f where
  mapM :: Monad m => (a -> m b) -> f a -> m (f b)
```

If you look at either the Haskell definition or the trait defining this notion in Cats — where it has the name `Traverse` — you may notice that there is yet another method:

```
class Functor f => Traversable f where
  traverse :: Applicative m => (a -> m b) -> f a -> m (f b)
```

This function is exactly the same as `mapM`, but it asks for an applicative functor instead of a full monad.* In fact, in the previous definitions of `mapM` for both lists and optional values, we only used methods from `Applicative` and not a single bind. The same definition works for both `mapM` and `traverse`. This duplication in Haskell is a historical accident, since monads were introduced in the language before applicative functors. Modern code uses `traverse` wherever possible.

But wait! If you look at Haskell's `Traversable` type class, there is yet another pair of functions:

```
sequenceA :: Applicative m => f (m a) -> m (f a)
sequence  :: Monad       m => f (m a) -> m (f a)
```

This is the generalization of the `sequence` function for lists. This duplication has the same historical roots as the one involving `traverse` and `mapM`. As an example, we can instantiate the function to work on `Maybe` values:

```
sequence :: Monad m => Maybe (m a) -> m (Maybe a)
```

The role of `sequence` is to "swap" the monad from inside the traversable to the outside.

*Remember, every monad is an applicative functor but not vice versa.

One important property of traversables is that you only need to define *one* of these functions to get their full functionality, akin to how a monad is definable either by its bind function or by using `join`. We already know one implementation from our discussion of lists:

```
mapM f = sequence . fmap f
```

In this definition, we map a monadic function `f` as if it were pure, and then we flip the monad part to the outside via `sequence`. The interesting part is that you can also define `sequence` in terms of `mapM`. Let us think about it — we need to find a function `g`, such that:

```
sequence = mapM g  ::  f (m c) -> m (f c)
```

Since the type of `mapM` is `(a -> m b) -> f a -> m (f b)`, we need `g` to be of type `m c -> m c`. We can see this by noticing that the `a` in `mapM` ought to be instantiated `m c`, and the `b` has to be exactly `c` to make the types match. Once we know that the type of `g` has to be `m c -> m c`, finding the function is easy: it is just the identity! As a result, `sequence = mapM id`.

Traversables are powerful, because they relate *two* functors. My recommendation is that you learn to spot those places where a container — the traversable — is used in a monadic context, because `sequence` and `traverse` will come in handy. One concrete example is taking a list of optional values and turning it into a regular list of values that is itself optional, the operation succeeding only if none of the values are missing. Such a function would have this type:

```
allJust :: [Maybe a] -> Maybe [a]
```

The implementation is simply `sequence`, where the list type assumes the role of container, and `Maybe` is the context where the container lives.

4.2.1 Doing Nothing While Traversing

We first introduced `Functor` as the class of types that allow mapping pure functions and then extended it to `Traversable` for types that also allow mapping monadic functions. We have been talking about monads as a means to add a certain context to a computation — the ability to fail, threading state, and so on. The question we pose in this section is whether we could have gone the other way, starting with `Traversable` and treating `Functor` as a special case when the monad does not add any contextual information.

To answer this question, we first need to describe the monad that does nothing. This may look like a mere academic exercise, but it is the key to some constructions such as monad transformers, which we will introduce in Chapter 11. Mathematicians often refer to these “do nothing” elements as identities, and following their lead, we

name our monad `Identity`. As discussed in the Introduction, due to the problem of overlapping instances in Haskell — a problem that does not exist in Scala — our first definition of `Identity` is "pseudo-Haskell," but it nevertheless serves the purpose of demonstrating its behavior:

```
-- Beginning of pseudo-Haskell
type Identity a = a  -- do nothing with the value

instance Monad Identity where
  return :: a -> a
  return x = x    -- also, return = id
  (>>=) :: (a -> b) -> a -> b
  f >>= x = f x   -- also, (>>=) = id

instance Functor Identity where
  fmap :: (a -> b) -> a -> b
  fmap f x = f x  -- also, fmap = id
-- End of pseudo-Haskell
```

```
object Id {  // Adapted from Scalaz
  type Id[X] = X

  implicit val idMonad: Monad[Id] with Functor[Id]
    = new Monad[Id] with Functor[Id] {
        def point[A](x: A): A = x
        def bind[A, B](x: A)(f: A => B): B = f(x)
        def map[A, B](x: A)(f: A => B): B = f(x)
      }
}
```

As we can see, none of the operations change the values involved in any way — they are all identity functions. When we apply `return`, there is no wrapping to be done, and when we apply a function to a value via `(>>=)` or `fmap`, we just do it.

Now take the `mapM` function of any traversable functor:

```
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
```

We can instantiate it to the `Identity` monad. Since `Identity a` is equal to `a`, we end up with the following type:

```
mapM :: Traversable t => (a -> b) -> t a -> t b
```

This function coincides with the map of a functor and thereby proves that any traversable is, in fact, also a functor.

The real `Identity` implementation. Alas, the previous Haskell code is not correct: the language does not allow using a type synonym — defined by `type` — as an instance declaration. This is so, because otherwise we could hide clashing instances by using different synonyms. Note that this restriction is not present in the Scala implementation, where we are free to define `Id[A]` as `A`.

The trick is to introduce a data type that holds a single value of the desired type. Using `newtype` instead of `data` ensures that no extra memory is allocated other than that which is used for the single wrapped value. It marks this new type as distinct for the compiler but equivalent to its underlying type at runtime:

```
newtype Identity a = I a
```

Now we can create an instance for this newly-defined type. However, we need to pattern match and wrap the elements in the constructor manually:

```
instance Monad Identity where
  return :: a -> Identity a
  return x = I x
  (>>=) :: Identity a -> (a -> Identity b) -> Identity b
  (I x) >>= f = f x

instance Functor Identity where
  fmap :: (a -> b) -> Identity a -> Identity b
  fmap f (I x) = I (f x)
```

Note that, in this case, we really *need* the `newtype`. Otherwise, we would be trying to create an instance that would match with any type `a`. In other words, the instance would match *every type* in the code, thus colliding with any other functors or monads already defined.

5 Interlude: Monad Laws

Up to this point, we have spoken a great deal about monads and their relationship to other abstract concepts, such as functors and traversables. But along the way, we have left out an important part of their behavior: the *laws* that all monads must abide. Getting to know monad laws is not just an academic exercise. It helps us understand how monads are expected to behave, so we are not surprised.

We start this chapter by looking at the general concept of a law, using functions — the core type of functional programming — as a source of examples. Functors, which manipulate functions, are the next rung of our ladder. Then we move on to monoids and monads, two constructions that are related in several ways when we consider their laws.

5.1 Laws for Functions

Before discussing even the simplest law, we should make it clear what we mean by a law. A *law* is a property of a function, or set of functions, that relates their behavior in different scenarios. In most cases, this relation takes the form of an *equality* between two expressions. Even more concretely, we say that `f x ...` $\equiv$ `g x ...` for some expressions `f` and `g` when given equal arguments `x ...`. Note that we use the symbol "$\equiv$" here to represent equality at a logical level — as in, "$2x$ is equal to $x + x$" — as opposed to the use of equality in function declarations, such as "the function `succ x` is defined as `x + 1`," and in (possibly mutating) variable assignments in imperative languages, both cases where the symbol "$=$" typically appears, instead.

Consider the following two functions from the standard functional toolbox:

```
(.) :: (b -> c) -> (a -> b) -> a -> c  -- function composition
f . g = \x -> f (g x)
```

```
id :: a -> a  -- identity function
id x = x
```

One law involving these functions — called *left identity* — states that for every function `f`:

```
id . f ≡ f
```

Since both sides of the equality operator are functions, this statement is equivalent to saying that for every function `f` and every argument `x`,[*] we can state that:

```
(id . f) x ≡ f x
```

This law also has a dual law[†] — *right identity* — that tells us that `f . id ≡ f` for every function `f`.

These laws tell us that our intuition about the identity function, that it does not do anything to the value it takes, is correct. If `id` were doing something other than "passing through" its argument, we would not be able to expect these laws to work for *every* `f`.

Functions satisfy another important law, called *associativity*. Given any three functions, with types `f :: c -> d` (no relation to the `f` used in the law above), `g :: b -> c`, and `h :: a -> b`, we have the property:

```
f . (g . h) ≡ (f . g) . h
```

Note the generality of this statement — it works regardless of the functions. We just need their types to match.

Once again, this law tells us that function composition does not conceal any hidden surprises. Imagine writing a function as a pipeline, or a composition, of smaller blocks:

```
g = f1 . f2 blah . f3 bleh . f4
```

If, by choosing different positions for the parentheses, we obtain different results, we would have to think twice about using this pipeline style. Fortunately, the parentheses do not matter at all, so we do not have to care about where they are introduced by the compiler. The idea is similar to writing:

```
1 + 2 + 3 + 4 + 5
```

We do not need to worry about which additions to perform first and which ones later, since we know that the result will be the same regardless of the order of operations.[‡]

[*] Technically, this is called *extensional* equality for functions: two functions are the same if they return the same output for every input. There is another kind of equality for functions — *intensional* — but we will not talk about it any further in this book.

[†] Category theory provides a formal definition for *duality*, but we do not need to go that deep here. Intuitively, the dual of a law states a symmetric version of the original law according to the order of the arguments, the shape of the types, or any other aspect.

[‡] Note that addition also satisfies *commutativity*, `x + y ≡ y + x` for any pair of numbers, whereas function composition does not.

Proving laws. What we call laws, mathematicians call *theorems*. This gives us the hint that laws need to be *proven*: we cannot just state a law, we need to check that it is, in fact, true.

There are many ways to prove that a law holds. The simplest technique, called *equational reasoning*, works by substituting equal expressions for equal expressions in tiny steps, until we obtain the equality we want. We follow a style in which every $\equiv$ sign includes the reason for the equality between the two expressions surrounding it. For example, the proof of function associativity looks like this:

```
  (f . (g . h)) x
≡ -- by definition of (.)
  f ((g . h) x)
≡ -- by definition of (.)
  f (g (h x))
≡ -- by definition of (.)
  (f . g) (h x)
≡ -- by definition of (.)
  ((f . g) . h) x
```

Equational reasoning is usually augmented by *induction*, which provides a way to prove laws that involve data types such as numbers, lists, and trees.

Proving laws is a wide area in itself that we do not aim to cover in this book. The interested reader may consult *Programming in Haskell* [Hutton, 2016], which provides a gentle introduction to the topic.

5.1.1 Functor Laws

The laws associated with a functor relate its only operation, `fmap` or `map`, to the identity and composition functions:

```
     fmap id ≡ id
fmap (f . g) ≡ fmap f . fmap g
```

In other words, identities are mapped to identities, and composition inside of `fmap` is mapped to the composition of the lifted functions. For that reason, we say that functors *preserve* identities and composition.

Another way to state these laws is that a functor must preserve the *structure* of the container it maps over untouched. It should only be able to affect whatever information is encoded inside of that container. As `id` returns its input as it is, the fact that mapping `id` over any structure keeps the *whole* structure the same — since `fmap id` equals `id` — tells us that `fmap` itself only has the power to modify the elements inside a container and not the container itself.

This idea leads us to a couple of *counterexamples*, implementations of `fmap` that have the right type but do not satisfy the functor laws. Both counterexamples show the list type constructor with a different functor instance than the usual one.

The first one turns everything into an empty list:

```
instance Functor [] where
  fmap _ _ = []
```

The first law is violated in this case:

```
   fmap id [1, 2]
≡  -- by definition of fmap
   []
≠
   id [1, 2]
≡  -- by definition of id
   [1, 2]
```

This definition of `fmap` changes the shape of the input list: `[1, 2]` has two elements, whereas `[]` has none. But the shape of a list is not only the number of its elements. Let's consider yet another instance of the list type constructor, one that reverses the list in addition to mapping the function:

```
instance Functor [] where
  fmap f xs = reverse (map f xs)
```

Both laws are violated, as the following examples show:

```
   fmap id [1, 2]
≡ [2, 1]
≠
   id [1, 2]
≡ [1, 2]

   fmap ((+1) . (+1)) [1, 2]
≡ fmap (+2) [1, 2]
≡ [4, 3]
≠
   fmap (+1) (fmap (+1) [1, 2])
≡ fmap (+1) [3, 2]
≡ [3, 4]
```

The shape of a list is defined not only by the number of its elements but also their relative positions.

Abstracting from this example, the functor laws guarantee that only elements are affected by a functorial operation. If we want the first law to hold, every constructor in the data type must be mapped to the exact same constructor by `fmap`. The second law tells us that if we apply two functions in a row, it does not matter whether we apply both at once over the elements or we traverse the structure twice.

5.2 Monoids

There are many more structures in programming and mathematics that come with a binary operation that is associative and have some notion of an identity element, in a similar fashion to functions.* In Haskell or Scala terms, we have a type T with an operation $\oplus$ that takes two Ts into another one and an element of T called e, such that:

$$\begin{aligned} e \oplus x = x = x \oplus e \qquad & \text{left and right identity} \\ (x \oplus y) \oplus z = x \oplus (y \oplus z) \qquad & \text{associativity} \end{aligned}$$

```haskell
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

```scala
trait Monoid[A] {
  def empty: A
  def combine(x: A, y: A): A
}
```

There is another related notion, *semigroup*, that denotes a type that has an associative operation but not necessarily an identity element.

The archetypal examples of monoids come from the realm of numbers. In this case, there are at least *three* different and useful monoidal structures:

1. Numbers form a monoid with respect to addition, where 0 is the identity.
2. They also form a monoid under multiplication, in which case the identity is 1.
3. The operation of taking the maximum or minimum also leads to a monoidal structure, if you consider $-\infty$ or $+\infty$, respectively, as identity elements.

In functional programming, we are especially fond of another monoid, lists:

```haskell
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Note that in contrast to the list *monad*, which employs a property of the type *constructor* `[]`, here we specify that the *ground* type `[a]` forms a monoid regardless of the type of its elements.

The power of monoids comes from the fact that they are everywhere! Not only lists and numbers, Booleans also form a monoid with many of their operations, and tuples with monoidal components are monoids, too. To offer a revealing data

*Note that functions do not form a monoid but a *category*, because not every pair of functions can be composed.

point, the standard library bundled with GHC 8.4.3 comes with 30 instances of the `Monoid` type class.

One practical application of monoids is answering the question: in which cases are the results of using a right fold (`foldr`) and a left fold (`foldl`) the same? Remember that if we have a list `[a, b, ..., z]` that we fold using an operation $\oplus$ and an initial element e, these two variations correspond to the following sequence of applications:

$$\begin{array}{ll} a \oplus (b \oplus (\cdots \oplus (z \oplus e))) & \text{right fold} \\ (((e \oplus a) \oplus b) \oplus \ldots) \oplus z & \text{left fold} \end{array}$$

If the $\oplus$ and e operations form a monoid for their corresponding type, then we are free to move the parentheses from one nesting order to the other. Furthermore, the values of both $z \oplus e$ and $e \oplus a$ are not affected by the identity element e, so we can just ignore it.

> **Exercise 5.1.** The type `Maybe a` has two different monoidal structures. Can you describe them? Hint: we say that one of these structures is right-biased and the other left-biased.

For the sake of completeness, there is a property of monoids and e similar to what we find with functors and identity. That is, any function that preserves monoidal structure is called a *monoid homomorphism*. Given a function `f`, we say it is a monoid homomorphism if it satisfies the following two laws:

```
        f mempty ≡ mempty
f (x `mappend` y) ≡ (f x) `mappend` (f y)
```

Note that the identity element and combination operations on either side of the equation may be different. For example, `length` is a monoid homomorphism between lists with the concatenation operation and numbers with addition, since:

```
length mempty ≡ length [] ≡ 0 ≡ mempty

length (x `mappend` y) ≡ length (x ++ y)
                       ≡ length x + length y
                       ≡ length x `mappend` length y
```

> **Exercise 5.2.** Show that the exponentiation function `exp`, which computes e^x, is a monoid homomorphism between the monoid of numbers with addition and the monoid of numbers with multiplication.

5.3 Monad Laws

As in the case of functors, every monad that deserves the name must not only implement certain methods from the corresponding type class or trait, but it must also do so in a way that satisfies certain laws. Note that these laws must be satisfied *in addition* to the functor laws, since every monad is also a functor.

There are different formulations of the same laws, and in this section we look at a couple of them. Using Haskell, the laws look like this. In most cases we can write the same code using `do` notation and explicit bind operations:

```
-- Left identity
   f x
≡
   do y <- return x
      f y
≡
   return x >>= f

-- Right Identity
   m
≡
   do x <- m
      return x
≡
   m >>= return

-- Associativity
-- using do notation
   do y <- (do x <- m
               f x)
      g y
≡
   do x <- m
      (do y <- f x
          g y)
-- using explicit binds
   (m >>= f) >>= g
≡
   m >>= (\x -> f x >>= g)
```

The first two laws guarantee that `return` and `point` do not interfere with the pure parts of the language. Returning or yielding a pure value, and then consuming it, should be indistinguishable from merely using the pure value. The associativity

law, in turn, tells us that nesting `do` or `for` blocks for the same monad is unnecessary and that all nestings have the same behavior. For that reason, we can safely write:

```
do x <- m
   y <- f x
   g y
```

We do not even have to think about where the compiler will insert parentheses. Once again, these laws allow us to write code without surprises, since the laws encode our intuition about sequential computations.

The counterparts of the monad laws written in Scala, using both `for` comprehensions and explicit monad operations, look like this:

```
// Left identity
   f(x)
≡
   point(x) bind f

// Right identity
   m
≡
   for { x <- m } yield x
≡
   m bind point

// Associativity
   (m bind f) bind g
≡
   m bind { x => f(x) bind g }
```

5.3.1 Kleisli Arrows

The justification for the monad laws is that they guarantee our computations will behave as expected when several actions are composed. But you, dear reader, may have noticed that these laws are called "left and right identity" and "associativity," yet they do not look at all like the corresponding laws for functions and monoids. However, the names make sense if we look at them from a slightly different point of view.

Let us consider functions that have a specific shape, namely `a -> m b` for a monad `m` and two types `a` and `b`. These are called the *Kleisli arrows* for that monad. For example, the action `validateName` with type `String -> Maybe Name` is a Kleisli arrow for the `Maybe` monad.

Kleisli arrows come with their own version of composition. The usual composition operator combines two pure functions and hence has this type:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Whereas this new version combines two Kleisli arrows:

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
f <=< g = \x -> do y <- g x
                   f y
-- Alternatively
f <=< g = \x -> g x >>= f
```

The notion of identity also makes sense in this setting. However, the identity we are looking for is not one of type `a -> a`, since it does not follow the shape of a Kleisli arrow. To fulfill that latter requirement, we need a function `a -> m a`. This is exactly the type of `return` or `point`!

If we now rewrite the monad laws using the special composition for Kleisli arrows and taking `return` as the identity, they read:

```
   return <=< f  ≡  f                 -- Left identity
   f <=< return  ≡  f                 -- Right identity
(f <=< g) <=< h  ≡  f <=< (g <=< h)   -- Associativity
```

We can think of these laws in two different ways. First of all, monad laws are nothing but the usual function laws with functions replaced by Kleisli arrows. Second, monads are similar to monoids, in which we combine Kleisli arrows — in other words, monadic actions — instead of more basic values such as integers and Booleans. This line of thought eventually leads us to the discussion in Chapter 17: a monad is just a monoid in the category in which objects are functors. But there are many things to discuss before we get there.

Part II

More Monads

6

Pure Reader-Writer-State Monads

The first part of this book deals with what unites monads. In this second part, we are going to focus on the specifics of the most common monads that we find in the wild, giving special attention to which operations they provide in addition to their monadic interfaces.

We start this quest by having a closer look at the State monad and its close colleagues, the Reader and Writer monads. In a certain sense, Reader and Writer provide just one part of a stateful interface, obtaining or saving a value, respectively. Another common characteristic among these three monads is that they are built from simple, functional building blocks: functions and tuples.

At this point, Haskell and Scala code look fairly similar, except for the small syntactic differences between them. For that reason, the code blocks are still only in Haskell, except for those places where the languages diverge considerably.

A note on packaging. In Scala, you can find most of the monads we describe in this part of the book in the usual distributions of both Scalaz and Cats. In Haskell, the situation is a bit more complicated. Lists, `Maybe`, and `Either` are distributed in the `base` package and included as part of the Prelude, the module that is imported by default into every Haskell file. There is a second set of basic monads that live in the `transformers` package, each of them in a different module. This means that if you want to use them, you need to add a dependency to the `transformers` package in your project file and import the corresponding module. Finally, some monads — like `Logic` — are distributed in their own package.

6.1 The State Monad

The State monad was our first example in Chapter 1. A value of `State s a` should be understood as a computation that produces values of type `a` while modifying an

internal state of type `s`. In fact, we can see `State s a` as the encoding of functions from an initial state to their return values, paired with the new state:

```
type State s a = s -> (a, s)
```

Haskell implementation warning. Due to restrictions in the way type class instances are resolved, it is not possible to write a `Monad` instance for a type synonym. In order to work around this problem, Haskellers use the `newtype` trick discussed in Section 0.3:

```
newtype State s a = State { runState :: s -> (a, s) }
```

The downside is that now every time we need to build or inspect a `State s a` value, we need to add or peel off the constructor. For example, here is the `Functor` instance:

```
instance Functor (State s) where
  fmap f (State x) = State (\s -> let (a, s') = x s in (f a, s'))
```

In contrast, in the description of `State` in Chapter 1, we wrote:

```
instance Functor (State s) where
  fmap f x = \s -> let (a, s') = x s in (f a, s')
```

The same trick is played for the rest of the monads in this chapter.

> **Exercise 6.1.** Write the `Monad` instance for this variation of `State s`.

State operations. Since we know the internals of the State monad, we could inspect or update the internal state directly. For example, this is a function that updates a counter — represented as an integral state — and gives back the current value:

```
nextValue :: State Int Int
nextValue = State (\i -> (i, i + 1))
```

This is not a good idea for exactly the same reasons that it is not a good idea to access private methods from a module or class. Instead, we should use the functions that `State` provides as an *interface* to build stateful computations. Doing so will pay off very soon, when we introduce monad transformers in Chapter 11.

The interface for State is comprised of three functions, namely:

```
   get ::             State s s
   put ::  s       -> State s ()
modify :: (s -> s) -> State s ()
```

The names are self-descriptive. `get` obtains the current value of the state and makes no change to it. `put` performs the dual operation: it changes the state of the computation but returns nothing interesting. `modify` also modifies the current state but does so by applying a function to it.

The availability of different ways to modify a value allows us to write the previous `nextValue` action in two different, both better ways:

```haskell
nextValue = do i <- get
               put (i+1)
               return i
```

```haskell
nextValue = do i <- get
               modify (+1)
               return i
```

Exercise 6.2. Define the function `modify` in terms of `get` and `put`.

Starting the computation. A value of type `State s a` describes how to obtain a result value and a modified state *given* an initial value. If you want to execute the action, you need to provide that value. There are a handful of ways to do so, depending on whether you are interested in the result value, the resulting state, or both:

```haskell
runState  :: State s a -> s -> (a, s)
evalState :: State s a -> s -> a
evalState c = fst . runState c  -- Keep only the value
execState :: State s a -> s -> s
execState c = snd . runState c  -- Keep only the final state
```

In the Haskell implementation, `runStateT` corresponds to the function that extracts the field from the `State` constructor. But even in Scala, the same functions are provided in order to abstract away the implementation details.

As an example of the usage of the whole `State` interface, let's look at the execution, step by step, of the following code. There is no real action here, since the initial state is just read and returned, never modified, but it allows us to see how the state flows through the different stages:

```haskell
runState (do x <- get
             return x) 0
```

As we discussed in Section 2.1.1, `do` notation is translated into a sequence of calls to the bind operation in the monad. This is the code we actually execute:

```haskell
runState (get >>= \x -> return x) 0
```

The trace of execution of this code is given in Figure 6.1. Note that in this trace, we are using the `State` that is wrapped in a `newtype`, hence the calls to the constructor `State` and the accessor `runState`. If you have not done so before, this might serve as inspiration to write the `Monad` instance for this version of `State`.

```
   runState (get                  >>=    \x -> return x) 0
≡ runState (State (\s -> (s, s)) >>=    \x -> return x) 0
≡ runState (State (\s' -> let (a, s'') = (\s -> (s, s)) s'
                          in runState ((\x -> return x) a)   s'')) 0
≡ runState (State (\s' -> let (a, s'') = (\s -> (s, s)) s'
                          in runState ((\x -> return x) a)   s'')) 0
≡                ((\s' -> let (a, s'') = (\s -> (s, s)) s'
                          in runState ((\x -> return x) a)   s'') 0)
≡                         let (a, s'') = (\s -> (s, s)) 0
                          in runState ((\x -> return x) a)   s''
≡                            runState ((\x -> return x) 0)   0
≡                            runState (       return 0)      0
≡                            runState (State (\s -> (0, s))) 0
≡                                                (\s -> (0, s))  0
≡                                                       (0, 0)
```

Figure 6.1: Trace of `runState (get >>= \x -> return x) 0`

Throughout the rest of this chapter, we will look at several other monads. Their descriptions follow the same pattern as for `State`:

1. A set of primitive operations on the monad, in this case `get` and `put`.
2. Some derived operations, such as `modify`.
3. A function that runs or executes the effects provided by the monad. In this case, we have `runState` and variations derived from it.

6.2 The Reader Monad

The `Reader` monad enhances a computation with a reference to an environment that can be queried at any moment but, in principle, cannot be changed. The main, primitive operation for `Reader` is, in fact, the one that obtains — or "reads" — this environment:

```
ask :: Reader r r
```

From this signature, we can see that `Reader` receives a type argument that specifies the type of the environment, as is the case for `State`. From this operation, we derive the subtly different `asks`, which applies a function to the environment before returning it:

```
asks :: (r -> a) -> Reader r a
asks f = f <$> ask
```

Also, in a similar fashion to `State`, we provide the value for the environment that is needed to start the computation via a specific function:

```
runReader :: Reader r a -> r -> a
```

By this point, you might have realized that `Reader` can be simulated with `State` if you never modify the environment. This gets even more obvious once we discuss the implementation.

`Reader` encapsulates the pattern of having one parameter in a function that is queried at several points but never changes. In most situations, the environment defines the global configuration of your application. Without this monad, we are forced to manage the arguments explicitly.

Consider the following piece of code, in which `cfg` represents a configuration value. Parts of this configuration are threaded to all three calls in its body:

```
handle :: Config -> Request -> Response
handle cfg req =
  produceResponse cfg (initializeHeader cfg) (getArguments cfg req)
```

Clearly, there is a lot of boilerplate here. With the `Reader` monad, we can remove most of it:

```
handle :: Request -> Reader Config Response
handle req = do header <- initializeHeader
                args   <- getArguments req
                produceResponse header args
```

Another way to look at `Reader` is to think of it as introducing an implicit parameter, which is threaded under the covers by the compiler. But in contrast to real implicit parameters, such as the ones in Scala, you can only have one per computation.

6.2.1 Focusing the Environment

Consider the following scenario, in which the configuration of your application is made of different parts related to different subsystems:

```
data Config = Config { userConfig :: UserConfig
                     , dbConfig   :: DatabaseConfig
                     , logConfig  :: LoggingConfig }
```

At first glance, if you decide to take the route of using a `Reader` to pass the configuration, you are forced to pass the whole of it to every operation:

```
handle = do q <- ...
            r <- query q
            ...
  where query :: DatabaseQuery -> Reader Config Result
        query q = ...
```

This solution forces `query` to include an additional call to `dbConfig` to obtain the part of the configuration it is interested in. This is inconvenient but could be improved with a dedicated `ask` for that field:

```
  where query :: DatabaseQuery -> Reader Config Result
        query q = do dbc <- asks dbConfig  -- or dbConfig <$> ask
                     ...  -- work with dbc
```

It would be better, however, to make `query` unaware of any other configuration parameters it does not use. That way, we reduce the chances of introducing coupling between the different subsystems that is too strong:

```
  where query :: DatabaseQuery -> Reader DatabaseConfig Result
        query q = do dbc <- ask  -- no call to dbConfig!
```

The problem is that the implementation of `handle` no longer compiles, since the types of the environment — `Config` and `DatabaseConfig` — differ. There is no way to bridge both worlds using `ask`, the only additional operation provided by the monad. One could, however, hack an approach by manually running the inner `Reader` computation:

```
handle = do q <- ...
            dbc <- asks dbConfig
            let r = runReader (query q) dbc
            ...
```

This pattern is encoded in a derived operation called `withReader`. The purpose of `withReader` is to define *local environments*, where we only need the information provided by part of the environment:

```
handle = do q <- ...
            r <- withReader dbConfig (query q)
            ...
```

In addition to `withReader`, which may alter the `Reader` environment arbitrarily, there is a specialized function `local` that maintains the same type. Note the difference in their signatures: `withReader` takes a function `r -> s`, whereas `local` requires `r -> r`:

```
withReader :: (r -> s) -> Reader s a -> Reader r a
local      :: (r -> r) -> Reader r a -> Reader r a
```

This functionality is useful when the configuration is slightly different in part of your code. One good example is logging — for a certain piece of code you might desire to have more verbose logging output than for the rest of your application:

```
handle = do q <- query
            r <- withReader dbConfig (query q)
            local (setLogLevel 3) (render r)
  where ... -- previous definitions
        setLogLevel :: Int -> Config -> Config
        setLogLevel lvl cfg = cfg { logConfig = ... }
        render :: Result -> Reader Config Html
        render r = ...
```

6.2.2 Implementation

Intuitively, the `Reader` monad works like a function for which one argument is always threaded implicitly. This intuition serves for writing an implementation — we will make `Reader r a` a simple function `r -> a`:

```
type Reader r a = r -> a
```

The `return` method must create a function `r -> a` from a single value `a`. Without any further requirement on `a`, the only choice is to create a constant function that always returns the given argument. For the case of bind, we need to propagate the environment to each of its arguments:

```
instance Monad (Reader r) where
  return x = \_ -> x
  x >>= f  = \env -> f (x env) env
  -- alternatively
  x >>= f  = \env -> let x' = f env in f x' env
```

You can compare this definition with the one for `State` given in Chapter 1. The shape is similar, although `Reader` does not need to thread any state from the first computation in the bind to the second, because the environment is immutable.

Let us look now at the primitive operations of the monad. We have `ask`, which gives back the environment. As a function, we simply return the environment as given to us:

```
ask :: Reader r r
ask = \env -> env
```

The other operation is `withReader`. We have discussed an implementation in terms of `ask` and `runReader`, but we can also write a more direct one:

```
withReader :: (r -> s) -> Reader s a -> Reader r a
withReader modifyEnv reader = \env -> reader (modifyEnv env)
```

Haskell implementation. Although, in this case, it is technically possible to just give an instance of `Monad` for `(->) r`, the `transformers` libraries prefer to follow the same convention as for `State` and use `newtype`. There is a bit more code, in that case, because we need to wrap and unwrap the values:

```
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
  return x = Reader (\_ -> x)
  x >>= f = Reader $ \env -> let x' = runReader x env
                             in runReader (f x') env
```

In addition, we need to update the primitives `ask` and `withReader` to support this implementation:

```
ask = Reader (\env -> env)
withReader m reader = Reader (\env -> runReader reader (m env))
```

6.3 The Writer Monad

The `State` monad is internally represented as `s -> (a, s)`. From that type, `Reader` encodes the passing of information, the `s -> ...` part. The other half, `(..., s)`, is called the `Writer` monad. In contrast to previous expositions, let us first write the `Monad` instance for this type and then discuss its possible usages.

First of all, a `Writer` is parametrized by the type of the elements in the first component of the tuple. This means that, similar to `State s a` and `Reader r a`, we actually build computations of the form `Writer w a`. In the previous sections, we described a high-level view before introducing the version with `newtype`. In this case, we start directly with the executable code:

```
newtype Writer w a = Writer { runWriter :: (w, a) }
```

If you lose track of what is going on, you can always ignore the wrapping and unwrapping — that is, forget about the `Writer` used both as a constructor and in matching, and focus only on the tuples. Note that we also reverse the position of `w` when compared with `State`, but this is irrelevant.

In order to implement `return`, we need to generate a value of type `(w, a)` from the argument `a`. Filling in the second component is easy:

```
instance Monad (Writer w) where
  return x = Writer (???, x)
```

But how do we generate a value for the first component? The difficult part is that the implementation has to work *for any type* you choose for `w`. But neither Haskell nor Scala provide such a feature — in fact, it would conflict with the concept of

parametric polymorphism.* The solution is to restrict the range of possible types for `w` to those that come equipped with a default value we can use. Fortunately, we have already discussed one such structure in Section 5.2, namely monoids:

```
instance Monoid w => Monad (Writer w) where
  return x = Writer (mempty, x)
```

The next operation we are going to consider is `fmap`, coming from `Functor`. Given the type, it is only possible to provide one implementation:

```
instance Functor (Writer w) where
  fmap :: (a -> b) -> Writer w a -> Writer w b
       -- (a -> b) ->     (w, a) ->      (w, b)
  fmap f (Writer (w, x)) = Writer (w, f x)
```

If we were to introduce the `Monoid` constraint here, we would be able to combine `w` with `mempty` via `mappend`. But remember that such an operation leaves the value untouched, so at the end of the day, we get a definition equivalent to the one above.

In addition to these two, we also have to implement either `(>>=)` or `join` to make `Writer` a monad. If we instantiate the type of `join` to our specific monad, we expect a tuple nested within a tuple, and we need to remove this nesting:

```
join (Writer (w1, Writer (w2, x)) = Writer (???, x)
```

Once again, we only have a single choice for the second component of the tuple. For the first component, we could use `w1` or `w2`. But this choice feels completely arbitrary. Instead, we can take advantage of the fact that the type of this first component is a monoid and *combine* the values, instead:

```
join (Writer (w1, Writer (w2, x)) = Writer (w1 `mappend` w2, x)
```

The reason we bring `join` to the table first instead of `(>>=)` is that for the latter, this choice is a bit less obvious. Let us look at those parts that are fixed by the types:

```
Writer (w1, x) >>= f = let Writer (w2, y) = f x in Writer (???, y)
```

By looking at the bind operation for `State`, we might be tempted to write `w2` in place of the question marks above. Such a definition is accepted by the compiler, but then the definitions for `(>>=)` and `join` would be inconsistent with each other — after all, we are forgetting the information in `w1`. The real definition for `Writer` combines `w1` and `w2` in a similar fashion to `join`:

```
Writer (w1, x) >>= f = let Writer (w2, y) = f x
                       in Writer (w1 `mappend` w2, y)
```

*This is not strictly correct: you can always throw an exception in Scala, use `error` or `undefined` in Haskell, or write an infinite loop in either. But none of those implementations is satisfactory.

As a final note, every time we use `mappend` above, we could have chosen to flip the arguments, writing `w2 `mappend` w1` instead. The choice we make in the `Writer` monad is arbitrary, although it is somehow justified by the sequential reading of the bind function.

As yet, we have not seen a way to influence the first component of the tuple that makes up the `Writer` monad. The `tell` operation breaks this barrier:

```
tell :: w -> Writer w ()
tell w = Writer (w, ())
```

The value given as an argument is combined with those produced in the other parts of the `Writer` action using the corresponding monoidal `mappend`. For example, consider the following piece of code that `tell`s two different numbers under the addition monoid:

```
example :: Writer (Sum Int) String
example = do tell (Sum 3)
             tell (Sum 4)
             return "seven"
```

In order to get such a computation running, we do not need to provide any further information. However, following the conventions in `transformers`, we provide a `runWriter` function to obtain the final value. In our implementation, `runWriter` is just the accessor for the `newtype`:

```
runWriter :: Writer w a -> (w, a)
```

Evaluating `runWriter example` above produces the value `(Sum 7, "seven")`, as expected. The values 3 and 4 are combined using the monoid operation — addition, in this particular case.

Logging using `Writer`. The `Writer` monad enables us to build a second return value for a computation by aggregating small pieces. We cannot access the current value of that part of the computation at any point, though. These two requirements are met for the main use case of `Writer`: producing a message log during a computation.

For example, consider a simple evaluator of arithmetic expressions. For various reasons, we do not want to fail when a division by zero is present. Instead, we will default to a 0 value:

```
data Expr = Lit Float | Add Expr Expr | ... | Divide Expr Expr

eval :: Expr -> Float
eval (Lit n) = n
...
```

```
eval (Divide x y) = case (eval x, eval y) of
                      (_, 0) -> 0  -- default to 0!
                      (u, v) -> u / v
```

The problem is that we cannot be sure, merely by inspecting the return value of `eval`, whether this error case has been executed. The solution we propose is to wrap the entire computation in a `Writer` monad and produce a log message in those scenarios:

```
eval :: Expr -> Writer [String] Float
eval (Lit n) = return n
...
eval (Divide x y) = do x' <- eval x
                       y' <- eval y
                       case (x', y') of
                         (_, 0) -> do tell ["Divide by zero"]
                                      return 0
                         (u, v) -> return (u / v)
```

Be aware that `Writer` only provides very basic logging capabilities. If you need a framework for logging real applications, we recommend instead that you look at `fast-logger` or `monad-logger` in Haskell or one of the many logging frameworks for the JVM.

6.3.1 Other Primitive Operations

The core of the `Writer` monad is its `tell` operation. In addition, the implementations available for both Haskell and Scala contain other methods. The main reason for its availability is that the simple data type used to implement the monad — a tuple — gives us the freedom to write other utilities.

The `transformers` library, which implements most monads in Haskell, does not provide an operation to reflect on the *current* value of a `Writer`. On the other hand, it does allow us to run a sub-computation and then look at its result:

```
do tell (Sum 2)
   (_, x) <- listen $ do tell (Sum 3)
                         tell (Sum 4)
   -- At this point, x ≡ Sum 7
   -- and the aggregated value is Sum 9
```

Another possibility, also provided by `transformers`, is to modify the value aggregated from a sub-computation before combining it using the monoidal `mappend` operation. In order to do so, we have two options, depending on whether or not the function to apply also comes from the monadic computation:

```
pass   :: Writer w (a, w -> w)               -> Writer w a
censor ::             (w -> w) -> Writer w a -> Writer w a
```

The simplest version is `censor`, in which the function is provided as an argument. The more complex `pass` receives as its argument an entire `Writer` computation, which provides both a normal result value of type `a` and the function of type `w -> w` to apply before aggregation.

6.4 All at Once: the RWS Monad

Keeping a state while having some configuration lying around, and producing logging output while running, describes a great majority of applications. In Chapter 11, we describe a general procedure for combining several monads — monad transformers. But given the pervasiveness of that use case, many libraries — including `transformers` and Cats — provide a combined Reader-Writer-State (`RWS`) monad.

The interface of the `RWS` monad is just the aggregation of the interfaces of `Reader`, `Writer`, and `State`. In addition, it provides three functions to embed computations from each of the monads that make up the mix:

```
reader :: Monoid w => (r -> a)      -> RWS r w s a
writer ::                (a, w)     -> RWS r w s a
state  :: Monoid w => (s -> (a, s)) -> RWS r w s a
```

The only remarkable feature of `RWS` is that it takes not two, but four type parameters. The first three define the types of the elements from each monadic component, and the last one is the type contained, as usual, by the monad.

6.5 Bi-, Contra-, and Profunctors

The functor type class provides the ability to modify the value contained in a monad by specifying a function to do the change. We know that every monad is also a functor, which means that we can map over any of the types described in this chapter. The types of `fmap` used in the context of each of these monads are:

```
fmap :: (a -> b) -> State  s a -> State  s b
fmap :: (a -> b) -> Reader r a -> Reader r b
fmap :: (a -> b) -> Writer w a -> Writer w b
```

Functors always operate on the type appearing in the last position of the type constructor. But these monads have a much richer structure than the simple functorial one!

Take, for example, `Writer w` for a given type of `w`. In a similar fashion to `fmap`, we can modify the aggregated value:

```
mapWriter :: (v -> w) -> Writer v a -> Writer w a
mapWriter f (Writer (v, a)) = Writer (f v, a)
```

Given `fmap` and `mapWriter`, `Writer` can be thought of as a functor in *both* type arguments. `Bifunctor` is an extension of `Functor` composed of those types that support mapping in both positions:

```
class Bifunctor f where
  first  :: (v -> w) -> f v a -> f w a
  second :: (a -> b) -> f v a -> f v b
  -- Alternatively, map over both at the same time
  bimap :: (v -> w) -> (a -> b) -> f v a -> f w b
```

The situation with `Reader` is similar, except for one small detail. Let us look closely at the signature of `withReader`:

```
withReader :: (r -> s) -> Reader s a -> Reader r a
```

The key part is that the map over `Reader` is inverted with respect to the given function. You give `withReader` a function that takes an `r` and produces an `s`, and then you can transform a `Reader` with environment `s` into a `Reader` with environment `r`. In this situation, we say that `Reader` acts as a *contravariant* functor in its first type argument. The converse situation of "normal" functors is referred to as *covariance*.

Variance in Scala. The words *covariance* and *contravariance* are used in Scala in a way that is related to the discussion above. The difference is that here we are talking about what happens to the type arguments when a *function* `a -> b` is applied, whereas in the usual Scala jargon we refer to the *inheritance* relation between types. Both ideas are related, though: if we think of converting from one type to another as applying a coercion function, the function-related notions become just the inheritance-related ones.

Another example of a contravariant functor is a predicate, that is, a function that returns a Boolean value:

```
newtype Predicate a = Predicate { runPredicate :: a -> Bool }
```

Let us think for a moment: if we have a function `a -> b`, what can we say about predicates over those types? Well, if we have a predicate for type `b`, we can generate a predicate for type `a` by first turning the `a` type into a `b` type and then applying the Boolean function:

```
through :: (a -> b) -> Predicate b -> Predicate a
through f (Predicate p) = Predicate (p . f)
```

Scala programmers have a `Contravariant` trait available in both Scalaz and Cats. The type class for Haskell is available in the `contravariant` package:

```
class Contravariant f where
  contramap :: (a -> b) -> f b -> f a

instance Contravariant Predicate where
  contramap = through
```

We discussed at the beginning of this section that `Writer` acts as a functor in both type arguments, so we call it a bifunctor. In contrast, `Reader` acts as a functor in the second type argument but as a *contravariant* functor in the first. `Reader` is thus not a bifunctor — we instead say it is a *profunctor*:

```
class Profunctor f where
  lmap   :: (v -> w) -> f w a -> f v a
  rmap   :: (a -> b) -> f v a -> f v b
  -- Alternatively, map over both at the same time
  dimap :: (v -> w) -> (a -> b) -> f w a -> f v b
```

Exercise 6.3. Consider the following type of functions that *return* a given type `r`:

```
newtype Returns r a = R (a -> r)
```

This `Returns` type is an example of a contravariant functor. Write the corresponding instance. Hint: the code is similar to that of `Predicate` above.

The nomenclature introduced in this chapter — bifunctors, contravariant functors, profunctors — is becoming increasingly popular for speaking about types with interesting relations with respect to changes of type parameters. When speaking about functors and monads, we tend to focus too much on what happens to the *last* type argument, because that is the one those structures play with. But we should not forget that such a focus is a side effect of the power of the languages we use to describe those structures. If we look from a more abstract perspective, functoriality appears in many more places.

7 Failure and Logic

Lists and optionals were some of our original examples when we introduced the concept of a monad. But in that moment, we only scratched the surface of what they give us. In this chapter, we deepen our knowledge of those types, in particular with respect to computations that may fail. We will see that the ideas of multiple return types — from lists — and possibly-absent values — from optionals — can be united under a single umbrella: `MonadPlus`.

7.1 Failure with Fallback

Up to now, we have been using `Maybe` and `Option` in a sequential manner. That is, we execute a bunch of computations one after the other, and if any of them fail, we stop at that point and state that we have nothing to return, by means of `Nothing` or None:

```
validatePerson name age
  = do name' <- validateName name
       age'  <- validateAge  age
       return (Person name' age')
```

Sometimes we have a plan B, another computation that we want to try if plan A fails. Consider a more complex name validation that takes into account the different structures of names in English, in which first names precede last names; in Spanish, in which a first name may be followed by a couple of last names; and in Dutch, where in between the first and last names, you may find a *tussenvoegsel*, such as "van." In order to take into account all possible cases, you would like to try all of them and return the first one that succeeds. It is time for a new combinator:

```
(<|>) :: Maybe a -> Maybe a -> Maybe a
Just x  <|> _     = Just x
Nothing <|> other = other
```

Now, we can describe the more complex validation procedure:

```
validateName :: String -> Maybe Name
validateName s = validateNameEnglish s
                 <|> validateNameSpanish s
                 <|> validateNameDutch s
```

This shows that a type like `Maybe`/`Option` has two, symmetrical modes of use:

- Success is represented by `Just` or `Some`. The `(>>=)` combinator continues the computation only if the previous step was successful.
- Failure is represented by `Nothing` or `None`. The `(<|>)` combinator tries another computation if the previous step failed. We also say that `(<|>)` recovers from failure.

The monadic structure of optional values gives us a nice interface, but it is quite uninformative when the computation fails — `Nothing` does not encode a reason for the failure. The `Either` data type remedies that problem by adding error information to the failure case:

```
data Either e r = Left e | Right r
```

> **Exercise 7.1.** Write the `Functor` and `Monad` instances for `Either e`. By convention, a successful return value is encoded by `Right`.

In the Scala world, Cats maintains the same nomenclature as Haskell. Scalaz deviates from this convention and instead uses mnemonic names to ease remembering which "side" of the failure/success case we are in:

```
sealed abstract class \/[A, B]  //
final case class -\/[A](a: A) extends (A \/ Nothing)
final case class \/-[B](b: B) extends (Nothing \/ B)
```

In the Haskell world — in particular, in the `transformers` library — the type `Either` is also known as `Except`.

The commonalities between these two monads are abstracted over by two different type classes (or traits) called `Alternative` and `MonadPlus`. The main difference between them lies in their prerequisites: an `Alternative` is an `Applicative` that supports some additional operations, whereas a `MonadPlus` is a subclass or subtrait of `Monad`:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

In fact, any `MonadPlus` instance must also be an instance of `Alternative`, although this requirement is not encoded in the definitions. In the Scala world, Cats does not include a specific `MonadPlus` type class, whereas Scalaz does, but in the latter it is just defined as the intersection of `Monad` and `Alternative`.

> **Exercise 7.2.** Write the `Alternative` instance for `Either e`. Hint: you need a default element to build a `Left` value. Requiring the error type to be a monoid is a common solution to this problem.

At this point, we would like you, dear reader, to stare for a while at the definitions of `Monoid` and `MonadPlus`. The similarity is striking:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

```
class MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Even though we have used the same name `m` for the argument to the class, `Monoid` is applicable to value types whereas `MonadPlus` is used with type constructors. But if you look at `m a` as a single type, you indeed have a monoid in your hands. In other words, the types of the methods in `MonadPlus` encode the idea that by using the type constructor `m`, you can get a monoid *regardless* of the element type to which it is applied!

The Scala implementations of `Alternative` are explicit, in this respect. In Cats, we find a `MonoidK` trait for type constructors that provides monoids for every type argument:

```
trait MonoidK[F[_]] {
  def empty[A]: F[A]
  def combineK[A](x: F[A], y: F[A]): F[A]
  def algebra[A]: Monoid[F[A]]
}
```

Now, you can recover `Alternative` by also requiring an `Applicative`:

```
trait Alternative[F[_]] extends Applicative[F] with MonoidK[F]
```

Scalaz uses a similar implementation but with different names. Instead of `MonoidK`, we have `PlusEmpty`, and `Alternative` is known as `ApplicativePlus`.

Alternative utilities. As is always the case with monads, much of the power of using abstractions such as `Alternative` and `MonadPlus` comes from the availability

of generic functions that work on any of their instances. In Haskell, these utilities can be found in either the `Control.Monad` or `Control.Applicative` modules. In Scala, they are part of their corresponding traits.

When dealing with error conditions, the following pattern is common:

```
if some condition
   then ... -- keep working
   else empty
```

This code could be part of the `validateAge` function, for example:

```
validateAge :: String -> Maybe Age
validateAge s = do n <- readMay s  -- turn the String into an Integer
                   if n >= 18
                      then Just n
                      else Nothing
```

The `guard` function encapsulates that same pattern:

```
guard :: Alternative m => Bool -> m ()
guard True  = pure ()
guard False = empty
```

This means that we can write `validateAge` in a more concise way:

```
validateAge s = do n <- readMay s  -- turn the String into an Integer
                   guard (n >= 18)
                   return n
```

In the next section, we are going to discuss the implications of making lists an `Alternative`. But for now, we can introduce two generalizations of common list operations that only require the structure provided by either `Alternative` or `MonadPlus`:

- The first function is `msum` or `asum` (depending on whether you require an additional `Monad` constraint), a generalization of `concat`. The type is a bit scary but the definition not so much:

  ```
  asum :: (Traversable t, Alternative m) => t (m a) -> m a
  asum = foldr (<|>) empty
  msum :: (Traversable t, MonadPlus   m) => t (m a) -> m a
  msum = foldr mplus mzero
  ```

  In other words, given a structure that contains elements wrapped in an `Alternative`, `asum` joins all of them via the `(<|>)` operation. When `t` is a list, this definition is equivalent to the following formulation:

  ```
  asum [m1, ..., mn] = m1 <|> ... <|> mn
  ```

If you specialize one step further, taking the `m` to be the list constructor, `asum` becomes the flattening function `concat :: [[a]] -> [a]`:

```
   asum [ [], ["Hello"], ["World"] ]
≡ ["Hello","World"]
```

If the list is comprised of `Maybe` values, the behavior of `(<|>)` makes `asum` return the first non-`Nothing` value in the list:

```
   asum [ Nothing, Just "Hello", Just "World" ]
≡ Just "Hello"
```

- Second, we can make `filter` work over any `MonadPlus`:

```
mfilter :: (MonadPlus m) => (a -> Bool) -> m a -> m a
mfilter p x = do x' <- x
                 if p x' then return x' else mzero
```

The inspiration for this comes from the fact that to filter a list, we can also go element by element, producing for each one either a singleton value or an empty list — akin to `return` and `mzero` — and then join all these elements together.

Note that `mfilter` makes use of *both* the monadic structure for the binding and the monoidal structure when using `mzero`. Thus, we really require `MonadPlus` in this case. `Alternative` is not strong enough. This is apparent if we look at the execution of `filter` over a `Maybe` value:

```
   mfilter (< 1) (Just 1)
≡ Just 1 >>= \a -> if (< 1) a then return a else mzero
≡ Just 1 >>= \a -> if (< 1) a then   Just a else Nothing
```

Note how `return` wraps the value into a `Just`, whereas `mzero` is defined as `Nothing`. This example also shows that `mfilter` may not "traverse" a data structure in the usual sense of the word.

In Haskell, the `errors` library provides versions of many common functions that replace exceptions or generalize a `Maybe` type into a generic `MonadPlus` instance. Some examples are the indexing operations for lists and the always demonized `head` function:

```
headZ :: MonadPlus m => [a] -> m a
atZ   :: MonadPlus m => [a] -> Int -> m a
```

If you have behaviors in your code that might fail, consider abstracting this failure into a generic `MonadPlus`. And without any doubt, adopt the `errors` library into your projects.

7.2 Logic Programming as a Monad

As hinted above, we can make lists an instance of `Alternative` and `MonadPlus`. Since list operations embody a type constructor with a monoidal shape, it is conceivable that when that monoidal structure is already known, it can be reused in their definitions:

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

```
instance Alternative [] where
  empty = []
  (<|>) = (++)
```

As a rough approximation, the absent value `Nothing` corresponds to an empty list, and a wrapped value `Just x` corresponds to a singleton list `[x]`. But we do not stop there, as the list monad also embodies the idea of a computation that returns several values — it is not limited to zero or one. By using this additional power, we can recreate *logic programming*[*] in our favorite language(s).

The core idea of logic programming is to express your program as a set of basic *facts*, plus a set of *rules* that allow us to derive more knowledge from those facts. When faced with a problem or programmatic question, a logic program starts a guided search through those facts and rules to find whether the given question holds or not — in other words, whether some logical formula can be derived from the given set of rules and facts. At certain points, this search procedure may signal a partial failure. In those cases, the program *backtrack*s in order to search another part of the solution space.

Consider the following example, which can be found in any textbook about Prolog, the most widely used logic programming language. Our facts are parent-child relationships among a set of people:

```
type Person = String

people :: [Person]
people = ["Alejandro", "Elena", "Quique", "John", "Mary", "Tom"]

pcRels :: [(Person, Person)]
pcRels = [("Alejandro", "Quique"), ("Elena", "Quique")
         ,("John", "Mary"), ("John", "Tom"), ("Mary", "Tim")]
```

Taking these facts into account, we can express the grandparent-grandchild relationship as a monadic computation over the facts:

```
gpgcRels :: [(Person, Person)]
gpgcRels = do (grandp,  parent) <- pcRels
              (parent', grandc) <- pcRels
              guard (parent == parent')
              return (grandp, grandc)
```

[*]There is also a mixed *functional-logic* paradigm, which includes languages such as Curry and Mercury.

How does this work? Remember that the list monad generates all the possible combinations of `grandp`, `parent`, `parent'`, and `grandc` from the set of parent-child relationships. Then, we only retain those combinations where `parent` and `parent'` coincide. The result is a set of tuples related by an intermediate person. If this expression reminds you of a join in SQL, you are absolutely right! Another important logic language is Datalog, used to query databases.

> **Exercise 7.3.** Define `siblingRels` to return one tuple for every two people who are siblings, that is, who share a common parent.

The bind construct of the list monad works like a conjunction: a value has to satisfy *all* the conditions in the `do` to make an appearance in the result list. Here are two additional definitions that find all the triples of numbers from a list such that two of them added together equal the third, or such that the sum of the squares of the first two equal the square of the third one (in that last case, we are referring to a Pythagorean triple):

```
sums, pyts :: [Integer] -> [(Integer, Integer, Integer)]
sums ns = do x <- ns
             y <- ns
             z <- ns
             guard (x + y == z)
             return (x,y,z)
pyts ns = do x <- ns
             y <- ns
             z <- ns
             guard (x * x + y * y == z * z)
             return (x,y,z)
```

However, if we want to return those triples that form triples of any of these sorts, we need something more than binds and guards. To produce the disjunction of two rules, we need `Alternative`:

```
triples ns = sums ns <|> pyts ns
```

In summary, you can model any logic programming problem — or just any search problem — by using the list monad and taking advantage of its monoidal structure.

Feeling brave after our new accomplishments, we decide to obtain not just the triples from a finite list but *any* triple among the natural numbers. In other words, we want to get the list `triples [1 ..]`, in Haskell notation. Alas, such a search fails to produce any triple after `(1,1,2)`. This is a consequence of the *unfairness* of the list monad.

We say that the list monad is unfair, because it keeps searching on a branch until it runs out of candidates. In this example, the execution proceeds by choosing

1 as the value for x in sums, then choosing 1 for y and also for z. That combination does not satisfy the condition 1 + 1 == 1, so it is discarded. Now, we backtrack in the innermost computation, leading to the value 2 for z. This combination is a triple, and it is thus reported. Then, we keep trying with larger numbers for z. Alas, we never choose a new value for x or y, since the list of numbers [1 ..] is infinite.

The solution is to use a more refined monad for search that includes *fair* combinators in addition to unfair ones. The package logict provides a Logic monad with those abilities. In order to understand how to use Logic, it is better to desugar the do notation into explicit binds:

```
sums ns = ns >>= \x -> ns >>= \y -> ns >>= \z ->
          guard (x + y == z) >> return (x,y,z)
```

However, there is now a mismatch, since ns is of type [Integer], whereas the bind requires a Logic computation. To perform the conversion, we first lift each Integer value into Logic Integer using return, and then we combine the whole list using the asum operation defined above. This trick works in general:

```
list :: [a] -> Logic a
list xs = asum (map return xs)

sums :: [Integer] -> Logic (Integer, Integer, Integer)
sums ns = list ns >>= \x -> list ns >>= \y -> list ns >>= \z ->
          guard (x + y == z) >> return (x,y,z)
```

However, this is not enough to make the computation produce more than one number. We need to make the enumeration of possibilities fair. To do so, we extract that part into its own function and replace the unfair bind operation (>>=) with its fair version (>>-). The final version reads:

```
fairTriples :: [Integer] -> Logic (Integer, Integer, Integer)
fairTriples ns = list ns >>- \x ->
                 list ns >>- \y ->
                 list ns >>- \z ->
                 return (x,y,z)

sums :: [Integer] -> Logic (Integer, Integer, Integer)
sums ns = fairTriples >>= \(x,y,z) ->
          guard (x + y == z) >> return (x,y,z)
```

And that's it! The search is now fair, so every combination of (x,y,z) is eventually tested. Note, however, that it usually takes some trial and error to figure out how to split the generation and checking correctly and when the fair bind (>>-) is needed to produce all the solutions we require.

Exercise 7.4. Rewrite the function `pyts`, which returns Pythagorean triples, using the `Logic` monad. Hint: you can reuse the `fairTriples` auxiliary function.

The `Logic` monad provides different ways to observe the results of a search, depending on whether we need just one solution, a fixed number of them, or we prefer to obtain all of them as a (lazy) list:

```
observe     ::         Logic a ->  a
observeMany :: Int -> Logic a -> [a]
observeAll  ::         Logic a -> [a]
```

The same unfairness problem also appears with disjunction. Since `sums` produces an infinite stream of triples, we never see any Pythagorean triple coming from `triples`. The solution is, again, changing to a fair disjunction operation:

```
triples ns = sums ns `interleave` pyts ns
```

Fair combinators such as `(>>-)` and `interleave` deactivate the traps that arise when combining search problems expressed in the list monad with an infinite solution space. In those cases, you do not want to look for a solution in a single branch exhaustively — since you might never finish searching it — but rather interleave the search among several branches.

In this section, we have only scratched the surface of the power of logic programming, in general, and the list and `Logic` monads, in particular, to describe and solve search problems. The interested reader is referred to *Adventures in Three Monads* [Yang, 2010] for many more examples and to *Backtracking, Interleaving, and Terminating Monad Transformers* [Kiselyov, Shan, Friedman, and Sabry, 2005] for a deeper exploration of the `Logic` monad, including a description of combinators for conditions and negations.

Another great source of knowledge about logic programming is the book, *The Reasoned Schemer* [Friedman, Byrd, Kiselyov, and Hemann, 2018], which describes miniKanren, a domain-specific language for logic programming, implementations of which exist in many languages. The website, `http://minikanren.org/`, maintains an index of the implementations. At the time of writing, there are ten listed for Haskell and three for Scala, with different levels of maturity.

Comprehensions and guards. In our description of comprehensions in Section 2.1.2, we omitted one of their most important ingredients: guards. A guard in a list comprehension filters out a subset of the values iterated over. For example, we might only be interested in those pairs of numbers that sum to a prime:

```
[(x, y) | x <- ns, y <- ns, prime (x + y)]
```

Following the Haskell Report literally, this expression is translated to:

```
concatMap (\x ->
  concatMap (\y ->
    if prime (x + y) then [(x, y)] else []) ns) ns
```

But, as we mentioned in Section 2.1.2, comprehensions can be generalized to work for any monad. In particular, we can replace the list operations with their generic versions — `concatMap` with `(>>=)`, `[x]` with `pure`, and `[]` with `mzero`:

```
ns >>= \x -> ns >>= \y -> if prime (x + y) then pure (x, y) else mzero
```

But wait! This last part is just the composition of `guard` with `pure`:

```
ns >>= \x -> ns >>= \y -> guard (prime (x + y)) >> return (x, y)
```

In fact, the generic translation from monad comprehensions to monad combinators uses `guard` in the translation of a conditional guard, like `prime (x + y)` in the code above.

In Scala, the situation is different. Each guard generates a call to a method akin to `mfilter`:

```
def filter(condition: A => Boolean): M[A]
```

```
for {                              for {
  x <- generator                     x <- generator.filter {
  if guard                                 x => guard
                          ⟹               }
  ???  // something else            ???  // something else
}                                  }
```

However, this way of translating comprehensions is not that different from Haskell's if we look at an alternative definition of `mfilter` in terms of `guard`:

```
mfilter :: (MonadPlus m) => (a -> Bool) -> m a -> m a
mfilter p m = do x <- m
                 guard (p x)
                 return x
```

This is exactly the same pattern — `guard` and then `return` — we see above.

7.3 Catching Errors

`Alternative` and `MonadPlus` embody a style of programming in which errors may occur, but we are never interested in inspecting or reasoning about them, only trying something else via `(<|>)`. But in many scenarios, we definitely need extra

functionality for inspecting errors, which we refer to generically as *catching* or *handling*, given the similarities to imperative exception mechanisms.

Several of the monads discussed in this chapter provide an interface for that purpose, in addition to their `MonadPlus` definitions. In this case, Haskell's `mtl` library, Scalaz, and Cats agree on a name for it — `MonadError` — but, unfortunately, not on the names of the operations:

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

```
trait MonadError[F[_], E] extends Monad[F] {
  def raiseError[A](e: E): F[A]
  def handleError[A](fa: F[A])(f: E => F[A]): F[A]
}
```

The first thing to notice is that `MonadError` takes one type parameter in addition to the monad itself.* This parameter defines the type of *error* we are able to throw and catch. For example, if we have `Either Problem Solution`, that type is `Problem`. But in some cases, the type is fixed. For example, we can see `Maybe` as a `MonadError` for which the error type is completely uninformative, either `()` or `Unit`, depending on the language.

The two methods of the class or trait provide the two parts of error-oriented programming: generating an error via `throwError` and doing something in response to one via `catchError`. Suppose we enhance our `validateName` function with possible reasons for failure:

```
data NameReason = NameTooLong | UnknownCharacters
```

Then, we can provide fallbacks for each specific case:

```
vn :: MonadError NameReason m => String -> m Bool
vn name = validateName name `catchError` (\e ->
            case e of
              NameTooLong       -> vn (cutName name)
              UnknownCharacters -> vn (normalize name)
            )
```

The behavior described above is something we *cannot* get using only `(<|>)`, since that operator provides no information about any previous computation. Note that in the implementation above, when we find an error, we call `vn` again — instead of `validateName` on its own — to ensure that if the `name` is both too long and has unknown characters, both errors are handled correctly.

*The `m -> e` part in the Haskell declaration is called a *functional dependency*. Roughly, it declares that the type of the monad determines the error type associated with it.

Exercise 7.5. Implement the `MonadError` class (or trait) for the types `Either` and `Maybe` (or `Option`).

Before we move on to the next chapter, let's return to the idea of bifunctors for a moment. The type `Either` is yet another example of a bifunctor, since we can map over both type arguments:

```
instance Bifunctor Either where
  first :: (e -> f) -> Either e a -> Either f a
  first f (Left  l) = Left  (f l)
  first _ (Right r) = Right r
-- instance Bifunctor Either where
  second :: (a -> b) -> Either e a -> Either e b
  second _ (Left  l) = Left  l
  second f (Right r) = Right (f r)
```

But there is more to it. Let's put, side to side, the type signatures of the monadic operations and those of `MonadError`, with the type of `catchError` slightly generalized to allow for the changing of the error type in the handler:

```
return :: a -> Either e a
(>>=)  :: Either e a
       -> (a -> Either e b)
       -> Either e b
```

```
throwError :: e -> Either e a
catchError :: Either e a
           -> (e -> Either f a)
           -> Either f a
```

It is indeed the same structure! `Either` is not only a bifunctor, but a *bimonad*, a type that provides both binding and returning for each of its two type parameters.

8

Monads for Mutability

What usually brings people into the realm of monads is the need for side-effectful computations as opposed to pure evaluation. In Haskell, every time you need to read a file or communicate through a network, you are forced to work in the `IO` monad, even though you do not really need to understand the concept deeply to start being productive.

At first sight, monads are very much about sequence and have a strong imperative flavor. By this point, dear reader, you have seen many other containers and contexts that are also monads. In this chapter, we go back to square one and look at those monads that provide one of the main features of imperative programming: mutable variables.

In principle, most of the discussion in this chapter is relevant only to pure languages and not so much to those that allow unrestricted side effects, like Scala. However, in recent years we have seen an uprising of pure solutions in that space, such as Scalaz ZIO.

8.1 Mutable References

The `State` monad allows us to hide a piece of data, which can be transformed during the computation but that we never have to thread explicitly. This monad simulates a mutable variable conceptually *without* actually mutating anything. One of its shortcomings, though, is that you can only keep *one* piece of state. If your algorithm requires keeping and updating several variables, using the `State` monad forces you to put everything into a single value — a tuple or an *ad-hoc* data type — and manually query or modify the corresponding field:

```
data AlgState = AlgState { counter :: Int
                         , visited :: [Node] }
```

```
travel :: [Node] -> State AlgState ()
travel (n:ns) = do ...
                   -- query and update the counter
                   c <- gets counter
                   modify (\s -> s { counter = c + 1 })
                   ...
                   -- push an element into a list of nodes
                   modify (\s -> s { visited = n:visited s })
                   ...
```

The `ST` monad,* found in the `Control.Monad.ST` module, has an interface much closer to that of conventional imperative languages. You can work with *references* — another name for mutable variables — by using the functions in `Data.STRef`. For example, to create a new reference you use:

```
newSTRef :: a -> ST s (STRef s a)
```

As you can see, we cannot create an uninitialized variable. We always require an initial value. The result is a `STRef` holding that single value. Both the `ST` monad and the `STRef` reference are marked with an additional type parameter that indicates the thread in which the computation takes place. As we will see later, this parameter is the key to guaranteeing purity in this scenario.

Along with creating one, we have the usual operations for reading, writing, and modifying a reference. All of them take as a first parameter a `STRef` value and do their work in the `ST` monad:

```
readSTRef   :: STRef s a             -> ST s a
writeSTRef  :: STRef s a -> a        -> ST s ()
modifySTRef :: STRef s a -> (a -> a) -> ST s ()
```

Running a `ST` computation. Up to this point, we have learned how to describe computations that use multiple variables:

```
weirdSum = do x <- newSTRef 1
              y <- newSTRef 1
              modifySTRef y (+1)
              (+) <$> readSTRef x <*> readSTRef y
```

For illustration, here is how one would write this problem in an imperative fashion:

```
int weirdSum() {
  int x = 1, y = 1;
  y += 1;
  return (x + y);  // reads are implicit
}
```

*Believe it or not, the acronym comes from *state threads*.

But describing such a computation is useless if we cannot execute it. We need a function similar to `runState` but for `ST` computations. You might very well have guessed the name: `runST`. The important property of that function is that it returns a pure value. If we execute `runST weirdSum`, the result is a simple, pure, and pristine `Int`:

```
> :type weirdSum
weirdSum :: ST Int
> :type runST weirdSum
runST weirdSum :: Int
```

Another point of view is that `runST` *delimit*s a piece of code in which mutability is allowed. However, it never allows this mutability to be apparent from the outside. Nevertheless, purity and mutable variables seem at odds with each other. We describe how we can reconcile these worlds in the next section.

Vectors. In most imperative programming languages, you may not only declare one variable at a time but also a sequence of them. We say that such a variable refers to an *array* or *vector*. In Haskell, the same interface is provided by the type `MVector` from the `vector` package. To create a new `MVector`, you can choose whether or not to initialize it to a common value:

```
new       :: Int      -> ST s (MVector s a)
replicate :: Int -> a -> ST s (MVector s a)
```

Reading and modifying the value in the vector is always done using a 0-based index. Apart from that additional argument, the interface is similar to that of `STRef`, which in turn copies the interface provided by the original `State` monad:

```
read   :: MVector s a             -> Int -> ST s a
write  :: MVector s a -> a        -> Int -> ST s ()
modify :: MVector s a -> (a -> a) -> Int -> ST s ()
```

All these operations perform bounds checks and will raise an exception for any index out of bounds. For maximum efficiency, you can disable those checks by using the `unsafe` family of operations — `unsafeNew`, `unsafeRead`, and so on.

8.1.1 Keeping References Pure

Most people think that there is something suspicious about `ST` the first time they encounter it. After all, we are injecting an imperative block into our functional code — the biggest sin — and getting away with it by calling `runST` at the end. Why does this ability not compromise the guarantees of pure, functional code?

The answer is that variables inside an `ST` block — a single piece of code delimited by a call to `runST` — have a limited scope. At the end of the computation, all mutable

variables are destroyed. And even more importantly, we cannot share variables between different ST blocks, which would amount to keeping a mutable reference lying around for an indefinite period of time. This check is not performed at runtime but statically guaranteed at compile time. Consider the following code:

```
notAllowed = let var = runST (newSTRef 0)  -- Block 1
             in runST (readSTRef var)      -- Block 2
```

If we try to compile it, GHC shouts loudly at us:

```
Couldn't match type ‘a’ with ‘STRef s Integer’
  because type variable ‘s’ would escape its scope
This (rigid, skolem) type variable is bound by
  a type expected by the context:
    forall s. ST s a
```

The magic is in the very special type of runST:

```
runST :: (forall s. ST s v) -> v
```

In this type, we have two variables, v and s. The variable v, which represents the output type of the computation, acts like any other type variable in Haskell. That is, it indicates that runST can produce any output type we want, and we are free to *choose* with which type we want to instantiate v. The case of s is different. Since there is a forall quantifier in front of it, the compiler requires that the computation we provide work for any value of s that runST wants. In order to check that this property is true, GHC creates a fresh variable, guaranteed to be different from every other variable in the system, every time we call runST — this is what we refer to as a rigid, or Skolem, variable.

The right way to look at s is to regard it as a token that represents a specific ST computation. Because of the way type checking works, which we have just described, the compiler assigns a different token to each block in the code. Now, remember the type of newSTRef:

```
newSTRef :: a -> ST s (STRef s a)
```

The reference is annotated with the token of the block it came from. When we want to use readSTRef from a different block, we cannot do this, since the token from the second block will not match the token of the reference. Saved by the types!

8.2 Interfacing with the Real World

The IO monad is as powerful as a spaceship but also as powerful as Pandora's box. In Haskell, the IO monad grants access to external libraries, to the file system, to the network, and to an imperative model of execution. We need it — we want to

communicate with other systems or with the user, don't we? — but we want to stay far away from it as much as possible.

It is impossible to describe all the possibilities inherent in Haskell's `IO` monad. For the sake of simplicity, we are going to restrict ourselves to simple actions. The following actions allow us to show and obtain information through the console:

```
putStr   :: String -> IO ()
putStrLn :: String -> IO ()  -- end with a newline
getChar  ::           IO Char
getLine  ::           IO String
```

Using these primitives, we can write a simple program that asks for a name and uses it to print a greeting:

```
greet :: IO ()
greet = do putStr "Enter your name: "
           name <- getLine
           putStrLn ("Hello, " ++ name ++ "!")
```

Another functionality that lives in the `IO` monad is randomness. Each data type that supports a notion of a random value has to implement the `Random` type class. This means that the following two operations are available for that type:

```
randomIO  :: Random a =>           IO a
randomRIO :: Random a => (a, a) -> IO a  -- within bounds
```

Purity. Haskellers often emphasize that their language is *purely* functional. A pure language is one that embodies the idea that "equals can be substituted for equals." This idea is also important in mathematics. For example, we know that $1 + 2 = 3$. This means that if we have an expression like $(1 + 2)^2$, we can just turn it into 3^2.

Almost everything in Haskell works in this way. For example, the definition of the `length` of a list tells us that:

```
length [] = 0
```

If we take the expression `(length []) * 2`, we can safely rewrite it as `0 * 2`. This property also holds for local bindings, so we can turn `let x = 0 in x * x` into `0 * 0`.

Imagine now that random generation would have the signature `random :: Random a => a`. The rule of "equals can be substituted for equals" tells us that `let r = random in r == r` could be rewritten to `random == random`. But those two expressions have completely different meanings. In the first one we produce a random value once, which is checked with itself for equality, and thus always returns `True`. In the second case, two random values are generated, so the outcome is equally random.

Haskell's solution is to mark those values for which purity does not hold with `IO`. Since `randomRIO` generates two values of type `IO a`, we cannot directly apply the equality operator to them, as no instance for `IO a` exists. In addition, the compiler knows that whereas it is safe to inline or manipulate any other expression in a program, it should never touch an `IO` action.

Description versus execution. `IO` values are treated like any other value in Haskell: they can be used as arguments to functions, put in a list, and so on. This raises the question of when the results of such actions are visible to the outside world.

Take the following small expression:

```
map putStrLn ["Alejandro", "John"]
```

If you try to execute it, you will see that nothing is printed to the screen. What we have created is a *description* of a list of actions that write to the screen. You can see this in the type assigned to the expression, `[IO ()]`. The fact that `IO` actions are not executed on the spot goes very well with the lazy nature of Haskell and allows us to write our own imperative control structures:

```
while :: IO Bool -> IO () -> IO ()
while cond action = do c <- cond
                       if c then action >> while cond action
                            else return ()
```

Such code would be useless if the actions given as arguments were executed immediately.

There are only two ways in which we can *execute* the description embodied in an `IO` action. One is entering the expression at the GHC interpreter prompt. The other is putting it in the call trace that starts in the `main` function of an executable. In any case, only those expressions that have `IO` as their outer constructor are executed. This is the reason why the previous expression would not print anything, even in `main`. To get the work done, we need to use `sequence_` or `mapM_`:

```
sequence_ (map putStrLn ["Alejandro", "John"])
-- or equivalently
         mapM_ putStrLn ["Alejandro", "John"]
```

This distinction between description and execution is at the core of the techniques explained in Chapter 13 for creating your own, fine-grained monads. But even for a monad with so many possible side effects like `IO`, it is useful for keeping the pure and impure parts of your code separated. For that reason, people even use `IO` in languages in which side effects are available everywhere, such as Scala. In particular, the Scalaz ZIO library defines an `IOApp` class as the entry point of a side-effectful computation, which is represented as an `IO` action. Note that in ZIO, the `IO` type takes two arguments — the first one represents the range of exceptions that may be thrown during the execution of the side effects:

```
trait IOApp extends RTS {
  def run(args: List[String]): IO[Void, ExitStatus]
  final def main(args0: Array[String]): Unit
    = unsafePerformIO(run(args0.toList)) ...
}
```

Wait a minute! You are now looking at the devil itself: `unsafePerformIO`. The type of that function is `IO a -> a`. In other words, it allows us to break the barrier between purity and impurity. You should know that this function exists only so you never use it. The situations in which you would need it are extremely rare and mostly involve interfacing with external systems. If the moment ever comes, you will know.

8.2.1 Exceptions

Dealing with external systems implies dealing with many more error conditions than when we use pure code. A simple pure operation like division only has one corner case, division by zero. In those cases, it makes sense to wrap the operation in the `Maybe` or `Option` monad. Opening a file, however, may lead to many more problematic scenarios: the file may not exist, it is locked, the user may not have enough permissions to access it, the operation needs to go through the network and times out, etc. The designers of `IO` in Haskell decided against `Maybe` for this use case and instead introduced exceptions into the language.

There are two sides to working with exceptions. The first one is *raising* or *throwing* an exception. In that moment, the flow of the program is cut off, and control is given to the nearest *exception handler*. Declaring which part of the code is linked to each handler is the second part of dealing with exceptions.

In Haskell, the functions used to throw and handle exceptions are centralized in a single module, `Control.Exception`. Throwing an exception can be done with the following two functions:

```
throw   :: Exception e => e ->     a
throwIO :: Exception e => e -> IO a
```

The first thing to notice is that exceptions can also be thrown from pure code, as `throw` witnesses. In fact, the error messages one sees when `undefined` or `error` are called are really coming from exceptions raised within those functions.

Second, not every type can be used as an exception. It must be an instance of the `Exception` type class. For every domain in which a new kind of exception makes sense, you have to declare the corresponding data type:

```
data DatabaseException = ConnectionError | TableNotFound | ...
                       deriving (Show, Typeable)
```

The last line, which auto-derives instances, is mandatory. After doing so, we just declare `DatabaseException` as an exception:

```
instance Exception DatabaseException
```

Haskell's base library comes with a wide range of built-in exception types, such as `IOException` and `ArithException`.

The most common way to handle an exception is to use `catch`. The first argument is the operation you wish to perform and the second one a function that is called only if that operation throws an exception:

```
(print (amount / numberOfPeople))
`catch` (\(e :: ArithException) -> putStrLn "wrong number of people")
```

Note that the argument to the handler is annotated with the type `ArithException`. This is how we inform the compiler which type of exception we wish to handle. In this case, any `IOError` — or other kind of exception — is propagated to the next handler. If you really want to handle *any* exception, you can use `SomeException` as the type.

The library defines `handle` as `catch` with the order of arguments swapped. In some cases, it might be more understandable to declare the handlers before the main computation:

```
handle (\(e :: ArithException) -> putStrLn "wrong number of people") $
       print (amount / numberOfPeople)
```

If you want to handle not one single type of exception, but several of them, it is advisable to use `catches` instead of a sequence of `catch` calls. The reason is that by sequencing `catch` actions, you create several enclosing regions of handling, and it becomes difficult to guess what happens if one handler raises yet another exception. In contrast, `catches` creates one single region, in which each handler is tried in succession:

```
complicatedComputation
`catches` [ Handler (\(e :: ArithException) -> ...)
          , Handler (\(e :: DatabaseException) -> ...) ]
```

One unfortunate consequence of the introduction of exceptions is that error handling in pure code — usually done via `Maybe` or `Either` — becomes rather different from that in `IO` computations. The first approach to reconcile them comes from the `try` function:

```
try :: Exception e => IO a -> IO (Either e a)
```

In this case, any exception during the execution of the given computation is turned into the `Left` branch of an `Either`. A more powerful solution is embodied by the `exceptions` package, which defines a couple of type classes for those monads that may handle pure or impure exceptions:

```
class Monad m => MonadThrow m where
  throwM :: Exception e => e -> m a
class MonadThrow m => MonadCatch m where
  catch :: Exception e => m a -> (e -> m a) -> m a
```

In addition to `IO`, the type `Either SomeException` is also a member of that class. In most cases, defining your functions in terms of these classes is a net gain, as it allows you to use `IO` in production code, while using `Either` — which is easier to inspect — during testing.

There is another problem with exceptions in Haskell, this one due to the laziness of the language. Consider the following variation of the code that handles divide-by-zero errors:

```
ae :: Rational -> Rational -> IO Rational
ae n d =
  return (n / d)
  `catch` (\(e :: ArithException) -> do
              putStrLn ("wrong number of people " ++ show e)
              return 0)
```

If you run this code in the interpreter with zero as the divisor, you will not see the expected error message. Instead, an exception is still thrown:

```
> ae 5 0
*** Exception: Ratio has zero denominator
```

Briefly, the problem is that at the moment in which `catch` does its job, the result of `n / d` is not evaluated yet. As a consequence, there is no exception being raised at that point. Only when we try to print the result does evaluation take place, but then it is too late: the safeguard of `catch` is already gone. To fix this problem, we need to *force* the evaluation of `n / d`. We do so by replacing `return` with `evaluate`.

This was only a quick introduction to the vast realm of exceptions in Haskell. In particular, we have not spoken about two sources of headaches within the `IO` exception mechanism: asynchronous exceptions and exceptions between different threads. My personal recommendation for avoiding those headaches is to use the `safe-exceptions` or `unliftio`* package instead of the built-in `Control.Exception` module. Both packages provide more intuitive guarantees for the behavior of error reporting and handling, with a similar interface to the one provided by default. An even stronger recommendation is to use exceptions as little as possible and only deal with them when forced to do so — when working in the impure `IO` world, for example — instead preferring the pure `MonadError` interface from Section 7.3.

*We discuss the `unliftio` package in depth in Section 12.3.

8.2.2 Impure References

We introduced `ST` as a way to have mutable references in a block of code while maintaining our purity guarantees. But when we work in `IO`, we might need to put those guarantees aside. For example, if we are building a web server, some state might be shared between all clients, like the database connection pool. The solution is simple — use an `IORef`, the unsafe version of `STRef`:

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a             -> IO a
writeIORef  :: IORef a -> a        -> IO ()
modifyIORef :: IORef a -> (a -> a) -> IO ()
```

In contrast with `STRefs`, `IORefs` are not tagged with the specific region they belong to. As a consequence, you can freely share them among different computations.

When you are writing a web server, things are not that easy, though. Each client is usually assigned a different *thread* of execution, and all of the active clients execute concurrently. Creating a new thread in Haskell is simple, as you just need to call the `forkIO` primitive, which returns an identifier that can be used for canceling or otherwise manipulating the execution of a thread from within the program that created it:

```
forkIO :: IO () -> IO ThreadId
```

Alas, `IORef` does not offer any guarantees about concurrency. There is some level of atomicity available through `atomicWriteIORef` and `atomicModifyIORef`. But the right solution is to move to `MVars`, variables that can be either empty or full and whose semantics are safe for multi-threaded programs. We will not delve here into parallelism and concurrency in Haskell, but *Parallel and Concurrent Programming in Haskell* [Marlow, 2013] gives a beautiful account of all the details.

8.2.3 Transactional Variables

If your goal is to communicate information between threads of a single application, however, `MVars` are not your best option, either. The best guarantees are offered by *software transactional memory*, STM for short, which brings the idea of atomicity from databases into regular code.

In the Haskell implementation, each transaction is represented as a computation in the `STM` monad. Continuing with the theme of variables, transactional variables are called `TVars`. They provide the same set of operations as `IORefs` and `STRefs`:

```
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a             -> STM a
writeTVar  :: TVar a -> a        -> STM ()
modifyTVar :: TVar a -> (a -> a) -> STM ()
```

Their difference from normal variables is that in a certain block of code, you may access and modify several transactional variables. In the following piece of code, we use two of them, a counter acting as a source of identifiers and a list of names paired with those identifiers:

```
addName :: TVar Integer -> TVar [(Integer, String)]
        -> String -> STM ()
addName counter names name = do
  i <- readTVar counter
  modifyTVar names ((counter, name) :)
  writeTVar counter (i + 1)
```

All operations in the block are treated as an *atomic* block that executes over a consistent view of the data. You do not have to worry about the scenarios in which the variable `counter` is updated between the read and the final write — `counter` always increases after a successful completion of `addName` — or in which different threads lock `counter` and `names` independently and lead to a deadlock. The runtime system will ultimately guarantee the atomicity of the transaction.

> **Exercise 8.1.** Modify the code above to guarantee that no duplicates exist in the list of names. In other words, the code should only insert a name if there is no pair in `names` that already has it as its second component.

Merely writing a `STM` computation does not lead to any mutation. In order to execute the transaction, you need to call the `atomically` function:

```
atomically (addName "Alejandro")
```

Such an operation works in the `IO` monad. At this point, you might be wondering: if we are going to end up in `IO` anyway, why bother creating another monad? The reason is that by providing a smaller monad that supports a smaller set of operations, we can get better guarantees about our program. In fact, creating restricted subsets of `IO` is a common approach for keeping under control the number of side effects a certain computation may have.

8.2.4 The Truth Behind GHC's IO

Back to simple `STRefs` and `IORefs`, it is clear that both types share the same concepts and operations. They're so close that the `primitive` package is able to provide a common interface by means of the `PrimMonad` and a shared `MutVar` type:

```
newMutVar    :: PrimMonad m
             => a -> m (MutVar (PrimState m) a)
readMutVar   :: PrimMonad m
             => MutVar (PrimState m) a -> m a
```

```
writeMutVar  :: PrimMonad m
             => MutVar (PrimState m) a -> a -> m ()
modifyMutVar :: PrimMonad m
             => MutVar (PrimState m) a -> (a -> a) -> m ()
```

What is even more interesting, the `MutVar` type is not declared using any kind of type magic — it just takes as an argument the `PrimState` of the corresponding monad, defined as:

```
PrimState (ST s) = s
PrimState IO     = RealWorld
```

It looks here like `IO` is a state-transformer monad that transforms the *entire world*. That intuition is correct — if we ask the GHC interpreter for information about `IO`, this is what we get:

```
> :info IO
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

The first time I saw this type, I was mind-blown. `IO` is really a normal `State` monad that threads the "real world" as an argument. The only difference is its use of *unlifted* pairs of type `(# a, b #)` instead of regular tuples, since they provide better runtime characteristics for this use case. Of course, it is not the entire existence of the universe that is threaded but rather a token that ensures that `IO` operations are performed in the correct order.

In turn, `ST s` is nothing but a synonym for:

```
newtype ST s a = ST (State# s -> (# State# s, a #))
```

Given that the types share most of their structure, it is possible to move from `ST` — which offers more guarantees — to `IO` — which offers fewer – via a function named `stToIO`. This function allows us to describe computations using the monad that provides the most guarantees but use those same computations in `IO` if you need its power for other reasons.

In this chapter, we focused on mutability, but this is only one of the features provided by `IO`. Its other main use is accessing resources outside of the program itself, such as files or networks. As we will see in the following chapter, `IO` on its own does not provide enough guarantees for even the most common of such scenarios, so we need to learn next about more restricted monads.

9 Resource Management and Continuations

The monads we examined in the previous chapters are usually considered the basics that every monadic functional programmer should know. Before moving on to the issue of combining monads, however, we will first have a look at monads that are useful in a less well-known scenario: resource management.

By resource management, we refer to the handling of external resources that require the use of a certain protocol to be accessed. The prime example is given by files: you need to open a file prior to any operation and close it once the job is done. The problem is that those operations, or the work itself, may go wrong, but we always want to return to an acceptable state without leaking either memory or file handles.

As we will see, the `IO` monad from Chapter 8 provides enough building blocks to handle one resource correctly. But when more than one is involved, we need the help of dedicated resource management monads. In turn, the operations provided by these monads can be understood as particular cases of *continuations*.

9.1 The Bracket Idiom

Each resource in our system is assumed to undergo three states: (1) acquisition, (2) use, and (3) release. Since these resources are external to our pure world, problems — in the form of exceptions — may arise at any moment. We want to ensure, however, that if the acquisition phase finishes correctly, the release phase is always run, regardless of any exception raised during the use phase.

In the world of functional programming, we can represent each of these phases as a function. The acquisition phase produces a value of the resource type `r` — since we are assuming that all work happens in the `IO` monad, this is simply a computation of the type `IO r`. Both the use and release phase need to take that

resource as an argument to be able to use it. When we execute the whole operation with the resource, we are only interested in the value we obtain from phase (2) — the release phase should happen under the hood. Putting all of this together, a function that implements the resource protocol should have the following type:

```
   IO r -> (r -> IO a) -> (r -> IO ()) -> IO a
-- ↳ acq.  ↳ use          ↳ release
```

This is almost the signature of the `bracket` function in Haskell's `Control.Exception` module![*] That function takes care of setting up exception handlers in such a way that the guarantees of release are always satisfied:

```
bracket :: IO r -> (r -> IO b) -> (r -> IO a) -> IO a
        -- ↳ acq.  ↳ release      ↳ use
```

The differences between the two signatures above are for practical reasons. First, we can generalize the type of the release function to return any type, instead of insisting on `()`. As we saw in Chapter 4, we can always forget any result from a monadic computation by applying `void`. Second, putting the use part last allows for a certain style of coding:

```
bracket acquire release $ \r -> do
  ...
```

Given that acquisition and release take relatively little code, this puts the focus on the use part, which is where the logic of the program really lives.

C# and F# are languages with exceptions — similar to our situation when working in the `IO` monad — and they thus require a similar set-up to deal with resources. The pattern in those languages looks like this:

```
var r = acquire
try {
  use
} finally {
  if (r != null) release
}
```

The first versions of C# did not support functions as arguments, so instead of a mere function like `bracket`, they made this pattern available through a keyword:

```
using (var r = acquire) {
  use
}
```

[*] Although you should be using the one from either the `unliftio` or the `safe-exceptions` package instead, since they take care of both synchronous and asynchronous exceptions correctly.

In this case, the release operation is assumed to be a call to a method named `Dispose` in the resource itself. F#, on the other hand, supported higher-order functions from the beginning. There, `using` is just a function:

```
let using r f = try f(r) finally r.Dispose()
```

In addition to `using`, F# provides a more powerful version. At any point in which you can create a variable using `let`, you can replace that keyword with `use`. The compiler will then ensure that the release of the resource in that variable is executed before the current code block ends. The resulting code style is similar to the one enabled by Haskell's `bracket`:

```
use r = acquire
// use r and forget about release :)
```

The support for resource management that `bracket` provides is highly generic. Many other libraries give specific versions of that function targeted to a single use case. For example, in the module `System.IO`, we can find functions for opening, working with, and closing files:

```
withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r
withFile fp mode = bracket (openFile fp mode) hClose
```

A similar example can be found in the `temporary` package, in which acquiring a resource means creating a new temporary file or folder and releasing it entails removing that temporary element from the file system:

```
withTempFile :: FilePath -> String -> (FilePath -> Handle -> IO a)
             -> IO a
```

The `bracket` function is without a doubt one of the most important functions for dealing with external resources in Haskell code. Alas, as we will see in the next section, it falls short in some use cases.

9.2 Nicer Code with Continuations

Consider a piece of code that needs to work with more than one resource. For example, you want to process some input file and write the results back to an output file. As we discussed above, the right way to handle that scenario is to write something like this:

```
withFile inFile ReadMode $ \inHandle ->
  withFile outFile WriteMode $ \outHandle ->
    doWork inHandle outHandle
```

Back in the beginning of the book, we introduced monads specifically to combat this nested style for `Maybe` and stateful computations.

We can do the same trick with the `Managed` monad, available in the `managed` package. Instead of nested calls to `bracket`, with this package we can write a linear description of a computation. The only extra requirement is to put the call to the `withFile` operation — or, in general, to `bracket` or any of its variants — inside a call to `managed` to indicate that we are dealing with a resource. At the top level, the call to `runManaged` executes the computation. The previous code then becomes:

```
runManaged $ do
  inHandle  <- managed (withFile inFile ReadMode)
  outHandle <- managed (withFile outFile WriteMode)
  liftIO $ doWork inHandle outHandle
```

You might have noticed an extra call to `liftIO` in the code above. Computations that work with resources typically live in the `IO` monad. In contrast, the `do` block above lives in the `Managed` monad. How do we reconcile these two worlds? The answer is this function, `liftIO`:

```
liftIO :: IO a -> Managed a
```

With `liftIO`, we can treat any `IO` computation as an operation inside of `Managed`. In fact, this is part of a more general pattern that we describe in depth in Chapter 11.

In order to understand how `Managed` works, we need to introduce the general concept of a *continuation*. The core idea is that whenever we talk about a value of type `a`, we could also express it as a function of type:

```
(a -> r) -> r
```

The reason is that having the value is equivalent to having a way to apply any possible function you want over that value. We often say that the function `a -> r` in the signature above is a continuation that consumes a value of type `a`. The entire type above is usually represented as `Cont r a`.

> **Exercise 9.1.** Prove that `forall r. Cont r a` is isomorphic to `a` for every type `a`. In other words, implement functions going back and forth between each representation:
>
> ```
> toCont   :: a -> (forall r. Cont r a)
> fromCont :: (forall r. Cont r a) -> a
> ```

Continuations may look alien at first glance: why replace a normal value with a functional type? But in fact, you might well be used to this idea in the form of *callbacks*. When you use a function with a callback, you do not get to see the

result of that function. Instead, you provide a function that consumes the value — in other words, a continuation. Here is an example adapted from the Node.js documentation:

```
http.get('http://haskell.org', res => {
  const statusCode = res.statusCode;
  // keep on going
})
```

This code starts a new HTTP request, its result made available through the `res` argument. Note that `res => ...` is JavaScript notation for an anonymous function. In Haskell terms, `get` would be a function of type `String -> (Result -> IO ()) -> IO ()`. To call it, most people use the application operation `($)` instead of parentheses:

```
get "http://haskell.org" $ \res ->
  let statusCode = statusCode res
   in ...
```

Note that the type we have given to `get` is equivalent to `String -> Cont (IO ()) Result`.

Usually, APIs either decide not to use callbacks at all or use them for every operation. In the latter case, programs often end up with a deeply nested structure, since the next operation to perform is given as an argument to the previous one:

```
action1 $ \res1 ->
  action2 $ \res2 ->
    ...
```

In the JavaScript world, this is known as *callback hell*.* Luckily for us, `Cont r` is a monad, and we can write instead:

```
do res1 <- action1
   res2 <- action2
   ...
```

In order to understand why this is the case, we need to look at the implementation of that `Monad` instance and what the `return` and bind operations mean in this specific scenario. Note that this implementation is pseudo-Haskell, we omit the wrapping and unwrapping of the `newtype` for `Cont`, in order not to get lost in details.

*Sounding even more dramatic, Mozilla's documentation at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises` refers to this pattern as the *callback pyramid of doom*.

The `return` operation takes a value of type `a` and has to build a value of type `Cont r a` or, in other words, `(a -> r) -> r`. If we put the whole type together, we see that the only possibility is to apply the continuation `a -> r` to the value given to `return`:

```
return :: a -> (a -> r) -> r
return x = \k -> k x
-- using known combinators
return   = flip ($)
```

For `(>>=)`, the best approach is to expand the definition of `Cont r` in the type that function ought to have for the corresponding instance:

```
(>>=) :: Cont r a         -> (a -> Cont r b)       -> Cont r b
      :: ((a -> r) -> r) -> (a -> (b -> r) -> r) -> (b -> r) -> r
```

We know for sure that the result of applying the first two arguments is going to be a function that consumes the given continuation of type `b -> r`:

```
x >>= f = \k -> ...
```

In this function, there are only two possibilities for obtaining a final value of type `r`: to provide `x` with a continuation of type `a -> r` or to provide `f` with both a value `a` and a continuation `b -> r`. The second option is not really possible, though, since there is no way to get our hands on a value of type `a`. So our first refinement is to rewrite the function:

```
x >>= f = \k -> x (\a -> ...)
```

As the code above shows, we can now access a value of type `a` by means of an abstraction. Thus, we are ready to provide `f` with the values it requires:

```
x >>= f = \k -> x (\a -> f a k)
```

The intuitive interpretation of this code is that it takes a function that expects a callback, and a function that consumes this value but creates another callback, and creates a bigger box in which you only need to provide that second callback.

The monadic spirit of callback-based programming has permeated other languages, even those that do not formally support monads. If you were to write the JavaScript code shown above in more contemporary terms, you would use a `Promise` object, instead:

```
http.get('http://haskell.org')
    .then(res => ...)
```

The property to notice is that `then` generates yet another promise, so you get to a position where you can apply the `then` method again and again. You are thus

replacing nested callbacks with linear calls to `then`. Well, this method is nothing other than the bind operation of the `Cont r` monad!

Going back to our initial question, how is `Managed` related to `Cont`? We can obtain the type of `withFile` applied to both an undefined file and an undefined mode by asking the interpreter:

```
> let file = undefined
> let mode = undefined
> :type withFile file mode
withFile file mode :: (Handle -> IO a) -> IO a
```

The type we get is exactly a continuation from the type of resources, `Handle` in this case, into `IO a`. We can therefore take all the scaffolding we have just introduced for generic continuations and use it for those specifically targeted at resource management. The same idea holds for the general `bracket` function and the many variations defined for specific resource types.

Resource pools. Sometimes, it is advisable for several parts of your program to share a small amount of resources. One typical example is connecting to a database: since each database connection is expensive, you would naturally prefer to keep the number of open connections low during program execution. The package `resource-pool` provides an easy way to handle this situation.

In order to share resources, we first need to create a *pool* that keeps track of their various states. We do so via the `createPool` function, which needs the acquire and release functions, plus some additional configuration:

```
pool <- createPool
          (connectPostgreSQL "host=localhost, port=1234")
          close
          1    -- number of sub-pools, 1 is usually OK
          0.5  -- seconds until an unused resource is collected
          10   -- maximum number of connections
```

Instead of using `bracket`, we now use `withResource`. The first argument to that function is the pool itself, which we obtained using `createPool`, and the second is the function `r -> IO a`, which consumes that resource. Given this shape, we can use `managed` as explained earlier in this section to prevent deeply nested programs:

```
do connection <- managed (withResource pool)
   liftIO (query "select * from clients" ...)
```

9.3 Early Release

Resource management with `bracket` — or, equivalently, via `Managed` — guarantees that the acquired resources are released in every possible situation, even in the

```
zipFiles :: FilePath -> FilePath -> FilePath -> IO ()
zipFiles fin1 fin2 fout =
  withFile fin1 ReadMode $ \in1 ->
    withFile fin2 ReadMode $ \in2 ->
      withFile fout WriteMode $ \out ->
        go in1 in2 out
  where -- both files still have content
        go in1 in2 out = do
          eof1 <- hIsEOF in1
          if eof1 then end in2 out
             else do
               eof2 <- hIsEOF in2
               if eof2 then end in1 out
                  else do hGetChar in1 >>= hPutChar out
                          hGetChar in2 >>= hPutChar out
                          go in1 in2 out
        -- only one file still has contents
        end in_ out = do
          eof <- hIsEOF in_
          if eof then return ()
             else do hGetChar in_ >>= hPutChar out
                     end in_ out
```

Figure 9.1: `zipFiles` without early release

presence of exceptions. But it also imposes a severe restriction on when release takes place: only after the whole computation has finished. This means that sometimes we hold onto resources for too long, potentially forbidding other parts of the application from accessing a resource that may be scarce. In this section, we discuss monads that add the possibility of *early release*, that is, of freeing a resource before the whole computation ends.

As a simplified example of when early release is desirable, consider the program given in Figure 9.1, which zips a pair of files. The code has three sections: the `zipFiles` function itself just opens the files in the corresponding mode and then gives control to `go`. The `go` function reads a single character from each file until one of them runs of out contents. Then, the `end` function is called, and it just moves all the remaining content from the larger file into the output.

In that piece of code, we are keeping one of the files — the shortest one — open longer than necessary. Once we have finished reading all of its contents, we could close it, since we never touch it again. But this is not a problem that we can solve by changing how we nest the `withFile` calls, since which file is to be closed first is only known at execution time.

```haskell
zipFiles' :: FilePath -> FilePath -> FilePath -> IO ()
zipFiles' fin1 fin2 fout = runResourceT $ do
    (rin1, in1) <- allocate (openFile fin1 ReadMode)  hClose
    (rin2, in2) <- allocate (openFile fin2 ReadMode)  hClose
    (rout, out) <- allocate (openFile fout WriteMode) hClose
    liftIO (go in1 rin1 in2 rin2 out rout)
  where -- both files still have content
    go in1 rin1 in2 rin2 out rout = do
      eof1 <- liftIO $ hIsEOF in1
      if eof1 then release rin1 >> end in2 out rout -- (!)
         else do
          eof2 <- liftIO $ hIsEOF in2
          if eof2 then release rin2 >> end in1 out rout -- (!)
                  else do liftIO $ hGetChar in1 >>= hPutChar out
                          liftIO $ hGetChar in2 >>= hPutChar out
                          go in1 rin1 in2 rin2 out rout
    end in_ out rout = do
      eof <- hIsEOF in_
      if eof then release rout >> return ()
             else do hGetChar in_ >>= hPutChar out
                     end in_ out rout
```

Figure 9.2: `zipFiles` with early release

There are two packages available in Hackage that provide early release functionality for resource management, `resourcet` via the `Resource` monad and `pipes-safe` via the `Safe` monad. Both packages share similar interfaces and even have a similar implementation. In this section, we focus exclusively on the former.

Compared to `bracket` and `managed`, each time a resource is acquired — in this case, via the `allocate` function — we get a *release key* in addition to the resource itself. We can work with this resource as we did with any of the previous approaches, and the monad ensures that the resource is released at the end of the block. In addition, we can call `release` with the previously obtained release key and force an early release. It is guaranteed that no resource is released more than once.

In Figure 9.2, we rewrite the code using the facilities available in the `resourcet` package. The allocation is similar to the usage of `Managed`, except that here we give the allocation and release functions as separate arguments. Also similar to `Managed`, we need to lift all the operations that work in `IO` using the `liftIO` function. The new code is marked with `(!)` symbols: when we know that we are finished with one of the files, we immediately release it instead of waiting until the end of the computation.

Throughout this chapter, we have seen that resource management is far from an easy task once you start considering how to hold onto resources for the least possible amount of time in all possible usage and error scenarios. This chapter also concludes our journey through the principal monads, each of them providing a specific and fundamental functionality. A natural question to ask next is, "What if I need features from several different monads in a single code block?" The answer is just a few pages away.

Part III

Combining Monads

10 Functor Composition

The first two parts of this book deal with the generalities of monads and the specifics of the best-known monads one might find in the wild. This third part focuses on the techniques needed to combine the behavior of several monads into a single structure. As you, dear reader, can imagine from the fact that we have a whole part of the book devoted to this problem, it is not an easy one to solve.

In this chapter, we are going to focus on two observations about the nature of monads: the order in which we compose them affects the outcome of executing their effects, and we cannot have an infrastructure for composing them that is generic for all monads. What is even more interesting, we can achieve the second goal for almost every other abstraction we have introduced in this book — functors, applicatives, and traversables.

10.1 Combining Monads by Hand

In principle, each monad adds some extra behavior to a pure computation. `Maybe`, `Option`, and `Either` provide ways to signal failure. `Reader` introduces an environment that can be queried but not modified. `State` threads a hidden piece of data between actions, something we can perceive as passing a mutable variable. The same idea is extended to multiple variables via the `ST` monad. And so on. In many scenarios, though, we need several of these behaviors at the same time.

Let us make things more concrete with an example. Consider the following data type, which represents arithmetic expressions by combining the four basic operations, integer literals, and variables represented by an abstract `Name` type, which we declare to be a `String`:

```
type Name = String
data Expr = Literal Integer | Var Name | Op Op Expr Expr
data Op   = Add | Subtract | Multiply | Divide
```

One of the simplest things we can do with such an expression is evaluate it, that is, compute the result — an integer value — of performing all the operations. In order to do so, we need to know the value that each variable takes during evaluation. For example, evaluating the expression $x \times x + 1$ with $x \mapsto 3$ should return $3 \times 3 + 1 = 10$. We call such a mapping from variables to values an *assignment*. The evaluation function is not completely straightforward, though, because the lookup in the assignment may fail, in addition to arithmetic errors such as division-by-zero:

```
type Assignment = [(Name, Integer)]
eval :: Expr -> Assignment -> Maybe Integer
eval (Literal n) _ = return n
eval (Var v)     a = lookup v a  -- returns Nothing if not present
eval (Op o x y)  a = do
  u <- eval x a
  v <- eval y a
  case o of
    Add       -> return (u + v)
    Subtract  -> return (u - v)
    Multiply  -> return (u * v)
    Divide    -> if v == 0 then Nothing
                           else return (u `div` v)
```

In this code, we make use of the `Maybe` monad to account for the possibility of failure without having to check whether the result of every step is `Just` or `Nothing`. Still, there is some repetition in this code, because we have to thread the assignment manually through every subcall. It would be nice if we could use the power of the `Reader` monad to do this for us:

```
eval :: Expr -> Reader Assignment (Maybe Integer)
```

Unfortunately, this means that the monad in the `do` block is `Reader`, not `Maybe`, so we go back to manually checking whether the result of every operation fails or not:

```
eval (Op o x y) = do
  u <- eval x
  v <- eval y
  case (u, v) of
    (Nothing, _) -> return Nothing
    (_, Nothing) -> return Nothing
    (Just u', Just v') ->
      case o of
        Add      -> return (Just (u' + v'))
        Subtract -> return (Just (u' - v'))
        Multiply -> return (Just (u' * v'))
        Divide   -> if v' == 0 then return Nothing
                               else return (Just (u' `div` v'))
```

Our end goal should be to write code that *neither* checks for failure conditions *nor* requires manual passing of the assignment.

The first step in this direction is to create a completely new data type:

```
data Evaluator a = Evaluator (Reader Assignment (Maybe a))
```

Its `Monad` instance is not very complicated and follows the code pattern from `Reader` and `Maybe`. Note that, as discussed in Section 6.2, we use `Reader` and `runReader` as part of the `newtype` implementing the outer monad:

```
instance Monad Evaluator where
  return x = Evaluator $ Reader $ \_ -> Just x
  (Evaluator x) >>= f = Evaluator $ Reader $ \a ->
    case runReader x a of
      Nothing -> Nothing
      Just y  -> let Evaluator e = f y
                 in runReader e a
```

Using this instance, we can rewrite the code above without any mention of the assignment — except for the `Var` case — or manual handling of failure. To raise errors, we could implement one of the type classes we know about, such as `MonadThrow` or `MonadFail`, but in this case we introduce our own failure function, `evalFail`:

```
eval :: Expr -> Evaluator Integer
eval (Literal n) = return n
eval (Var v)     = do
  a <- Evaluator $ fmap Just $ ask
  case lookup v a of
    Nothing -> evalFail
    Just v' -> return v'
eval (Op o x y)  = do
  u <- eval x
  v <- eval y
  case o of
    Add      -> return (u + v)
    Subtract -> return (u - v)
    Multiply -> return (u * v)
    Divide   -> if v == 0 then evalFail
                          else return (u `div` v)

evalFail :: Evaluator a
evalFail = Evaluator $ Reader $ \_ -> Nothing
```

This solution does not scale, however, because for every different composition of monads, we will need to write a different data type and its corresponding monad

instance. In this case, the last step is not hard, but other scenarios, such as the combination of `State` and `List`, are far from trivial. It is not an easily maintainable solution either, since adding a new behavior in some part of your code requires duplicating most of the scaffolding for a new monad.

Non-commutativity. There is an important remark to be made at this point: the combination of two monads depends not only on the monads themselves but also on the order in which those monads appear in the type. For example, we can combine the list monad — which provides non-determinism — and the `Maybe` monad — which provides failure — in two different ways: `[Maybe a]` and `Maybe [a]`. The former type describes computations that have different outcomes, each of them possibly failing. In contrast, the latter is for computations that succeed or fail as a whole and in the former case produce zero or more values.

The list monad could also be combined with `State`. The difference between `[State s a]` and `State s [a]` reflects whether the state is shared between different outcomes of the functions. In the former type, each computation in the list holds its own state. On the other hand, `State s [a]` shares a single state among all values. For example, to implement Dijkstra's algorithm to compute which nodes are reachable in a graph, we should use this latter type, since the state — the set of visited nodes — is shared during the traversal.

Type constructor composition. In the example heading this section, we make use of the type `Reader Assignment (Maybe Integer)`. The inner `Maybe` monad is given as part of the result type of the outer `Reader`. This notation obscures the fact that we are trying to combine two monads, `Reader` and `Maybe`, that produce an `Integer` value with possible side effects. To make the connection more explicit, we introduce a new data type that combines two type constructors into a single one. Here we depart from Haskell's `base` library, where this type is known as `Compose` instead of `(:.:)`:

```
newtype (f :.: g) a = Compose (f (g a))
```

Now we can write the type above as `(Reader Assignment :.: Maybe) Integer`. Furthermore, we can speak of the type `Reader Assignment :.: Maybe` independently of its last type argument. With these terms in place, we can now raise the central question of this chapter: does the following `Monad` instance exist?

```
instance (Monad f, Monad g) => Monad (f :.: g) where ...
```

Note that in Scala, you do not need such syntactic workarounds, because the language can reason about the composition of type constructors during implicit search, thanks to *type-level lambdas*. For example, the `Functor` instance for composition is declared in Scalaz directly as:

```scala
private trait ComposeFunctor[F[_], G[_]]
  extends Functor[({type T[a] = F[G[a]]})#T] {
  implicit def F: Functor[F]
  implicit def G: Functor[G]
  override def map[A, B](fga: F[G[A]])(f: A => B): F[G[B]]
    = F(fga)(G.lift(f))
}
```

Cats provides an additional data type in case we prefer a Haskell-like definition:

```scala
final case class Nested[F[_], G[_], A](value: F[G[A]])
```

In this case, we say that `Nested[F, G, ?]`* is the one with a `Functor` instance.

10.2 Many Concepts Go Well Together

Before we delve into monads, let us consider those cases where there are *no* problems. If we know that both `f` and `g` are functors, we can make a new functor out of their composition. The function that we need to write for such an instance has the type:

```haskell
(a -> b) -> (f :.: g) a -> (f :.: g) b
-- alternatively, removing the Compose layer
(a -> b) ->   (f (g a)) ->    (f (g b))
```

```scala
// in Scala, we compose functors directly
(A => B) =>     F[G[A]] =>      F[G[B]]
```

The instance we need to write works in two steps. First, we lift the original function `a -> b` to `g a -> g b` by calling `fmap` for that functor. Then, we lift the resulting function again but this time using the mapping operation for `f`. As code, it looks like this:

```haskell
instance (Functor f, Functor g) => Functor (f :.: g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)
```

As usual, when dealing with `newtype`, we need to manually wrap and unwrap the contents. On a conceptual level, though, the `Compose` constructor plays no role.

Traversables — introduced in Section 4.2 — provide a similar interface to functors but work with functions of the form `a -> f b`, where `f` is an applicative functor. As a reminder, the function that performs the lifting in that scenario is this one:

```haskell
traverse :: Applicative m => (a -> m b) -> f a -> m (f b)
```

*This syntax comes from the `kind-projector` compiler plug-in. It means that `F` and `G` are fixed, but the last type variable may vary.

Given these similarities with `Functor`, we can reuse the idea of mapping twice to obtain an instance for the composition of two traversable functors:

```
instance (Foldable (f :.: g), Traversable f, Traversable g)
         => Traversable (f :.: g) where
  traverse f (Compose g) = Compose <$> traverse (traverse f) g
```

The only remarkable part of this code is that we need to lift the `Compose` constructor to work under the applicative `m`, which we do by means of `(<$>)`.

The applicative functor is another structure that works well under composition. The simplest operation to define on the combination is `pure` — `Applicative`'s version of `return`. We start with a value of type `a` and have to build one of type `f (g a)`. Easy! We just call `pure` twice, each time adding a different functor layer:

```
instance (Applicative f, Applicative g) => Applicative (f :.: g) where
  pure x = Compose $ pure (pure x)
```

The other operation that makes up an applicative functor is `(<*>)`. Ignoring for a moment the `newtype` wrappers, we need to provide a function with this signature:

```
(<*>) :: f (g (a -> b)) -> f (g a) -> f (g b)
```

The trick here is to start with `(<*>)` for the functor `g`. Such an operation needs to have the type `g (a -> b) -> g a -> g b`. We then take that operation and lift it through the functor `f`, which "adds one layer" to each argument and the result:

```
Compose f <*> Compose x = Compose $ liftM2 (<*>) f x
                        -- or using applicative operators
                         = Compose $ (<*>) <$> f <*> x
```

Alternatives — which we discussed in Section 7.1 — extend the notion of an applicative functor with operations for failure and fallback. This is an interesting case: if we compose two `Alternative` functors, we obtain yet another one, but we can also obtain one with even fewer constraints. We just need the inner functor to be an applicative:

```
instance (Alternative f, Applicative g) => Alternative (f :.: g) where
  empty = Compose empty
  Compose x <|> Compose y = Compose (x <|> y)
```

In fact, we do not need *anything* from the inner functor to write a type-correct definition. However, we need at least an applicative if we want the laws governing these type classes to be respected.

It seems like the promise of composition is achieved: start with a set of primitive functors — which might also be traversables, applicatives, or alternatives — and compose them as desired. The resulting combination is guaranteed to support at least the same operations as its constituents. If only that were true of monads.

10.3 But Monads Do Not

As you might have already guessed, it is not possible to take two monads `f` and `g` and compose them into a new monad `f :.: g` in a generic way. By *generic way*, we mean a single recipe that works for every pair of monads. Of course, there are some pairs of monads that can be combined easily, like two `Reader`s, and others that need a bit more work, like lists and optionals, as shown at the beginning of the chapter. But, stressing the point once again, there is no uniform way to combine any two of them.

In order to understand why, we are going to consider in greater detail the idea of monads as boxes, first discussed in Chapter 1:

```
class Monad m where
  return :: a -> m a
  join   :: m (m a) -> m a
```

We have just seen how to combine the `fmap` operations of two functors and the `pure` operations — `return` for monads — of two applicative functors. The latter method, `return`, poses no problem for composition: just take the definition of `pure` we described for the composition of two applicative functors. Therefore, we must conclude that `join` is the one to blame.

The `join` operation for the composition of two monads has the following type:

```
join :: (f :.: g) ((f :.: g) a) -> (f :.: g) a
```

If we leave out for a moment the `newtype`, this type amounts to:

```
join :: f (g (f (g a))) -> f (g a)
```

In a monad, we only have methods that add layers of monads — `return` and `fmap` — and a method that flattens two *consecutive* layers of the same monad. Alas, `f (g (f (g a)))` has interleaved layers, so there is no way to use `join` to reduce them to `f (g a)`.

As you can see, the reason why we cannot combine two monads is not very deep. It is just a simple matter of types that do not match. But the consequences are profound, since it tells us that monads cannot be freely composed.

10.3.1 Distributive Laws for Monads

One way to work around this problem is to provide a function that *swaps* the middle layers:

```
swap :: g (f a) -> f (g a)
```

This way, we can first turn `f (g (f (g a)))` into `f (f (g (g a)))` by running `swap` under the first layer. Then we can `join` the two outer layers, obtaining `f`

`(g (g a))` as a result. Finally, we `join` the two inner layers by applying the `fmap` operation under the functor `f`.

In code, this description amounts to:

```
join = fmap join . join . fmap swap
```

When such a function, `swap`, exists for two monads `f` and `g`, we say that there is a *distributive law* for those monads. In other words, if `f` and `g` have a distributive relationship, then they can be combined into a new monad.

One example of a pair of monads that admits a distributive law consists of two `Reader` monads. To describe such a law, we need a function mapping a value of type `Reader r (Reader s a)` into `Reader s (Reader r a)`. But remember that `Reader t a` is just a synonym for a function `t -> a`, which implies that the distributive law we are looking for is really a simple flip:

```
swap :: (r -> s -> a) -> (s -> r -> a)
swap f = \s r -> f r s
```

This shows us that we can compose two `Reader` monads into a single one. In fact, this is not an unexpected result, since having two pieces of data, `r` and `s`, in some environment is equivalent to having a pair `(r, s)`.

Some monads even admit a distributive law with *any other* monad. The simplest example is given by `Maybe`. We can move `Maybe` from the outer to the inner layer of a monad composition by using the following definition for `swap`:

```
swap :: Monad m => Maybe (m a) -> m (Maybe a)
swap Nothing  = return Nothing
swap (Just x) = fmap Just x
```

But note that we cannot, in general, write the dual function that takes `m (Maybe a)` and returns `Maybe (m a)`. This highlights, once again, that the order in which we compose monads matters.

> **Exercise 10.1.** The `Writer` monad admits a distributive law with any other monad from `Writer w :.: m` to `m :.: Writer w`.
> The `Reader` monad admits a dual distributive law from `m :.: Reader r` to `Reader r :.: m`.
> Write the corresponding `swap` functions, and think about which converse operations do not exist, in general.

The list monad. The discussion above contains a small lie. We definitely need a `swap` function if we want to combine two monads, but this is not enough. The reason is that a well-typed implementation may lead to a combined monad that violates one of the monad laws.

The list monad is a well-known example of this problem.* In principle, you can write a generic definition in the spirit of `Maybe`, above:

```
swap :: Monad m => [m a] -> m [a]
swap []     = return []
swap (x:xs) = (:) <$> x <*> swap xs
--          = sequence    -- other possible implementation
```

Using this definition, we can write an instance for a monad wrapped in a list. As usual, we need to introduce a `newtype` in Haskell to make the code compile:

```
newtype Listed m a = Listed { unListed :: [m a] }

-- This is using the Monad definition via fmap and join
instance Monad m => Monad (Listed m) where
  fmap f (Listed x) = Listed $ fmap (fmap f) x
  return x = Listed $ return [x]
  join = Listed . fmap join . join . fmap swap . unListed . fmap unListed

-- This is using the common Monad definition
-- Taken from the source of 'transformers'
instance Monad m => Monad (Listed m) where
  return a = Listed $ return [a]
  (Listed xs) >>= f = Listed $ do
    x <- xs
    y <- mapM (unListed . f) x
    return (concat y)
```

Exercise 10.2. Write the `Functor` and `Applicative` instances for `Listed m`. Use the code from Section 10.2 to guide you.

Alas, the definition of `Monad` for `Listed` shown above breaks the monad laws, in some cases. The shortest counterexample comes from Petr Pudlák, who has shown how the composition of the list monad with itself does not satisfy the associativity property of its Kleisli arrows:

```
v :: Int -> Listed [] Int
v 0 = Listed [[0, 1]]
v 1 = Listed [[0], [1]]
```

*There are a number of online articles with titles such as *`ListT` done right*.

```
main = do
    print $ ((v <=< v) <=< v) 0
    -- = [[0,1,0,0,1],[0,1,1,0,1],[0,1,0,0],
    --    [0,1,0,1],[0,1,1,0],[0,1,1,1]]
    print $ (v <=< (v <=< v)) 0
    -- = [[0,1,0,0,1],[0,1,0,0],[0,1,0,1],
    --    [0,1,1,0,1],[0,1,1,0],[0,1,1,1]]
```

If Kleisli arrow composition were associative, then both lines would print the same result. In this case, they do return the same elements but not in the same order. It is also remarkable that even combining a monad with itself can lead to problems.

For the specific situation of the list monad, the cases for which its combination with another monad `m` lead to a new monad have been described. In particular, for the `swap` procedure to be correct, `m` needs to be a *commutative* monad, that is, the following two blocks must be equal. The difference lies in the distinct order in which `xs` and `ys` are bound:

```
do x <- xs                do y <- ys
   y <- ys        ≡          x <- xs
   return (x, y)             return (x, y)
```

The `Maybe` monad is commutative, since the order of failure does not matter for a final absent value, and in the case in which both elements are `Just` values, the result is the same. On the other hand, the list monad is not commutative: both blocks will ultimately result in the same elements, but the order in which they are produced will be different.

Other techniques, besides using distributive laws, for combining two monads are described in the paper *Composing monads* [Jones and Duponcheel, 1993]. With the passage of time, some of these techniques have warped into monad transformers. But distributive laws still remain as a distinct notion, because they also appear in the categorical description of monads.

11

A Solution: Monad Transformers

It would be impolite to thoroughly describe a problem — the composition of monads — and then not describe at least one of the solutions. That is the goal of this chapter, to describe how *monad transformers* allow us to combine the operations of several monads into one single monad. It is not a complete solution, however, since we need to change the building blocks: instead of composing several different monads into a new monad, we actually enhance one monad with an extra set of operations via a transformer.

Alas, there is one significant downside to the naïve, transformers approach: we cannot abstract over monads that provide the same functionality but are not identical. This hampers the maintainability of our code, as any change in our monadic needs would lead to huge rewrites. The classic solution is to introduce type classes for different sets of operations. Consider that solution carefully, as it forms the basis of one of the styles for developing your own monads, as we will discuss in Chapter 13.

One word of warning before proceeding: monad transformers are *a solution* to the monad composition problem. But they are not *the solution*. Another approach, *effects*, is gaining traction in the functional programming community. The right way to design an effects system is an idea that is still in flux, however, as witnessed by its many different implementations.

11.1 Monadic Stacks

In the previous chapter, we introduced the `Evaluator` monad to combine the features of a `Reader` — which holds an assignment from variables to integral values — and a `Maybe` — because some operations, such as division, may fail:

```
data Evaluator a = Evaluator (Reader Assignment (Maybe a))
```

Unfortunately, we had to write a completely new monad instance. Otherwise, we would only get the functionality from the top-level monad, `Reader` in this case.

Consider now a similar scenario, in which the assignment is not only read but may also be updated. We need a `State` monad instead of a `Reader`. However, the ability to fail is encoded in a similar fashion, by having `Maybe` in the return type position:

```
data Evaluator2 a = Evaluator2 (State Assignment (Maybe a))
```

This suggests a way to add failure to any other monad — just change the return type to `Maybe`. This is what we call the `MaybeT` or `OptionT` *monad transformer*:

```
data MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

```
case class OptionT[M[_], A](runMaybeT: M[Option[A]])
```

The key point here is that if `m` is a monad, then `MaybeT m` is also guaranteed to be a monad:

```
instance Monad m => Monad (MaybeT m) where
  -- return :: a -> MaybeT m a
  return x = MaybeT $ return (Just x)
  -- (>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
  (MaybeT x) >>= f = MaybeT $ do
    y <- x
    case y of
      Nothing -> return Nothing
      Just z  -> runMaybeT (f z)
```

It is useful to compare the code for the `MaybeT m` instance with that of `Maybe`:

```
instance Monad Maybe where
  return x = Just x
  y >>= f =
    case y of
      Nothing -> Nothing
      Just z  -> f z
```

Apart from the wrapping and unwrapping of the `MaybeT` constructor, the only thing that `MaybeT` adds are calls to `return` and the use of `<-`, because we are no longer in a pure context. This similarity is used below to derive the `Monad` instance of `Maybe` directly from that of `MaybeT`.

> **Exercise 11.1.** Rewrite the last implementation of the `eval` function in Section 10.1 to have this type:
>
> ```
> eval :: Expr -> MaybeT (State Assignment) Integer
> ```

Remember that most monads come with an associated `runMonad` function that executes the computation. For example, `runState` executes a stateful computation when given the initial value of a state. When we transform our monad, however, we can no longer run the computation in the same way. We need to "peel off" the transformer layer beforehand. For that reason, the `Maybe` transformer also comes with an associated `runMaybeT` function. The expression that runs the `eval` function over another expression `e` and an initial assignment `a` becomes:

```
runState (runMaybeT $ eval e) a :: (Maybe Integer, Assignment)
```

Note that the order of `run` applications — `runState` after `runMaybeT` — is reversed with respect to the type of `eval e` — `MaybeT (State Assignment Integer)`. The reason is that each `run` function removes the outer layer: first we go from `MaybeT (State Assignment Integer)` to `State Assignment (Maybe Integer)`, and then we execute the stateful computation to reach `(Maybe Integer, Assignment)`.

Note that `MaybeT` itself is not a monad, if only because of the form of the type argument it takes. A monad becomes a ground type once a parameter is given, like `Maybe Int` or `Option[T]`. Scala makes this explicit, because you need to write a hole to refer to the type itself, `Option[_]`. In contrast, `MaybeT` requires *two* arguments: a monad that is wrapped and then a ground type. Thus, we write `MaybeT (Reader Assignment) Int` or `OptionT[Reader[Assignment,_],_]`. Not only are there two holes instead of one, but the type that fills the first position has a hole itself.

The Haskell language has a notion — *kinds* — to specify the form of the type arguments that are allowed in each position. A ground type, such as `Int` or `Bool`, is said to have kind `*` (pronounced "star"). Ground types are those composed of actual *values* that are held in memory. A type constructor like `Maybe`, on the other hand, has type parameters that must be specified before we can speak of actual values. In the case of `Maybe`, we need one argument, which ought to be a ground type itself, to form things like `Maybe Int` or `Maybe Bool`. We say that the kind of `Maybe` is `* -> *`. Note the similarity with the usual typing discipline, as we can conclude only from its kinds that `Maybe Int` is a ground type. Any functor, applicative, traversable, or monad shares this same kind, `* -> *`.

Now let us return to `MaybeT`. It takes two type arguments, the first one with the shape of a monad, and the second being ground. Its kind is thus `(* -> *) -> * -> *`. We can play here the same trick that we play with the types of functions and add some parentheses:

```
Maybe, Reader r, State s
       :: (* -> *)             -- or any monad
MaybeT :: (* -> *) -> * -> *   -- or any transformer
```

This kind gives us yet another reason to call `MaybeT` a monad transformer. It you apply `MaybeT` to just one argument — which ought to be a type constructor — you get back a type constructor with kind `* -> *`. This is exactly the kind of a monad. `MaybeT` takes a monad and turns it into a new kind of monad.

Monad	*Transformer*	*Package*
`Maybe a`	`MaybeT m a = m (Maybe a)`	`t`
`Either e a`	`ExceptT e m a = m (Either e a)`	`t`
`State s a = s -> (a, s)`	`StateT s m a = s -> m (a, s)`	`t`
`Reader r a = r -> a`	`ReaderT r m a = r -> m a`	`t`
`Writer w a = (a, w)`	`WriterT w m a = m (a, w)`	`t`
`Logic a`	`LogicT m a`	`l`
`Cont r a`	`ContT r m a`	`t`
`= (a -> r) -> r`	`= (a -> m r) -> m r`	
	`ResourceT m a`	`r`
	`SafeT m a`	`p`

Table 11.1: Summary of monad transformers
(`t` = `transformers`, `l` = `logict`, `r` = `resourcet`, `p` = `pipes-safe`)

11.1.1 Other Monad Transformers

Almost every monad has an associated monad transformer. In Haskell, most of them are defined in the `transformers` library, in the `Control.Monad.Trans` module. Table 11.1 provides a summary of those transformers. Scalaz and Cats also come with a corresponding set of transformers.

Note that, in contrast to `MaybeT`, not all monad transformers lead to a type of the form `m Something`. For example, `ReaderT` and `StateT` "inject" the monad `m` within the result type of the arrow. We leave out the translations of `LogicT`, `ResourceT`, and `SafeT`, since they are considered internal details of their respective implementations.

Our description of resource management monads in Chapter 9 left types conspicuously absent. Here is the reason: since using those monads only makes sense when working within a larger monad with resources to manage, the corresponding libraries only provide transformer versions. As an example, an operation that returns the contents of a file is assigned the type `ResourceT IO String`.

> **Exercise 11.2.** Write the `Monad` instances for the combinations of the above transformers with an arbitrary monad. Hint: proceed as we have done with `MaybeT` — start with the implementations of `return` and `bind` for the monad and replace pure applications and bindings with `do` notation.

The list transformer. There is an important monad missing from the table of transformers above: the list monad. The problem is that the naïve implementation, `ListT m a = m [a]`, only works correctly when `m` is a commutative monad, for the reasons outlined at the end of the previous chapter.

Here is the `Monad` instance of that `ListT`, for reference:

```
instance Monad m => Monad (ListT m) where
  return a = ListT $ return [a]
  (ListT xs) >>= f = ListT $ do
    x <- xs
    y <- mapM (runListT . f) x
    return (concat y)
```

Fortunately, there are several implementations of list-like monad transformers that satisfy the right laws. The first solution is to use the `LogicT` transformer, since it provides a superset of the features provided by the list monad. The package `list-transformer` builds on the following pair of data types:

```
newtype ListT m a = ListT { next :: m (Step m a) }
data    Step m a  = Cons a (ListT m a) | Nil
```

Comparing the `Monad` instance for this version of `ListT` to the previous one helps us to grasp the difference between the correct and incorrect implementations of the list monad transformer:

```
instance Monad m => Monad (ListT m) where
  return = ListT $ return (Cons x empty)
  (ListT xs) >>= f = ListT $ do
    x <- xs
    case x of
      Nil       -> return Nil
      Cons y ys -> next (f y <|> (ys >>= f))

-- The definition above uses the Alternative instance
instance Monad m => Alternative (ListT m) where
  empty = ListT (return Nil)
  (ListT xs) <|> ys = ListT (do
    x <- xs
    case x of
      Nil       -> next ys
      Cons z zs -> return (Cons z (zs <|> ys)) )
```

Whereas the former (wrong) definition needs a *single effect* to produce the entire list, the implementation in `list-transformer` executes a monadic action *at each step*.

The package `list-t` implements this idea, but instead of two data types, it uses one:

```
newtype ListT m a = ListT (m (Maybe (a, ListT m a)))
```

The empty list case, which corresponds to `Nil` in `list-transformer`, is represented as `Nothing` in `list-t`. Conversely, `Just` is used to represent `Cons`. We refrain from showing the instance declaration in this case, since it is quite similar to the previous one.

11.1.2 The Identity Monad

The discussion of the relation between `Traversable` and `Functor` in Section 4.2.1 led us to introduce the "do nothing" monad, usually referred to as the `Identity` monad. As a reminder, here are the definitions of `Identity` in Haskell — as usual, we require a `newtype` wrapper — and Scala:

```
type Identity a = I a

instance Functor Identity where
  fmap f (I x) = I (f x)
instance Monad Identity where
  return x     = I x
  (I x) >>= f  = f x
```

```
object Id {  // Adapted from Scalaz
  type Id[X] = X

  implicit val idMonad: Monad[Id] with Functor[Id]
    = new Monad[Id] with Functor[Id] {
        def point[A](x: A) = x
        def bind[A, B](x: A)(f: A => B): B = f(x)
        def map[A, B](x: A)(f: A => B): B = f(x)
      }
}
```

This monad adds no extra operations or effects. It just replicates the pure language.

> **Exercise 11.3.** Check that `T` and `Identity T` are isomorphic for any type `T`. In other words, write these two functions:
>
> ```
> toIdentity   :: a -> Identity a
> fromIdentity :: Identity a -> a
> ```
>
> Check that applying one after the other, in any order, always gives back the same result.

Haskell's `transformers` library uses `Identity` to remove the duplication of code between a monad and its transformer version. The trick is that the monad is

nothing more than the transformer applied over the `Identity`. The only possible effects we get from this combination are those from the transformer, since `Identity` carries none of its own. In fact, if you dive a bit into the internals of the library, you will find the following definitions:

```
type State  s = StateT  s Identity
type Reader r = ReaderT r Identity
type Writer w = WriterT w Identity
```

For peace of mind, let us check that `MaybeT Identity` and `Maybe` form the same monad. The definition of `MaybeT` tells us directly that `MaybeT Identity a = Identity (Maybe a)`. Furthermore, the previous discussion tells us that `Identity (Maybe a)` is isomorphic to `Maybe a` — they are the same type, except for the `newtype` wrappers. We need to check, though, that the implementations of the `Monad` methods are also the same:

```
   m >>= f
≡ -- definition of MaybeT and Identity data types
   (MaybeT (I x)) >>= f
≡ -- definition of (>>=) for MaybeT
   MaybeT $
     do y <- I x
        case y of
          Nothing -> return Nothing
          Just z  -> f z
≡ -- replacing do notation with binds
   MaybeT $
     I x >>= \y ->
     case y of
       Nothing -> return Nothing
       Just z  -> f x
≡ -- definitions for (>>=) and return
   MaybeT $ (\y ->
     case y of
       Nothing -> Nothing
       Just z  -> f z) x
≡ -- since (\u -> ... u ...) t = ... t ...
   MaybeT $ case x of
     Nothing -> Nothing
     Just z  -> f z
```

The final result is exactly the same as the bind operation of the `Maybe` monad, except for the additional construction and elimination of `newtype` wrappers for `MaybeT` and `Identity`.

Exercise 11.4. Prove (or at least convince yourself) that the definitions of `return` and `fmap` also coincide.

As a final note, not only do we have an `Identity` monad, but we can also define an `IdentityT` monad transformer:

```
type IdentityT m a = m a
```

This transformer serves almost no practical purpose, unlike the `Identity` monad. It goes very well, though, with the theme of things that compose and have an identity element that underlies many of the abstractions in the Haskell and Scala style of functional programming. Section 17.4 describes the role of `IdentityT` from a theoretical point of view.

11.1.3 Higher Towers of Monads

Since the combination of a transformer and a monad gives you back another monad, nothing stops you from applying yet another transformer to the result. For example, you may have a computation that obtains data for a person, that needs access to a global environment (like a pool of database connections), that produces some logging,*, and that also may fail. This computation can be assigned a type like this:

```
ReaderT Environment (WriterT [Message] Maybe) Person
```

We could also write this type completely in terms of transformers, with the help of the `Identity` monad:

```
ReaderT Environment (WriterT [Message] (MaybeT Identity)) Person
```

We call such a type a *monadic stack*, and we say that `Maybe` is the *bottom layer* and `ReaderT` the *top layer* of this stack.

We can expand the definition of each transformer to obtain the "actual type" under the hood:

```
   ReaderT Environment (WriterT [Message] (MaybeT Identity)) Person
≡ -- ReaderT r m a = r -> m a
   Environment -> (WriterT [Message] Maybe) Person
≡ -- WriterT w m a = m (a, w)
   Environment -> Maybe (Person, [Message])
```

Note that this type is *not* the same as the following:

```
Reader Environment (Writer [Message] (Maybe Person))
```

*As we discussed in Section 6.3, `Writer` is not recommended for real logging, only for small pieces of pure code. We use it here for illustration purposes.

Expanded, the above amounts to `Environment -> (Maybe Person, [Message])`.

Looking at the expanded type helps us understand the relation between the different layers of the stack. This specific combination needs an environment to even start producing a return value. However, we only get back the execution log if the result is successful, since `[Message]` appears inside the `Maybe`. This is not usually what we want. We would generally prefer to get back a log even (more) when the execution fails. A monadic stack that enables this functionality can be achieved by swapping the order of the bottom two layers:

```
   ReaderT Environment (MaybeT (Writer [Message])) Person
≡ -- ReaderT r m a = r -> m a
   Environment -> MaybeT (Writer [Message]) Person
≡ -- MaybeT m a = m (Maybe a)
   Environment -> Writer [Message] (Maybe Person)
≡ -- Writer w a = (a, w)
   Environment -> (Maybe Person, [Message])
```

This example shows us that we have to be careful when putting together big stacks. As we already know, the composition of monads is not commutative, so the order matters. Furthermore, it is difficult to understand only from the type, expressed as a combination of transformers, which layers can interact with one another. In general, bottom layers cannot access values from upper layers, but the effects of those upper layers may still influence the execution of the bottom layers.

11.2 Classes of Monads, MTL-style

Going back to the beginning of our discussion about monad transformers, consider the stack we designed for evaluating an arithmetic expression that might fail:

```
type Evaluator a = MaybeT (Reader Assignment) a
```

In order to implement one such evaluator, we need to access the assignment in the environment, but we might also fail. Since our monadic stack is supposed to support both sets of operations, we could write:

```
eval :: Expr -> Evaluator Integer
eval (Literal n) = return n
eval (Var v)     = do
  a <- ask
  case lookup v a of
    Nothing -> mzero
    Just v' -> return v'
```

```
eval (Op o x y)  = do
  u <- eval x
  v <- eval y
  case o of
    Add      -> return (u + v)
    Subtract -> return (u - v)
    Multiply -> return (u * v)
    Divide   -> if v == 0 then mzero else return (u `div` v)
```

If you write this code in a file and ask GHC to compile it, it is accepted without complaint. However, it is far from trivial why this is the case. After all, according to the previous chapters in this book, the function `ask` has the type `Reader r r`, whereas we use it in the code above as `MaybeT (Reader r) r`.

This problem is not specific to `Reader` or `MaybeT`. Whenever you combine one or more transformers, you want to be able to use the operations from all the monads you combine — if possible, without any special ceremony just because you happen to be using monad transformers.

We already hinted at the solution when we introduced `MonadPlus` and `MonadError` in Chapter 7: we introduce type classes or traits that represent each set of primitive operations in a monad. These are customarily known as `MonadX`, where `X` is the name of the monad we are abstracting over. For example, here are the Haskell type class and Scala trait that correspond to `Reader`:

```
class Monad m => MonadReader r m | m -> r where
  ask   :: m r
  local :: (r -> r) -> m a -> m a
```

```
trait MonadReader[F[_], S] extends Monad[F] { self =>
  def ask: F[S]
  def local[A](f: S => S)(fa: F[A]): F[A]
}
```

In Section 6.2, we introduced a derived operation for the `Reader` monad, namely `asks`. Once you generalize the primitive operations, you can also make `asks` polymorphic over the monad in which you work, provided that it is an instance of `MonadReader`:

```
asks :: MonadReader r m => (r -> a) -> m a
```

```
trait MonadReader[F[_], S] extends Monad[F] { self =>
  ???
  def asks[A](f: S => A): F[A] = map(ask)(f)
}
```

Of course, in order to make the operations work for specific monadic stacks, we have to write the instances. For each of these classes, we need two kinds of instances:

1. You need those instances that really implement the operation. For example, a monadic stack that contains `ReaderT` is an instance of `MonadReader`:

   ```
   instance Monad m => MonadReader r (ReaderT r m) where
     ask       = ReaderT return
     local f m = ReaderT $ runReaderT m . f
   ```

2. If the top layer of your stack is not `ReaderT`, you need to pass through that layer and find a `MonadX` below. For example, if `MaybeT m` is a `MonadReader`, that means `m` must be a `MonadReader`, since `MaybeT` does not provide such functionality:

   ```
   instance MonadReader r m => MonadReader r (MaybeT m) where
     ask       = MaybeT $ runMaybeT ask
     local f m = MaybeT $ runMaybeT (local f m)
   ```

In general, this means that if we have n sorts of monads, we need to write n different classes and n instances for each of those classes. This gives us a grand total of n^2 instances to write. Fortunately, it is not our task to write all of those instances. The `mtl` package makes them available for Haskell, whereas Scalaz contains the necessary traits and implicit values for Scala. In Chapter 12, we will look at techniques to prevent this sort of code duplication.

Table 11.2 summarizes each of the type classes and traits that come with the main implementations of each monadic concept. For each of them, we also state the primitive operations that its instances must support.

Automatic derivation for your own monadic stacks. A common architectural pattern in typed functional programming is a custom monad stack that contains all the functionality you need for an application. One possibility for defining such a stack is to use a type synonym, as we did above for `Evaluator`. Good practices tell us that an even better option is to wrap that stack into a `newtype`, because this prevents the details of the monad from leaking all over your program:

```
newtype Evaluator a
  = Evaluator { unEvaluator :: MaybeT (Reader Assignment) a }
```

By doing this, however, we step into a thick wall: the `Monad`, `MonadReader`, and `MonadPlus` instances no longer apply, since the compiler sees `Evaluator` as a type completely different from any other. Note that Scala does not suffer this problem so much, for two reasons. First, it is more common to define `Evaluator` as a type synonym, instead of declaring a completely new type, as in Haskell. Second, the

Type class or trait	*Instances*	*Operations*

`MonadPlus m`	`MaybeT m`
	`Monoid e => ExceptT e m`
	`ListT m`
		`mzero :: m a`
		`mplus :: m a -> m a -> m a`

`MonadError e m`	`ExceptT e m`	`throwError :: e -> m a` `catchError :: m a -> (e -> m a) -> m a`
`MonadState s m`	`StateT s m`	`get :: m s` `put :: s -> m ()`
`MonadReader r m`	`ReaderT r m`	`ask :: m r` `local :: (r -> r) -> m a -> m a`

`MonadWriter w m`	`Monoid w => WriterT w m`	`tell :: w -> m ()`
		`listen :: m a -> m (a, w)`
		`pass :: m (a, w -> w) -> m a`

`MonadLogic m`	`LogicT m`	`msplit :: m a -> m (Maybe (a, m a))`
`MonadCont r m`	`ContT r m`	`callCC :: ((a -> m b) -> m a) -> m a`

Table 11.2: Summary of main monad classes

degree of control of implicit search in Scala is much higher, so we can help the compiler find the corresponding instances.

In modern versions of GHC, we can use *automatic derivation* to solve this problem. Since a `newtype` just wraps an existing type, we can instruct the compiler to "copy" the instance of the underlying type to the newly-created one. This feature requires a language pragma:

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

Then, we enumerate all the instances we want to derive after the data type definition:

```
newtype Evaluator a
  = Evaluator { unEvaluator :: MaybeT (Reader Assignment) a }
    deriving ( Functor, Applicative, Alternative, Monad
             , MonadPlus, MonadReader Assignment )
```

Scala does not have a feature similar to `newtype`, so developers use monad transformers directly. This prevents the problem from spreading to Scala. As a result, we do not need to give any special guidance to the compiler.

The `MonadIO` type class. There is one monad that does not have a corresponding transformer: `IO`. We can only add new features to `IO` by wrapping it with a transformer. In many ways, `IO` is similar to the `Identity` monad, but it provides an impure rather than a pure base, instead.

One difference between `IO` and all the other monads is that the latter usually have a small set of operations, whereas `IO` supports all kinds of side-effectful computations, from the passing of global variables to network communication. The solution of having a type class for all operations in the monad does not scale here. Instead, `mtl` provides a special `MonadIO` type class:

```
class MonadIO m where
  liftIO :: IO a -> m a
```

Using `liftIO`, we can make any `IO` operation work in a monad that has `IO` as the bottom layer. Imagine we keep some configuration in a `Reader` — we could read the file mentioned there using:

```
do file <- asks configFile
   liftIO $ readContents file
```

Another ability that `IO` provides is throwing and catching exceptions. As we discussed in Section 8.2, there are several type classes that abstract over the specific structure we use for this task. These type classes provide an interface in the very same `mtl`-style we have been discussing. Table 11.3 summarizes the operations in each class and their main instances.

Type class or trait	*Instances*	*Operations*
`MonadIO m`	`IO`	`liftIO :: IO a -> m a`

`MonadThrow m`	`IO, STM`
	`[], Maybe`
	`Either SomeException`
		`throwM :: Exception e => e -> m a`

`MonadCatch m`	`IO, STM` `Either SomeException`	`catch :: Exception e =>  m a -> (e -> m a) -> m a`
`MonadResource m`	`ResourceT m`	`liftResourceT :: ResourceT IO a -> m a`

Table 11.3: Summary of monad classes related to IO

11.2.1 Mixing Several Reader Monads

The definition of the `MonadReader` type class in Haskell has a decoration attached to it: `| m -> r`. This annotation is called a *functional dependency*. The way to read it is that given a monad `m`, we also know the type of `r`. In other words, the type `r` is a function of type `m`.

This additional element is necessary, because when we write a function that is parametric over the `Reader` monad, the monad in the result type is simply `m`, in contrast to the version that is not polymorphic, which returns `ReaderT r m`. Compare, for example, the following two signatures for `ask`, the former dictating that `ReaderT` must appear on top, whereas the latter is parametric over the stack:

```
ask :: ReaderT r m r
ask :: MonadReader r m => m r
```

The functional dependency above is needed to get reasonable type inference. We do not want to dive too deeply into this topic, but without the dependency, the compiler is unable to figure out that two operations in the same `Reader` monad must share its type. This means that every time we use `ask`, we need to include an annotation:

```
n <- (ask :: m Int)
```

On the other hand, functional dependencies *forbid* us from declaring that a certain computation needs access to *two* pieces of the environment in an easy way. Suppose we were to declare a function with this type:

```
eval :: (MonadReader Assignment m, MonadReader Cache m)
     => Expr -> m Int
```

The compiler would stop us. The reason is that the functional dependency declares that given an `m`, we can infer the type of the environment, which we call `r` in the definition of `Reader`. By applying this idea to both `MonadReader` declarations, we reach the conclusion that the types of `Assignment` and `Cache` should coincide. But those types are completely different, and as a result, the type signature is rejected.

Classy lenses. The first solution to this problem accepts the fact that we can only have one `MonadReader r m` per signature, but it plays the trick of keeping the `r` under-constrained. To make things concrete, let us look again at our specific example.

Instead of asking for `Assignment` and `Cache` directly, we use type classes to access them:

```
class HasAssignment r where
  assignment :: r -> Assignment
class HasCache r where
  cache :: r -> Cache
```

Then we declare a reader type to comply with both classes:

```
eval :: (MonadReader r m, HasAssignment r, HasCache r)
     => Expr -> m Int
```

The downside is that instead of just calling `ask`, we need to apply the corresponding accessor function, `assignment` or `cache`:

```
eval (Var v) = do env <- assignment <$> ask
                  ... -- work with env
```

Another problem is that we need to define the `HasX` classes and instances for all the data types that carry one of the pieces of information. Some instances are really dumb, like those that look for an element of type `X` inside a value of type `X`:

```
instance HasAssignment Assignment where
  assignment = id
instance HasCache Cache where
  cache = id
```

The `lens` and `microlens-th` packages provide functionality to write that code automatically. In both cases, we just need to call `makeClassy` with the name of the type we want to abstract over.

Ether. Another possibility is to use the `ether` package, in addition to `mtl`. This package provides a notion of *tags* to refer to layers in a stack. If there is some ambiguity regarding the layer you want to use for a certain operation, you can use a tag to state explicitly which one you want.

The usage of `ether` is quite simple. First, you need to create data types to represent the tags at the type level. Since we do not need them at runtime, we can just create them without constructors:

```
data TheAssignment
data TheCache
```

The next step is to tag those layers that conflict using the data types previously declared. Note that here we use a different `MonadReader` class than the one from `mtl` — we prefix it with `Ether` to avoid confusion:

```
eval :: ( Ether.MonadReader TheAssignment Assignment m
        , Ether.MonadReader TheCache      Cache      m )
     => Expr -> m Int
```

Note also that we still need to write the reader type, *in addition* to the tag. Finally, we use the specific versions of the monad operations from `ether`. By switching on the `TypeApplications` compiler extension available since GHC version 8.0, we can use the types representing the tags to disambiguate which `MonadReader` we are referring to:

```
eval (Var v) = do env <- Ether.ask @TheAssignment
                  ... -- work with env
```

There are similar libraries available in Hackage, most notably `monad-classes`. However, `ether` offers, in our opinion, the best trade-off between complexity and extra power.

The problems outlined in this section are also visible in other monad classes that use functional dependencies, such as `MonadState` and `MonadWriter`. The solutions in all cases are similar to the ones described above. In fact, `lens` and `microlens` assume that any `MonadState` constraint is going to be polymorphic, and they provide utility functions for that use case.

11.2.2 Another Note on Packaging

At this point, we have introduced several Haskell packages related to monads and transformers. Inevitably, you will ask yourself the question of which of them you need for the functionality you require. Before aswering that question, let us dissect the three moving parts in Haskell's approach to combining monads:

1. Concrete monads, such as `Reader`. Each of them has a corresponding `Monad` instance.
2. Monad transformers, such as `ReaderT`. Each of them comes with an instance of `Monad` that requires the underlying type to also be a monad. A function operating on a transformer stack is not polymorphic, as it requires an exact match on the structure of the stack to be called.
3. Classes of monads, such as `MonadReader`, which allow us to use the same function over different monadic stacks. These classes introduce polymorphism for the primitive monadic operations.

Basic concrete monads — lists, `Reader`, optionals — live in either the `base` or `transformers` package. The corresponding transformers live in the `transformers` library. For classes of monads, you can choose between `mtl` or `ether`. The two latter packages also expose the functionality of the others. Adding `mtl` as a dependency is the most common approach in Haskell projects for using combinations of monads.

Outside this basic set, packages usually provide the three components we need: a concrete monad, the corresponding monad transformer, and a class to work with different stacks. This is the case for `Logic` and `ResourceT`, for example. Note that in many cases, those packages transitively depend on `mtl`, in order to provide instances for different monad classes.

As we have discussed previously, the problem of combining monads is not considered solved in the functional programming community. The classes of monads provided by `mtl` are without a doubt the most common solutions. Packages such as `ether` and `monad-classes` remove some restrictions, most notably having only

one layer of each type within a stack. If you prefer type families to multi-parameter type classes, `mtl-tf` provides classes of monads where the additional information (such as the type of the environment in a `Reader`) is expressed with associated types. Older packages such as `monads-fd`, most of them deprecated in favor of `mtl`, also survive in the ever-growing repository of Haskell packages.

11.3 Parsing for Free!

Parsers are pieces of code that take some unstructured data — usually raw strings — and turn it into structured data — like a JSON value or a tree representing a program. Parsing is one of the areas where functional programming shines, especially in the form of parser combinators. The goal of this section is not to be a deep dive into parsing in Haskell or Scala (you should use your favorite library for that purpose) but to see how easy it is to implement a simple parsing strategy with monad transformers.

Graham Hutton, in his book *Programming in Haskell* [Hutton, 2016], includes a little poem to remember the type we use to represent a parser in functional languages:

> A parser for things
> is a function from strings
> to lists of pairs
> of things and strings.

Unraveling the rhyme, we can figure out the type, `Parser a = String -> [(a, String)]`. The first `String` represents the input. From the input, we get different, partial results — each consisting of some value and a leftover string. This leftover string is then consumed by further parsing operations. Looking carefully, we can write this down as a combination of two monads:

```
type Parser a = StateT String [] a
              -- given that StateT s m a = s -> m (a, s)
              ≡ String -> [(a, String)]
```

If you have ever tried to write the `Functor`, `Applicative`, `Monad`, and `MonadPlus` instances for the `Parser` data type above, you know that it is painful to get them right. But we never have to do it in the first place, since they come for free if we formulate parsers using monad transformers!

In order to build actual parsers, we need two primitive operations. The first one checks that the first character of an input string satisfies a given predicate and returns it, if successful:

```
satisfies :: (Char -> Bool) -> Parser Char
```

```
satisfies p = StateT $ \s ->
                case s of
                  "" -> []                          -- fail
                  (c:cs) | p c       -> [(c, cs)]  -- success
                         | otherwise -> []         -- fail
```

Using this primitive, it is easy to write a parser that checks for a specific character:

```
char :: Char -> Parser Char
char c = satisfies (== c)
```

> **Exercise 11.5.** Actually, we can also take `char` as the primitive operation. Write `char` directly in terms of `StateT`. Then, use `char` and the `guard` function to define `satisfies`.

The second primitive operation is `end`, which allows us to check that the input string has been completely consumed. It is not possible to write it using the previous combinators, since they always consume some part of the input:

```
end :: Parser ()
end = StateT $ \s -> case s of
                       "" -> [((), "")]
                       _  -> []
```

Don't worry if you do not completely understand the parser code, as this is just for illustration. If you want to understand parser combinators, the aforementioned *Programming in Haskell* or my own book, *Beginning Haskell*, provide tutorials. Using these two simple primitives, however, we can already write a parser for hexadecimal numbers:

```
hex :: Parser String
hex = (\a b c d -> [a, b, c, d])
        <$  (char '0') <*  (char 'x' <|> char 'X')
        <*> hexdigit <*> hexdigit <*> hexdigit <*> hexdigit
  where hexdigit = satisfies (`elem` "abcdefABCDEF012345789")
```

Apart from operations to build parsers, we also need to *execute* them, that is, to obtain the structured data encoded in a given string. We do not need to look too far for a means to do this, because the `StateT` runner already has the correct type:

```
runParser :: Parser a -> String -> [(a, String)]
runParser p s = runStateT p s
```

If we are only interested in those runs that consume the full input, we just need to filter the resulting list:

```
runParser' :: Parser a -> String -> [a]
runParser' p s = do (x, "") <- runParser p s
                    return x
           -- or [x | (x, "") <- runParser p s]
```

As you can see, once the whole ecosystem of monad transformers is in place, you get a lot of code for free. In addition, the code is easy to modify: for example, if you need to add an environment to your parsing in order to provide some parsing options, you can just add a new `ReaderT` layer on top of the existing ones. Ensuring that the operations of every monad layer are available at the upper layer, however, can be tedious — as we will see in the next chapter.

12

Generic Lifting and Unlifting

The last chapter introduced one of the main tools for dealing with complex interactions of effects, *monad transformers*. It is rather difficult to escape them if you architect your application using Haskell, Scalaz, Cats, or similar libraries that build on the concept of monads.

This chapter is a bit different from the previous ones in that we look at the implementations of monad transformers in order to highlight some of their shortcomings. Note that you can be productive with transformers by knowing only how to use them and maybe a bit about `lift` for advanced use cases — in other words, by understanding only the previous chapter and the first section of the current one.

The n^2 problem for transformer instances. The heart of the problem is that dealing with monad transformers using `mtl`-like type classes or traits requires *too much* code. On the other hand, if we do not define those classes, we can only write code that works on specific monad stacks, even though the same code would work verbatim for different ones.

As we explained in the previous chapter, for each monad `Mn`, we usually define a transformer version `MnT` which wraps other monads inside:

```
instance Monad m => Monad (MnT m) where ...
```

Usually `Mn` is redefined in terms of that transformer:

```
type Mn = MnT Identity
```

In addition, a type class or trait `MonadMn` defines its primitive operations:

```
class Monad m => MonadMn m where
  op :: ...
```

If all this setup makes any sense at all, every stack that contains `MnT` should be a member of `MonadMn`. The layer with `MnT` could be the top one:

```
instance Monad m => MonadMn (MnT m) where
  op = ... -- implement the functionality
```

Or it could be in a lower layer. In that second case, we need to "peel" one layer away and keep looking. That upper layer can be any of the other transformers:

```
instance MonadMn m => MonadMn (MaybeT m) where
  op = ... -- bridge with MaybeT
instance MonadMn m => MonadMn (ReaderT r m) where
  op = ... -- bridge with ReaderT
instance MonadMn m => MonadMn (StateT s m) where
  op = ... -- bridge with StateT
-- and many more instances
```

Now, let us count. Suppose we have n different transformers, and for each we want to define a type class. Each type class needs n instances, one that does the actual work and the rest that just constitute a bridge. As a result, we need n^2 instances.

Of course, if we restrict ourselves to the transformers introduced in the previous chapter, somebody else has already written that code for us, so we might say that this is not a problem. In fact, since much of this code is boilerplate, we might well write some macros that write the code for us. But this is not regarded as a good solution, since we cannot express it inside the boundaries of our language. Taking it to the extreme, a solution using macros is like making the `map` function a macro. Even worse, if we want to roll our own monads, as we will do from the next chapter on, we might end up having to implement instances for all the existing classes, a tedious and error-prone task.

12.1 MonadTrans and Lift

Up to this point, a monad transformer has been defined as a type constructor that takes a monad and builds a new monad by adding some additional functionality. However, this definition does not guarantee that the operations of the wrapped monad are available in the larger stack in a generic fashion. In other words, there is no generic way to *lift* operations from the wrapped monad. This is the core of the n^2 instances problem: because we do not have a generic way to move operations across layers of a monadic stack, we need to write down how *each* operation moves across *each* layer.

On the other hand, if you write the code that moves operations across `MaybeT`, you see a fair number of commonalities. Note that the implementation in the `mtl` package uses the technique described later in this section — these examples are just illustrative of the most direct solution:

```
instance MonadReader r m => MonadReader r (MaybeT m) where
  -- ask :: MaybeT m r ≡ m (Maybe r)
  ask    = MaybeT $ Just <$> ask -- 2nd ask is from the inner monad
```

```haskell
instance MonadWriter w m => MonadWriter w (MaybeT m) where
  -- tell :: w -> MaybeT m ()
  tell x = MaybeT $ Just <$> tell x -- 2nd tell is from the inner monad

instance MonadError e m => MonadError e (MaybeT m) where
  -- throwError :: e -> MaybeT m a
  throwError e   = MaybeT $ Just <$> throwError e
  -- catchError :: MaybeT m a -> (e -> MaybeT m a) -> MaybeT m a
  catchError m h = MaybeT $ catchError (runMaybeT m) (runMaybeT . h)
```

Lifting an operation across `MaybeT` simply requires wrapping the return value in an additional `Just` layer. At least for the simpler operations, catching errors requires a more convoluted solution (in fact, the next section is devoted to the problems that lifting operations like `catchError` bring to the table). We can abstract this operation via a function:

```haskell
liftThroughMaybeT :: Functor m => m a -> MaybeT m a
liftThroughMaybeT x = MaybeT $ Just <$> x
```

And now we can write `ask = liftThroughMaybeT ask`, `tell = liftThroughMaybeT . tell`, and so on. If we were to repeat this procedure with the `ReaderT` transformer, we would come to the conclusion that there is a generic lifting function:

```haskell
liftThroughReaderT :: m a -> ReaderT r m a
liftThroughReaderT x = ReaderT (\_ -> x)
```

> **Exercise 12.1.** Convince yourself of the previous claim about `ReaderT` by writing and comparing several `MonadMn` instances for `ReaderT`.

The `transformers` package in Haskell and Scalaz in Scala define abstractions for those monad transformers that admit such a lifting operation. In both cases, the chosen name is `MonadTrans`, although the lifting functions are slightly different:

```haskell
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

```scala
trait MonadTrans[T[_[_], _]] {
  def liftM[G[_]: Monad, A](a: G[A]): T[G, A]
}
```

The previously-defined lifting functions for `MaybeT` and `ReaderT` become the `MonadTrans` instances for each transformer:

```haskell
instance MonadTrans MaybeT where
  lift x = MaybeT $ Just <$> x
instance MonadTrans (ReaderT r) where
  lift x = ReaderT (\_ -> x)
```

By a careful usage of `lift`, we can combine operations from different monads. Suppose you have two monadic computations with the following types:

```
f1 ::                         Maybe Int
f2 :: Int -> ReaderT String Maybe Bool
-- We do not really care about what they do at this point,
-- but here are some well-typed possibilities
f1   = return 5
f2 x = return (x == 0)
```

You want to use them in a larger computation, but the following code does not compile:

```
g = do x <- f1
       f2 x
```

```
Couldn't match type ‘ReaderT String Maybe’ with ‘Maybe’
```

Since each monadic block must be assigned a *single* monad, the compiler cannot decide here whether it should be `Maybe` or `ReaderT String Maybe`. The solution is to lift `f1`, that is, to add an additional `ReaderT String` layer on top of `Maybe`:

```
g' = do x <- lift f1
        f2 x
```

By nesting calls to `lift`, we can add as many upper layers as we want to a computation.

The second advantage of `MonadTrans` is that we no longer need to write n^2 instances to obtain a working monad transformer ecosystem. Whereas before we needed n instances per `MonadMn` class — one per monad transformer — we now require just two. Take, for example, `MonadReader`. The first instance is the one claiming that a `ReaderT` layer provides `MonadReader` operations:

```
instance Monad m => MonadReader r (ReaderT r m) where
  ask = ... -- as before
```

The rest of the instances only exist to lift operations across layers. But now we have a generic way to perform this lifting, so they can be summarized into a single instance:

```
instance (Monad (t m), MonadReader r m, MonadTrans t)
         => MonadReader r (t m) where
  ask = lift ask
```

Unfortunately, this elegant solution is at odds with the type inference machinery in Haskell and implicit resolution in Scala. Take a concrete stack, such as `ReaderT Config Maybe a`. When faced with a call to `ask`, the compiler has to decide which instance to apply. Alas, *both* of the instances defined above match:

1. The one for `ReaderT r m` clearly matches, since it has the same shape as the concrete stack. In fact, our aim would be to make this instance the only one that matches.

2. The one for a generic `MonadTrans` also matches, since `ReaderT r` is a monad transformer with a lifting operation. Of course, in most cases this choice does not make sense, since the inner monad is not going to satisfy the `MonadReader r m` constraint.

In practice, compilers restrict themselves to decisions that are right on the nose and report an ambiguity when faced with these two instances. This is the only way they can prevent compilation from entering into an infinite loop. Thus, while elegant, the `MonadTrans` solution to the pollution of instances makes programmers' lives harder, and implementations such as `mtl` prefer hard-coding the n^2 instances.

As a historical note, `MonadTrans` was created before monadic classes. The latter were in fact designed to eliminate the many uses of `lift` that would otherwise cramp computations using monad transformers. For example, if we want to write a computation with the type `WriterT w (ReaderT r) a`, *every* call to `ask` would need to be preceded by a call to `lift`. Even worse, if we decided to change the monadic stack, we might need to change all the calls to `lift` to make them match the new structure.

These days, the main reason to use `MonadTrans` in practice is for its generic `lift` operation, which can be used to inject an operation from a monad into a larger stack. In combination with the unlifting described in the following sections, we now have a nice set of functions to decouple monadic operations from concrete stacks.

12.2 Base Monads: MonadIO and MonadBase

In the previous chapter, we introduced the `MonadIO` type class, which allows us to lift a computation in the `IO` monad into larger stacks:

```
class MonadIO m where
  liftIO :: IO a -> m a
```

This type class looks similar to `MonadTrans`, except that we do not lift a computation from *any* monad into a larger one — we specifically lift `IO` operations.

`IO` is the best-known representative of a set of monads — usually called *base monads* — that do not come with transformer versions, as opposed to `Reader`, `State`, and all the others we discussed in the previous chapter. As a consequence, they can only appear at the *bottom* of a stack, never as upper or intermediate layers. Other examples of base monads are `ST` and `STM`, which were introduced in Chapter 8.

`MonadIO` is a good solution for computations that ultimately run in `IO`, but it is interesting to consider stacks with other base monads. For example, we might be interested in expressing computations that use mutable variables in such a way that they can be used with both `IORefs` and `STRefs`. One possibility is to have dedicated type classes for each base monad — `MonadST`, `MonadSTM`, etc. To fight the duplication that such a solution would lead to, the package `transformers-base` defines a "one size fits all" type class for any base monad:

```
class (Monad b, Monad m) => MonadBase b m | m -> b where
  liftBase :: b a -> m a
```

An instance of this type class relates two monads, `b` and `m`, and states that operations in `b` — the base monad — can be lifted to `m` — the complete stack. For example, there is an instance `MonadBase IO (ReaderT String IO)` that declares `IO` as the base monad of the whole stack.

Exercise 12.2. Write the code for the following instances:

```
instance MonadBase IO IO
instance MonadBase (ST s) (ST s)
```

Hint: notice that the types in both arguments coincide. Then write the code for the instance that lifts an operation from a monadic stack `m` one additional layer `t`:

```
instance (MonadBase b m, MonadTrans t) => MonadBase b (t m)
```

At the end of Chapter 8, we introduced the `PrimMonad` type class, which defines those monads that have a notion of mutable variables, such as `IO` and `ST s`. Using that type class, we could create computations that are independent of the kinds of variables we use:

```
updateCounter :: PrimMonad m
              => Int -> MutVar (PrimState m) Int -> m ()
```

Now, we can go one step further and represent computations independently of the monad stack, just by requiring the base monad to be primitive:

```
updateCounter :: (MonadBase p m, PrimMonad p)
              => Int -> MutVar (PrimState p) Int -> m ()
```

In fact, almost any computation (we will discuss why some functions need stronger guarantees in the next section) that previously worked on `IO`, `ST`, or `STM` could be generalized to work over any stack, provided that the base monad coincides with it:

```
putStrLn ::                    String -> IO ()
putStrLn :: MonadIO       m => String ->  m ()
putStrLn :: MonadBase IO m => String ->  m ()
```

Alas, Haskell's `base` library defines the first version, so we need to write `liftIO` or `liftBase` explicitly every time we want to use `putStrLn` in a larger stack — unless we decide to swap `base` with `lifted-base`, a package that provides the same functionality but with more general types, by exchanging explicit `IO` with `MonadBase IO`.

12.3 Lifting Functions with Callbacks

Giving transformers the ability to lift operations from inner monads via `lift` is a good first step, but it only works for those scenarios where the monad appears *only* in the *return type* of the function. However, consider an operation such as `mplus`:

```
mplus :: Monad m => m a -> m a -> m a
```

Here, the monad `m` also appears in the argument position. Lifting this operation should change *all* occurrences of `m` to `t m`, resulting in `t m a -> t m a -> t m a` (as opposed to changing only the result type, `m a -> m a -> t m a`). Alas, `lift` is not enough to get us there. Suppose we try to use it anyway to lift `mplus` to obtain the desired `mplus'`:

```
mplus' :: (MonadTrans t, Monad m) => t m a -> t m a -> t m a
mplus' x y = lift (mplus x y)
```

This code is ill-typed. By using `lift`, we can perform an operation in the wrapped monad `m`, `mplus` in this case, and put its result inside `t`. This means that the `mplus` in the right-hand side of the definition is working with its original type, `m a -> m a -> m a`. But `x` and `y` have type `t m a` instead, so we cannot use them directly.

In essence, what we are looking for is a dual to the `lift` function that works in the opposite direction. For that reason, we usually call such an operation an *unlifting*. Whereas `lift` has type `m a -> t m a`, such an `unlift` would be `t m a -> m a`. This unlifting function should allow us to write `mplus` as follows:

```
-- Note: unlift is *not yet* defined
mplus' :: (MonadTransUnlift t, MonadPlus m) => t m a -> t m a -> t m a
mplus' x y = lift $ mplus (unlift x) (unlift y)
```

As we will see in a moment, however, it is not possible to have an `unlift` operation that works as seamlessly as the code above suggests. There are several solutions to this problem, which trade off power (how many transformers you can use and how many operations can be lifted) against simplicity.

Another source of functions that require unlifting are those that require a callback function as an argument. The simplest example is catching an error using `MonadError` operations:

```
catchError :: MonadError e m => m a -> (e -> m a) -> m a
```

If we want to lift this operation through a transformer, our aim is to write a version with the following type:

```
catchError' :: t m a -> (e -> t m a) -> t m a
```

As in the case of `mplus`, those `t m` types appearing in the arguments to the function make our life much harder. To be able to use the original `catchError` as part of the new definition, we need a way to unlift the arguments into the base monad:

```
-- Note: unlift is *not yet* defined
catchError' c h = lift $ catchError (unlift c) (unlift . h)
```

The same pattern of callbacks as arguments is common when dealing with resource management in `IO`, as we discussed in Chapter 9. Take, for example, the `bracket` function:

```
-- This is the original type
bracket  :: IO r -> (r -> IO b) -> (r -> IO a) -> IO a
-- This is the type we want to achieve
bracket' :: MonadUnliftIO m
         => m  r -> (r ->  m b) -> (r ->  m a) ->  m a
```

We can use `liftIO` or `liftBase` — depending on whether we want to use the more specific `MonadIO` or the more generic `MonadBase` — to inject the return value of type `IO` into a larger stack. For the arguments, though, we need the converse operation: transforming a value from the monadic stack `m` into the simpler `IO`. Otherwise, we would not be able to call the `bracket` function on `IO`. Such a transformation is the *unlifting* we have been looking for. In the following sections, we discuss several ways to implement it, each with different trade-offs.

12.3.1 Targeted Solutions

One of the advantages of `MonadTrans` is that by providing only one additional operation, you unleash a lot of power to move operations among transformers in a generic and reusable way. Unfortunately, providing an `unlift` operation with similar characteristics leads to complicated designs that push the type system to its limits. As a consequence, some libraries in the Haskell ecosystem have decided to target only one specific area of unlifting, resulting in simpler-to-understand interfaces.

We already introduced the type class that targets the error handling scenario, `MonadError`:

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

```
trait MonadError[F[_], E] extends Monad[F] {
  def raiseError[A](e: E): F[A]
  def handleError[A](fa: F[A])(f: E => F[A]): F[A]
}
```

If you look at the code that implements this type class for each transformer, you will find no abstraction. Each transformer defines its own, independent implementation of how `catchError` is lifted through it.

It might come as a surprise, but GHC defines an exception mechanism much along the lines of Scala, Java, or C#. The main difference is that whereas exceptions can be thrown at any point in a program, you need to be in the `IO` monad to handle them. The `exceptions` library defines type classes similar to `MonadError` but targeted at these exceptions instead of at mere errors:

```
class Monad m => MonadThrow m where
  throwM :: Exception e => e -> m a
class MonadThrow m => MonadCatch m where
  catch :: Exception e => m a -> (e -> m a) -> m a
class MonadCatch m => MonadMask m where
  mask :: ((forall a. m a -> m a) -> m b) -> m b
```

The last operation, `mask`, disables the transmission of exceptions raised by other threads in the program. It is essential, however, to ensure diligent resource management even in the presence of multiple running threads. Since the proper usage of `mask` is tricky, we suggest that you never call the function explicitly in practice but rely instead on higher-level interfaces designed for your intended use cases.

As discussed above, the other main source of functions with callbacks is `IO`. The `unliftio` package defines a specific type class, `MonadUnliftIO`, for unlifting from several layers up to `IO`. In the actual implementation, there is another method in the class called `askUnliftIO` that provides the same functionality but with a different signature. In this section, we restrict ourselves to `withRunInIO`, mostly due to pedagogical rather than technical reasons:

```
class MonadIO m => MonadUnliftIO m where
  withRunInIO :: ((forall a. m a -> IO a) -> IO b) -> m b
```

A call to `withRunInIO` is usually written in the form of an anonymous function. The argument to this function — which takes the type `forall a. m a -> IO a`

— represents how to move down the monadic ladder all the way to `IO`. Using that helper, we have to build an `IO` operation, which is then lifted to the entire stack.

Take for example the `bracket` function we described at the beginning of this section. Its original type is `IO r -> (r -> IO b) -> (r -> IO a) -> IO a`. The following expression lifts the operation to work on any monad that supports unlifting to `IO`:

```
bracket' :: MonadUnliftIO m
         => m r -> (r -> m b) -> (r -> m a) -> m a
bracket' acq release use
  = withRunInIO $ \run -> bracket (run acq)
                                  (run . release)
                                  (run . use)
```

As we explained above, each of the callback functions — acquisition, release, and resource usage — is unlifted via `run`. The entire computation is wrapped inside `withRunInIO`.

> **Exercise 12.3.** Write the lifted version of the following function, using `MonadUnliftIO`:
>
> ```
> withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r
> ```
>
> Note that you have to define *both* the new signature and its implementation.

`MonadUnliftIO` is the first example of a class of transformers that is not inhabited by all the transformers we introduced in the previous chapters. In fact, the set of stacks that supports `withRunInIO` is small: it only allows `ReaderT` layers down to a base `IO` at the bottom. We will describe the problem in depth later in this chapter, but in brief, a monad for which the execution of the same computation twice may change its visible outcome cannot have a well-behaved `MonadUnliftIO` instance. This includes transformers that keep an internal state or provide error handling functionality. For that reason, we say that `MonadUnliftIO` only works for *stateless* monads.

As in the case of `lifted-base`, the package `unliftio` provides versions of `IO` operations in the base library already lifted and ready to use. Note, however, that `unliftio` is, at the moment of writing, a young package, so the ecosystem around it is still in flux.

12.3.2 A Solution for Stateless Monads

There are two details worth mentioning about `MonadUnliftIO`. First, it only targets monad stacks with `IO` at the bottom. The second is that this unlifting operation is

only available for stateless monads. The `monad-unlift` package retains the latter restriction while dropping the former.

During this chapter, we have been working with two different ideas:

1. Lifting through one transformer layer, for which we mainly use `lift` from the `MonadTrans` type class. In this case, if we want to lift an operation through several layers, we need the same number of nested calls to `lift`.

2. Lifting an operation in the base monad — usually `IO` — through all the layers until we reach the one we require. This lifting is available via `liftIO` from `MonadIO` or, in general, via `liftBase` from `MonadBase`.

This dichotomy is also reflected in a duo of type classes for unlifting in the `monad-unlift` library, `MonadTransUnlift` and `MonadBaseUnlift`:

```
-- Dual of lift from MonadTrans
askUnlift     :: (MonadTransUnlift t, Monad m)
              => t m (Unlift t)
-- Dual of liftBase from MonadBase
askUnliftBase :: MonadBaseUnlift b m
              =>   m (UnliftBase b m)
```

The `MonadUnliftIO` type class introduced in the previous section can be seen as a particular case of `MonadBaseUnlift`, for when the base monad is `IO`.

The usage of this library is similar to that of `unliftio`, except that the `ask` operations only provide you with a function to unlift, whereas the final lift needs to be done manually. Let us see how we would write a version of `bracket` using this functionality:

```
bracket' :: MonadBaseUnlift IO m
         => m r -> (r -> m b) -> (r -> m a) -> m a
bracket' acq release use
  = do UnliftBase run <- askUnliftBase
       liftBase $ bracket (run acq)
                          (run . release)
                          (run . use)
```

In the code above, we can see that the result of `askUnliftBase` is a small data type containing the unlifting function. The reason why the function cannot be returned directly, instead of wrapped inside `UnliftBase`, has to do with specifics of the type inference process in Haskell. In the call to `bracket`, we use `run` repeatedly to unlift the operations, but the result of that call is of type `IO a`, hence the final `liftBase` to pull it again under the reign of the full monad.

In order to illustrate the use of `MonadTransUnlift`, let us lift the `catchError` operation through stateless transformers. The code in this case is similar to what we saw previously: we call `askUnlift` to obtain the unlifting function, run the

original operation `catchError` after unlifting its parameters, and finally go back to the main monad by using `lift`:

```
catchError' :: (MonadTransUnlift t, MonadError e m, Monad (t m))
            => t m a -> (e -> t m a) -> t m a
catchError' c h = do Unlift run <- askUnlift
                     lift $ catchError (run c) (run . h)
```

Stateless `StateT`**.** One interesting idea from the `monad-unlift` ecosystem is how it works around the restriction of monads being stateless to provide its own versions of `Writer` and `State`. The trick is simple: use a `Reader` monad that holds a mutable variable. The type of references you want to use (`IORef`, `STRef`, and so on) is given as an additional variable:

```
data WriterRefT ref w m a = WriterRefT (ref w -> m a)
data StateRefT  ref s m a = StateRefT  (ref s -> m a)
```

Since in both cases, we just hold a mutable variable in the environment, the implementation ends up being the same in both cases. However, the interface that `monad-unlift` provides is different: `MonadWriter` in the former case and `MonadState` in the latter. In order to execute the computation, we need to choose the kind of variables to use and thus the base monad in which the operations are to be performed:

```
runStateIORefT :: (Monad m, MonadBase IO     m)
               => StateRefT IORef     s m a -> s -> m (a, s)
runStateSTRefT :: (Monad m, MonadBase (ST p) m)
               => StateRefT (STRef p) s m a -> s -> m (a, s)
```

In principle, this idea of simulating state by using mutable variables is safe. Furthermore, in those environments where we are using `IO` anyhow, it can provide a good performance boost. Still, the use of these transformers remains quite rare in practice.

12.3.3 A General Solution

This is the end of our trip of increasing power — and more convoluted design — in unlifting. The package `monad-control` provides two type classes, `MonadTransControl` and `MonadBaseControl`, which allow us to lift and unlift across almost every transformer. But with great power comes great responsibility: we can write code that exhibits a surprising — some would say plainly wrong — behavior, without even noticing.

The basic idea of `monad-control` is similar to `unliftio` and `monad-unlift`: it provides a way to get a function that performs the unlifting. However, once we depart from the world of stateless monads, a function with the type `t m a -> m a` is not enough, for the following reasons:

1. If we are unlifting from a `WriterT` transformer, we need to give back both the result value and any changes in the value that are accumulated along the way. We therefore require a function with the type `WriterT m a -> m (a, w)`, instead. The same holds for `StateT`. The function in the unlifted position also ought to define how the state changes.

2. In the case of `MaybeT` or `ExceptT`, the unlifting operation should have a corresponding type, `MaybeT m a -> m (Maybe a)` or `ExceptT e m a -> m (Either e a)`, respectively. This tells us whether the inner operation should be lifted back as a success or a failure.

The connection between each transformer and the additional information it needs in its unlifting operation is given by the `StT` type family. Roughly, a type family in Haskell works as a function over types, with the difference that each of the equations of the function — called *instances* in this scenario — may be defined in a different module. Here are the main instances for `StT` taken from the `monad-control` documentation:

```
StT IdentityT   a = a
StT (ReaderT r) a = a
StT MaybeT      a = Maybe a
StT (StateT s)  a = (a, s)
StT (WriterT w) a = Monoid w => (a, w)
```

This offers another reason why `IdentityT` and `ReaderT` are called stateless transformers. Their `StT` instances add no additional state to the return value of an unlifted operation.

Lifting and unlifting via `monad-control` is done in three steps:

1. Obtain the unlifting function using `liftWith`. In contrast to `unliftio` and `monad-unlift`, the obtained function is not of the form `t m a -> m a` but instead a slightly modified `t m a -> m (StT t a)`, to account for the additional information saved by the transformer.

2. Run your operations in the lower monad, unlifting operations as required. In most cases, the code looks similar to other approaches, but you end up returning something of the form `m (StT t b)`, instead of simply `m b`. For example, when performing the operation in the `StateT s` transformer, you return a value `m (b, s)`.

3. Reconstruct the internal state of the transformer by calling `restoreT` with the additional information returned by the unlifting function. This step is crucial, though unenforced by the type system, for making the lifted version of an operation work as expected.

Let us look at our favorite examples, starting with `catchError`. Note that in the following code, the composition with `run` changes the type of the call to `catchError` to be of the form `m (StT t a)`. For example, if we were running `catchError` below a `StateT s` layer, this type is `m (a, s)` to account for the additional internal state. Instead of just returning the result of that operation, we first need to put back this internal state in the `StateT s` layer of the transformer by calling `restoreT`:

```
catchError' :: (MonadTransControl t, MonadError e m, Monad (t m))
            => t m a -> (e -> t m a) -> t m a
catchError' c h
  = do x <- liftWith $ \run -> catchError (run c) (run . h)
       restoreT (return x)
```

The same pattern works for operations that ought to be lifted from a base monad, instead of through a transformer layer. Here is the lifted version of `withFile`, which takes care of acquiring and releasing a file handle, where the actual work to be done is given as a function:

```
withFile' :: MonadBaseControl IO m
          => FilePath -> IOMode -> (Handle -> m r) -> m r
withFile' fp m r
  = do x <- liftBaseWith $ \run -> withFile fp m (\h -> run (r h))
       restoreM x
```

This combination of `liftBaseWith` and `restoreM` is so common that the library defines a synonym for it, `control`. Using it, the lifting and unlifting are almost transparent:

```
withFile' :: MonadBaseControl IO m
          => FilePath -> IOMode -> (Handle -> m r) -> m r
withFile' fp m r = control $ \run -> withFile fp m (\h -> run (r h))
```

Sometimes, things can get a bit tricky, though. Consider the following (incorrect) implementation of `bracket'`, following the pattern we used for `withFile'`:

```
bracket' :: MonadBaseControl IO m
         => m r -> (r -> m b) -> (r -> m a) -> m a
bracket' acq release use
  = do x <- liftBaseWith $ \run ->
           bracket (run acq) (\r -> run (release)) (\r -> run (use))
       restoreM x
```

It looks fine: we just unlift every operation by means of the `run` function provided by `liftBaseWith`, and eventually we restore the state with `restoreM`. However, the compiler complains:

```
Couldn't match type ‘r’ with ‘StM m r’
In the first argument of ‘bracket’, namely ‘(run acq)’
```

The problem is that by applying `run` to the acquisition function, we no longer get back a resource of type `r` but a value of type `StM t r` that also holds the transformer state. The use and release functions do not expect such a value, only the resource `r`. Thus, before feeding in the resource, we need to remove the transformer state by means of `restoreM`:

```
bracket' :: MonadBaseControl IO m
         => m r -> (r -> m b) -> (r -> m a) -> m a
bracket' acq release use
  = do x <- liftBaseWith $ \run ->
         bracket (run acq)
                 (\r -> run (restoreM r >>= release))
                 (\r -> run (restoreM r >>= use))
       restoreM x
```

But even though this implementation type checks, it is *still wrong*. There are three moments where some transformer state might be modified, corresponding to the three functions that we unlift via `run`. The result of `bracket` returns the value from the `use` branch. This means that the effects that may have happened during acquisition and release are completely ignored!

The excellent talk, *Everything you didn't want to know about monad transformer state* [Snoyman, 2017] contains a simple piece of code that showcases this problem:

```
foo :: StateT [String] IO ()
foo = bracket' (modify (++ ["1"])) -- acquire
               (modify (++ ["3"])) -- release
               (modify (++ ["2"])) -- inner
main = do res <- execStateT foo []
          print res
```

Most people would expect that after the entire acquisition/use/release cycle, the value of `res` would be `["1","2","3"]`. Instead, you get `["2"]` printed to the screen, witnessing that only the changes to the state made by the use function remain.

The takeaway from this problem is that trusting `monad-control` too much may lead to surprising results. My recommendation is to keep unlifting to stateless transformers, or use a package targeted to a specific kind of function, such as `exceptions` and `unliftio`.

12.4 More on Manipulating Stacks

The `MonadTrans` type class allows us to lift an operation from a monad `m` into the larger monad `t m`. However, `lift` is restricted to adding layers *on top of* the

existing ones. But what if the layer we want to add has to be not at the top, but at the bottom of the stack or in an intermediate position? The `mmorph` package provides a set of operations to handle those cases.

There is one easy case: if there is *no* layer at all — that is, the computation is pure — we can create that single layer using `return`, since it has the type `a -> m a`. It is interesting to create a version of `return` that converts from the `Identity` monad into any other monad, for reasons we will see below:

```
-- Identity was defined in previous chapters
newtype Identity a = I { runIdentity :: a }
generalize :: Monad m => Identity a -> m a
generalize (I x) = return x
```

Now, let us consider the converse situation to the one at the beginning of this section. Remembering that `Reader`, `State`, and so on are really synonyms for transformers applied to the `Identity` monad, the monadic computations you want to compose are as follows:

```
g1 ::        Reader  String          Int
   -- or     ReaderT String Identity Int
g2 :: Int -> ReaderT String Maybe    Bool
```

The intuition here is that we need to apply `generalize` to swap the first `Identity` for `Maybe`, but we need to do so under the common `ReaderT` prefix. Such an operation is called `hoist` in the `mmorph` package and has the type:

```
hoist :: Monad m => (forall a. m a -> n a) -> t m b -> t n b
```

This type looks a bit more complicated than usual, but reading it will show you that it encodes the same idea. You give as an argument a function that converts monadic computations from `m` to `n`, and `hoist` takes care of applying it within the scope of the transformer `t`. One possible argument is `generalize`:

```
hoist generalize :: Monad n => t Identity b -> t n b
```

In conclusion, the code we need to write to use `g1` and `g2` becomes:

```
do x <- hoist generalize g1
   g2 x
```

The combination of `hoist` and `lift` becomes useful once you start working with stacks of three or more layers (although humongous monad stacks are usually a code smell). Imagine you need to compose two computations with these types:

```
h1 :: StateT Counter                 Maybe  Int
h2 :: StateT Counter (ReaderT Config Maybe) Bool
```

Operation	*Action*
`lift`	Add a layer on top.
`generalize`	Add a layer at the bottom.
`hoist f`	Apply an operation `f` one layer deeper.
`squash`	Turn two identical layers into one.

Table 12.1: Summary of morphisms for manipulating monadic stacks

You need to insert a `ReaderT Config` layer into the type of `h1` before proceeding. You cannot use `lift` directly, because that would produce a `ReaderT Config (StateT Counter Maybe)` — the stack in reverse order. The `generalize` function does not help either, since `ReaderT` does not have to appear at the bottom. The solution is to, first, go down one layer using `hoist`:

```
hoist :: (forall a. m a -> n a)
      -> StateT Counter m n -> StateT Counter n b
```

And *then* introduce the additional layer via `lift`:

```
hoist lift h1 :: StateT Counter (ReaderT Config Maybe) Int
```

The trick is always the same: `hoist` until you get to the layer you want to modify, and then use `lift` to insert a new layer, or use `generalize` to exchange an `Identity` for any other monad.

The `mmorph` library provides yet another operation on monadic stacks, which is not that common in practice. The function `squash` turns two identical layers into a single one:

```
squash :: Monad m => t (t m) a -> t m a
```

This operation resembles the `join` operation for monads, which also turns a nested monadic type into a single type, `m (m a) -> m a`, but at the level of monad transformers. This observation actually has a deep significance: it means that transformers supporting `squash` are "monads over monads." We should not go down the rabbit hole this time, though. For the full significance of this topic, we prefer to keep you waiting until Chapter 17.

12.4.1 Monad Morphisms

Not every implementation of `lift` that has the right type is fine to use within monadic stacks: it needs to respect the monadic structure. In particular, we should ensure that binds and returns are respected:

```
lift $ do x <- m          ≡      do x <- lift m
          f x                        lift (f x)

lift (return x)           ≡      return x
```

In this case, we say that `lift` is a *monad morphism* or *monad transformation*. Another way in which we can formulate these laws is by using the Kleisli arrow composition operator, `(<=<)`:

```
lift . return       ≡      return
lift . (f <=< g)    ≡      lift . f <=< lift . g
```

The function `lift` is not the only interesting monad morphism out there. All the functions in Table 12.1, which we have introduced to manipulate monadic stacks, are also examples of monad morphisms. Note that `hoist f` is a monad morphism only if its argument `f` is a monad morphism, too.

Another type class with instances that must satisfy the monad morphism laws is `MonadUnliftIO`. If we call `run`, the function that performs the unlifting via `withRunInIO`, we can reformulate the laws as follows:

```
run (return x)      ≡      return x
run . (f <=< g)     ≡      run . f <=< run . g
```

Just asking for those laws to be satisfied is not enough to rule out those monads that are not stateless, however. But `MonadUnliftIO` comes with an additional law, which ensures that any context in the monad is respected through the unlifting-lifting combination:

```
withRunInIO (\run -> liftIO (run m))      ≡      m
```

Roughly, this law tells us that if we just unlift the computation and lift it again, things are kept unchanged. But on non-stateless monads, `withRunInIO` would need to "save" the state, and any further modification done by `m` would be lost.

In this part of the book, we have considered the issue of combining operations from different monads into a single computation. As we have seen, the task is far from trivial. The most common solution — monad transformers — requires some advanced programming patterns. A few pages further on, we change the question completely, asking whether we can *define* our own monads and why this is useful. Unexpectedly, the issue of combining custom monads is easier to resolve than combining the usual `State`, `Reader`, `Maybe`, and friends.

Part IV

Rolling Your Own Monads

13 Defining Custom Monads

Up to this point, we have only looked at monads that have already been defined for us. This provides the benefit of having a uniform way to look at many disparate notions of computation. In this final part of the book, we go one step further: monads are a general tool for modeling behaviors in software, and thus you should also be able to define your *own* monads in your own code. This chapter introduces the basic concepts we use to describe custom monads and three different ways — final, initial, and operational — in which they can be turned into code. As part of our journey, we will explore the trade-offs between them.

As a side note, the terminology used in this chapter is not yet set in stone, and in many cases, different languages and different projects may use different names to refer to the same idea. This has the unfortunate consequence that drawing relationships between different libraries and their documentation becomes unnecessarily difficult, especially for the uninitiated.

Note also that the notion of final and initial style descriptions is not tied directly to monads, as every data type can be described in a similar fashion. As usual in this book, we focus in this chapter only on the application of those ideas to monads, in particular, to the crafting of custom monads.

13.1 Introduction

Let us begin right away with what will be a running example in this part of the book: a new monad that encodes the rules of tic-tac-toe, a well-known pen-and-paper game for two players. The game starts with an empty 3×3 board, and players take turns drawing a mark — usually a circle or a cross — in an unclaimed space. They try to win by putting their marks in three, consecutive positions on the board in a row, in a column, or diagonally. If all positions have a mark but no three consecutive positions have the same mark, the game ends in a draw.

A computation in the `TicTacToe` monad describes the *actions* that a player takes in a particular run of the game. Since the gameplay has an inherently sequential nature, we can only decide what to do next after each move of our opponent. The concept of a monad fits very well into that idea. However, having a `bind` and a `return` is not enough to describe a game of tic-tac-toe. We need additional functionality in order to:

- Query the state of the board, whether each of the positions is already taken and by whom:

  ```
  data Position = Position ... -- exercise, see below
  data Player = O | X  deriving (Eq, Ord)
  info :: Position -> TicTacToe (Maybe Player)
  ```

- Indicate that we want to claim an empty position.

- Know whether we have already won or lost:

  ```
  data Result = AlreadyTaken { by :: Player }
              | NextTurn
              | GameEnded { winner :: Player }
  take :: Position -> TicTacToe Result
  ```

We refer to the functionality that a particular monad provides as the *primitive operations* of the monad — we sometimes drop "primitive" and just speak of the operations of the monad. For example, `get`, `put`, and `modify` are the operations of the `State` monad, and `mzero` and `mplus` are the operations of `Maybe` and `List`. By putting together chains of primitive operations, we can describe an entire computation.

> **Exercise 13.1.** Complete the definition of the `Position` data type. A value of this type represents a position in the 3×3 board used to play tic-tac-toe.

At this point, you might be convinced that we *can* define a monad for playing tic-tac-toe. The natural question then is whether we *should* define one — what do we gain by modeling our problem with a monadic computation, instead of hacking our way directly to the solution? There are reasons for doing so that come from two areas: *Domain-Driven Design* (also known as DDD) and *Domain-Specific Languages* (usually shortened to DSLs). DDD provides a philosophy and framework for how to design software, while DSLs are a way to turn those ideas into runnable code.

Every piece of software we write aims to describe and implement the processes of a particular *domain*. An air reservation back-end works with flight numbers, routes and legs, and traveler information. A food delivery back-end works with restaurants, routes, and the scheduling of drivers. As developers, our main task is to "translate" information about how a domain works into a language that a computer

can understand. This is not an easy task: clients use high-level descriptions, which usually leave many details implicit. On the other hand, a programming language needs everything to be defined in terms of classes, types, functions, or predicates (depending on the language). DDD and DSL claim that we can build better software by *reducing* the amount of translation that we need to perform. In essence, we should make the domain the center of our development and model every process in the *language* of the domain, not in the mental framework provided by our programming language.

DDD goes one step further, up to the realization that even in a single piece of software, there might be different — possibly overlapping — domains for each subsystem. In our example above, the food delivery back-end may use a language to describe relations with providers different from that used to describe client invoices. Those different sub-domains may share terms, but that is only a superficial similarity: only within each domain does a term have a definite and concrete meaning. DDD calls each of these domains with concrete and well-understood terms a *bounded context*. We will stop here, but the interested reader can find lots of literature about DDD, where tools for modeling and relating bounded contexts are described in depth.

If every bounded context has its own terms, a natural decision is to build a *different language* for each domain at hand. This is what the DSL community aims to do: provide tools to ease the development of mini-languages with a clear goal. The concept is not new: CSS and SQL are DSLs defined for the realms of styling and layout, and database querying and updating, respectively. In the world of functional programming, it is common to work with *embedded* DSLs, which are nothing more than libraries with a nice, intuitive interface. The main advantage of embedded DSLs is that a single piece of code may use several DSLs at once, increasing the level of abstraction.

DDD teaches us that every domain is made out of three different kinds of information: pure data values, objects with identity, and processes or services. Monads are a great way to describe this third kind.* And this is where programming languages without a convenient way to express monads fall short.

In conclusion, our goal is the same as for any development technique: increase productivity and reduce the number of bugs. By “explaining to the computer” a particular approach to playing tic-tac-toe in the same terms that human players would do, we aim to bridge the gap between those who have the ability to code and those who have the ability to play tic-tac-toe well. That way, we can benefit much more from expert players than if those experts would have to be introduced to programming.

*Functional programming also models pure data nicely by leveraging immutability, and it provides tools such as Software Transactional Memory to talk about objects with identity.

13.1.1 Syntax and Interpretation

One important distinction, which we discussed previously in this book, is that *describing* a computation is different from *executing* a computation. Sometimes, this difference is obvious: building a SQL query does not mean that its effects make their way to any database. Moreover, many languages do not distinguish between "this action requires input from the console" and actually asking a user for input from the console. Custom monads, as described in this and forthcoming chapters, have this distinction at their core.

Every custom monad we will define comes with a set of *operations* that it supports, which provide the building blocks to create complete computations. People also refer to this set as the *algebra* of the monad or the *syntax* of the monad ("algebra" is mostly heard in Scala circles, whereas "syntax" is heard in Haskell and academic circles).

The term "syntax," which comes from linguistics, suggests that knowing the operations of a monad tells you which are the "well-formed sentences," the computations that are grammatically correct. Alas, syntax is nothing without a real meaning, a "semantics." A *semantics* for a monad, also known as a *handler* or an *interpreter*, defines how to execute each computation built from the monad syntax.

Note here a small difference. We talk about *the algebra* of the custom monad but also about *an interpreter* for it. In fact, one of the great benefits of this separation between syntax and semantics is that a single description of a computation may be interpreted in many different ways. For example, a `TicTacToe` value can be executed as an opponent in a video game, taking as input the actions of the player. Even more, different handlers could provide different interfaces, like a web page or a desktop application. That same `TicTacToe` computation could also be executed within a testing framework, in order to check that the logic of the game functions as expected.

Another advantage of this separation is that we can implement analyses and optimizations of the syntax of the monad that apply regardless of the interpretation we make of it. Those *transformations* ought to capture core invariants of the domain at the highest level. For example, in tic-tac-toe, we know that claiming the same position twice would make the second one fail. Or in the `State` monad, a `get` after a `put x` is guaranteed to return `x`.

Note that the most common monads — `State`, `Reader`, `Maybe`, and so on — do not follow this strict separation. The description of a computation cannot be separated from the way it executes. Each monad provides only one interpretation, which is the only way to run the monad. Think of `runState`, `runReader`, and so on.

We mentioned, in the introduction to this chapter, that there are different approaches for implementing custom monads in languages like Haskell and Scala. For each variation, we will describe how the syntax of the monad is reflected in code, how each interpretation is built and run, and how transformations may be applied to a computation.

Exercise 13.2. Think of a domain in which you want to model a behavior or process (a game is a good example if you cannot find another one). Think about all the parts you would need to implement a custom monad for that behavior: the primitive operations and at least a couple of different interpretations. This is an exercise that is better done with a colleague or even in a larger group.

13.1.2 Restrictive Monads

Describing behaviors and processes within a DSL is our main goal in developing a custom monad. But there is another interesting goal, which is providing a *restricted view* of a set of operations. The prime example is introducing a monad `FS` for file system operations to replace the broader `IO`. To keep it simple, our custom `FS` monad provides just two operations. The types are based on the similar operations in the `System.IO` module, although we have changed the return type to `Either` to be able to handle error conditions:

```
writeFile :: FilePath -> String -> FS (Either FSError ())
readFile  :: FilePath           -> FS (Either FSError String)
```

By construction, we want to ensure that a computation in the `FS` monad may only read and write files as side effects. Another example is `ST`, which provides just the subset of `IO` operations to create and manipulate mutable references.

In general, a restrictive monad — for lack of a better name — comes only with a subset of the operations that the larger monad provides. Another way to phrase this is that the algebra of the restrictive monad is a *subalgebra* of the larger one. The main interpretation of the restrictive monad merely dispatches the operations in the larger monad. For example, executing an `FS` operation simply entails translating the operations into the corresponding `IO` interpretations and executing them.

Running the examples. In this and the following chapters, many names are shared between Haskell and Scala code blocks. For example, in both cases we use `FilePath` to refer to paths in the file system. The purpose is pedagogical: we want to highlight the similarities between both languages.

However, this means that code using those names is not directly executable. This can be solved by introducing type synonyms either to the real Haskell types:

```
-- All from the System.IO module
type FilePath = String
type FSError  = IOError
```

Or to their Scala counterparts:

```
type FilePath = File
type FSError  = Exception
```

13.2 Final Style

Final style is one of the primary methods for embedding monadic languages into functional languages that support higher-kinded polymorphism, such as Haskell and Scala. The main driving force behind final style monads is Oleg Kiselyov.* This style bears many resemblances to `mtl` type classes such as `MonadReader` but adds the ability to have more than one interpretation per monad. A major advantage of final style, as we will see in Chapter 14, is the extensibility of the monads: it is very easy to combine operations from different monads into a single computation. This style is sometimes known as *final tagless* or, in Scala circles, *object algebras*.

In final style, the *syntax* — or *algebra* — of your custom monad is defined by a *type class* or *trait*, depending on the programming language you are using. Every position where your monad would appear is replaced by the variable you choose in the type class or trait. For example, here is the syntax of our `TicTacToe` monad in Haskell and Scala:

```
class TicTacToe m where
  info :: Position -> m (Maybe Player)
  take :: Position -> m Result
```

```
trait TicTacToe[F[_]] {
  def info(p: Position): F[Option[Player]]
  def take(p: Position): F[Result]
}
```

A *computation* that uses these primitive operations receives a polymorphic type that reflects this fact as a `TicTacToe` constraint. This is again quite similar to the type one would get with `mtl`-style monad transformers:

```
takeIfNotTaken :: (Monad m, TicTacToe m)
               => Position -> m (Maybe Result)
takeIfNotTaken p = do i <- info p
                      case i of
                        Just _  -> return Nothing
                        Nothing -> Just <$> take p
```

```
def takeIfNotTaken[F[_]: TicTacToe: IO]
      (p: Position): F[Option[Result]] = { ??? }
```

In this case, we have decided to keep the `TicTacToe` primitives separate from those of `Monad`. This means that most computations get both constraints in their types. Another possibility is to make `Monad` a superclass of `TicTacToe`:

*Readers interested in going really deep into this topic should visit his web site: `http://okmij.org/ftp/tagless-final/`, at the time of writing.

```
class Monad m => TicTacToe m where ... -- As before
```

The drawback is that using any `TicTacToe` operation drags the `Monad` requirement in. This is not always desirable, for example, if you are composing operations using only `Applicative` combinators.

The final ingredient is how to *interpret* a final style monad: as an *instance* of the corresponding type class or trait. Since we may have more than one instance for a single type class or trait, we naturally retain the ability to have *multiple interpretations* of the same syntax. That way, the two elements of a custom monad — algebra and interpretations — correspond quite directly to the two elements of the type class mechanism — a type class and several instances, respectively.

There is only one, slight caveat. The way we define the `TicTacToe` type class requires the instances to apply to type constructors — this is clearer in the Scala code, where the variable is annotated with the number of its type arguments as `F[_]`. Furthermore, interpreting a computation such as `takeIfNotTaken` requires a `Monad` instance in addition to `TicTacToe`, since otherwise we would not be able to interpret either the sequential composition of operations or `return`. As a consequence, an interpretation needs to map your custom monad *into another monad*. This is a constant in most interpretations, regardless of the style you use to implement it.

One possible way to run a `TicTacToe` computation is to keep the current state of the board as part of a `State` and use the console to ask for the input of the other player. We need a third piece of information that does not change, one that represents which player our computation refers to. We can keep that configuration in a `Reader` monad. In our terms, we can interpret our monad using `ReaderT Player (StateT Board IO)`:

```
-- The type Map comes from the 'containers' package
type Board = Map Position Player

instance TicTacToe (ReaderT Player (StateT Board IO)) where
  info p = lookup p <$> get
  take p = do l <- info p
              case l of
                Just p' -> return AlreadyTaken { by = p' }
                Nothing -> do me <- ask
                              modify (insert p me)
                              liftIO (putStrLn "Your next move:")
                              ...
```

Since any computation described using only operations from the `TicTacToe` type class is polymorphic over the interpretation, we can execute it simply by instructing the compiler which instance to use. Let's say we have a function that implements a given strategy to play the game:

```
logic :: (Monad m, TicTacToe m) => m Result
```

Since `ReaderT Player (StateT Board IO)` implements the `TicTacToe` and `Monad` classes, we can just use `logic` as a value of this type. In turn, that means that we can execute the game by then running each of the transformer layers:

```
runTicTacToe :: IO (Result, Board)
runTicTacToe = runStateT (runReaderT logic O) emptyBoard
  where emptyBoard = Map.empty
```

> **Exercise 13.3.** Implement an algebra in final style for the custom monad you devised in the previous section. Write a couple of computations that work in that monad.

Our second example of a custom monad is `FS`, the purpose of which is to restrict the usage of `IO` operations to only those that read or write from a file. The first step is to write the syntax of the monad as a type class or trait:

```
class FS m where
  writeFile :: FilePath -> String -> m (Either FSError ())
  readFile  :: FilePath           -> m (Either FSError String)
```

```
trait FS[F[_]] {
  def writeFile(handle: FilePath, contents: String): F[FSError \/ Unit]
  def readFile(handle: FilePath): F[FSError \/ String]
}
```

If we now try to ascribe an `FS` type to a computation that also does random number generation, we are stopped by the compiler. In the following definition of `f`, we require the computation to work with any interpretation that understands `FS` syntax, whereas the use of `randomRIO` would force the entire computation to work on `IO`:

```
f :: (Monad m, FS m) => FilePath -> m ()
f path = do number <- randomRIO (1, 100)
            writeFile path (show number)
```

The compiler stops us from mixing `IO` with other monads:

```
Couldn't match expected type ‘m’ with actual type ‘IO’
‘m’ is a rigid type variable
```

Whereas `IO` actions cannot be mixed freely with `FS` actions, `IO` is the perfect target for an interpretation that simply executes the actions in the real file system. In this case, we use the syntax to ensure that we do not mix file system operations with other actions, but at the time of execution, we put that barrier aside, since it has already been checked:

```
instance FS IO where
  writeFile path contents
    = (Right <$> System.IO.writeFile path contents)
      `catch` \ex -> return (Left ex)
  readFile  path
    = (Right <$> System.IO.readFile path)
      `catch` \ex -> return (Left ex)
```

This interpretation is great for executing the code in production, but what about providing another interpretation that mocks the real file system? If we had it, we could test a program under different starting conditions and check that the end result is the desired one, without ever polluting our real hard disk. We can naïvely implement a file system as a map from file paths to strings, which reflects the actual contents of the files:

```
data MockFileSystem = Map FilePath String
```

A pure interpretation of `FS` can be obtained through `State MockFileSystem`. The read and write operations just update the information in that map:

```
instance FS (State MockFileSystem) where ...
```

> **Exercise 13.4.** Complete the interpretation of `FS` as a `State` monad.

Since we may have more than one interpretation for each custom monad, it is fair to ask whether we are paying something in terms of performance. Fortunately, in the case of final style monads, the answer is "no" (well, maybe just a little, tiny bit). Instances provide interpretations directly, without building any intermediate data structures. This is one of its differences with initial style monads, which we describe in the next section.

13.3 Initial Style

Describing a monad in initial style amounts to taking the opposite approach of final style: instead of keeping the type polymorphic over the interpretation, we have a data type that describes the computations. As we will see, the main benefit of encoding a monad in initial style is the ease it affords for inspecting, transforming, and optimizing a computation before it is executed.

To obtain the data type that corresponds to the *algebra* or *syntax* of a custom monad `M`, you need to turn each primitive operation into a constructor for that data type:

- Each constructor has one field per argument that the primitive operation takes.

- Each constructor receives an additional field of the form `R -> M a`, where `R` is the return type of the primitive operation.

In addition to those constructors, there is one constructor with a single field of type `a`.

For a concrete example, let us look at the tic-tac-toe monad. This monad provides two primitive operations:

```
info :: Position -> TicTacToe (Maybe Player)
take :: Position -> TicTacToe Result
```

When described in initial style, they give rise to the following data type:

```
data TicTacToe a = Info Position (Maybe Player -> TicTacToe a)
                 | Take Position (Result       -> TicTacToe a)
                 | Done a
```

```
sealed abstract class TicTacToe[A]  //
case class Info[A](p: Position, k: Option[Player] => TicTacToe[A])
  extends TicTacToe[A]
case class Take[A](p: Position, k: Option[Player] => TicTacToe[A])
  extends TicTacToe[A]
case class Done[A](x: A) extends TicTacToe[A]
```

The intuition of this approach for modeling operations is that for each operation, we need to save all of its parameters, and then we need to know how to *continue* working once we have computed the result. In fact, in many cases, the last field of each constructor is referred to as a *continuation*. The final `Done` constructor is used to signal that a computation is finished — otherwise the recursive structure of the type will prevent us from describing a finite computation.

Note that if the primitive operations return no value, a direct translation to an initial style monad would result in a continuation of type `() -> M a` or `Unit => M[A]`. In practice, in those situations, we simply use `M a` instead. The reason is that a function from the unit type is equivalent to a single value.

> **Exercise 13.5.** Prove that correspondence by writing two functions, `from :: (() -> a) -> a` and `to :: a -> (() -> a)`.

If we claim that `TicTacToe` is, in fact, a monad, we should be able to write down its `Monad` instance. Once again, there is a general recipe that works for every monad in initial style. The first step is making the `Done` constructor — or the corresponding constructor in the data type — describe the `return` operation:

```
instance Monad TicTacToe where
  return = Done
```

The tricky part is in the bind, where we should take a computation `TicTacToe a` and a function `a -> TicTacToe b` and produce a new `TicTacToe b` computation.

One case is simple, that of the `Done` constructor, since we have a value of type `a` at hand:

```
(Done x) >>= f = f x
```

Here comes the magic: for the constructors that represent operations, we leave all the fields as we obtain them and compose the function inside the continuation. Intuitively, we enlarge the continuation to include the new operation that was requested in the bind. In the code, this is reflected by recursively calling the bind function:

```
(Info p k) >>= f = Info p (\pl -> k pl >>= f)
(Take p k) >>= f = Take p (\r  -> k r  >>= f)
```

Exercise 13.6. Write the `Functor` instance for `TicTacToe`. If you feel brave, try `Applicative`.

Using the constructors in the `TicTacToe` data type directly can be quite painful, though. Consider the `takeIfNotTaken` function we developed in final style. The same function written directly in initial style does not use any of the monadic niceties we are used to:

```
takeIfNotTaken :: Position -> TicTacToe (Maybe Result)
takeIfNotTaken p = Info p $ \i ->
                   case i of
                     Just _  -> Done Nothing
                     Nothing -> Take p $ \r ->
                                  Done (Just r)
```

Fortunately, there is a way to provide a nicer interface to an initial style monad, an interface that resembles the way we are used to writing this kind of computation, without explicit manipulation of the continuations as we do above. The trick is to notice that `return` always works as the continuation, regardless of the constructor at hand:

```
info :: Position -> TicTacToe (Maybe Player)
info p = Info p return  -- or Info p Done
take :: Position -> TicTacToe Result
take p = Take p return  -- or Take p Done
```

The interface is now the same as the final style version, except for the type of each operation. In fact, the code for `takeIfNotTaken` can be copied directly from the final style section. The threading of information between different constructors,

which before we had to write explicitly as a continuation, is now delegated to the bind operation of the monad.

People sometimes refer to `info` and `take` as the *smart constructors* of `TicTacToe`. It is customary for initial style monads not to expose the actual constructors but only these smart constructors, which correspond to each operation in the algebra.

After the syntax, it is time for the semantics. In principle, since the initial style monad is a data type, nothing forces an interpretation to have a specific type. Any function of the form `TicTacToe a -> Something` can be regarded as an interpretation. However, the most common interpretations are those that take the form:

```
interpret :: TicTacToe a -> OtherMonad a
```

```
def interpret[A](program: TicTacToe[A]): OtherMonad[A]
```

Functions with this shape are called *natural transformations*, and they play an important role in the theory of monads, as we will see in Chapter 17. Both Scalaz and Cats provide special syntax for natural transformations, which allows us to write:

```
def interpret: TicTacToe ~> OtherMonad
```

As an example of interpretation, let us write one mapping of our custom monad `TicTacToe` to `ReaderT Player (StateT Board IO)`. The code follows closely the instance we defined for `TicTacToe` in final style:

```
runGame :: TicTacToe a -> ReaderT Player (StateT Board IO) a
runGame (Done x)   = return x
runGame (Info p k) = do pl <- lookup p <$> get
                        runGame (k pl)
runGame (Take p k) = do pl <- lookup p <$> get
                        case lp of
                          Just p' -> runGame (k $ AlreadyTaken { by = p' })
                          Nothing -> do me <- ask
                                        modify (insert p me)
                                        ...
```

Note the similarities between all the operations. After performing some work specific to that operation, we always call the continuation `k` with the value we computed, which returns a new `TicTacToe a` value. We then interpret this new computation recursively, by calling `runGame`.

Restrictive monads can also be written in initial style using the same ideas. Above, we designed an `FS` monad that restricts side effects to input/output over files. We can represent its operations by creating a data type:

```
data FS a = WriteFile FilePath String (Either FSError ()     -> FS a)
          | ReadFile  FilePath        (Either FSError String -> FS a)
          | FSDone                    a
```

```
sealed abstract class FS[A]  //
case class WriteFile[A](handle: FileHandle, contents: String,
  k: (FSError \/ Unit) => FS[A]) extends FS[A]
case class ReadFile[A](handle: FileHandle,
  k: (FSError \/ String) => FS[A]) extends FS[A]
case class FSDone[A](x: A) extends FS[A]
```

And we can also write corresponding smart constructors to make the operations easier to use in do notation:

```
writeFile :: FilePath -> String -> FS (Either FSError ())
writeFile path contents = WriteFile path contents FSDone
readFile :: FilePath -> FS (Either FSError String)
readFile path = ReadFile path FSDone
```

> **Exercise 13.7.** Write a computation that mixes file input via the FS monad with some operation in IO. What kind of error do you expect to arise? Use the compiler to check your answer.

We have already mentioned that the most interesting interpretations of initial monads are natural transformations to other, different monads. In the case of FS, we are interested in interpreting it within IO, so the input/output is actually executed. The code for this is similar to that of the final style interpreter:

```
interpret :: FS a -> IO a
interpret (FSDone x) = return x
interpret (WriteFile path contents k)
  = do System.IO.writeFile path contents
       interpret (k (Right ()))
    `catch` \ex -> interpret (k (Left ex))
interpret (ReadFile path k)
  = do contents <- System.IO.readFile path
       interpret (k (Right contents))
    `catch` \ex -> interpret (k (Left ex))
```

As in the case of the TicTacToe interpreter, the interpretation of each operation ends with a recursive call to `interpret`.

> **Exercise 13.8.** Write an interpretation of FS, mapping it to the State monad. Hint: base your code on the final style interpretation, as we have done above for the TicTacToe monad.

13.3.1 Free Monads

Our previous discussion emphasizes that initial monads and their interpretations have a characteristic shape, a design pattern if you wish to call it that. But where there are commonalities, there is always a possibility for abstraction, especially in languages that support higher kinds such as Haskell and Scala. As we will see, abstracting the commonalities among different initial style monads naturally leads to the notion of a *free monad*.

Warning. This section refers to free monads as implemented by Haskell's `free` package. The free monad construction in Scalaz and Cats implements the operational, or *freer monad*, concept that we explore in the next section.

All initial monads share the following four commonalities:

1. A constructor that signals the end of the computation.
2. A (single) continuation as the final element in each constructor except the last one.
3. Smart constructors to make the surface interface similar to that of other monads.
4. Interpretations are natural transformations to other monads.

In the first place, we are going to deal with items (1) and (2), that is, with how to abstract the syntactic shape of initial style monads. The trick is to separate the data type into two parts. The first one holds just the operations, with an additional twist — the recursive appearances of the data type are replaced by a variable `r`:

```
data TicTacToeF r = Info Position (Maybe Player -> r)
                  | Take Position (Result       -> r)
```

Each of these data types is what we call a *pattern functor*, because it describes the shape of an operation. As its name suggests, every type of this form is also a functor.

Exercise 13.9. Write the `Functor` instance for `TicTacToeF`.

If you are using Haskell, this instance can be derived automatically for you. To do so, add the line `deriving Functor` at the end of the data type definition. This mechanism is also available for `Traversable` but not for `Applicative` or `Monad`. For this automatic deriving to work, you need to include a language pragma at the top of your file: {`-# LANGUAGE DeriveFunctor #-`} or {`-# LANGUAGE DeriveTraversable #-`}. Of course, if you could derive `Monad` instances automatically, this book would be much thinner.

Getting back on track, it seems that where we had two problems, we now have three, since we have lost the recursive nature of the initial style syntax. We can regain all of it at once by using this magic data type:

```
data Free f a = Free (f (Free f a))
              | Pure a
```

First of all, this data type is parametric over whichever pattern functor `f` that you use. Therefore, once the definition for `Free` is in place, we do not need to write it any more. The `Pure` constructor fulfills requirement (1), as it adds to any pattern functor an additional constructor to return a pure value. Finally, the recursive structure is made explicit in the `Free` constructor.

Of course, in order to use `Free` to build monads, we need to write the corresponding instances. The code is similar to an initial style monad:

```
instance Functor f => Functor (Free f) where
  fmap f (Pure x) = Pure (f x)
  fmap f (Free x) = Free (fmap (fmap f) x)

instance Functor f => Applicative (Free f) where
  pure = Pure
  Pure f <*> Pure x = Pure (f x)
  Pure f <*> Free x = Free (fmap (fmap f) x)
  Free f <*> x      = Free (fmap (<*> x) f)

instance Functor f => Monad (Free f) where
  return = Pure
  Pure x >>= f = f x
  Free x >>= f = Free (fmap (>>= f) x)
```

Now we can replace all uses of `TicTacToe` with the combination of `Free` and the pattern functor:

```
type TicTacToe = Free TicTacToeF
```

The most important feature of `Free` is that we are guaranteed to obtain a monad if we promise that the pattern functor is indeed a functor. This is the reason for calling `Free f` the *free monad over a functor* — because you get the `Monad` instance for free! That instance abstracts the common shape we have seen in initial style monads, where the bind is applied recursively until we find a `Pure` constructor.

In order to convince ourselves, let us see how this definition translates into the specific case of the `Info` constructor in `TicTacToeF`. The `fmap` operation for that constructor reads:

```
fmap f (Info p k) = Info p (f . k)
```

Now, let us substitute this definition for that of the bind function:

```
   Free (Info p k) >>= f
≡ -- by the definition of (>>=)
   Free (fmap (>>= f) (Info p k))
≡ -- by the definition of fmap
   Free (Info p ((>>= f) . k))
≡ -- by introducing a parameter in the last expression
   Free (Info p (\pl -> (>>= f) (k pl)))
≡ -- rewriting the previous expression into a more common form
   Free (Info p (\pl -> k pl >>= f))
```

If you compare the last expression above to the one we wrote explicitly for the initial style monad, you can see that they are the same, except for the additional `Free` constructor. Remember that, previously, we had to work hard to reach the correct expression, but here, `fmap` can be derived by GHC, and the bind function is written once and for all for any initial style monad.

> **Exercise 13.10.** Trace the execution of `fmap f (Free (Info p k))` using the same style as above. Convince yourself that this definition just lifts the `fmap` from `TicTacToeF` into the free monad.

Encoding using `Free` does not eliminate the need for smart constructors, however. In the case of initial style monads, using constructors directly gives rise to a somewhat awkward style. This problem gets worse with free monads, since every time you use an operation, you need to write the constructor plus a layer of `Free`. A smart constructor for the `Info` operation looks like this:

```
info :: Position -> TicTacToe (Maybe Player)
info p = Info p return
```

The common pattern here is to initialize the continuation argument to `return`. As in the case of bind, we can achieve generality by using `fmap` instead of the name of the constructor. The generic "smart constructor builder" is called `liftF` in Haskell's `free` library:

```
liftF :: Functor f => f a -> Free f a
liftF = Free . fmap return
```

The pattern now is to use `liftF` over each constructor of the pattern functor to obtain the smart constructors we want, using `id` as the last argument:

```
info :: Position -> Free TicTacToeF (Maybe Player)
info p = liftF (Info p id)
```

To be fair, in this case, we do not reduce the length of the implementation simply by writing `Free (Info p return)`. However, this generality will pay off when we

introduce other forms of free monads that provide different trade-offs in terms of efficiency. Each of those other forms provide their own version of `liftF`, so the code above readily generalizes. In contrast, the implementation using the `Free` constructor is only valid for the flavor of free monads described in this section.

The final step in every discussion about implementing custom monads is how to encode interpretations. One possibility is to pattern match directly on the free monad structure. For example, we can rewrite our `runGame` interpretation as follows:

```
runGame :: Free TicTacToeF a -> ReaderT Player (StateT Board IO) a
runGame (Pure x)          = return x
runGame (Free (Info p k)) = do pl <- lookup p <$> get
                               runGame (k pl)
runGame (Free (Take p k)) = ...
```

There is some repetition, though. First of all, we usually want to map `Pure` — the `return` of the free monad — into the `return` operation of another monad. Second, we always end the branch for each constructor by recursively calling the interpretation. Those are the patterns we were trying to abstract by introducing the `Free` data type. Our goal is to reduce our implementation to the real meat, the mapping from `TicTacToeF` to another monad:

```
runGame' :: TicTacToeF a -> ReaderT Player (StateT Board IO) a
runGame' (Info p k) = do pl <- lookup p <$> get
                         return (k pl)
             -- or = k <$> (lookup p <$> get)
             -- or = k . lookup p <$> get
runGame' (Take p k) = ...
```

We then need a way to tie `runGame'` with the free monad structure to obtain an interpretation akin to `runGame`. We call this operation `foldFree`:

```
foldFree :: Monad m => (forall r. f r -> m r) -> Free f a -> m a
foldFree _         (Pure x) = return x
foldFree interpret (Free x) = do x' <- interpret x
                                 foldFree interpret x'
                    -- or = interpret x >>= foldFree interpret
```

Then we have `runGame = foldFree runGame'`.

> **Exercise 13.11.** Rewrite `FS` as a free monad. Start by introducing the corresponding pattern functor, and then define the interpreters using `foldFree`.

In this section, we introduced free monads as the natural product of abstracting over the commonalities of initial style monads. The hope is to remove the esoteric

glow around them — there are some technicalities in the definition of `Free`, but the idea is not that profound. As we have just seen, it has already paid off: by using `liftF` or `foldFree`, we can remove most of the boilerplate associated with describing computations as data types.

13.3.2 Streams as Initial Style Monads

This section is long, but there is one, small thing that we still need to show. The notion of initial style monad is used in many libraries that provide a streaming interface to consume or produce data from different sources, like in-memory lists, files, or the network. Let us explore them with our new "custom monad" glasses.

In Haskell, there are two main contenders for the streaming library space, `pipes` and `conduit`. In both cases, the monad that describes streaming computations is marked with input and output types. But there the similarities stop, as each library has a different set of features. Let us look first at the `Proxy` type, which is the core of the `pipes` library:

```
data Proxy a' a b' b m r
  = Request a' (a  -> Proxy a' a b' b m r )
  | Respond b  (b' -> Proxy a' a b' b m r )
  | M          (m    (Proxy a' a b' b m r))
  | Pure       r
```

This is indeed an initial style monad: there is a `Pure` constructor for building `Proxy`s from pure values, and the `Request` and `Respond` constructors take continuations. The `pipes` library uses a model of bidirectional streaming, in which some data flows from `a` into `a'` and vice versa for `b` and `b'`. As in other initial style monads, `Request` and `Respond` have a continuation as their second argument. The other interesting feature is that a `Proxy` may embed the operations of another monad `m` inside of itself. This is useful if we need to perform resource management while streaming, for example.

The core data type of `conduit` is called `Pipe` — the terminology gets a bit confusing here, so we have to be careful about the distinctions between data types and packages. In this case, we get even more type parameters:

```
data Pipe l i o u m r
  = HaveOutput     (Pipe l i o u m r) o
  | NeedInput (i -> Pipe l i o u m r) (u -> Pipe l i o u m r)
  | Done r
  | PipeM       (m (Pipe l i o u m r))
  | Leftover       (Pipe l i o u m r) l
```

Above, `i` and `o` refer to the basic input and output types. However, in addition to consuming inputs and producing outputs, a `Pipe` may consume upstream values of type `u` — for this reason, `NeedInput` takes not one but two continuations — and

produce leftovers of type `l` — and hence we need two constructors, `HaveOutput` and `Leftover`, to produce both types of values. Upstream and leftover values allow us to chain `Pipes` together in an efficient way. For example, if a `Pipe` finishes its job early and just wants to pass the rest of the stream without inspection, it can use the `Leftover` instead of the `HaveOutput` interface.

In the Scala world, we have `fs2` as a streaming library, which is compatible with both Scalaz and Cats. If we look inside the `Pipe` object, we find the following definitions (reduced for the sake of conciseness):

```
sealed abstract class Stepper[-A, +B]  //
final case class Suspend[A, B](force: () => Stepper[A, B])
  extends Stepper[A, B]

sealed abstract class Step[-A, +B] extends Stepper[A, B]
final case object Done extends Step[Any, Nothing]
final case class Fail(err: Throwable) extends Step[Any, Nothing]
final case class Emits[A, B]
  (segment: Segment[B, Unit], next: Stepper[A, B])
  extends Step[A, B]
final case class Await[A, B]
  (receive: Option[Segment[A, Unit]] => Stepper[A, B])
  extends Step[A, B]
```

Here, there is an extra layer of indirection, because every step in the computation is guarded by a `Suspend` constructor to stop the pipe from evaluating earlier than expected. In the `Step` trait, we find a `Done` constructor to signal that the stream has ended and then consume input or produce output via `Awaits` or `Emits`, respectively. As an additional feature, `fs2` provides the ability to throw an error while streaming.

13.4 Operational Style and Freer Monads

Initial style monads reify computations as a data type, and with the free monad construction, we remove most of the boilerplate associated with that technique. Alas, there is one important drawback: initial style forces us to write constructors with awkward types, using continuations to keep track of successive computations. *Operational style* (along with *freer monads*, the corresponding boilerplate-removal technique) uses a data type to represent computations, but the shape mirrors much more closely the way we describe those computations using `do` notation or `for` comprehensions.

Returning again to our tic-tac-toe example, we would like to develop a monad to represent the whole game. The operations that such a monad ought to support are the following:

```
info :: Position -> TicTacToe (Maybe Player)
take :: Position -> TicTacToe Result
```

In operational style, each primitive operation turns into a constructor with exactly *the same type*. In order to complete the set of operations, we also add constructors for `return` and the bind function. Note that in Haskell, we actually get a Generalized Algebraic Data Type (or GADT, for short), since the constructors refine the type we build. No such distinction is needed in Scala:

```
data TicTacToe a where
  Info :: Position -> TicTacToe (Maybe Player)
  Take :: Position -> TicTacToe Result
  Done :: a        -> TicTacToe a
  Bind :: TicTacToe a -> (a -> TicTacToe b) -> TicTacToe b
```

```
sealed abstract class TicTacToe[A]  //
case class Info(p: Position) extends TicTacToe[Option[Player]]
case class Take(p: Position) extends TicTacToe[Result]
case class Done[A](x: A) extends TicTacToe[A]
case class FlatMap[A, B](x: TicTacToe[A], f: A => TicTacToe[B])
  extends TicTacToe[B]
```

We can provide a monad instance for this data type by just using the corresponding constructors in the operations. We might want to have smart constructors for uniformity with other styles. In both cases, the definition of each function just calls the corresponding constructor:

```
instance Monad TicTacToe where
  return = Done
  (>>=)  = Bind

-- using the same type signatures as before
info = Info
take = Take
```

One advantage of operational style as compared to initial style is that it is fairly easy to see how programs written with either `do` notation or `for` comprehensions translate into the corresponding data type. As an example, consider the `takeIfNotTaken` function we developed earlier:

```
takeIfNotTaken p = do i <- info p
                      case i of
                        Just _  -> return Nothing
                        Nothing -> fmap Just (take p)
```

You can desugar it into a series of binds and map operations:

```
takeIfNotTaken p = info p >>= \i ->
                   case i of
                     Just _  -> return Nothing
                     Nothing -> fmap Just (take p)
```

The last line can be turned into `take p >>= (return . Just)`. When the monad is described in operational style, each primitive and monadic operation turns into a constructor. This means that the code above is equivalent to the following:

```
takeIfNotTaken p = Info p `Bind` \i ->
                   case i of
                     Just _  -> Done Nothing
                     Nothing -> Take p `Bind` (Done . Just)
```

This shows that in terms of syntax, code written in operational style has the most straightforward conversion to a data type. In contrast, initial style brings continuations to the table, and final style relies on the type class or trait mechanism available in the underlying language.

Apart from the `Monad` instance, we also need the corresponding `Applicative` and `Functor` instances. In order to make `TicTacToe` into an applicative functor, we should recall that structure's correspondence with the monadic operations we outlined in Chapter 3. The `pure` operation corresponds directly to `return`, and the `(<*>)` operation also has a direct definition in terms of `(>>=)`:

```
instance Applicative TicTacToe where
  pure = Done
  f <*> x = do f' <- f
               x' <- x
               return (f' x')
```

The `do` notation used in `(<*>)` can be translated back to a sequence of binds:

```
  f <*> x = f >>= \f' -> x >>= \x' -> return (f' x')
```

We can rewrite this definition using only the constructors in the `TicTacToe` data type:

```
  f <*> x = f `Bind` (\f' -> x `Bind` (\x' -> Done (f' x')))
```

> **Exercise 13.12.** Write the `Functor` instance for `TicTacToe`. Hint: remember that you can always define `fmap f x` as `x >>= return . f`.

As in the case of initial style monads, interpretations do not require the target to be a monad. However, the most interesting semantics are actually natural transformations from the custom monad into another one — remember that a natural

transformation, in this case, just refers to an interpretation that is parametric over the return type of the monad. Here is the operational style version of the `runGame` interpretation we have already described in the other styles:

```
runGame :: TicTacToe a -> ReaderT Player (StateT Board IO) a
runGame (Done x)   = return x
runGame (Bind x f) = runGame x >>= runGame . f
runGame (Info p)   = lookup p <$> get
runGame (Take p)   = do pl <- lookup p <$> get
                        case pl of
                         Just p' -> return $ AlreadyTaken { by = p' }
                         Nothing -> do me <- ask
                                       modify (insert p me)
                                       ...
```

There are two, key points in this implementation. First of all, the interpretation of the primitive operations `Info` and `Take` do not care about any further operations that might come after them. They only perform their single duty. This is similar to how final style monads operate — in fact, the code of `Info` and `Take` is identical to the final style interpretation. The second important point is that we need to convert the `Done` and `Bind` operations explicitly into the corresponding operations of the target monad, `ReaderT Player (StateT Board IO)`, in this case.

> **Exercise 13.13.** Rewrite the `FS` monad in operational style.

From our description of operational style monads, there is clearly room for abstraction: all of them introduce constructors similar to `Done` and `Bind` above. Following the lead of initial style monads — the similarities of which can be abstracted into free monads — we can also abstract the commonalities of operational style monads into the so-called freer monad.

As in the case of free monads, the custom part of each freer monad is described in its own data type. But in contrast to free monads, we do not have to introduce any extra type variables or change the data type in any way. The constructors look exactly as they did for the operational style monads, which in turn means that they take the type of the primitive operations. Haskellers tend to say that this data type describes the *instructions*, whereas Scala practitioners refer to them as *actions*, hence the different suffixes:

```
data TicTacToeI a where
  Info :: Position -> TicTacToeI (Maybe Player)
  Take :: Position -> TicTacToeI Result
```

```
sealed abstract class TicTacToeA[A]  //
case class Info(p: Position) extends TicTacToeA[Option[Player]]
case class Take(p: Position) extends TicTacToeA[Result]
```

The freer monad just embeds those constructors into a common data type for the `return` and bind operations.

This data type is available under the name `Program` in `operational`:*

```
data Program instr a where
  Done  :: a -> Program instr a
  Bind  :: Program instr b -> (b -> Program instr a) -> Program instr a
  Instr :: instr a -> Program instr a
```

> **Exercise 13.14.** Write the `Monad` instance for `Program instr`. Note that this instance does not depend on which set of instructions, `instr`, you work on. Hint: as in the case of operational style monads, this instance is obtained by using some of the constructors in `Program instr`.

Interestingly enough, both Scalaz and Cats use the freer monad concept to model their `Free` type, instead of the "free monad from a functor" introduced in the previous section. This is a source of confusion for beginners, since different communities refer to related but slightly different concepts by the same name:

```
sealed abstract class Free[S[_], A]
final case class Return[S[_], A](a: A) extends Free[S, A]
final case class Suspend[S[_], A](a: S[A]) extends Free[S, A]
// This class is called FlatMapped in Cats
final case class Gosub[S[_], A, B](a: Free[S, A], f: A => Free[S, B])
  extends Free[S, B]
```

If we want to model tic-tac-toe using Scala idioms, the most common approach is to use this definition of `Free`:

```
type TicTacToe[A] = Free[TicTacToeA, A]
```

The main advantage of abstracting the commonalities of operational style monads into a single data type is that we no longer have to write the boilerplate part of the interpretations, the mapping of `return` and the bind into the corresponding target monad. Instead, we can focus on writing a natural transformation from the set of operations, instructions, or actions into the target monad and leave the rest to the libraries. This is what the `foldMap` function from Scalaz provides:

```
final def foldMap[M[_]](f: S ~> M)(implicit M: Monad[M]): M[A] =
  step match {
    case Return(a) => M.pure(a)
    case Suspend(s) => f(s)
    case a@Gosub(_, _) => M.bind(a.a foldMap f)(c => a.f(c) foldMap f)
  }
```

*Actually, the `operational` package defines a monad transformer `ProgramT`, but here we show a simplified version that does not embed other monads.

Note that the definition of the function does not pattern match directly on the `Return`, `Suspend`, and `Gosub` cases. Instead, it first preprocesses the value via the `step` function:

```
@tailrec final def step: Free[S, A] = this match {
  case x@Gosub(_, _) => x.a match {
    case b@Gosub(_, _) => b.a.flatMap(a => b.f(a).flatMap(x.f)).step
    case Return(b) => x.f(b).step
    case _ => x
  }
  case x => x
}
```

The goal of `step` is to prevent stack overflows that arise from left-nested binds. Take a series of three computations tied to `Gosub` with nesting associated to the left operand, that is, `Gosub(Gosub(f, g), h)`. If we directly apply the natural transformation, we would get a similarly nested call to `flatMap`: `(f flatMap g) flatMap h`. In order to perform the transformation, we separate out the `f flatMap g` and then recurse over it. We need to finish transforming that part before we can move to apply the computation to `h`. If we have more levels of nesting, we need to place further recursive calls into the stack.

By means of the monad laws, we know that, in fact, `(f flatMap g) flatMap h` is equivalent to `f flatMap (g flatMap h)`. However, in this second case, right after we transform `f`, we can extract its result and move to the next part of the computation. We do not need to do anything else on that call, since the function is *tail recursive*. In practice, this means that transforming a right-nested bind can be turned into a loop, instead of consuming precious stack space. This kind of problem pops up in many other contexts when designing custom monads, and we explore the whole space of solutions in Chapter 15.

Why "freer"? The concept of a free monad arose from category theory. In that world, we get a monad for free when given a functor, and we refer to such a functor as a pattern functor. In the case of operational style monads, we do not require any special feature from the instruction data type (`TicTacToeI` is definitely not a functor). Because Haskell's `Program` and Scala's `Free` are so lenient about the type they wrap, they allow us to get a monad for free without even requiring a functor instance. So it is even "freer" than the free monad!

13.4.1 Operational Monads as Free Monads

The definition of `step` in the Scala version of the freer monad suggests another representation for that concept, in which the binds are right-associated by construction. In this representation, we do not have a different constructor for `Instr`

and `Bind` — we merge both into `Impure`. In order to distinguish this new version from `Program`, we call it `Freer` in the Haskell version:

```
data Freer instr a where
  Pure   :: a -> Freer instr a
  Impure :: instr a -> (a -> Freer instr b) -> Freer instr b

instance Monad (Freer instr) where
  return = Pure
  Pure   x   >>= f = f x
  Impure x k >>= f = Impure x (f <=< k)
```

```
sealed abstract class Free[S[_], A]  //
final case class Pure[S[_], A](a: A) extends Free[S, A]
final case class Impure[S[_], A, B](c: S[A], f: A => Free[S, B])
  extends Free[S, B]
```

Exercise 13.15. Write the `Functor` and `Applicative` instances for `Freer instr`.

Exercise 13.16. Write functions `twoToThree` and `threeToTwo` that convert between the freer monads with two and three constructors. Hint: for the latter, it is useful to follow the same shape of recursion as `step`.

This new representation — which is isomorphic to the one with three constructors — is useful for highlighting the relationship between freer and free monads. As we discussed earlier, the "freer" name suggests that we do not impose any requirements on the instruction data type, whereas the free monad requires a functor. So, in order to bridge the gap and express the freer monad in terms of the free monad, we need a way to build a functor out of thin air. Fortunately, there is such a construction, with the categorically-inspired name of `Coyoneda`.* The definition of `Coyoneda` is written as a GADT, because the type variable `b` in the constructor does not appear in the return type (only `f` and `a` do). We say that `b` is *existentially quantified*:

```
data Coyoneda f a where
  Coyoneda :: (b -> a) -> f b -> Coyoneda f a
```

`Coyoneda f` is a functor regardless of which type constructor `f` you choose to instantiate it:

*Crossing even more categorical keywords off our list, `Coyoneda f` is technically the *left Kan extension* of `f` along the identity functor. You do not need to know any of these terms to follow along with this book, but if you are interested, *Kan extensions for program optimisation* [Hinze, 2012] provides an introduction to this concept in Haskell.

```
instance Functor (Coyoneda f) where
  fmap h (Coyoneda g x) = Coyoneda (h . g) x
```

In the definition of `fmap`, suppose that `h` has type `a -> b` and `g` has type `c -> a`. From the definition of the `Coyoneda` constructor, the type of `x` must be `f c`. Now, the composition `h . g` has type `c -> a`. This new function still has `c` as its source type, so `x` can still be used as second argument to `Coyoneda`.

The freer monad is now just the result of applying the free monad construction after creating a functor via `Coyoneda`:

```
type Freer f = Free (Coyoneda f)
```

The `free-operational` package in Hackage makes use of this correspondence to implement the functionality of the `operational` package by reusing most of the implementation from `free`.

Of course, dear reader, you should not blindly believe statements about the correspondence between the two constructions. Let us prove that this result indeed holds, by expanding the definitions of `Free` and `Coyoneda`:

```
   Freer f a
≡ -- by definition
   Free (Coyoneda f) a
≡ -- introducing the constructors for Free
   Pure a | Free (Coyoneda f (Free (Coyoneda f)) a)
≡ -- by definition, Free (Coyoneda f) = Program f
   Pure a | Free (Coyoneda f (Program f a))
≡ -- expanding Coyoneda f
   Pure a | Free (b -> Program f a) (Program f b)
≡ -- reordering the fields in the Free constructor
   Pure a | Free (Program f b) (b -> Program f a)
```

Et voilà! You get exactly the same shape as the two-constructor version of the freer monad. This shows that, indeed, the two constructions are deeply related.

13.5 Transforming and Inspecting Computations

One of the advantages of separating descriptions of computations from their actual execution is that you can perform some transformations on the former. As we will see, the amount of introspection that free monads allow is limited, but we can turn to *free applicatives* if we really require optimizations in our programs.

Consider a simple language to describe manipulations of stacks of integers, in which we can either push values onto or pop values from the top of the stack. Using `Free`, we could define this language in the following manner:

```
data IStackF r = Pop (Integer -> r) | Push Integer r deriving Functor
type IStack = Free IStackF
-- Smart constructors
pop :: IStack Integer
pop = liftF (Pop id)
push :: Integer -> IStack ()
push v = liftF (Push v ())
```

This simple language is enough to implement a *Reverse Polish Notation calculator*. Reverse Polish Notation (RPN) is a way of writing arithmetic formulas that postpones the operation on the operands. For example, $(3 + 2) \times (1 + 4)$ is written as $3\ 2\ +\ 1\ 4\ +\ \times$. The advantage of RPN is that you do not need any parentheses, in contrast to common mathematical notation. In order to evaluate a formula in RPN, you create a stack of integers from which each operation pops the required number of operands, with the result of the operation being pushed back on top. When the list of instructions is finished, the top of the stack is taken as the final value:

```
data RPNInstruction = Number Integer | Plus | Times

evaluate :: [RPNInstruction] -> IStack Integer
evaluate []               = pop
evaluate ((Number n) : r) = push n >> evaluate r
evaluate (Plus       : r) = ((+) <$> pop <*> pop) >>= push
                            >> evaluate r
evaluate (Times      : r) = ((*) <$> pop <*> pop) >>= push
                            >> evaluate r
```

Exercise 13.17. Write an interpreter from `IStack` to `State [Integer]`. Use that interpreter to write a function from `[RPNInstruction]` to `Integer`.

The `IStack` language is amenable to some optimization. For example, if you pop right after a push, there is no need to actually perform the operation. Instead, we can thread the value without ever touching the stack. In other words `push v >>= pop` $\equiv$ `return v`. We can implement that optimization as a recursive operation over `IStack` computations.

```
optimize :: IStack a -> IStack a
optimize (Pure x)          = Pure x
optimize (Free (Push v (Free (Pop k))))
                           = optimize (k v)
optimize (Free (Pop k))    = Free (Pop (\n -> optimize (k n)))
optimize (Free (Push v k)) = Free (Push v (optimize k))
```

This example also shows the restrictions we have on inspecting an operation. Whenever an operation produces a value that is passed on to the next computation — like `Pop` in this case — we cannot look further ahead. The reason is that the definition of the next computation depends on that same value, so we cannot inspect it beforehand. The previous definition of `optimize` works around this problem a bit by calling `optimize` after obtaining the value `n` in the `Pop` branch, but this is not always possible. In some circles, you may hear the expression "free monads can be inspected up to the information-theoretic limit."

> **Exercise 13.18.** Rewrite the `IStack` monad using the operational/freer construction, and implement `optimize`. Is is possible to inspect the computation further in this style?

The conventional wisdom is that being inspectable is a property of initial style monads and free monads, their close relatives, but final style monads provide no such capability. However, this is not true. First of all, you could inspect a final style monad by interpreting it into initial style and back again into final style:

```
class MonadIStack m where  -- Final style definition
  pop'  :: m Integer
  push' :: Integer -> m ()

instance MonadIStack IStack where  -- Interpretation into initial style
  pop'  = pop   -- as previously defined
  push' = push  -- as previously defined

-- The inverse operation, initial to final style
toMonadIStack :: (Monad m, MonadIStack m) => IStack a -> m a
toMonadIStack = foldFree toMonadIStack'
  where toMonadIStack' (Pop    k) = k <$> pop'
        toMonadIStack' (Push v k) = k <$  push' v
```

Using this approach, the optimization procedure we devised for initial style can be applied to final style, too:

```
optimize' :: (Monad m, MonadIStack m) => m a -> m a
optimize' = toMonadIStack . optimize
```

However, this approach is clearly not optimal. First of all, it requires two definitions of the same custom monad, one in initial and one in final style. Furthermore, conversion to initial style involves the creation of an intermediate value, which might grow rather large depending on the computation.

Once again, Oleg Kiselyov's work on final style interpreters provides a solution to this problem. In our case, if we find ourselves in the situation of a `pop` after

a push, we want to short circuit the evaluation. Thus, to optimize the code we are required at each step to remember the previous one. We call that piece of information the *context*:

```
data LastOp = Return | LastPop | LastPush Integer
```

The trick now is to create an interpretation of `MonadIStack` for functions with an additional argument representing the context. That is, we write an interpretation that is still parametric but knows a bit about its own inner structure. To do so, we introduce `WithContext`, which is similar to the `StateT` monad transformer, except that its value is updated by its operations:

```
newtype WithContext c m a = C { unC :: c -> m (a, c) }

instance Monad m => Monad (WithContext LastOp m) where
  return x  = C $ \_ -> return (x, Return)
  C x >>= f = C $ \context -> do (x', context') <- x context
                                 unC (f x') context'

instance (Monad m, MonadIStack m)
         => MonadIStack (WithContext LastOp m) where
  pop' = C $ \context -> case context of
                LastPush n -> return (n, Return)
                _          -> (, LastPop) <$> pop'
  push' v = C $ \_ -> (, LastPush v) <$> push' v
```

Exercise 13.19. Write the `Applicative` and `Functor` instances for `WithContext LastOp m`. Hint: use the same methodology that we use to write the corresponding instances for `TicTacToe` at the beginning of Section 13.4.

Performing the optimization now amounts to interpreting the monad in an initial context. In this case, we choose `Return`, since it does not affect our optimization. In other cases, we might need a special starting value:

```
optimize :: (Monad m, MonadIStack m)
         => WithContext LastOp m a -> m a
optimize p = do (x, _) <- unC p Return
                return x
```

Exercise 13.20. Convince yourself that the previous code works. Here, we are using the monadic computation `p` with the type `WithContext LastOp m`. Since the result is another computation that is parametric over `MonadIStack` coupled with the final state, we can just ignore the second part.

13.5.1 Free Applicatives

We introduced *applicative functors* in Section 3.2 as a succinct notation for lifting pure functions into the monadic world. We discussed back then that every monad is an applicative but not the other way around. This means that we can express any applicative computation as a monadic one, but not all computations expressed using monadic combinators can be written exclusively using applicatives (hence the need for the `ApplicativeDo` compiler extension described in Section 3.3.2). This implies that the structure of applicative computations is more restricted, and this fact helps us to inspect and transform them in many more ways than we can monadic computations.

The bind operation in a monad allows the next computation to depend entirely on the previous value, as witnessed by its type, `m a -> (a -> m b) -> m b`. In other words, until we know the value of the argument of type `a`, we know *nothing* about what to do next. Thus, binds are the place where our analyses and transformations have to stop. On the other hand, the `(<*>)` operation for applicatives does not allow its function parameters to depend on the values that you obtain, since its type is `f (a -> b) -> f a -> f b` — note that the entire `a -> b` is now *inside* the `f`. As a result, we can inspect much more of the computation!

The definition of the free applicative in Haskell's `free` package is given as the `Ap` data type:

```
data Ap f a where
  Pure :: a -> Ap f a
  Ap   :: f a -> Ap f (a -> b) -> Ap f b
```

If you swap the order of the arguments, you can see that `Ap` almost has the same type as `(<*>)`:

```
flip Ap :: Ap f (a -> b) -> f a -> Ap f b
  (<*>) ::    f (a -> b) -> f a ->    f b
```

The difference is that one of the arguments — the one that is a function — recursively calls `Ap`, whereas the other is just a simple value inside a pattern functor. The `Applicative` instance highlights this fact by swapping arguments in the `(<*>)` definition:

```
instance Functor f => Functor (Ap f) where
  fmap f (Pure a) = Pure (f a)
  fmap f (Ap x y) = Ap x (fmap (f .) y)

instance Functor f => Applicative (Ap f) where
  pure = Pure
  Pure f <*> y = fmap f y
  Ap x y <*> z = Ap x (flip <$> y <*> z)
```

An interesting fact about free applicatives is that the definition above is not the only possible one. The article *Flavours of free applicative functors* [Cheplyaka, 2013] describes four different versions of this structure. For example, a variation of the previous definition can be achieved with the same data type for `Ap`, by changing the `Applicative` instance slightly:

```
instance Functor f => Applicative (Ap f) where
  pure = Pure
  x <*> Pure y = fmap ($ y) x
  x <*> Ap y z = Ap y ((.) <$> x <*> z)
```

In both of the two instance definitions, an expression in applicative style with the shape `f <$> x <*> y <*> z` is represented by a term of the form `z `Ap` (y `Ap` (x `Ap` Pure f))`. The difference between these instances is when and how the term is normalized to this form.

Following the same recipe as for describing the freer monad construction using Coyoneda, we can also go a step further and define the *freer applicative*, `Ap (Coyoneda f)`. In fact, the `free-operational` package in Hackage does exactly this.

As an example of a free applicative functor, let us build a small library for handling command line arguments, in the spirit of Haskell's `optparse-applicative` library. To simplify, we consider only two types of command line arguments: (1) flags, which take no arguments; and (2) options, each given by its name followed by the value of the option. For example, in the command line invocation, `run --background --file out`, `--background` is a flag and `--file` is an option followed by the value `out`.

We want a way to describe command line arguments and how they affect the configuration of a program. We will benefit from inspecting their structure when we need to show the help documentation for the program. We begin by introducing a pattern functor for the two kinds of arguments. Each kind of argument consumes a different kind of value:

```
data Arg a = Flag   String (Bool -> a)
           | Option String (Maybe String -> a)
           deriving Functor
```

In the case of a flag, it is just a Boolean that indicates whether it appears in the command line. For the options, we need to know whether it is present, and if so, which is its associated value. The next step is introducing the free applicative:

```
type CommandLine = Ap Arg
```

Finally, as we did with free monads, we should create smart constructors to make it easy to work with the library. In this case, the smart constructors just apply the identity function to the argument obtained:

```
flag :: String -> CommandLine Bool
flag s = Ap (Flag s id) (Pure id)
option :: String -> CommandLine (Maybe String)
option s = Ap (Option s id) (Pure id)
```

We can now build a small program that reads command line arguments and creates a `Configuration` value out of them. This is similar to a parser, for people familiar with them:

```
data Config = Config { background :: Bool, file :: String }

readCommandLine :: CommandLine Config
readCommandLine = Config <$> flag "background"
                         <*> (maybe "out" id <$> option "file")
```

The interesting fact is that even though the definition of `readCommandLine` uses `(<$>)` inside of `(<*>)`, the `Applicative` instance takes care of normalizing everything to a `Pure` function applied to a bunch of values of type `Arg`. We can benefit from this fact to write a function that traverses a `CommandLine` and returns a list of all possible flags and options:

```
argNames :: CommandLine a -> [String]
argNames (Pure _)        = []
argNames (arg `Ap` rest) = argNames rest ++ [name arg]
  where name :: Arg a -> String
        name (Flag   nm _) = nm
        name (Option nm _) = nm
```

Writing such a function would be impossible if `CommandLine` were a monad. But we do not really need monadic power here, since we expect command line arguments not to depend on one another. This is exactly the amount of power that `Ap` gives us.

In this chapter, we introduced both free monads and free applicatives *for a functor*. These two structures themselves create either a monadic or applicative structure starting with a pattern functor. However, there is another possible free construction, the *free monad of an applicative*. This notion is defined in the `Control.Monad.Free.Ap` module of the `free` library. The data type is the same as for a free monad of a functor, but there are differences in the `Monad` instance:

```
instance Applicative f => Monad (Free f) where
  return = pure  -- (!)
  Pure a >>= f = f a
  Free m >>= f = Free (fmap (>>= f) m)
```

The important part is marked with an exclamation point above. In the free monad of a functor, the `return` operation uses the `Pure` constructor. But here, it piggybacks on the `pure` operation of the underlying applicative functor.

This new instance for `Free` lays the path to a combination of applicative and monadic components, starting from a pattern functor `f` and constructing the type `Free (Ap f)` — or even further, starting with just a data type of instructions and building `Free (Ap (Coyoneda f))`. Ideally, this structure provides inspection as much as possible — with the free applicative component — but still allows us data dependencies when we need them — with the free monad part. The functional programming community is still getting a handle on this promising idea. Only the future will show us its real applications.

In this chapter, we have seen how you can create your own monads using a handful of approaches: final style, initial style, free monads, and operational style. The next question we will tackle is how to combine several of your own, custom monads. This is especially interesting when you develop a domain-specific language that may specify multiple behaviors. Spoiler alert: final style monads are easy to combine, but the others will give us more than one headache.

14 Composing Custom Monads

A custom monad helps you refine the language you use to develop your application in order to target your specific domain. But sometimes, you need to describe a computation operating in several domains. Another possibility is that you need to combine two, restrictive monads in order to take advantage of operations from both. In both scenarios, you need a composite, custom monad. One solution, drawing directly from Chapter 11, is to define custom monads as transformers and use the same layered techniques as discussed in that chapter.

This chapter introduces other ways in which we can compose custom monads defined in any of the three styles presented in the previous chapter. As we will see, this sort of composition is much simpler than for generic monads, due to its more rigid structure. At the end of the chapter we present *extensible effects*, which take this idea to the extreme: there is only one monad, `Eff`, parametrized by the set of operations allowed at each point.

Remember that a custom monad is defined by its algebra (also called syntax), which defines the primitive operations it provides. On top of that, one or more interpretations (also referred to as semantics) describe how to execute those operations, usually in the context of a more generic monad, such as `IO`. When we speak about composing two custom monads, we expect the combination to have these properties:

- Its algebra should combine the operations of each of its constituent monads. Furthermore, you should be able to describe a computation in the combined monad without any special ceremony when compared to using only one of them.
- Defining the interpretation of each custom monad for the same target should suffice to define the interpretation of the combination. For example, if two monads have an interpretation to `IO`, there should be a way to get a single interpretation to `IO` out of both.

Let us dive in and see how many of these requirements we can satisfy when we use each of these three approaches to creating custom monads.

14.1 Final Style

You will be happy to learn that composing final style monads is easy. In fact, you need to do *nothing* to compose monads in final style!

Consider the following restrictive monad for random number generation. The only primitive operation available is obtaining an integer value from a range. Using this functionality, we could implement random generation for other data types, but that is out of the scope of this book:

```
class RandomGen m where
  random :: Int -> Int -> m Int
```

```
trait RandomGen[F[_]] {
  def random(lower: Int, upper: Int): F[Int]
}
```

Our goal is to describe a computation that generates a random integer and then writes it to a file. Since `RandomGen` alone does not provide that functionality, we would like to use the `FS` monad we introduced in the previous chapter. The whole program should look like this:

```
randomWrite path = do number <- random 1 100
                      writeFile path (show number)
```

In fact, the previous code compiles without any change, which means that we could combine operations from both monads freely. The type of `randomWrite` is:

```
randomWrite :: (Monad m, FS m, RandomGen m)
            => FilePath -> m (Either FSError ())
```

Each custom monad that is necessary to capture all the operations appears as a constraint in the type. The trick here is that the type class and trait mechanism both already provide an integrated approach to combination, and we can piggyback on top of it for our own goals.

Composing interpretations is not a big deal, either. Suppose you have instances of each custom monad to `IO`:

```
instance FS        IO where ...
instance RandomGen IO where ...
```

Then, you can interpret `randomWrite` directly to `IO` without any additional declaration. Once again, the trick is that type classes and traits already provide a form of

combination — for example, if you declare a type as supporting equality and order, you do not have to do anything special to use those two features in a single piece of code.

In summary, final style monads can be composed in a simple way. This is one of their primary virtues, that they require no complicated machinery, nor do they introduce any runtime overhead. Thus, it is possible to define as many custom monads as your domain requires.

14.2 Initial and Operational Style

Both initial and operational style use data types with a common structure. In the case of initial style, each of the constructors includes a continuation, and in the case of operational style, we always find constructors corresponding to the bind and `return` operations of the monad. Unfortunately, this rigid structure is what makes it difficult to combine monads in any of those styles.

As an example of the problem in initial style, consider the `FS` monad introduced in the previous chapter (shown only in Haskell, for the sake of conciseness):

```
data FS a = WriteFile FilePath String (Either FSError ()     -> FS a)
          | ReadFile  FilePath        (Either FSError String -> FS a)
          | FSDone                    a
```

We would like to combine this functionality with the `RandomGen` monad introduced in the previous section. In initial style, the declaration of that monad is as follows:

```
data RandomGen a = Random Int Int (Int -> RandomGen a)
                 | RGDone         a
```

```
sealed abstract class RandomGen[A]
case class Random[A](lower: Int, upper:Int, k: Int => RandomGen[A])
  extends RandomGen[A]
case class RGDone[A](x: A) extends RandomGen[A]
```

If we want to put those two declarations together into a new `FSRandom` monad, we encounter two problems:

1. We have two different constructors corresponding to `return`, namely `FSDone` from `FS` and `RGDone` from `RandomGen`. But `FSRandom` should have only one — and there is no obvious way to remove constructors from the original data types or even to choose between them.
2. The continuations in `FS` and `RandomGen` refer back to their own monads. But this is not correct if we want the operations to live in `FSRandom`. If you use `WriteFile`, for example, the computation could only continue with more `FS` operations.

Similar problems arise when we consider monads in operational style. The constructors corresponding to `return` and bind each need to be fused into a single operation in the combined monad. Unfortunately, there is no way to combine two constructors in either Haskell or Scala.*

Combining monads written directly in initial or operational style is, in fact, a lost cause. So let us look instead at custom monads built using either the free or freer construction. The idea is that instead of combining the whole monad structure, we only need to combine the parts that describe the operations — the pattern functors for free monads and instructions, or actions, for freer monads:

```
-- pattern functors for FS and RandomGen in initial style
data FSF r = WriteFile FilePath String (Either FSError ()      -> r)
           | ReadFile  FilePath        (Either FSError String  -> r)
data RandomGenF r = Random Int Int     (Int -> r)

data FSRandomF r  = FSF r :+: RandomGenF r
type FSRandom     = Free FSRandomF
```

This approach solves both problems. First of all, the `Done` constructor is given by the `Free` data type, not by the instruction sets. Therefore, by construction, there is only one constructor corresponding to the `return` operation. The `Free` data type is also responsible for typing the recursion — both `FSF` and `RandomGenF` keep the continuation type open by means of an additional type variable `r`. The only missing piece is how to define the `(:+:)` operation, which ought to combine the constructors from both pattern functors.

14.2.1 Coproducts of Functors

The argument to a free or freer monad† should be a type constructor, not a ground type. In Haskell terms, we say that in `Free f`, `f` must be of kind `* -> *`. In Scala terms, if we have `Free[F]`, the argument must have the shape `F[_]`. This rules out the use of `Either` or `\/`, since those types combine ground types. That is, we can combine types in the form `Either Int Bool`, or even `Either (FSF Int) (RandomGenF Bool)`, but not the type constructors themselves:

```
> :kind! Free (Either FSF RandomGenF)  -- Ask for the kind
```

The compiler complains, in fact, about a mismatch in the kinds:

```
Expected kind '*', but 'FSF' has kind '* -> *'
In the first argument of 'Either'
```

*There is some ongoing research on a theory of data type *ornaments* that could make it possible. But at the time of writing, there is no practical implementation of these ideas.

†Remember that the `Free` monad in Scalaz and Cats is actually an example of the latter.

What we need is a data type that takes the type constructors as arguments and chooses which one to apply to a final result type. In recent versions of Haskell's `base` package, you can find a version of that data type, called `Sum`, although in most places this type is called `(:+:)`, instead:

```
data (f :+: g) a = InL (f a) | InR (g a)
```

Scalaz and Cats define types with similar characteristics but using different implementations. Instead of creating a completely new constructor, those types piggyback on their corresponding definitions of a disjunctive type:

```
// In Scalaz
final case class Coproduct[F[_], G[_], A](run: F[A] \/ G[A])
// In Cats
final case class EitherK[F[_], G[_], A](run: Either[F[A], G[A]])
```

These three examples show the three common names used to refer to this notion. Given two functors `f` and `g`, we have built their *sum*, *coproduct*, or higher-kinded version of `Either` (and hence the `K` at the end of the name in Cats).

> **Exercise 14.1.** Show that `Sum f g` — and any of the Scala variants — is a functor, if both `f` and `g` are functors. In other words, implement the following instance:
>
> ```
> instance (Functor f, Functor g) => Functor (f :+: g) where
>   fmap :: (a -> b) -> (f :+: g) a -> (f :+: g) b
> ```

The coproduct of functors provides the combination operation that we were looking for. The same construction works for free and freer monads, since in both cases we need to combine type constructors. Now, if we were to define a computation that uses `FS` and `RandomGen` operations, we can assign it the type `Free (FSF :+: RandomGenF) a`, in Haskell, or the type `Free[Coproduct[FS, RandomGen], A]`, in Scala.

The next question is how to combine interpretations for several pattern functors into a single one. As usual, we are interested in those interpretations that can be described as natural transformations, that is, as functions for a certain monad `M` with the type:

```
interpret :: Free (f :+: g) a -> M a
```

```
def interpret[F[_], G[_], A](program: Free[Coproduct[F, G], A]): M[A]
```

In the previous chapter, we discussed that the free or freer construction already helps us in building such an interpretation. Instead of concerning ourselves with the scaffolding part of the monad — `return`s and binds — we only need to provide an interpretation of the pattern functor or set of instructions:

```
interpret = foldFree interpret'
  where interpret' :: (f :+: g) a -> M a
```

As a consequence, when we use free or freer monads, the question of how to combine interpretations becomes the question of how to combine interpretations from `f` and `g` into a common `M`. Such a function just needs to redirect each request to the corresponding monad:

```
combine :: (f a -> m a) -> (g a -> m a) -> (f :+: g) a -> m a
combine f _ (InL x) = f x
combine _ g (InR x) = g x
```

14.2.2 Automatic Injection

Thanks to coproducts, we can start to describe at least the type you use to combine several custom monads. Unfortunately, their readability is not good. Since operations do not come from either `FSF` or `RandomGenF` but from a different type `FSF :+: RandomGenF`, we are forced to add an additional layer for the constructors of the sum. In turn, that means that we need to redefine all of our smart constructors to build operations in the combined monad:

```
-- FS operations
writeFile :: FilePath -> String
          -> Free (FSF :+: RandomGenF) (Either FSError ())
writeFile path contents = liftF (InL $ WriteFile path contents id)

readFile :: FilePath
         -> Free (FSF :+: RandomGenF) (Either FSError String)
readFile path = liftF (InL $ ReadFile path id)

-- RandomGen operations
random :: Int -> Int
       -> Free (FSF :+: RandomGenF) Int
random lower upper = liftF (InR $ Random lower upper id)
```

This is far from ideal, because it means that we need to redefine smart constructors every time we make a new combination of custom monads. If we now want to use `FS` with a new `Network` monad, the previous definitions of `writeFile` and `readFile` are no longer valid, since they refer specifically to `RandomGen`.

The ideal solution is to make the smart constructors generic with regard to the actual combination of functors used as an argument to `Free`, provided that `FSF` is one of its components. We usually refer to this problem as *automatic injection*: we want the operation to inject itself into the corresponding part of the `Sum` type, and we want the compiler to perform this procedure for us automatically. A well-known technique is described in *Data types à la carte* [Swierstra, 2008].

This construction, when encoded in Haskell, defines a type class `f :<: g1 :+: g2 :+: ...` representing a type constructor `f` that is part of a larger sum. The only method in this type class determines how to reach the right place, essentially by a combination of `InL` and `InR`:

```
class f :<: g where
  inject :: f a -> g a
```

The simplest option is for `f` and `g` to coincide. In that case, the injection does not work:

```
instance f :<: f where
  inject = id
```

The next possibility is for `f` to appear as the left operand to the sum. In that case, we inject the value into the combined functor by means of `InL`:

```
instance f :<: (f :+: g) where
  inject = InL
```

Finally, if `f` is not the left operand, we need to look for it in the right-hand side. This search is recursive, since we might need to traverse several layers of sums until we reach our destination. In each layer, we need to add a layer of `InR`. This presentation assumes that all uses of `(:+:)` are right-nested, that is, that we always write sums of the form `g1 :+: (g2 :+: (g3 :+: ...))`. This is the default parenthesization when writing code by hand and thus works most of the time:

```
instance (f :<: h) => f :<: (g :+: h) where
  inject = InR . inject
```

For more complicated scenarios, there is another construction described in *Composing and Decomposing Data Types* [Bahr, 2014] and available in Hackage as part of the `compdata` package.

Scalaz and Cats use a similar construction under the names `Inject` and and `InjectK`, respectively. In addition to injection, the Scala counterparts also define the inverse operation, called *projection*. When you project a value to a certain functor `f`, you get an answer only if the value was previously injected from `f`. In this section, we do not use this additional power, but other constructions do:

```
sealed abstract class Inject[F[_], G[_]] extends (F ~> G) {
  def inj[A](fa: F[A]): G[A]
  def prj[A](ga: G[A]): Option[F[A]]
}
```

```
sealed abstract class InjectInstances {
  implicit def reflexiveInjectInstance[F[_]]
    :  Inject[F, F] = ???
  implicit def leftInjectInstance[F[_], G[_]]
    : Inject[F, Coproduct[F, G, ?]] = ???
  implicit def rightInjectInstance[F[_], G[_], H[_]]
    (implicit I: Inject[F, G]): Inject[F, Coproduct[H, G, ?]] = ???
}
```

Armed with automatic injection, we can finally create a good interface for our smart constructors. Instead of fixing a certain monadic stack, we generalize to allow any sum in which one of the operands includes the operation. For example, `writeFile` and `readFile` are no longer restricted to `FS` but work for any `Free f` where `FSF` is part of `f`:

```
-- FS operations
writeFile :: (Functor f, (FSF :<: f))
          => FilePath -> String -> Free f (Either FSError ())
writeFile path contents = liftF (inject $ WriteFile path contents id)

readFile :: (Functor f, (FSF :<: f))
         => FilePath -> Free f (Either FSError String)
readFile path = liftF (inject $ ReadFile path id)

-- RandomGen operations
random :: (Functor f, RandomGenF :<: f)
       => Int -> Int -> Free f Int
random lower upper = liftF (inject $ Random lower upper id)
```

The `randomWrite` function that we introduced at the beginning of this chapter — and which motivated our journey — can finally be assigned a type:

```
randomWrite :: (Functor f, FSF :<: f, RandomGenF :<: f)
            => FilePath -> Free f (Either FSError ())
```

Here, the `(:<:)` construction is summoned twice, once per set of instructions that the monad should make available. This type signature documents exactly the requirements of our computation. Furthermore, we are not tied to a specific coproduct of pattern functors. We can use `randomWrite` with any free monad that supports filesystem operations and random generation.

14.3 Extensible Effects

Extensible effects, or *algebraic effects and handlers*, offer an alternative approach to monad transformers for combining operations into a single computation. In

the last few years, many libraries have been written to investigate this approach. At the time of writing, the community seems to have settled on a few designs: `freer-simple` and `freer-effects` in Haskell, `eff` in Scala, and the effects system in PureScript.

In short, extensible effects frameworks use a single monad — usually called `Eff` — that is parametrized by a list of allowed *effects*. Those effects correspond to sets of operations, in a similar fashion to the sets of instructions or actions used to describe freer monads. Computations may request the operations by sending *messages*, which are ultimately turned by *handlers* into actual work. From the other side, the handlers define what to execute per operation, much like interpreters for freer monads.

In contrast to what we have discussed up to this point, the extensible effects frameworks replace the coproduct construction by a `Union` functor parametrized by all the components. More concretely, whereas before we wrote `f :+: g :+: h`, we would now write `Union '[f, g, h]`. Note that in the latter type, we use a new construction: a type-level list, that is, a list of types. The definition of the `Union` data type closely follows that of `(:+:)`. There are two constructors that tell you whether you are in the right position in the list — `This` — or you need to look further — `That`:

```
data Union (rs :: [* -> *]) x where
  This :: f x        -> Union (f : rs) x
  That :: Union rs x -> Union (f : rs) x
```

The `eff` library in Scala does not use type-level lists. Instead, it contains variations of `Union` for up to 12 components, named `fx1` to `fx12`. For the sake of conciseness in this section, we will talk mostly about the Haskell implementation and only describe the differences in the Scala version.

The next step in building our extensible effects monad is to define the notion of a *member* of a type-level list. This is a predicate over types that says that a certain type constructor `f` is part of a list. This definition can be turned into a type class with instances that implement linear search, in a similar fashion to how we defined `(:<:)` to find an element in a coproduct. As in that case, we need an injection function in order to build smart constructors later on:

```
class Member f rs where
  inj :: f x -> Union rs x
instance Member f (f : rs) where
  inj = This
instance {-# OVERLAPPABLE #-} Member f rs => Member f (r : rs) where
  inj = That . inj
```

In Scala's `eff`, the `Member` constraint is written as `MemberIn[F,R]` or `F |= R`. In that library, the definition of `Union` follows a slightly different strategy — it is represented as different case classes that perform a binary search over a list of

effects — but the overall interface is the same. In languages that have integrated support for rows, like PureScript, the declarations of `Union` and `Member` use that feature instead of baking in their own.

The final trick is to declare the `Eff` monad as the freer monad over the union of a set of effects. That is, we make the monad remember which operations are allowed inside of it. We can use either `Program` or `Freer`, as described in Section 13.4. Here we choose the latter:

```
type Eff rs = Freer (Union rs)
```

The same data types we used to declare operations for freer monads serve as effects in this new setting. We can readily reuse `FS` and `RandomGen` from the previous sections:

```
data FS a where
  WriteFile :: FilePath -> String -> FS (Either FSError ())
  ReadFile  :: FilePath           -> FS (Either FSError String)
data RandomGen a where
  Random :: Int -> Int -> RandomGen Int
```

A computation using any of these receives the type `Eff '[FS, RandomGen] a`. Once again, we remark that there is no essential difference between `Freer (FS :+: RandomGen) a` and the former type, merely a different way to combine the sets of operations.

As usual, smart constructors give us syntax that more closely resembles that of the common monad. If in the case of coproducts we used `(:<:)` to allow injection on any sum with a definite member, in the case of extensible effects we use `Member`, instead:

```
writeFile :: Member FS rs
          => FilePath -> String -> Eff rs (Either FSError ())
writeFile path contents = Impure (inj (WriteFile path contents)) Pure

readFile :: Member FS rs
         => FilePath -> Eff rs (Either FSError String)
readFile path = Impure (inj (ReadFile path)) Pure
```

The pattern is always the same: obtain the instruction from the data type of operations, inject it into `Union`, and finally bind it to a simple return instruction. In fact, we can abstract this pattern into a function:

```
send :: Member f rs => f a -> Eff rs a
send x = Impure (inj x) Pure

random :: Member RandomGen rs => Int -> Int -> Eff rs Int
random lower upper = send (Random lower upper)
```

The case of Scala's `eff` is similar, but there are two main differences: the first one is that you need more type annotations as compared to Haskell. The second is that the library uses a specific idiom to declare membership, which leads to better syntax:

```
type _fs[R] = FS |= R  // declare a type alias for membership of FS

def writeFile[R: _fs](path: FilePath, contents: String)
  : Eff[R, FSError \/ Unit]
  = Eff.send[FS, R, FSError \/ Unit](WriteFile(path, contents))
```

Using these smart constructors, computations written with extensible effects do not differ from the other approaches. The type, though, changes to reflect this new framework:

```
randomWrite :: (Member FS rs, Member RandomGen rs)
            => FilePath -> Eff rs (Either FSError ())
randomWrite path = do number <- random 1 100
                      writeFile path (show number)
```

But a value with type `Eff [e1, e2, ...] a` merely describes a computation that uses operations from the effects `e1`, `e2`, ... In order to perform some actual job, we need to handle them. Each *handler*, in this setting, takes care of one of the effects. To make things simpler, let us begin with one of the simpler effects, `Reader`, which provides one operation, `Ask`, to obtain a value from the surrounding context:

```
data Reader r x where
  Ask :: Reader r r
```

Before we continue, we need to introduce an auxiliary operation, `proj`. This function is the inverse of `inj`. It checks whether the value in a `Union` corresponds to a given type and either returns it or gives us back a `Union` with one fewer element:

```
proj :: Union (f : rs) x -> Either (Union rs x) (f x)
proj (This x) = Right x
proj (That y) = Left  y
```

It is customary to call the handlers after the name of the effect. In our case, `runReader` takes an `Eff` value that contains `Reader` and strips it out. We then have one fewer effect in the list. The rest of them are handled further:

```
runReader :: r -> Eff (Reader r : rs) a -> Eff rs a
runReader r m = loop m
  where loop (Pure   x)   = return x
        loop (Impure a k) = case proj a of
                              Right Ask -> loop (k r)
                              Left  op  -> Impure op (loop . k)
```

You should think of `runReader` as a function that takes messages from a queue, handles some of them, and dispatches those that are for `Reader` requests. This handling keeps going until there are no more messages. For that reason, the `Impure` branch always calls `loop` recursively.

In fact, `Impure` is where all the magic happens. Remember that in the freer monad, an `Impure` value is made of an instruction — which we refer to as `a` in the code — and a continuation `k`. The first thing we do, by using the `proj` operation defined earlier, is to check whether the request is intended to be captured by `runReader`. If that is the case, indicated by a `Right` value, then we do our job and call the continuation. The job is simply feeding the environment `r`.

A `Left` return value from `proj` indicates that the message uses another effect different from `Reader`. In that case, we leave the operation untouched — we return another `Impure` with the same `op` — but modify the continuation slightly. The reason is that there might be further `Reader` requests later, and we also need to handle those. For that reason, we compose our `loop` function with the original continuation.

By repeatedly calling handlers, you reach a point where you have exhausted all the effects. This is visible in the type of the computation, `Eff [] a`. Since there are no effects, that computation can be turned into a pure value:

```
run :: Eff '[] a -> a
run (Pure x) = x
-- There is no way to get Impure here
```

The handler for `Reader` effects is one of the simplest. In general, running a handler changes the return type. Here are the signatures for the handlers of the `State`, `Error`, and `NonDet` effects in the `freer-simple` package:

```
runState    :: s -> Eff (State s : rs) a -> Eff rs (a, s)
runError    ::      Eff (Error e : rs) a -> Eff rs (Either e a)
makeChoiceA :: Alternative f
                 => Eff (NonDet  : rs) a -> Eff rs (f a)
```

There are, in fact, a lot of commonalities between these `run` functions and the corresponding functions used with monad transformers. Usually, after executing all of your effects, you end up with a type like `Either String [Int]`, which indicates possible failure and non-determinism in the answer. In a similar fashion to monad transformers, the order in which you run the effects determines the final outcome.

An interesting feature of the extensible effects framework is that handlers are not required to completely remove one layer of effects. They can transform one effect into another or even keep the list of effects untouched but do some work inside. An example of the former is given by the following handler, which instead of removing all `Reader` effects, translates them into a `State` effect:

```
data State s x where
  Get :: State s s
  Put :: s -> State s ()
```

In particular, a call to `Ask` is turned into a call to `Get`. Note that, as part of the translation, we need to perform new injections, either via `inj` or using the `That` constructor directly:

```
runReaderS :: Eff (Reader r : rs) a -> Eff (State r : rs) a
runReader m = loop m
  where
    loop :: Eff (Reader r : rs) a -> Eff (State r : rs) a
    loop (Pure   x)   = return x
    loop (Impure a k) = case proj a of
                          Right Ask -> Impure (inj Get) (loop . k)
                          Left  op  -> Impure (That op) (loop . k)
```

After this translation, you can call `runState` to execute the computation. This ability of effect handlers makes it easy to create you own restrictive monad that can ultimately be expressed in terms of other handlers.

This point of view — computations request operations via messages and handlers intercept those that are targeted to them — provides a different answer to the problem of control operations, such as `catchError`. The problem of how to lift `catchError` generically required such a convoluted solution in the monad transformers arena that we devoted the whole of Chapter 12 to it. The solution in this new framework is that `throwError` is an operation, but we have been all wrong about `catchError`: it is a handler! This makes sense, as catching an error is exactly like intercepting the `throwError` message and doing something in response.

Let us discuss a simpler version of `catchError`, called `handleError`. The difference between them is that the former may throw a new exception, whereas the latter has to handle the error completely. This is visible in the types, since after `handleError` performs its duty, the effect `Error e` is no longer part of the list:

```
data Error e x where
  Error :: e -> Error e a

handleError :: Eff (Error e : rs) a -> (e -> Eff rs a) -> Eff rs a
handleError m c = loop m
  where
    loop (Pure   x)   = return x
    loop (Impure a k) = case proj a of
                          Right (Error e) -> c e
                          Left  op        -> Impure op (loop . k)
```

The key point here is that when we get an `Error` message, we forget about the rest of the computation to be found in the continuation `k`, and we turn instead

to the exception handler `c`. For the record, the type of the more complex handler, `catchError`, is as follows:

```
catchError :: Member (Error e) rs
           => Eff rs a -> (e -> Eff rs a) -> Eff rs a
```

Note that the list of effects `rs` does not change after the handler runs. In contrast, the `Error e` effect is removed from the list in `handleError`. Since `rs` in `catchError` contains `Error e` in every place it arises, we are allowed to throw new exceptions from the exception handler.

We have seen that when talking about effects, there are three modes of operation: injection for the smart constructors, removal for the `run` family of handlers, and interception while keeping the same list of effects, like `handleError` does. Scala's `eff` library is more explicit than its Haskell counterpart and provides three different traits to declare in which mode we are operating. When we write `F |= R`, we refer to the first case, we want `F` to appear in `R`. If we write `F <= R`, that means that `F` appears in `R`, and we want to remove it. Finally, `F /= R` indicates that we are going to keep `F` in the list `R`, but we are going to modify its operation in some way.

The framework outlined here works flawlessly until you need to perform some file system operation, random generation, network communication, etc. In those cases, you want your handler to run in the `IO` monad.* The `FS` and `RandomGen` effects are prime examples. The naïve approach to writing a handler for those operations leads to the following signatures:

```
runFS        :: Eff (FS        : rs) a -> IO (Eff rs a)
runRandomGen :: Eff (RandomGen : rs) a -> IO (Eff rs a)
```

The problem is that now you cannot (easily) handle computations that use both `FS` and `RandomGen`, because the output of one of them is already in `IO`.

Both the Haskell and Scala variants of extensible effects include a special mechanism for handlers that run inside of a monad. The essence of the idea is to wrap such monads as effects. As a result, handling any other effect using that monad becomes a translation from the effect to the wrapped monad. In code, we introduce a `Lift` type to wrap monadic actions:

```
data Lift m a = Lift (m a)
```

The handlers for `FS` and `RandomGen` now become translations from those effects into `Lift IO`:

```
runFS        :: Member (Lift IO) rs
             => Eff (FS        : rs) a -> Eff rs a
runRandomGen :: Member (Lift IO) rs
             => Eff (RandomGen : rs) a -> Eff rs a
```

*Because you are using some sort of `IO`, even in Scala, right?

In contrast to the case of computations that only used "pure" effects, the final type of a computation after handling all effects is not `Eff '[] a` but `Eff (Lift IO : '[]) a`. This means that we cannot just extract the final value of a computation with `run`, we have to execute the monadic actions. We therefore require a different `runM`:

```
runM :: Monad m => Eff (Lift m : '[]) a -> m a
runM m = loop m
  where
    loop (Pure x)     = return x
    loop (Impure a k) = case proj a of
                          Right (Lift a') -> do x <- a'
                                                loop (k x)
                          Left _ -> error "this can never happen"
```

The only caveat to this solution is that you have to forbid more than one appearance of `Lift` in your list of effects. Otherwise, you would not be able to execute the computation — remember that there is no general way to combine two monads. For that specific goal, both Haskell and Scala libraries define a `LastMember` constraint that checks that a given effect is the last one in the list. Since you cannot have two different effects as the last member at the same time, this effectively prevents the wrong situation from occurring.

Extensible effects are an exciting new approach to combining different sets of operations. You only need to know about one monad, `Eff`, which encapsulates the ideas of returning pure values and binding computations in a chain. On the other hand, thanks to libraries like Freestyle that automate away most of the boilerplate, the approach of using free or freer monads and coproducts still retains most of its benefits without the need for a new mental framework.

15 Performance of Free Monads

The previous two chapters have a clear message: build you own monads to bring the language in which you program closer to the language in which your problems are modeled. Both free and freer monads take care of most of the boilerplate, leaving only the juicy bits to be developed.

Unfortunately, the basic implementation of free monads does not perform well under all possible circumstances. For example, interleaving inspection and generation of a computation imposes a performance loss. In this chapter, we review other encodings for the same concept that lead to more efficient outcomes. This chapter contains some complex pieces of code, but before diving into monads, we will gain some intuition by looking at similar performance problems in the context of lists.

15.1 Left-nested Concatenation

Lists are one of the bedrocks of functional programming. In this section, we are going to explore some of the problems that arise from a naïve implementation of some operations, in particular of list concatenation. As we will see in the next section, the bind operation in a free monad has a similar behavior to concatenation. This means that many of the problems also translate to that new setting. Fortunately, the solutions do, too.

Throughout this book, we have discussed lists in the context of non-determinism, but for this chapter, we are going back to the basics. Remember that the data type of lists has two constructors, which build an empty list or build a list by appending one element to the front of an existing list:

```
data [a] = [] | a : [a]
```

One of the basic operations over two lists is to concatenate them. In order to do so, we recurse over the left argument until we have no more elements to append to the front:

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

A quick analysis of this function reveals that when you append two lists, `xs` and `ys`, you need to perform as many steps as there are elements in `xs`. We say that concatenating two lists takes time proportional to the length of the first list (i.e., the first argument to the concatenation function).

Reversing a list is another simple operation on lists. The beginners' implementation works by concatenating, at each step, the first element to the end of the list returned from recursion. We need to use list concatenation, because the list data type does not offer any primitive facility for placing elements at the end of a list:

```
reverse :: [a] -> [a]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

This implementation, albeit simple, is far from efficient. Take, for example, a list with three elements, `[a, b, c]`. This is the evaluation trace for reversing it:

```
   reverse [a, b, c]
≡ reverse [b, c] ++ [a]
≡ (reverse [c] ++ [b]) ++ [a]
≡ ((reverse [] ++ [c]) ++ [b]) ++ [a]
≡ (([] ++ [c]) ++ [b]) ++ [a]
≡ -- concatenation takes 0 steps
   ([c] ++ [b]) ++ [a]
≡ -- concatenation takes 1 step
   [c, b] ++ [a]
≡ -- concatenation takes 2 steps
   [c, b, a]
```

The problem is that the calls to concatenation are nested in such a way that every subsequent step gets a larger list on the left-hand side. This means that every step costs you more, since the concatenation operation is linear over the length of its left argument, which grows at every step. In general, for a list of length n, this definition of reverse takes a number of steps equal to:

$$1 + 2 + \cdots + (n - 1) = \frac{n \cdot (n - 1)}{2}$$

In other words, reversing a list takes time proportional to the *square* of the length of the list. Really bad, indeed.

Reversing is just one manifestation of a general problem with list concatenation. Whereas right-nested concatenation — `[a] ++ ([b] ++ [c])` — performs well, left-nested concatenation — `([a] ++ [b]) ++ [c]` — performs poorly. Even more, both expressions give the same result, thanks to the associativity of the operation. Our ultimate goal is to be able to nest concatenations as we want but still have good performance.

Note that for the specific case of reverse, there is an ad-hoc solution that uses an accumulator:

```
reverse xs = reverse' xs []
  where reverse' []     acc = acc
        reverse' (y:ys) acc = reverse' ys (x:acc)
```

In this section, though, we aim for a generic solution we can use for any function that produces left-nested concatenations. Furthermore, we want this solution to be as transparent as possible, so we can keep writing our code in the same style as we use for normal lists.

15.1.1 Difference Lists

Difference lists were introduced in 1984 by John Hughes as a remedy for the poor performance of left-nested concatenation. A difference list is merely a function from lists to lists:

```
data DList a = DList ([a] -> [a])
```

However, not every function is allowed inside of the `DList` constructor. A compatible function should be one that prepends some values to the argument it receives. In particular, we obtain the underlying list represented by a difference list by starting this process with the empty list:

```
toList :: DList a -> [a]
toList (DList dl) = dl []
```

This offers us a first definition of an "empty" difference list. We need to find a function that, when it is applied to `[]`, gives us back another empty list. This is none other than the identity function:

```
empty :: DList a
empty = DList id
```

We can also inject an already constructed list into a difference list by representing it with the function that concatenates that list. In other words:

```
fromList :: [a] -> DList a
fromList xs = DList (xs ++)
```

Our goal was to obtain better performance for concatenating lists. How do difference lists help in that respect? Consider the case in which you start with two regular lists, `xs` and `ys`, which you then turn into difference lists:

```
   fromList xs   `concatenate` fromList ys
≡ DList (xs ++) `concatenate` DList (ys ++)
```

What definition of `concatenate` gives the result `DList ((xs ++ ys) ++)`? The solution is to define this function as function composition:

```
concatenate :: DList a -> DList a -> DList a
DList xs `concatenate` DList ys = DList (xs . ys)
```

By performing this trick, you turn an operation that is linear over the length of the first argument into a constant operation. Let us look at what happens if we try to nest three lists in the wrong way:

```
   (fromList xs   `concatenate` fromList ys)   `concatenate` fromList zs
≡ (DList (xs ++) `concatenate` DList (ys ++)) `concatenate` DList (zs ++)
≡ DList ((xs ++) . (ys ++)) `concatenate` DList (zs ++)
≡ DList (((xs ++) . (ys ++)) . (zs ++))
≡ DList (\ws -> xs ++ (ys ++ (zs ++ ws)))
```

The left-nested concatenations using `concatenate` are turned into *right-nested* concatenations at the normal list level. This suggests that a good definition of `reverse` is:

```
reverse xs = toList (reverse' xs)
  where reverse' []     = empty
        reverse' (y:ys) = reverse' ys `concatenate` fromList [y]
```

With a simple change of representation, our naïve definition of `reverse` is made asymptotically more efficient. In the case of free monads, we need to do a bit more work to find the right "difference" representation, but the concept is quite similar: use a function, like we do in `DList`, to represent the operations over a data structure. But instead of a list, we base our construction on the `Free` data type.

15.1.2 Church and Scott Encodings

Right after `map` and `filter`, functional programmers are introduced to folds (also known as `reduce` in some languages). The most commonly used folding function in Haskell, `foldr`, combines the elements of a list nesting to the right:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

So we can define the sum of all the elements of a list as `foldr (+) 0` or the concatenation of the lists `xs` and `ys` as `foldr (:) ys xs`.

Beneath its simple appearance, `foldr` holds a secret: describing how to fold a list is equivalent to describing the list *itself*. Let us define the type of encoded lists:

```
data EList a = EList (forall b. (a -> b -> b) -> b -> b)
```

A value of this type must be a function that, *for any type* `b`, is able to perform the right fold. We can move back and forth between the usual and the encoded representation of lists via the following functions:

```
toEList :: [a] -> EList a
toEList xs = EList (\f v -> foldr f v xs)

fromEList :: EList a -> [a]
fromEList (EList fold) = fold (:) []
```

The type `EList` is called the *Church encoding*[*] of lists. We can define such an encoding for any algebraic data type by swapping the normal data type definition with its fold function.

Of course, many encodings to and from lists are possible.[†] The reason that this encoding is renowned is that it also provides a translation for every primitive list operation: building (`cons` and `nil`), checking whether a list is empty, and destructuring (`head` and `tail`):

```
nil               = EList (\_ v -> v)
cons x (EList xs) = EList (\f v -> f x (xs f v))

null (EList xs)   = xs (\_ _ -> False) True
head (EList xs)   = xs (\x _ -> x) (error "empty list")
tail (EList xs)   = EList (\f v -> xs (\h t -> \g -> g h (t f))
                                      (\_ -> v) (\_ v' -> v'))
```

Thus, every function that operates on lists can be translated into a function operating on Church-encoded lists. The bad news is that computing the tail of a list involves a huge amount of work. Church encoding works well for composing fold functions but not so much for other functions.

Instead of folding on its own, it is useful to focus on the general shape of pattern matching. In fact, we can define a function `matchList` that turns the pattern matching primitive into a higher-order function:

[*] To be completely fair to history, Church introduced this encoding in an untyped setting. Boehm and Berarducci were the designers of the typed version. But it is now common to refer to both as Church encodings.

[†] You can read about them, and their advantages and pitfalls, in *Church Encoding of Data Types Considered Harmful for Implementations* [Koopman, Plasmeijer, and Jansen, 2014].

```
matchList :: (a -> [a] -> b) -> b -> [a] -> b
matchList _ z []     = z
matchList f _ (x:xs) = f x xs
```

Note the lack of recursion in the second branch, in contrast to what happened with `foldr`. As we did for Church encoding, we can represent lists as functions with a shape similar to `matchList`. And also as in the case of Church encoding, we drop the final `[a]` argument from the type:

```
data SList a = SList (forall b. (a -> [a] -> b) -> b -> b)
```

Defining a data type by the type of its pattern match operation is known as *Scott encoding*. You can think of it as representing a data type by its continuations. For example, if you want to define a function from a list `[a]` into some type `b`, you need to specify what to do in case the list turns out to be empty and also what to do for the "cons" case. Those are precisely the two parameters required by the function in `SList`.

> **Exercise 15.1.** Define the primitive operations `nil`, `cons`, `null`, `head`, and `tail` for Scott-encoded lists. Compare these definitions with the Church encodings of the same functions.

Whenever you Scott-encode a data type, you are trading the performance of pattern matching for the performance of application, since pattern matching is now represented as a higher-order function. If your operation on lists relies heavily on pattern matching, without big computation steps in between, it might benefit from using the Scott encoding.

15.1.3 Queues, Trees, Sequences

At the beginning of this chapter, we focused on a specific use case of lists that leads to poor performance, namely the left-nesting of concatenation. This is, of course, not the only scenario with the same implications. For example, if you need to pop and push elements from both sides of a sequence, a list is a poor choice as a data structure. In contrast, if you want to do a lot of searches, you might be better off using a search tree instead.

The point is that there is no catch-all trick or data structure that performs well *in every possible situation*.[*] Take time to weigh your needs when choosing how to represent your ordered sequences of data. This same advice also holds true when speaking about free monads.

[*]Although finger trees as implemented in `Data.Sequence`, in the `containers` package, gets close.

15.2 Left-nested Binds

While exploring the `reverse` operation for lists, we realized that nesting concatenation operations to the left leads to poor performance. The same problem arises with the bind operation of monads. After all, free and freer monads are also lists of instructions, where bind sequences two of them.

Let us make this problem more concrete by looking at a specific example.* Consider the following data type of binary trees:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
            deriving Functor
```

> **Exercise 15.2.** `Tree` can be rewritten as a free monad. What is the pattern functor? To which constructor in `Free` does `Leaf` correspond?

This `Tree` type can be given a monad instance, in a similar fashion to what we did for `Free`. The `(<*>)` and bind operations are pushed down the leaves, and at that point we apply the function as required:

```
instance Applicative Tree where
  pure = Leaf
  Leaf f   <*> t = fmap f t
  Node l r <*> t = Node (l <*> t) (r <*> t)

instance Monad Tree where
  return = Leaf
  Leaf x   >>= f = f x
  Node l r >>= f = Node (l >>= f) (r >>= f)
```

The `fullTree` function creates a tree for which all leaves are at a given height. In this example, the content of all the leaves is equal to the given height, but this is not relevant for our explanation:

```
fullTree :: Integer -> Tree Integer
fullTree 0 = Leaf 0
fullTree n = do fullTree (n - 1)
                Node (Leaf n) (Leaf n)
```

Consider now the trace for a call like `fullTree 3`:

```
   fullTree 3
≡ fullTree 2 >>= Node (Leaf 3) (Leaf 3)
   -- from now on, nll n ≡ Node (Leaf n) (Leaf n)
```

*This example was first described in *Asymptotic Improvement of Computations over Free Monads* [Voigtländer, 2008] and extended in *Problem Set: The Codensity Transformation* [Yang, 2012].

```
≡ (fullTree 1 >>= \_ -> nll 2) >>= \_ -> nll 3
≡ ((fullTree 0 >>= \_ -> nll 1) >>= \_ -> nll 2) >>= \_ -> nll 3
≡ ((return 0   >>= \_ -> nll 1) >>= \_ -> nll 2) >>= \_ -> nll 3
```

As you can see, `fullTree` produces left-nested binds. This is problematic, because by construction, each call to the bind function has to traverse the entire structure until it gets to the leaves. As a result, we are traversing the same part of the structure over and over, in the same way that the left-nested concatenation in `reverse` forced us to traverse the same part of a list several times.

Unfortunately, the problem is not in the specifics of the `Tree` data type. Here is the definition of bind for the free monad, as introduced in Chapter 13:

```
Pure x >>= f = f x
Free x >>= f = Free (fmap (>>= f) x)
```

The shape of the function closely matches the one for `Tree`. At every step, the bind is pushed down recursively until it finds a `Pure` constructor (which was called `Leaf` in the case of `Tree`). If we write a monadic computation that leads to left-nested binds, the same problem arises. Since we cannot guarantee that all our binds are right-nested, we ought to find a solution to this performance problem. We will use the tricks from the list case as inspiration.

15.2.1 The Codensity Transformation

Without further delay, here is the data type that solves our problems with left-nested binds:*

```
data Codensity m a
  = Codensity { runCodensity :: forall b. (a -> m b) -> m b }

instance Functor (Codensity k) where
  fmap f (Codensity m) = Codensity $ \k -> m (\x -> k (f x))

instance Applicative (Codensity f) where
  pure x = Codensity $ \k -> k x
  Codensity f <*> Codensity x
         = Codensity $ \bfr -> f (\ab -> x (\y -> bfr (ab y)))

instance Monad (Codensity m) where
  return x = Codensity $ \k -> k x
  Codensity x >>= f
           = Codensity $ \k -> x (\y -> runCodensity (f y) k)
```

*The instances are taken from the `kan-extensions` package by Edward Kmett.

This data type forms a monad regardless of the type constructor `m` given as an argument. With only this code to judge, what we have here just looks like one more case of pointless type trickery. Let us set as our goal to convert a `Codensity m a` into a regular `m a`. This makes `Codensity` more useful, once we consider the argument `m` to be a monad. In that case, we can feed the `return` function, of type `a -> m a`, to the `Codensity` to get back the regular `m a`:

```
lower :: Monad m => Codensity m a -> m a
lower (Codensity x) = x return
```

The other direction for conversion also requires a monadic interface. We start with `m a` and we want to obtain something of type `(a -> m b) -> m b`. The combination of both elements resembles the type of the bind operation. In fact, this conversion is the partial application of bind:

```
lift :: Monad m => m a -> Codensity m a
lift m = Codensity (m >>=)
```

Now, we start to see the similarities to the difference list transformation. First, we use a function instead of a value of the `Free` data type. Second, we build new computations by partially applying the operation with the behavior we want to improve — concatenation for difference lists, bind for `Codensity` monads. Finally, we go back to the regular representation by feeding in the "identity element" for each operation — the empty list when we're talking about concatenation, `return` for monadic bind.

As in the case of difference lists, the result of using `Codensity m` over `m` itself is that left-nested binds are turned into right-nested binds. In cases like the `Tree` monad in the introduction, this may even change a quadratic algorithm into a linear one. Seeing that this is the case is much harder than for difference lists, though. What is important, however, is the practical application more than the proof itself. For those interested, the original paper [Voigtländer, 2008] contains all the details.

The `MonadFree` interface. With the introduction of `Codensity`, you now have two ways to express a free monad over a functor `f`. The first option is to use `Free f`, but you can also use `Codensity (Free f)` to improve certain usage scenarios. Choosing one over the other in an application can lead to premature optimization: you decide which data structure to use before knowing where it will be used later on. The `free` package for Haskell solves this problem in a way that has appeared throughout this book several times, by introducing a new type class.

The `MonadFree` type class ties a functor `f` with one of its possible free monad implementations. The basic operation in this class is `wrap`, which embeds a computation where the first layer is the functor `f` into the free monad itself. For example, it embeds `f (Free f) a` as `Free f a`:

```
class MonadFree f m | m -> f where
  wrap :: f (m a) -> m a
```

We already know a couple of instances of this class:

```
instance Functor f => MonadFree f (Free f) where
  wrap = Free
instance (Functor f, MonadFree f m) => MonadFree f (Codensity m) where
  wrap  t = Codensity $ \h -> wrap (fmap (\p -> runCodensity p h) t)
         -- Based on the code of 'kan-extensions'
```

If you write your function against the `MonadFree` interface, you can effectively delay your choice of structure for computation. The `Codensity` monad takes advantage of this fact to provide the following combinator:

```
improve :: Functor f => (forall m. MonadFree f m => m a) -> Free f a
improve m = lower m
```

The type signature tells us that `m` is a computation independent of the representation of free monads. For that reason, `improve` can use it with `Codensity (Free f)`, providing better nesting of binds. Finally, the `Codensity` layer is stripped out via `lower`, leaving us with a normal `Free f` computation.

The last question is how to make our operations independent of the free monad representation. Up to this point, we have been building our smart constructors by hand. In Section 13.3.1, though, we introduced the `liftF` combinator for that situation:

```
liftF :: Functor f => f a -> Free f a
liftF = Free . fmap return
```

This function can be generalized to work over any `MonadFree` by using the embedding function provided by that type class:

```
liftF :: (Functor f, Monad m, MonadFree f m) => f a -> m a
liftF = wrap . fmap return
```

By using `MonadFree` along with the different approaches to the combining operations presented in Chapter 14, our smart constructors obtain a huge degree of generality. We do not fix either the complete set of operations in the monad — we just declare the ones we need — or the way in which instructions are tied together — the free monad part.

15.2.2 Church and Scott Encodings

Church and Scott encodings alleviate the need to pattern match over every list constructor, replacing them with applications of functions that specify how to continue the operation of each one. It is instructive to consider how to apply this trick to the `Free` data type:

```
data Free f a = Pure a | Free (f (Free f a))
```

1. In the Scott encoding, the value is replaced by a function that specifies all the possibilities for continuing the computation. The arguments to those functions correspond to the values we would obtain by pattern matching:

   ```
   data ScottFree f a
     = ScottFree { runScottFree :: forall r. (a -> r)
                                -> (f (ScottFree f a) -> r)
                                -> r }
   ```

2. Church encoding goes one step further and replaces the data type with its fold function. In practice, this means that recursive appearances of `Free` are also substituted by the result type `r`:

   ```
   data ChurchFree f a
     = ChurchFree { runChurchFree :: forall r. (a -> r)
                                  -> (f r -> r) -> r }
   ```

   This data type is available under the name `F` in Haskell's `free` package.

This technique is not only available to free monads. It can be used for any custom monad, leading to similar improvements, and is known as a *continuation-passing transformation*. For example, the `parsec` parser combinator library defines its core data type as:

```
newtype ParsecT s u m a
    = ParsecT { unParser :: forall r. State s u
                         -> (a -> State s u -> ParseError -> m r)
                         -> (ParseError -> m r)
                         -> (a -> State s u -> ParseError -> m r)
                         -> (ParseError -> m r)
                         -> m r }
```

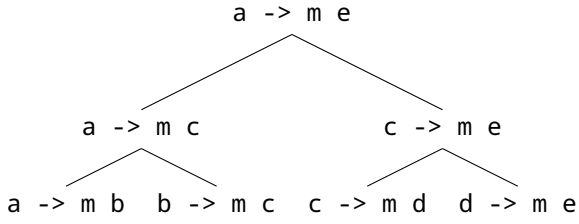
In this case, we are dealing with a monad transformer, and for that reason the result type — which we mark as `r` — is wrapped in an additional layer `m`. Disregarding that fact, `ParsecT` defines a data type with four constructors, which correspond to the four possible states of a parser: an OK state, an error state, and states for the parser having consumed some input or not.

15.2.3 Reflection Without Remorse

We finished our discussion of the performance of lists by remarking that, depending on the usage pattern of our data, a data structure such as a queue or a binary tree could be a better choice than a list. We also described how a free monad is essentially a list of instructions. The remaining question is then: what does a "binary tree of monads" look like?

Answering this question is not useless. As the paper *Reflection without remorse* [Ploeg and Kiselyov, 2014] explains, if your application interleaves building computations and inspecting them — for example, to optimize the computation as it is built — then none of the previous solutions perform well. A binary tree works best, because it allows you to access intermediate results, instead of forcing the computation of the whole list of instructions, in order to optimize them.

Binary trees of monadic computations are also known as *type-aligned sequences*. In such trees, each leaf holds a computation `a -> m b`. But each leaf may need to use different types, `a` and `b`. For the whole tree to be valid, the output type of one leaf must coincide with the input type of its immediate sibling. This is the "type-aligned" part in the sequence. Graphically, a type-aligned sequence with four leaves might look like this:



Each internal node is just the composition of the two subtrees beneath it. For example, the node whose children are the leaves `a -> m b` and `b -> m c` represents a computation `a -> m c`. This same idea is repeated until we reach the root of the tree.

In code terms, type-aligned sequences have the same constructors as binary trees, but they take care of the alignment of types. In Haskell, this is represented by the following GADT:

```
data TASeq m a b where
  Leaf :: (a -> m b)                  -> TASeq m a b
  Node :: TASeq m a b -> TASeq m b c -> TASeq m a c
```

One place where type-aligned sequences are immediately useful is in freer monads. Recall the definition of freer using two constructors, introduced in Section 13.4.1:

```
data Freer instr a where
  Pure   :: a -> Freer instr a
  Impure :: instr a -> (a -> Freer instr b)
         -> Freer instr b
```

We can replace the second argument of `Impure` with a type-aligned sequence:

```
data TASFreer instr a where
  TASPure   :: a -> TASFreer instr a
  TASImpure :: instr a -> TASeq (TASFreer instr) a b
            -> TASFreer instr b
```

By doing so, we get better support for inspecting a computation before interpreting it.

15.2.4 Which One to Choose?

In these last few sections, we introduced several techniques for tackling some of the performance problems of free monads. Unfortunately, there is no easy recipe for deciding which is the best for each application. `Codensity`, for example, deals with the left-nested bind problem, but most programs use a mix of left and right-nested binds. Scott and Church encodings usually lead to slightly better performance than basic free monads, but they may also lead to increased memory usage. The best approach for deciding which free monad implementation to use, therefore, is to benchmark them in realistic situations. In the Haskell world, `criterion` is a good library to use to measure performance.

We have already discussed how `parsec` uses Scott/Church encodings to squeeze out more performance. Other libraries, like `conduit`, use the `Codensity` approach, instead. The implementation of extensible effects in `freer-simple` uses type-aligned sequences. These differences are due to the different usage patterns of each library. In each case, the authors performed some tests on their real-world performance characteristics before choosing one approach over the other.

Part V

Diving into Theory

16

A Roadmap

This book has approached monads from a pragmatic point of view. We came to the monad concept by abstracting over a common pattern that appears in many different data types. Historically, monads were first defined in a branch of mathematics called *category theory* and then imported to programming by Eugenio Moggi [Moggi, 1991] and Philip Wadler [Wadler, 1995].

Note that from a strictly practical point of view, Parts I to IV of this book are enough for a functional programmer. The goal of the last part of this book is to look at monads from other points of view, in order to gain a deeper understanding. Each chapter is devoted to a particular perspective on the concept of monads.

Chapter 17. The first way we will look at monads is as generalized monoids. Under this generalization, the `mempty` and `mappend` operations from the `Monoid` type class correspond to the `return` and `join` operations from the `Monad` type class. In fact, category theorists *define* monads as a particular case of monoids.

§17.2 and §17.3. Unfortunately, before we can dive into these ideas, we need to introduce several categorical terms, such as *category* and *functor*. We will see that category theorists use these notions in a more general sense. For example, the `Functor` concept we use in functional programming is just a particular case of categorical functor: an endofunctor. The first sections of Chapter 17 are devoted to introducing the terms, along with several examples. Do not be afraid of the formal nature of the definitions — it is enough for a first read just to forge an intuition of these concepts.

§17.4. The final section of Chapter 17 applies these notions to a specific category, known as the *category of endofunctors*. In particular, a monad is defined as a monoid in that category. However, we will not stop there, as understanding that

particular category also sheds light on the complex monad transformer operations described in Chapter 12.

Chapter 18. The second way to look at monads is as the composition of two functors that are themselves related in a particular way, called an *adjoint* relation. A prime example is given by the `State` monad:

```
newtype State s a = State { runState :: s -> (a, s) }
```

Instead of a monolithic block, we can see `State` as the composition of the functor `(, s)` — which pairs any value `a` with an output value of type `s` — and the functor `(s ->)` — which adds an input of type `s`. Since these two functors are adjoint, their composition is automatically a monad. This notion of adjointness is the main topic of the chapter.

§18.1. We first introduce the notion decoupled from monads. Since this is a highly abstract idea, we discuss several examples to make it more concrete.

§18.2 and 18.3. Then we describe the construction that builds a monad out of an adjunction. This path will naturally lead us to a discussion of *comonads*, since they can also be built out of adjunctions, by a similar mechanism.

§18.4. Part IV introduced the concept of *free* monad on an intuitive level: it allows us to build a monad "for free." Once we introduce the notion of adjoint functor, we can sharpen this concept. The construction of a free monad is the adjoint to the *forgetful* functor we will describe.

If you want to dive deeper into the categorical side of functional programming, *Category Theory for Programmers* [Milewski, 2014] provides an excellent account. For the more mathematically inclined, the classic reference is *Categories for the Working Mathematician* [Mac Lane, 1998]. A more modern point of view is given by *Category Theory in Context* [Riehl, 2016].

17 Just a Monoid!

> A monad is just a monoid in the category of endofunctors, what's the problem?
>
> Fictionally attributed to Philip Wadler by James Iry, based on *Categories for the Working Mathematician* [Mac Lane, 1998]

This chapter has the explicit goal of explaining all the pieces that make up the famous sentence heading this chapter. If you just want to gain some intuition, the first section provides a conceptual overview without using too much jargon.

Alas, it is impossible to understand the connection between monoids and monads at a deeper level without introducing several notions such as *category* and *functor*. We introduce all these terms one by one, until we finally reach our goal: the definition of monad as a monoid in a certain category.

17.1 Quick Summary

Before getting into all the murky details, we will develop some intuition about why monoids and monads are related at all. In particular, we are going to show how `Monoid`'s `mempty` value is related to `Monad`'s `return` function and how `Monoid`'s `mappend` is related in the same manner to `Monad`'s `join`.

Remember that monoids were defined in Section 5.2 as types along with a couple of operations. We represent this concept in Haskell using the `Monoid` class:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

Let's first make a couple of changes to this definition. The identity element `mempty` is given here as a constant element, but it may also be seen as a function from `()` to `m`. This is, in fact, a general construction — any value `a` is equivalent to a function `() -> a`:

```
toUnitFunction :: a -> (() -> a)
toUnitFunction x = \() -> x

fromUnitFunction :: (() -> a) -> a
fromUnitFunction f = f ()
```

Furthermore, the composition of both functions in any possible order is equivalent to the identity function. In other words, you get back the same value you put in:

```
   toUnitFunction (fromUnitFunction f)
≡ -- definition of fromUnitFunction
   toUnitFunction (f ())
≡ -- definition of toUnitFunction
   \() -> f ()
≡ -- remove explicit application
   f

   fromUnitFunction (toUnitFunction x)
≡ -- definition of toUnitFunction
   fromUnitFunction (\() -> x)
≡ -- definition of fromUnitFunction
   (\() -> x) ()
≡ -- function application
   x
```

Instead of using the usual curried syntax of functions, we pass the two values to `mappend` using a tuple. The final shape of the `Monoid` class then becomes:

```
class Monoid m where
  mempty  :: ()     -> m
  mappend :: (m, m) -> m
```

Monads live in a higher world, where some of the constructions have been generalized to work over functors — think of type constructors for the time being — instead of ground types. The first concept to be generalized is that of function. Functions between functors are called *natural transformations*. A natural transformation must work parametrically over any element type.

In code, the type of natural transformations between `f` and `g` is defined as:

```
type f :=>: g = forall a. f a -> g a
```

As an example, we can write a natural transformation between the list and `Maybe` functors:

```
safeHead :: [] :=>: Maybe  -- that is, [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

> **Exercise 17.1.** Write a natural transformation between the list functor and itself. Think first of the type of that function and then about implementations that fit this type.

The second construction to generalize is the idea of tupling. In the world of ground types, `(,)` is a type constructor that turns two ground types into another one, and `()` is a ground type. Furthermore, the types `(a, ())`, `((), a)`, and `a` are all equivalent, in the sense that you can move from one to the other without losing any information.

> **Exercise 17.2.** Write the functions that witness the equivalence between the three types `(a, ())`, `((), a)`, and `a`.

We need to find a similar construction but that works over functors: a way to combine two of them and a type constructor that works in the same way that `()` does for tuples. We have already described such constructions, functor composition and the identity functor, in other parts of this book:

```
data (f :.: g) a = Compose (f (g a)) deriving Functor
data Identity  a = I a               deriving Functor
```

Let us see, for example, that `f :.: Identity` is equivalent to `f` itself by defining the translations back and forth between them. Since we are now working with functors, those operations must be natural transformations:

```
removeIdentityR :: Functor f => (f :.: Identity) :=>: f
removeIdentityR (Compose x) = fmap (\(I y) -> y) x

composeWithIdentityR :: Functor f => f :=>: (f :.: Identity)
composeWithIdentityR x = Compose (fmap I x)
```

> **Exercise 17.3.** Show that the composition of `removeIdentityR` and `composeWithIdentityR` is equivalent to the identity natural transformation, regardless of the order in which the composition is done. Hint: follow the same steps as we did to check the similar property for `toUnitFunction` and `fromUnitFunction`.

After all this work, let us see what we get when we replace the constructions in `Monoid` with the ones we have just defined for functors. This is a literal replacement of function arrows with natural transformations and tupling with functor composition:

```
mmempty  :: Monad m => Identity  :=>: m
mmappend :: Monad m => (m :.: m) :=>: m
```

We just need to unravel what all these symbols actually mean. Natural transformations are functions parametric over element type, and furthermore `Identity a` is equivalent to `a`, and `(m :.: m) a` is equivalent to `m (m a)`:

```
mmempty  :: Monad m => Identity  a -> m a
      -- i.e.          a -> m a
mmappend :: Monad m => (m :.: m) a -> m a
      -- i.e.   m (m a) -> m a
```

Those functions are simply `return` and `join` from `Monad`! Monads are indeed just monoids in which we work with functors instead of ground types.

17.2 Categories, Functors, Natural Transformations

Category theory is a branch of mathematics, quite young in comparison to other, better-known fields such as algebra or analysis. The notions used in category theory were introduced in the 1940s by Samuel Eilenberg and Saunders Mac Lane to describe concepts coming from topology (yet another branch of mathematics). It soon became clear that those ideas could be applied to many more areas of knowledge — indeed, some people have proposed category theory as a foundation for *all* mathematics and logic!

In a narrow sense, category theory is the branch of mathematics that studies categories and several related notions such as functors, natural transformations, and monads. In a wide sense, category theory is the study of formal *structures* and the *relations* and *transformations* between those structures. Its final aim is to find analogies between different kinds of structures and to describe general phenomena from different fields of mathematics and logic. Getting back to programming, category theory provides a lot of *reusable patterns* for our code.

The three fundamental concepts in category theory are categories themselves, functors — that is, transformations between categories — and natural transformations — which relate two different functors. The goal of this section is to offer an introduction to these three concepts. Each definition we introduce from now on is made of two components: some data that must be provided and some laws that the data must satisfy.

A *category* $\mathcal{C}$ is given by four pieces of data:

1. A set of objects $\mathcal{O}$. We follow the convention of using capital letters $A, B, \ldots$ for objects.
2. A set of morphisms or arrows $\mathcal{M}$. Each arrow comes with two associated objects from $\mathcal{O}$: its source or domain and its target or codomain. We write $A \xrightarrow{f} B$ to represent an arrow f with source A and target B.
3. For every object A in the category, we need to specify an identity arrow $A \to A$. We represent that arrow by id_A or 1_A and drop the subscript when the object can be inferred from the context.
4. Given two arrows $A \xrightarrow{f} B$ and $B \xrightarrow{g} C$, we must have a way to compose them into a new arrow $A \xrightarrow{g \circ f} C$.

Furthermore, identity and composition must satisfy two sets of laws:

1. *Associativity*: given three arrows $A \xrightarrow{f} B$, $B \xrightarrow{g} C$, and $C \xrightarrow{h} D$, the compositions $h \circ (g \circ f)$ and $(h \circ g) \circ f$ must be equal.*
2. *Identity*: given an arrow $A \xrightarrow{f} B$, the two composed morphisms $\mathsf{id}_B \circ f$ and $f \circ \mathsf{id}_A$ must be equal to f.

In Haskell, we can represent the notion of a category by means of a type class. To be completely fair, we can only give the data part of the definition, as the laws cannot be checked by the compiler. In order to do that, we would need a more powerful type system — usually one that supports so-called dependent types — such as that of Coq, Agda, or Idris. The definition of this type class is explicit about how all these data must fit together:

```
class Category o (m :: o -> o -> *) where
  id  :: m a a
  (.) :: m b c -> m a b -> m a c
```

We have a type for objects `o` and a type for morphisms `m`, each of them tagged with two members of `o`. This is the reason for the kind annotation, `o -> o -> *`. Then, we require definitions for identity and composition. In Haskell's `base` library, the `Category` class is defined by the types of the morphisms:

```
class Category m where
  id  :: m a a
  (.) :: m b c -> m a b -> m a c
```

*The notion of *equality* in mathematics can be subtle, as [Mazur, 2007] shows us. In the examples below, we use the notion of extensional equality we introduced in Chapter 5.

Those two definitions are equivalent, though, because the compiler creates an implicit type argument, which in this case would represent the object type. We prefer to use the most explicit one, in order to recall which information is available at each point in the explanation. The definition in the base library is more convenient when programming, since you have to write less code, and in any case the type of `o` can be inferred just by looking at the kind of `m`.

Considering the names we have chosen for the methods in `Category`, you may be wondering whether Haskell functions are a good fit for the morphisms of a category. And indeed they are. We can build a category in which objects are Haskell ground types (types with kind `*`) and morphisms are functions between them:

```
instance Category * (->) where
  id      x = x
  (g . f) x = g (f x)
```

The identity and composition morphisms are the usual ones for functions. The missing step is proving that the category laws hold, but we already discussed them in Chapter 5.

The laws of a category look fairly similar to those of monoids. We can wonder then what the relationship is between these two concepts. The answer is that a category is some sort of generalized monoid, in which *not* all elements may be combined with each other. Looking at it from the other side, a monoid can be seen as a category `Single` in which there is only one object, which we represent as `One` in code:

```
data Single = One
```

This implies that every arrow is of the form `One` $\longrightarrow$ `One`, so all of them can be freely combined. In this case, we are going to have one arrow per element in the monoid. The source and target types have to appear to make `MonoidArrow` the shape required by `Category`, but their only possible instantiation is `One`:

```
data MonoidArrow m (a :: Single) (b :: Single) = MonoidArrow m
```

The final trick is lifting the monoid operations into the category. The identity arrow corresponds to the identity element of the monoid. The composition of two arrows is merely the combination of the two monoid elements they represent:

```
instance Monoid m => Category Single (MonoidArrow m) where
  id = MonoidArrow mempty
  (MonoidArrow x) . (MonoidArrow y) = MonoidArrow (x `mappend` y)
```

Exercise 17.4. Prove that the category laws hold whenever `m` satisfies the monoid laws.

Much of the power of category theory lies in the fact that many concepts in mathematics and programming can be seen as a category. For example, any partial order can be written as a category where an arrow $X \to Y$ means that $X \leqslant Y$ with respect to that order. Another example is logical implication, in which an arrow between two propositions $P \to Q$ exists only if Q follows logically from P. Finally, every collection of elements — types, monoids, rings, etc. — forms a category with respect to those functions that respect the structure embodied by that collection.

The next basic concept in category theory is the *functor*. Intuitively, a functor is a mapping between two categories that respects all the structure embodied in them. More concretely, a functor F from a category $\mathcal{C}$ into another category $\mathcal{D}$ is given by two pieces of data:

- A mapping $F_{\mathcal{O}}$ from the objects of $\mathcal{C}$ to the objects of $\mathcal{D}$.
- A mapping $F_{\mathcal{M}}$ from the arrows of $\mathcal{C}$ to the arrows of $\mathcal{D}$. Given an arrow $A \xrightarrow{f} B$ in $\mathcal{C}$, the source and target must be mapped as $F_{\mathcal{O}}(A) \xrightarrow{F_{\mathcal{M}}(f)} F_{\mathcal{O}}(B)$.

Usually, we abuse notation and write both $F_{\mathcal{O}}$ and $F_{\mathcal{M}}$ as simply F. In most cases, we also drop the parentheses for function application, as we do in Haskell. For example, we say that $A \xrightarrow{f} B$ in $\mathcal{C}$ is mapped to $F\,A \xrightarrow{F\,f} F\,B$.

Along with the data, a functor F has to satisfy two laws:

- Identities must be respected: given an object A that is mapped to $F\,A$, its identity arrow id_A must be mapped to $\mathrm{id}_{F\,A}$.
- Composition must be respected: the mapped arrow $F(g \circ f)$ must be equal to $F\,g \circ F\,f$.

Once again, we can write a Haskell type class that defines the concept of a functor. There are many elements that are related: objects and morphisms for both categories and object and arrow mappings. We need to make them explicit:

```
class (Category o1 m1, Category o2 m2)
      => Functor o1 (m1 :: o1 -> o1 -> *)
                 o2 (m2 :: o2 -> o2 -> *)
                 (objMap :: o1 -> o2) where
  arrowMap :: m1 a b -> m2 (objMap a) (objMap b)
```

Our first example of a functor builds on two monoids that we will define as categories. The first is the monoid of Booleans with conjunction, and the other is the monoid of natural numbers with product. These monoids are called `All` and `Product`, by convention, and are defined as follows in Haskell's `base` library:

```
newtype All = All Bool
newtype Product a = Product a
```

```
instance Monoid All where
  mempty = All True
  mappend (All x) (All y) = All (x && y)

instance Num a => Monoid (Product a) where
  mempty = Product 1
  mappend (Product x) (Product y) = Product (x * y)
```

Both use the same underlying set of objects, so in principle the object map of this functor is the identity function. Alas, the limitations of the Haskell language force `objMap` to be a data type. We cannot reuse the `Identity` type constructor, because its kind is `* -> *`. The trick here is to introduce a new constructor `Single` that takes yet another `Single` in order to obtain a type constructor of the right kind:

```
data Single = One | Other Single
```

Now, we can finally write down the declaration of the `Functor` instance. Note that the weird type of the `MonoidArrow` arguments in the result type is not a problem, as they do not influence the type of the elements inside of it:

```
instance Functor Single (MonoidArrow All)
                 Single (MonoidArrow (Product Integer))
                 Other where
  arrowMap :: MonoidArrow All a b
           -> MonoidArrow (Product Integer) (Other a) (Other b)
```

We need to map every arrow from the Boolean monoid to a monoid in the natural numbers. Remember that one arrow in this case corresponds to one element of the monoid. This means that we need to map both `True` and `False` to natural numbers. We can do, for example, the following:

```
  arrowMap (MonoidArrow (All True))  = MonoidArrow (Product 1)
  arrowMap (MonoidArrow (All False)) = MonoidArrow (Product 0)
```

Dear reader, you can check that this definition satisfies the functor laws. For example, the identity of the Boolean monoid — `True` — corresponds to the identity in the `Product` monoid — 1, in this case.

One important subclass of functors, actually needed to define what a monad is, is that of endofunctors. An endofunctor is simply a functor from a category $\mathcal{C}$ to the same $\mathcal{C}$:

```
type Endofunctor (o :: *) (m :: o -> o -> *) (objMap :: o -> o)
   = Functor o m o m objMap
```

The `Functor` type class in Haskell does not, in fact, correspond to the categorical notion of a functor. Each Haskell instance is actually an endofunctor for the category of types. In code:

```
type HaskellFunctor objMap = Endofunctor (*) (->) objMap
```

Take, for example, the list constructor `[]`. The object mapping is just the application of that type constructor, that is, it maps a type `A` into `[A]`. If we read out the shape of the arrow mapping for that concrete instance, we get:

```
arrowMap :: (a -> b) -> ([a] -> [b])
```

This is exactly the type of `fmap` for lists! Every other Haskell `Functor` may be seen through the same lens as a categorical functor.

In the same way that functors map categories to categories, *natural transformations* map functors to functors. The two functors F and G that take part in a natural transformation η must transform the same underlying categories. That is, both F and G have to take the same category $\mathcal{C}$ into the same category $\mathcal{D}$. In order to define a natural transformation, the only data we require is a family of arrows, one per object X in $\mathcal{C}$, of the form $F\,X \to G\,X$. In other words, we have to specify how to take objects transformed by F to objects transformed by G. Each of these arrows is called a *component* of η at X, written η_X.

But not any such family of arrows defines a natural transformation. They need to satisfy the *naturality condition*, which requires that for every arrow $X \xrightarrow{f} Y$ from $\mathcal{C}$:

$$\eta_Y \circ F\,f = G\,f \circ \eta_X$$

This condition states that it does not matter whether we transform objects from F to G before or after transforming an arrow.

Consider the functors corresponding to `[]` and `Maybe` in Haskell. As we discussed above, they are endofunctors for the category of Haskell types. We can define a natural transformation `safeHead` from `[]` to `Maybe` by describing how to map every `[a]` to `Maybe a` — those are the components of `safeHead` at each type `a`:

```
safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

We need an additional check to ensure that `safeHead` describes a natural transformation. It has to be the case that for every function $X \xrightarrow{f} Y$:

$$\texttt{safeHead}_Y \circ \texttt{fmap}_{[]}\,f \equiv \texttt{fmap}_{\texttt{Maybe}} \circ \texttt{safeHead}_X$$

Remember that the arrow part of a functor is described by the `fmap` method of Haskell's `Functor` type class. To finish this proof, we only have to consider two cases, whether the argument is an empty list or has at least one element:

```
-- Case of empty list []
-- Left-hand side
```

```
   safeHead (fmap f [])
≡ safeHead []
≡ Nothing
-- Right-hand side
   fmap f (safeHead [])
≡ fmap f Nothing
≡ Nothing

-- Case of non-empty list (x:xs)
-- Left-hand side
   safeHead (fmap f (x:xs))
≡ safeHead (f x : fmap f xs)
≡ Just (f x)
-- Right-hand side
   fmap f (safeHead (x:xs))
≡ fmap f (Just x)
≡ Just (f x)
```

When restricted to Haskell `Functors`, that is, endofunctors for the category of Haskell types and functions, a natural transformation always has the shape introduced above, `F a -> G a`, for those functors. Literally, this only defines the components of the natural transformation. But something interesting happens when we are implementing those components: the semantics of Haskell prevent us from having different implementations depending on the chosen `a`. For example, we could not have given different code for the specific case of `[Int]` in `safeHead`. We say that functions of the form `F a -> G a` are *parametric* over the type `a`.

In a happy twist of events, parametricity is a stronger property than naturality. That is, every function with the shape of the components of a natural transformation defines, in fact, a natural transformation — without any further ado. This is an example of a *free theorem*, some property we know about a function by looking only at its type signature. This suggests parametric mappings between functors as the Haskell definition of natural transformations:

```
type f :=>: g = forall a. f a -> g a
```

As in the case of the `Functor` type class, `(:=>:)` refers to a specific sort of natural transformation, in particular, to those natural transformations between endofunctors for the category of Haskell types and functions. But this is more than enough to understand the construction of monads as monoids.

17.3 Monoids in Monoidal Categories

Categories, functors, and natural transformations are the basic ingredients for any discussion about category theory. Now we can start diving into the more specific

constructions that we need in order to describe a monad as a monoid. The first question is, in fact: what does the word *monoid* mean when applied in the context of a category?

We have discussed monoids in several contexts. Here is the usual definition:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

This definition is not amenable for generalizing from the category of Haskell types to other categories. The problem lies in the form of the functions: `mempty` takes *no* arguments, and `mappend` takes *two* argument. But arrows in a category always take *exactly one* argument.

The easiest fix is turning `mappend` into a function with just one parameter. We can require the two values to be combined in a tuple instead of given as separate, curried arguments. For the case of `mempty`, we described at the beginning of this chapter how any value of type `a` can be seen as a function of type `() -> a`. The categorical point of view for a monoid is thus:

```
class Monoid m where
  mempty  :: ()     -> m
  mappend :: (m, m) -> m
```

It looks like we have solved the problem of the shape of the types of `mempty` and `mappend`. Alas, our solution implicitly assumes that the category in which we are working has some notion of *tuple* and some notion of *unit* value. Not all categories do well enough in life to have those constructions, but we call those that do *monoidal categories*. The goal of this section is to describe this notion and give some examples of different monoidal categories. Once we have those, we can see a monoid as a generalization of the `Monoid` type class in which the tuple and unit value are instantiated to those notions in the corresponding monoidal category.

To be precise, a monoidal category $\mathcal{C}$ is described by two pieces of data, in addition to the elements that make $\mathcal{C}$ a regular category:

1. A bifunctor $\otimes$ from $\mathcal{C} \times \mathcal{C}$ to $\mathcal{C}$. Bifunctors are simply functors that take two categories as arguments instead of a single one. In the case of Haskell types, this bifunctor is the tuple constructor `(,)`.

2. An object 1 from $\mathcal{C}$. In the running example in this section, this object is the type `()`.

As usual, a bunch of laws must be satisfied to really speak about a monoidal category:

1. The object 1 must act as an identity for the bifunctor $\otimes$. That is, the objects $1 \otimes X$ and $X \otimes 1$ must both be equivalent to X, where X is an arbitrary object of $\mathcal{C}$.

2. The bifunctor $\otimes$ has to be associative. Given any three objects A, B, and C from $\mathcal{C}$, the objects $A \otimes (B \otimes C)$ and $(A \otimes B) \otimes C$ must be equivalent.

The meaning of "equivalent" in these laws has a strict meaning: a natural transformation with an inverse should exist between all the constructions. However, this level of detail is not required to understand the monoid construction, and so we will gloss over it a bit.

A given category may have more than one way to be monoidal. Even when concerning ourselves only with the category of types, we can already find another such structure in addition to tuples. This monoidal structure is given by a bifunctor and an object from the category of types, that is, a type. For the bifunctor, we choose `Either` — this type constructor has two type parameters, as required — and for the object that works as an identity, the `Void` type:

```
data Void
```

There is nothing missing in the definition above: the `Void` type has no constructor. We can never create a value of that type, and conversely there is no way to pattern match on it.

To check that `Either` and `Void` fit together, we need to check that `Either Void a`, `Either a Void`, and `a` are isomorphic. Let us start with the conversion between `Either Void a` and `a`:

```
removeVoidL :: Either Void a -> a
removeVoidL (Right x) = x

composeWithVoidL :: a -> Either Void a
composeWithVoidL x = Right x
```

The definition of `removeVoidL` is complete, even though we do not have a case for `Left`. Had we such a value, it would include a value of type `Void`. But we just said that we cannot construct an element of that type, so pattern matching on it is useless.

Exercise 17.5. Finish checking that `Either` gives rise to a monoidal category, `Void` being the identity:

1. For the identity laws, by writing functions from `Either a Void` to `a`, and vice versa.
2. For associativity, by writing functions from and to different nesting structures:
   `Either a (Either b c)` and `Either (Either a b) c`.

Wrapping up, we have seen that the notion of *monoid* only makes sense when we are inside a *monoidal category*. Given the monoidal structure of the bifunctor $\otimes$ with the object 1, a monoid is defined by one object M and two arrows:

$$1 \xrightarrow{\eta} M \quad \text{and} \quad M \otimes M \xrightarrow{\mu} M$$

We call the arrow η the *unit* or *identity* of the monoid and the arrow μ the *multiplication* of the monoid. We can now provide a precise definition of Haskell's `Monoid` type class in categorical terms: it is formed by monoids in the monoidal category of Haskell types with respect to tuples.

17.4 The Category of Endofunctors

We are getting closer to our goal of defining monads as monoids. We just need to choose the right (monoidal) category to make the definition work, and that is the category of *endofunctors*. Let us build such a category $\mathsf{End}(\mathcal{C})$ from any category $\mathcal{C}$:

1. The objects of this category are the endofunctors of $\mathcal{C}$, that is, functors that take $\mathcal{C}$ into itself. In the case we are mostly concerned with, that of Haskell types, this means that the category of endofunctors is made of those type constructors that are instances of the `Functor` type class, like `[]` or `Maybe`.
2. The arrows of $\mathsf{End}(\mathcal{C})$ are natural transformations between functors. In our Haskell setting, those are functions of the form `F a -> G a` for each two, given endofunctors.

The definition of a category also requires an identity arrow for each object. That is, given each endofunctor F, we need a natural transformation from F to F. We can choose the identity natural transformation, which sends each object $F\,X$ to $F\,X$ itself. In the same fashion, we can also define the composition of two natural transformations: the components at each object X are just the composition of the components of each transformation. This definition safisfies the naturality property of natural transformations.

Exercise 17.6. Write out this construction for the case of Haskell types and `Functor`:

1. The identity natural transformation is a function:

   ```
   ident :: Functor f => f :=>: f
   ```

2. The composition of two natural transformations is a function:

   ```
   compos :: (Functor f, Functor g, Functor h)
          => (g :=>: h) -> (f :=>: g) -> (f :=>: h)
   ```

The next step is making $\mathrm{End}(\mathcal{C})$ into a *monoidal* category. The main trick here is the choice of the bifunctor and object for this construction. In order to define a bifunctor, we need a way to combine two endofunctors. We already described one such approach in Chapter 10, the *composition* of endofunctors:

```
data (f :.: g) a = Compose (f (g a)) deriving Functor
```

The other element we require for this construction is an identity for the composition. Such an object has to be an endofunctor — this is what the category $\mathrm{End}(\mathcal{C})$ is made from. And it has to be one that adds no further information to the composition. This is exactly the *identity* endofunctor:

```
data Identity a = I a deriving Functor
```

To keep this chapter concise, we will not give a detailed account of how `(:.:)` and `Identity` satisfy the laws for monoidal categories. Availing ourselves of the well-known idiom from math books, we leave it as an exercise for the reader.

We said that a monad is a monoid in $\mathrm{End}(\mathcal{C})$. Let us unravel the categorical definition and see that it coincides with the definition from Chapter 1. The first operation in such a monoid is the unit, $1 \xrightarrow{\eta} M$. In this category, 1 is the identity functor, and arrows are natural transformations. In other words, the type of η is actually `Identity :=>: M`. Since a natural transformation in Haskell is represented by a polymorphic function, this type is equivalent to:

```
η :: Identity a -> M a
```

The last step is noticing that `Identity a` is simply a fancy wrapper around `a`. Thus, the unit of a monoid in the category $\mathrm{End}(\mathcal{C})$ is a function with the type `a -> M a` for an endofunctor `M`. In other words, it is the `return` operation of the monad `M`.

The second operation of the monoid corresponds to the `join` of `M`. The multiplication takes the form $M \otimes M \xrightarrow{\mu} M$. In the category $\mathrm{End}(\mathcal{C})$, the bifunctor $\otimes$ corresponds to functor composition, and as before, the arrows are actually natural transformations. The multiplication is thus of the form `(M :.: M) :=>: M`. We unpack for the last time the definition of `(:=>:)` as a polymorphic function to get:

```
μ :: (M :.: M) a -> M a
```

Looking above at the definition of functor composition, we see that `(M :.: M) a` is the same as writing `M (M a)` — in the same sense that `(f . g) x` is the same as `f (g x)`. Therefore, the type of μ is `M (M a) -> M a`. This is the type of `join`.

17.4.1 Functors in $\mathrm{End}(\mathcal{C})$

Back in Chapter 12, we discussed the `hoist` operation, which allows us to apply a conversion between monadic computations within a transformer:

```
hoist :: Monad m => (forall a. m a -> n a) -> t m b -> t n b
```

If you look at the documentation of the `mmorph` package where it resides, you will see that `hoist` is, in fact, a member of a type class called `MFunctor`. This is because `hoist` is simply the arrow part of a functor but in the category $\mathsf{End}(\mathcal{C})$!

Every time we have a polymorphic function between functors, we are speaking of a natural transformation. Let us use that equivalence to rewrite the type of `hoist` a bit:

```
hoist :: Monad m => (m :=>: n) -> (t m :=>: t n)
```

Compare this type with that of `fmap`, that is `(a -> b) -> (f a -> f b)`. They are pretty similar, the only difference being the kind of arrows in each mapping: normal arrows for `fmap`, natural transformations for `hoist`. This tells us that we can see a monad transformer as a variation of the concept of a type constructor, where the types are always replaced by monads.

Since the notion of a functor in $\mathsf{End}(\mathcal{C})$ makes sense, could we also bring in the concept of a monad? Such a monad in the category of endofunctors would be defined by two operations, which we called η and μ in our discussion of monads. For this, we require notions of identity and composition that work at the level of monad transformers and a notion of a transformer morphism. The identity monad transformer `IdT` must be built in such a way that, given a monad, it returns the same one. Likewise, we need a type `ComposeT` that combines two monad transformers. Their definitions are similar to the `Identity` type and the `(:.:)` operator we defined for monads but working with higher kinds:

```
data IdT          m a = IdT (m a)
data ComposeT f g m a = ComposeT (f (g m) a)
```

Remember that, in our discussion of monads, the arrows were natural transformations, that is, functions polymorphic over the type of the values contained in the monad. To continue our construction over monad transformers, we need a similar notion but where polymorphism occurs also at the level of the monad being wrapped. We call those *transformer morphisms*, and they are defined in Haskell as follows:

```
type t :==>: s = forall m a. t m a -> s m a
```

With all those ingredients, a monad `T` in $\mathsf{End}(\mathcal{C})$ — which despite the "monad" in its name is actually a *monad transformer*, such as `MaybeT` or `StateT` — is defined by the following pair of functions:

```
η :: IdT :==>: T
μ :: ComposeT T T :==>: T
```

Replacing the definitions in the types above, and removing unnecessary wrapping from `IdT` and `ComposeT` as usual, we reach the final types:

```
η :: m a       -> T m a
μ :: T (T m) a -> T m a
```

Exercise 17.7. Relate the operations described in Chapter 12 to η and μ.

The first function, which lifts a computation from a monad into a monad wrapper in a transformer, is simply `lift`. The second operation is called `squash` in the `mmorph` library, and we used it to flatten two consecutive layers in a monadic stack into a single layer.

This chapter was intense. We introduced, without much formality, the basic notions of category theory. Then, we moved on to the categorical generalization of a monoid in order to reveal that a monad is just a monoid. This point of view has already paid off, since we are now able to motivate the definitions of some "weird" functions from the `mmorph` library: they are just the components of the notion of monad lifted to the world of transformers.

18 Adjunctions

One of the first examples of monads in this book was the `State` monad. We defined it as a function taking a state and returning a value and a new state:

```
newtype State s a = State { runState :: s -> (a, s) }
```

This definition can be separated into two components: the type constructor `(s ->)`, which supplies the old state, and the type constructor `(, s)`, which couples the new state with the return value. Both of them are not only type constructors but *functors*. The `State` monad thus arises as the composition of two different functors into one single functor.

This might look like a specific trait of the `State` monad, but in fact, it is a general construction. Category theory defines pairs of functors as being *adjoint* when they satisfy a relation between their arrows: `(s ->)` and `(, s)` are a prime example of that relation. Whenever you have two adjoint functors, you can build a monad by composing them.

This chapter introduces the *adjoint functor* construction at a high-level, especially with regard to their relationship with monads. As we will see by the end, this same notion forms the basis of *free* constructions — free monads, in particular.

18.1 Adjoint Functors

Consider two categories, $\mathcal{C}$ and $\mathcal{D}$, and two functors in opposite directions: F maps $\mathcal{C}$ into $\mathcal{D}$, whereas G maps $\mathcal{D}$ into $\mathcal{C}$. We say that there is an adjunction between F and G, written $F \dashv G$, whenever we have an isomorphism between the sets of arrows:

$$(\text{in } \mathcal{D})\ F\,C \to D \quad \text{and} \quad C \to G\,D\ (\text{in } \mathcal{C})$$

In this situation, we say that F is *left adjoint* and G is *right adjoint*. If F and G are endofunctors, the situation becomes easier, since there is only one category

involved in the definition, and thus we do not have to specify each time where the objects and arrows come from.

The package `adjunctions` provides a type class `Adjunction`, which arises from specializing the definition above to the case in which the source and target categories are both the category of Haskell types. Remember that Haskell's `Functor` defines exactly the endofunctors in that category:

```
class (Functor f, Functor g) => Adjunction f g where
  leftAdjunct  :: (f a -> b) -> (a -> g b)
  rightAdjunct :: (a -> g b) -> (f a -> b)
```

The two methods in `Adjunction` witness the isomorphism between the sets of arrows. In essence, `f` and `g` are adjoints if you can turn any `f a -> b` into `a -> g b` and vice versa. Since we should have an isomorphism, the composition of the two methods, in any order, should be the identity.

Our first example of an adjunction is the one between partially applied pairs `(t,)`* and functions `(t ->)`. In fact, we will define a family of adjunctions, one for each choice of `t`. Here, the pair is the left adjunct and the function the right adjunct, `(t,)` $\dashv$ `(t ->)`. If this adjunction actually exists, that means that we have the following functions:

```
instance Adjunction ((,) t) ((->) t) where
  leftAdjunct  :: ((t, a) -> b) -> (a -> t -> b)
  rightAdjunct :: (a -> t -> b) -> ((t, a) -> b)
```

> **Exercise 18.1.** Write the implementation of the two functions above. Check that they do, in fact, form an isomorphism.

Another simple adjunction in the Haskell world is given by `Void1`, the functor with no constructors, and `Unit1`, the functor with exactly one constructor. These are higher-kinded versions of `Void` and `()`:

```
data Void1 a
data Unit1 a = Unit1
```

The adjunction in this case is `Void1` $\dashv$ `Unit1`. Let us write the proof step by step. First, we need to write the two functions that witness the isomorphism between the arrows:

```
instance Adjunction Void1 Unit1 where
  leftAdjunct  :: (Void1 a -> b) -> (a -> Unit1 b)
  leftAdjunct  _ = \_ -> Unit1
  rightAdjunct :: (a -> Unit1 b) -> (Void1 a -> b)
  rightAdjunct _ = \v -> case v of { }
```

*It does not matter whether we use `(t, )` or `(, t)` in this construction. We use the former, because the code and equivalences are slightly simpler.

In the last line, we use an *absurd pattern*, a pattern match with no choice. Since `Void1` has no constructors, there is no way to deconstruct a value of this type. Of course, there is no way to construct such a value, either. We now need to check that the composition in both directions is equivalent to the identity:

```
   (leftAdjunct . rightAdjunct) f
≡ leftAdjunct (rightAdjunct f)
≡ \_ -> Unit1

   (rightAdjunct . leftAdjunct) g
≡ rightAdjunct (leftAdjunct g)
≡ \v -> case v of { }
```

At first sight, it looks like the isomorphism does not hold. However, on a closer inspection, the type of `f` (`a -> Unit1 b`) *forces* `f` to return `Unit1` regardless of the input parameter. After all, there is only one way to build such a `Unit1 b` value, and it receives no parameters whatsoever. Thus, `f` is equivalent to the result of right and then left-adjoining it. We can reason similarly with `g`: since it receives a value of type `Void1 a`, the only thing it can do is pattern match with no choice.

The last, classic example of adjunction is the pair formed by the diagonal functor and the pair functor. This example is interesting because it shows an adjunction in which the two categories involved are not the same. To describe this construction, we first need to introduce the idea of the *product* of categories. Given two categories $\mathcal{C}$ and $\mathcal{D}$, the product $\mathcal{C} \times \mathcal{D}$ is a category in which objects are pairs of objects (C, D), where C is an object in $\mathcal{C}$ and D an object in $\mathcal{D}$, and similarly, arrows are pairs of arrows (f, g), one per category. Identities and compositions are defined component-wise:

$$\mathsf{id}_{(C,D)} = (\mathsf{id}_C, \mathsf{id}_D) \qquad (f_1, g_1) \circ (f_2, g_2) = (f_1 \circ f_2, g_1 \circ g_2)$$

The *diagonal* functor `Diag` goes from any category $\mathcal{C}$ into the product of $\mathcal{C}$ with itself, $\mathcal{C} \times \mathcal{C}$. Its action is simply to duplicate objects and arrows: an object C is sent to (C, C), and an arrow f is sent to (f, f). The *product* functor `(,)` takes objects in $\mathcal{C} \times \mathcal{C}$ and builds the internal product of them in $\mathcal{C}$. In Haskell terms, this means that it takes two values, `x` and `y`, of the same type and puts them into the tuple `(x, y)`. This implies that the action of that functor on arrows must be:

```
map :: (a -> b, c -> d) -- Arrow in C x C
    -> (a, c) -> (b, d) -- Arrow in C
map (f, g) = \(x, y) -> (f x, g y)
```

Each function of the pair works independently over each argument. This function is known as `(***)` in the `Control.Arrow` module.

The adjunction in this case takes the form `Diag` $\dashv$ `(,)`. By substituting the moving parts in the original definition of adjunction, we see that we need to

provide an isomorphism between:

$$(\text{in } \mathcal{C} \times \mathcal{C}) \quad \texttt{Diag}\, C \to D \quad \text{and} \quad C \to \texttt{(,)}\, D \quad (\text{in } \mathcal{C})$$

Remember also that objects in $\mathcal{C} \times \mathcal{C}$ are actually pairs of objects from $\mathcal{C}$. Thus, we can rewrite the definition, being more explicit about pairs. At the same time, we may also expand the definition of `Diag` C:

$$(C, C) \to (D_1, D_2) \quad \text{and} \quad C \to \texttt{(,)}\, (D_1, D_2)$$

Remember that the left-hand side of the adjunction works in $\mathcal{C} \times \mathcal{C}$. This means that even though we write $(C, C) \to (D_1, D_2)$, we really mean a pair of arrows $(C \to D_1, C \to D_2)$. Putting all of this together, the functions witnessing the isomorphism have the types:

```
leftAdjunct  :: (c -> d1, c -> d2) -> (c -> (d1, d2))
rightAdjunct :: (c -> (d1, d2)) -> (c -> d1, c -> d2)
```

Exercise 18.2. Write the implementation of the two functions above.

Exercise 18.3. We can play a similar game with the `Either` type. In that case, the adjunction is reversed, `Either` $\dashv$ `Diag`. Spell out the details of this adjunction.

You might be tempted to think that since `Diag` $\dashv$ `(,)` and `Either` $\dashv$ `Diag`, therefore `Either` $\dashv$ `(,)`. Alas, this is not the case, as from $A \dashv B$ and $B \dashv C$, we cannot deduce in general that $A \dashv C$. In more mathematical terms, the $\dashv$ relation is *not transitive*.

18.2 Monads from Adjunctions

In the previous section, we defined an adjunction as coming from an isomorphism between certain arrows in the underlying categories. We called one of the components of this isomorphism `leftAdjunct`:

```
leftAdjunct :: (f a -> b) -> (a -> g b)
```

Let us consider what happens in the specific case in which `b` is set to `f a`. Then, the argument required by `leftAdjunct` becomes of type `f a -> f a`, a constraint we can simply satisfy by using the `id` function. As a result of the application, we obtain the following function:

```
η :: a -> g (f a)
η = leftAdjunct id
```

Since we are dealing with a polymorphic function between functors, we may also see η as a natural transformation, η `::  Identity :=>:  (g :.:  f)`.

> **Exercise 18.4.** By using `rightAdjunct` instead of `leftAdjunct`, we can build a natural transformation in the converse direction, ϵ `::  (f :.:  g) :=>:  Identity`. Spell out the details of this construction.

The function η defined above is called the *unit* of the adjunction `f` $\dashv$ `g`, and the function ϵ is the corresponding *counit*. One of the main payoffs of category theory is that you can define an adjunction using two different sets of data:

1. An isomorphism between `f a -> b` and `a -> g b`.
2. Two natural transformations `Identity :=>:  (g :.:  f)` and `(f :.:  g) :=>:  Identity`.

We have already seen how to go from (1) to (2). Let us prove the converse. That is, we want to define `leftAdjunct` and `rightAdjunct` in terms of η and ϵ.

In the case of `leftAdjunct`, we need to turn a function `f a -> b` into `a -> g b`. This gives us a skeleton for the function:

```
leftAdjunct f = \x -> ... -- we need to build something of type 'g b'
```

Since `x` has type `a`, we can apply the unit function to it. We then get a value of type `g (f a)`. Since `f` has type `f a -> b`, we can make it work under `g` to obtain a value of type `g b`. We can only do so via `fmap`, since `g` is a functor:

```
leftAdjunct f = \x -> fmap f (η x)
```

> **Exercise 18.5.** Finish the other part of the proof by defining `rightAdjunct` using ϵ.

Let us set η and ϵ aside for a while. We are going to define a new function μ with a rather weird type:

```
μ :: g (f (g (f a))) -> g (f a)
μ = fmap ϵ
```

Why does this definition work? The whole expression is surrounded by `fmap`. By doing this, we work "inside the functor." As a consequence, its argument should be a function `f (g (f a)) -> f a`. We can build this function by using ϵ `::  f (g b) -> f b`, which "peels off" the top two layers. As in other constructions, we have to instantiate the variable `b` in ϵ to `f a`.

Now, let us rewrite the type of μ into a new form, step by step:

```
   g (f (g (f a))) -> g (f a)
≡ -- as a natural transformation
   (g :.: f  :.:  g :.: f) :=>: (g :.: f)
≡ -- reassociating parentheses
   (g :.: f) :.: (g :.: f) :=>: (g :.: f)
```

Finally, let us introduce a synonym, `type t = g :.: f`:

```
μ :: (t :.: t) :=>: t
```

This is exactly the type of the `join` operation of a monad! By following similar steps, we also see that η `:: Identity :=> t` is the type of the `return` operation. This tells us that if we have an adjunction `f` $\dashv$ `g`, we can always build a new *monad* by taking the composition `g :.: f`. This demonstrates a straightforward relationship between adjunctions and monads.

This construction establishes a formal ground to the `State` example mentioned at the beginning of this chapter. We discussed the adjunction `(s,)` $\dashv$ `(s ->)`, and we are guaranteed to obtain a monad by taking `type t = (s ->) :.: (s,)`. Let us see what happens when we use this type on a generic type argument `a`:

```
   ((s ->) :.: (s,)) a
≡ -- definition of :.:
   (s ->) ((s,) a)
≡ -- reducing the innermost type
   (s ->) (s, a)
≡ -- reducing the outermost type
   s -> (s, a)
```

Except for the fact that the elements in the pair are reversed, we obtain the exact definition of the `State` monad. I don't know about you, dear reader, but the first time I was shown this construction, it felt like magic to me!

18.2.1 Comonads

Throughout this book, we have silently ignored a close cousin to monads: *comonads*. This notion arises when we consider the dual of all monad operations — where *dual* is the categorical way to say "reversing the arrows." As in the case of monads, there are three operations from which you can choose two different sets to define a comonad. We follow here the naming convention of the `comonad` package. Other sources name the comonad operations by simply prepending "co" to the monadic names:

```
class Comonad w where
  extract   :: w a -> a                    -- coreturn
  duplicate :: w a -> w (w a)              -- cojoin
  extend    :: (w a -> b) -> w a -> w b    -- cobind
```

The main example of a comonad in functional programming is given by non-empty lists. The reason why we need to impose non-emptiness is to be able to define `extract`, otherwise we have no head to take. The corresponding instance is as follows:

```
instance Comonad [] where  -- non-empty
  extract :: [a] -> a
  extract (x:_) = x
  duplicate :: [a] -> [[a]]
  duplicate xs = [xs]
  extend :: ([a] -> b) -> [a] -> [b]
  extend f xs = [f xs]
```

We will not dive more deeply into the topic of comonads than this. Any of the books mentioned in Chapter 16 discusses them in more detail.

The interesting insight is that we can repeat the same construction that gave us a monad for `g :.: f` coming from an adjunction `f` $\dashv$ `g` but reversing all the arrows. The result is a *comonad* over `f :.: g`. For example, from `(s,)` $\dashv$ `(s ->)`, we obtain:

```
   ((s,) :.: (s ->)) a
≡ (s,) ((s ->) a)
≡ (s,) (s -> a)
≡ (s, s -> a)
```

This is known as the `Store` comonad, usually defined using two fields in a data type instead of a single tuple:

```
data Store s a = Store (s -> a) s
```

This comonad has found many uses lately in Haskell programming, from the definition of lenses to describing automata. The goal in this section was not to show the specifics of this comonad, however, but rather to show how a step by step exploration of the landscape painted by category theory allows us to discover new concepts.

18.3 The Kleisli Category

We saw that for every adjunction, we can build a monad (and also a comonad). We can also ask the converse question: does every monad arise from an adjunction?

The surprising answer to that question is yes — and in many different ways! In this section, we explore one of the solutions, involving the Kleisli category we introduced in Section 5.3.1.

In general, the *Kleisli category* of a monad M over a category $\mathcal{C}$ consists of the same objects in $\mathcal{C}$ but where the arrows have the form $A \to M\,B$. Back in Section 5.3.1, we called them *Kleisli arrows*. Let us turn this idea into Haskell code:

```
newtype Kleisli m a b = Kleisli (a -> m b)

instance Monad m => Category * (Kleisli m) where
  ...
```

In order to finish the definition of the Kleisli category, we need to describe how identity arrows look and how to compose two of them. For identity, we need `id :: Kleisli m a a`. In other words, we need a function `a -> m a`. In this case, we can use one of the monad operations, namely `return`:

```
  id = Kleisli return
```

For composition, we can recycle the fish operator `(<=<)` we defined previously. That operation has the shape `(b -> m c) -> (a -> m b) -> a -> m c`, exactly the one needed to combine two monadic computations:

```
  Kleisli f . Kleisli g = Kleisli (f <=< g)
```

We first introduced Kleisli arrows when describing monad laws. What seemed like an ad-hoc set of rules are now simply the rules of any category. For example, the left identity law reads `id . f` $\equiv$ `f`. If we substitute the general names for identity and composition for the specific versions for `Kleisli`, we get:

```
   id . Kleisli f
≡ Kleisli return . Kleisli f
≡ Kleisli (return <=< f)
≡ Kleisli f
```

Our end goal is to recover `m` as an adjunction. We will do so as an adjunction between $\mathcal{C}$ and `Kleisli m`. The first step is defining two functors for both directions:

1. First, we want to map $\mathcal{C}$ to `Kleisli m`. For the object part, we take each A in $\mathcal{C}$ to the same A in `Kleisli m`. This means that for the arrow part, we need to take each `a -> b` into `Kleisli m a b`.
2. The converse direction maps `Kleisli m` into $\mathcal{C}$. In this case, the object part takes each A in `Kleisli m` to `m` A. This implies that each arrow `Kleisli m a b` has to be converted into an arrow `m a -> m b`, but in $\mathcal{C}$.

The code defining these two functions uses a combination of monadic operations. Unfortunately, since the functors do not work only in the category of Haskell types, we cannot express them as instances of the `Functor` type class:

```
bareToKleisli :: Monad m => (a -> b) -> Kleisli m a b
bareToKleisli f = Kleisli $ \x -> return (f x)
          -- or = Kleisli $ return . f
kleisliToBare :: Monad m => Kleisli m a b -> m a -> m b
kleisliToBare (Kleisli f) x = x >>= f
```

The adjunction takes the form `bareToKleisli` $\dashv$ `kleisliToBare`. Once again, we cannot express it as an instance of the `Adjunction` type class introduced earlier in this chapter, since it only allows endofunctors. Remember that if we have an adjunction $F \dashv G$, where F maps $\mathcal{D}$ to $\mathcal{E}$, we require functions:

$$(\text{in } \mathcal{E})\ F\,C \to D \quad \text{and} \quad C \to G\,D\ (\text{in } \mathcal{D})$$

Let us unravel this definition for the specific case of the Kleisli adjunction. Here $\mathcal{D}$ is simply $\mathcal{C}$, and $\mathcal{E}$ is `Kleisli m`. The object part of functor F — `bareToKleisli` in this case — just keeps the object untouched. The functor G — `kleisliToBare` in this case — embeds the objects in the monad `m`. The shape of the adjunction is thus an isomorphism between:

$$\texttt{Kleisli m}\ C\ D \quad \text{and} \quad C \to \texttt{m}\ D$$

Finally, remember that `Kleisli m a b` is simply a synonym for `a -> m b`. The adjunction is an isomorphism between arrows of the form `a -> m b`, on the left, and arrows of the form `a -> m b`, on the right. This means that the identity is the isomorphism we need.

Just for the record, the other well-known construction that builds an adjunction from a monad uses the *Eilenberg-Moore* category. Although important in theory, this category does not have as many practical uses as Kleisli's, so we will not describe it in any detail.

18.4 Free Monads

At the end of our categorical journey, we will now look at free monads while wearing the glasses of adjunctions. In fact, all *free* constructions arise in a similar way to the one we described for monads, although we are not going to delve into the generic approach.

We obtained monads as monoids in the category of endofunctors of $\mathcal{C}$, which we called $\mathsf{End}(\mathcal{C})$, in the previous chapter. It is therefore not strange that $\mathsf{End}(\mathcal{C})$ makes an appearance once again, as one of the categories involved in the adjunction. The other category involved, $\mathsf{Monad}(\mathcal{C})$, is strikingly similar:

- The objects of the category are those endofunctors from $\mathcal{C}$ that can be turned into a monad. For example, `Maybe` is an object in Monad($\mathcal{C}$), whereas `ZipList` — for which a `Functor` instance exists, but not a `Monad`, as we discussed in Section 3.2 — is not an object in Monad($\mathcal{C}$).
- The arrows in Monad($\mathcal{C}$) are those natural transformations that respect the monadic structure. When instantiated to the category of Haskell types, that means that arrows are those polymorphic functions `m :: f a -> g a` between two monads that preserve the monadic structure, `m . return` $\equiv$ `return` and `m . f <=< m . g` $\equiv$ `m . (f <=< g)`.

  The definition of these so-called *monad morphisms* arises from generalizing the monoid morphisms from Section 5.2. In that case, for example, we proved that `f mempty` $\equiv$ `mempty`. Now our morphisms act polymorphically, and thus we use function composition, and the identity and combination operations are changed into the `return` and Kleisli composition of the monad.

The structure of Monad($\mathcal{C}$) is special when compared to End($\mathcal{C}$): it is formed by choosing only a subset of the objects and of the arrows between those objects. We say that Monad($\mathcal{C}$) is a *subcategory* of End($\mathcal{C}$). In this and other similar situations, we can build a *forgetful functor* that "forgets" the additional structure imposed, in this case, by monads as opposed to functors. Formally, we define the functor Forget from Monad($\mathcal{C}$) to End($\mathcal{C}$) as follows:

- Each object in Monad($\mathcal{C}$) is mapped to the same object in End($\mathcal{C}$), but it is seen as a mere endofunctor.
- Each arrow in Monad($\mathcal{C}$) is mapped to the same arrow in End($\mathcal{C}$), but it is seen as a mere natural transformation.

Similar forgetful functors arise every time we have a richer structure that we can remove — for example, when mapping from the category of monoids to the category of semigroups, in which we "forget" about the identity element.

We define the *free monad* construction as the left adjoint of the forgetful functor, Free $\dashv$ Forget. Instantiating the general definition of adjunction, this means that there is an isomorphism between:

$$(\text{in Monad}(\mathcal{C}))\ \text{Free}\, C \to D \quad \text{and} \quad C \to \text{Forget}\, D\ (\text{in End}(\mathcal{C}))$$

Note that the forgetful functor leaves the objects untouched, except for the removal of the additional monadic structure. Therefore, we can see Forget D as simply D. Furthermore, the arrows in both End($\mathcal{C}$) and Monad($\mathcal{C}$) are natural transformations, which we write as polymorphic functors in Haskell code. It is also important that C is an object in End($\mathcal{C}$), and thus an endofunctor, whereas D is an object in Monad($\mathcal{C}$), and thus a monad. Putting all of this together, the two parts of the adjunction receive these types:

```
leftAdjunct  :: (Functor c, Monad d)
             => (forall a. Free c a -> d a)
             -> (forall a.      c a -> d a)
rightAdjunct :: (Functor c, Monad d)
             => (forall a.      c a -> d a)
             -> (forall a. Free c a -> d a)
```

We discussed the `rightAdjunct` operation back in Chapter 13. There we called it `foldFree`, since it folds an interpretation of each of the operations described by `c` into the monad `d` throughout the computation described by a `Free c`.

For `leftAdjunct`, we need to bring back the operation that lifts a single instruction from `c` into a complete `Free c` construction:

```
liftF :: Functor c => c a -> Free c a
liftF = Free . fmap return
```

The left adjunct can now be defined by first lifting the single operation and then applying the operation given as the first argument:

```
leftAdjunct f = f . liftF
```

There are two additional proofs to be done before we are able to say that these two operations form an adjunction. First of all, `rightAdjunct` produces an arrow in $\mathsf{Monad}(\mathcal{C})$, so we need to check that the operation defines a monad morphism. Second, we need to check that `leftAdjunct` and `rightAdjunct` are inverses of each other. Confirming these proofs is once again left as an exercise for you, dear reader.

This last construction showcases the power of category theory for giving a sound basis to operations that otherwise look ad-hoc. Monads, monad morphisms, free monads — all arise as generalizations of simple notions, bound together by categories and adjunctions.

Bibliography

Allen, Christopher and Moronuki, Julie. *Haskell Programming from First Principles*. Gumroad (2015).
URL http://haskellbook.com

Bahr, Patrick. "Composing and Decomposing Data Types: A Closed Type Families Implementation of Data Types à La Carte." In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14. ACM, New York, NY, USA (2014) pages 71–82.
URL http://doi.acm.org/10.1145/2633628.2633635

Cheplyaka, Roman. "Flavours of free applicative functors." (2013).
URL https://ro-che.info/articles/2013-03-31-flavours-of-free-applicative-functors

Friedman, Daniel P., Byrd, William E., Kiselyov, Oleg, and Hemann, Jason. *The Reasoned Schemer*. MIT Press (2018).

Gurnell, Dave and Welsh, Noel. *Essential Scala*. Underscore Consulting LLP (2015).

Hinze, Ralf. "Kan Extensions for Program Optimisation or: Art and Dan Explain an Old Trick." In *Proceedings of the 11th International Conference on Mathematics of Program Construction*, MPC'12. Springer-Verlag, Berlin, Heidelberg (2012) pages 324–362.
URL http://dx.doi.org/10.1007/978-3-642-31113-0_16

Hutton, Graham. *Programming in Haskell*. Cambridge University Press (2016).
URL http://www.cs.nott.ac.uk/~pszgmh/pih.html

Jones, Mark P. and Duponcheel, Luc. "Composing monads." *Technical report*, Yale University (1993).

Kiselyov, Oleg, Shan, Chung-chieh, Friedman, Daniel P., and Sabry, Amr. "Backtracking, Interleaving, and Terminating Monad Transformers (Functional Pearl)." In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05. ACM, New York, NY, USA (2005) pages 192–203.
URL http://doi.acm.org/10.1145/1086365.1086390

Koopman, Pieter, Plasmeijer, Rinus, and Jansen, Jan Martin. "Church Encoding of Data Types Considered Harmful for Implementations: Functional Pearl." In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages*, IFL '14. ACM, New York, NY, USA (2014) pages 4:1–4:12.
URL http://doi.acm.org/10.1145/2746325.2746330

Kurt, Will. *Get Programming with Haskell*. Manning (2018).

Lipovača, Miran. *Learn You a Haskell for Great Good!* No Starch Press (2011).
URL http://learnyouahaskell.com

Mac Lane, Saunders. *Categories for the Working Mathematician*. Springer (1998).

Marlow, Simon. *Parallel and Concurrent Programming in Haskell*. O'Reilly (2013).

Marlow, Simon, Brandy, Louis, Coens, Jonathan, and Purdy, Jon. "There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access." In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14. ACM, New York, NY, USA (2014) pages 325–337.
URL http://doi.acm.org/10.1145/2628136.2628144

Mazur, Barry. "When is one thing equal to some other thing?" (2007).
URL http://www.math.harvard.edu/~mazur/preprints/when_is_one.pdf

McBride, Conor. "Answer to "help on writing the colist monad"." (2011).
URL https://stackoverflow.com/a/6573243

McBride, Conor and Paterson, Ross. "Applicative Programming with Effects." *Journal of Functional Programming* (2008). 18(1):1–13.
URL http://dx.doi.org/10.1017/S0956796807006326

Milewski, Bartosz. *Category Theory for Programmers* (2014).
URL https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/

Moggi, Eugenio. "Notions of Computation and Monads." *Inf. Comput.* (1991). 93(1):55–92.
URL http://dx.doi.org/10.1016/0890-5401(91)90052-4

Ploeg, Atze van der and Kiselyov, Oleg. "Reflection Without Remorse: Revealing a Hidden Sequence to Speed Up Monadic Reflection." In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14. ACM, New York, NY, USA (2014) pages 133–144.
URL http://doi.acm.org/10.1145/2633357.2633360

Riehl, Emily. *Category Theory in Context*. Dover Publications (2016).
URL http://www.math.jhu.edu/~eriehl/context.pdf

Serrano Mena, Alejandro. *Practical Haskell*. Apress (2019).

Snoyman, Michael. "Everything you didn't want to know about monad transformer state." (2017).
URL https://www.snoyman.com/reveal/monad-transformer-state

Swierstra, S. Doaitse and Duponcheel, Luc. "Deterministic, Error-Correcting Combinator Parsers." In *Advanced Functional Programming, Second International School-Tutorial Text*. Springer-Verlag, Berlin, Heidelberg (1996) pages 184–207.
URL http://dl.acm.org/citation.cfm?id=647699.734159

Swierstra, Wouter. "Data Types à la Carte." *Journal of Functional Programming* (2008). 18(4):423–436.
URL http://dx.doi.org/10.1017/S0956796808006758

Voigtländer, Janis. "Asymptotic Improvement of Computations over Free Monads." In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, MPC '08. Springer-Verlag, Berlin, Heidelberg (2008) pages 388–403.
URL http://dx.doi.org/10.1007/978-3-540-70594-9_20

Wadler, Philip. "Monads for Functional Programming." In *First International Spring School on Advanced Functional Programming Techniques*. Springer-Verlag, Berlin, Heidelberg (1995) pages 24–52.
URL http://dl.acm.org/citation.cfm?id=647698.734146

Yang, Edward Z. "Adventures in Three Monads." *The Monad.Reader* (2010).
URL http://web.mit.edu/~ezyang/Public/threemonads.pdf

Yang, Edward Z. "Problem Set: The Codensity Transformation." (2012).
URL http://blog.ezyang.com/2012/01/problem-set-the-codensity-transformation/