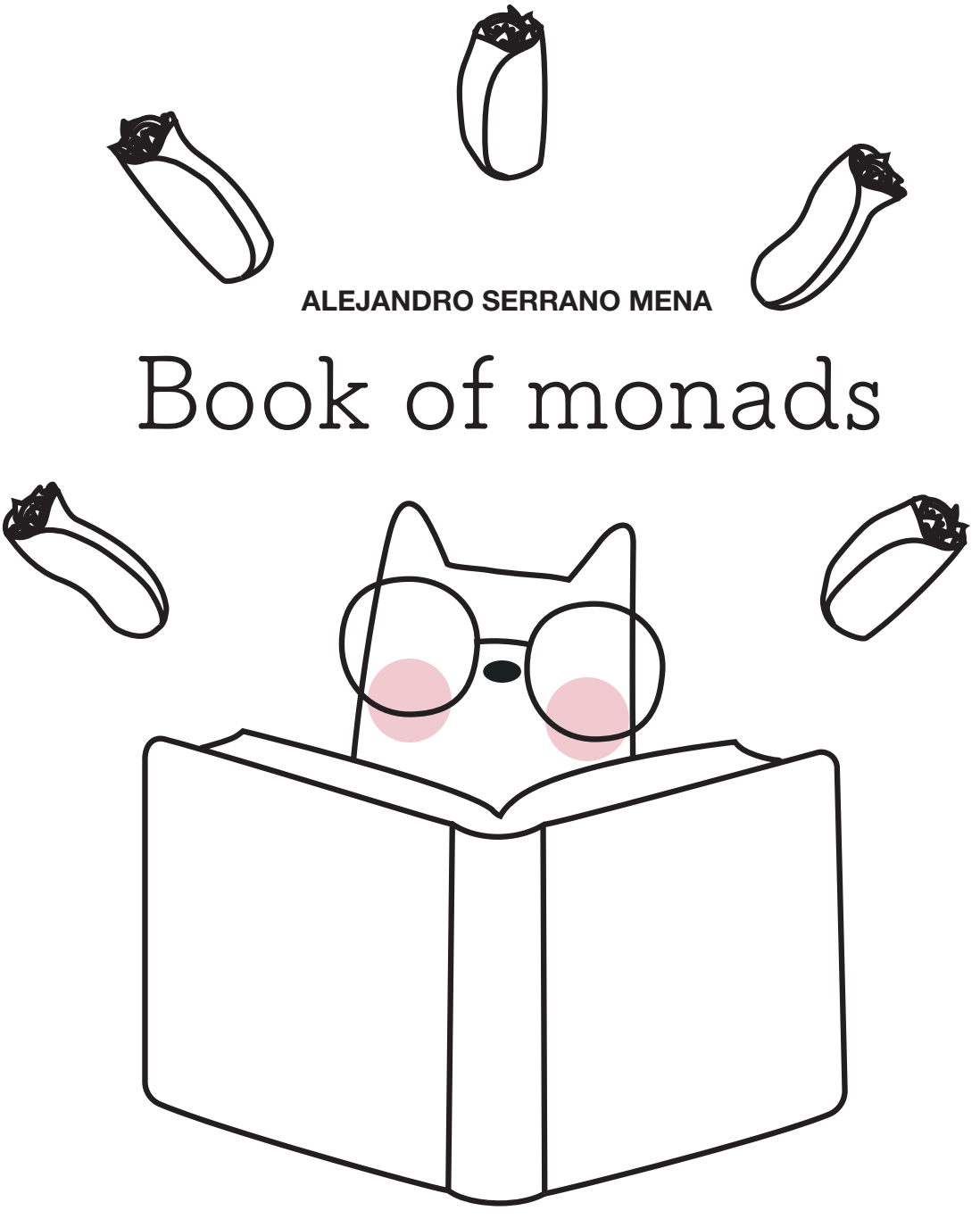
ALEJANDRO SERRANO MENA

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 **1**

0.1 Type Classes 3
0.2 Higher-kinded Abstraction 6
0.3 Haskell's Newtype 8
0.4 Language Extensions in Haskell 9

I What is a Monad? **11**

1 Discovering Monads **13**

1.1 State Contexts 13
1.2 Magical Multiplying Boxes 18
1.3 Both, Maybe? I Don't Think That's an Option 20
1.4 Two for the Price of One 24
1.5 Functors 26

2 Better Notation **29**

2.1 Block Notation 30
2.2 Pattern Matching and Fail 39

3 Lifting Pure Functions **41**

3.1 Lift2, Lift3, ..., Ap 41
3.2 Applicatives 44
3.3 Applicative Style 45
3.4 Definition Using Tuples 49

4 Utilities for Monadic Code **53**
4.1 Lifted Combinators . . . 53
4.2 Traversables . . . 59

5 Interlude: Monad Laws **65**
5.1 Laws for Functions . . . 65
5.2 Monoids . . . 69
5.3 Monad Laws . . . 71

II More Monads **75**

6 Pure Reader-Writer-State Monads **77**
6.1 The State Monad . . . 77
6.2 The Reader Monad . . . 80
6.3 The Writer Monad . . . 84
6.4 All at Once: the RWS Monad . . . 88
6.5 Bi-, Contra-, and Profunctors . . . 88

7 Failure and Logic **91**
7.1 Failure with Fallback . . . 91
7.2 Logic Programming as a Monad . . . 96
7.3 Catching Errors . . . 100

8 Monads for Mutability **103**
8.1 Mutable References . . . 103
8.2 Interfacing with the Real World . . . 106

9 Resource Management and Continuations **115**
9.1 The Bracket Idiom . . . 115
9.2 Nicer Code with Continuations . . . 117
9.3 Early Release . . . 121

III Combining Monads **125**

10 Functor Composition **127**
10.1 Combining Monads by Hand . . . 127
10.2 Many Concepts Go Well Together . . . 131
10.3 But Monads Do Not . . . 133

11 A Solution: Monad Transformers **137**
11.1 Monadic Stacks . . . 137
11.2 Classes of Monads, MTL-style . . . 145
11.3 Parsing for Free! . . . 154

12 Generic Lifting and Unlifting **157**
12.1 MonadTrans and Lift . . . 158
12.2 Base Monads: MonadIO and MonadBase . . . 161
12.3 Lifting Functions with Callbacks . . . 163
12.4 More on Manipulating Stacks . . . 171

IV Rolling Your Own Monads **175**

13 Defining Custom Monads **177**
13.1 Introduction . . . 177
13.2 Final Style . . . 182
13.3 Initial Style . . . 185
13.4 Operational Style and Freer Monads . . . 195
13.5 Transforming and Inspecting Computations . . . 202

14 Composing Custom Monads **211**
14.1 Final Style . . . 212
14.2 Initial and Operational Style . . . 213
14.3 Extensible Effects . . . 218

15 Performance of Free Monads **227**
15.1 Left-nested Concatenation . . . 227
15.2 Left-nested Binds . . . 233

V Diving into Theory **241**

16 A Roadmap **243**

17 Just a Monoid! **245**
17.1 Quick Summary . . . 245
17.2 Categories, Functors, Natural Transformations . . . 248
17.3 Monoids in Monoidal Categories . . . 254
17.4 The Category of Endofunctors . . . 257

18 Adjunctions **261**
18.1 Adjoint Functors . . . 261
18.2 Monads from Adjunctions . . . 264
18.3 The Kleisli Category . . . 267
18.4 Free Monads . . . 269

Bibliography **273**

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.

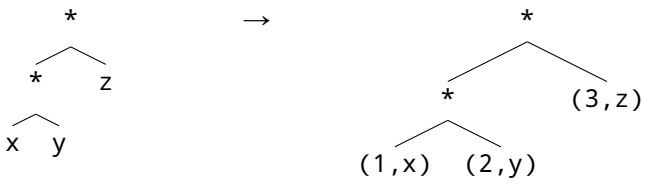


Figure 1.1: Example of left-to-right relabeling

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.