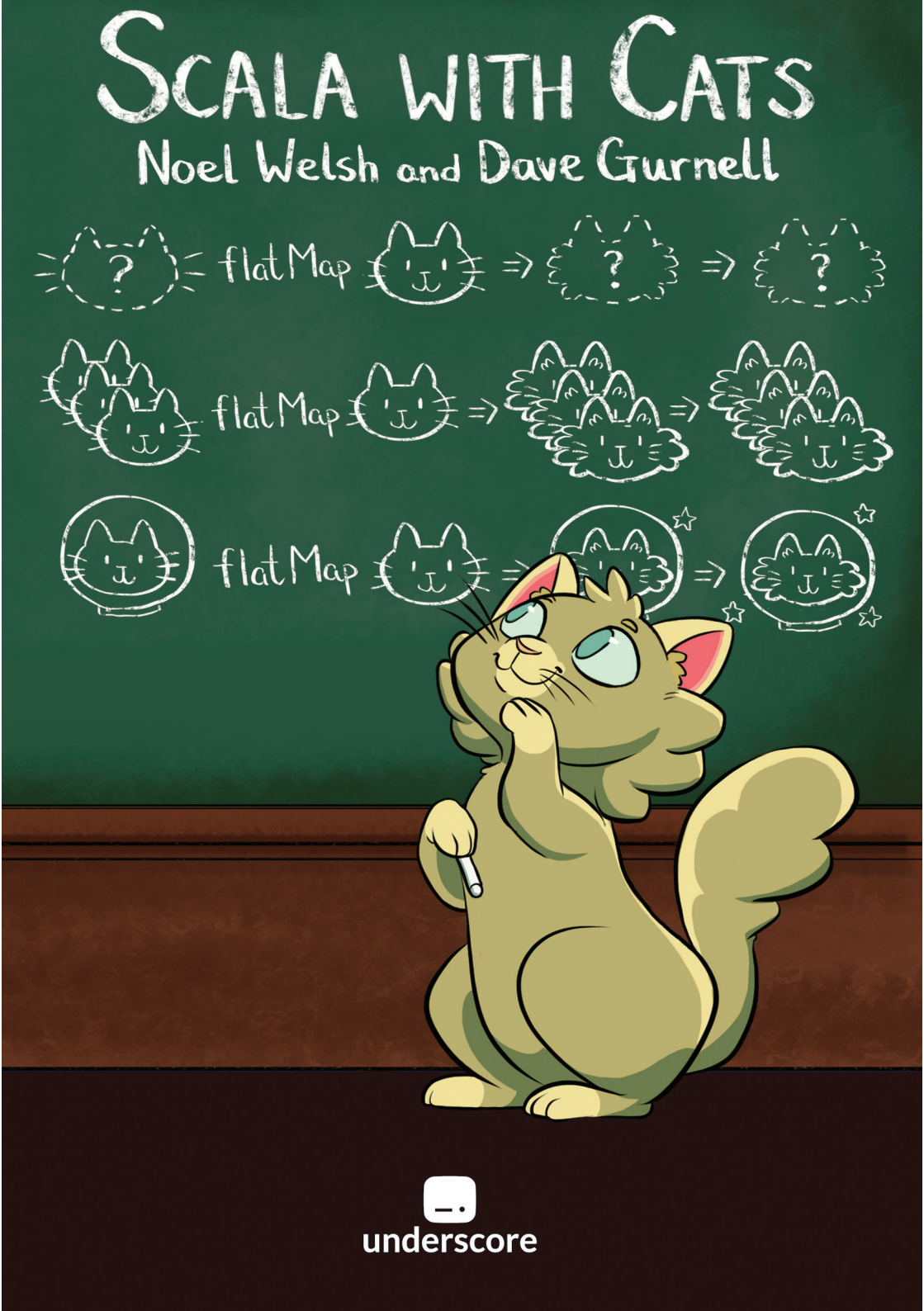
Scala with Cats

Noel Welsh and Dave Gurnell



underscore

Scala with Cats

November 2017

Artwork by Jenny Clements.

Published by Underscore Consulting LLP, Brighton, UK.

Contents

Preface

The aims of this book are two-fold: to introduce monads, functors, and other functional programming patterns as a way to structure program design, and to explain how these concepts are implemented in Cats.

Monads, and related concepts, are the functional programming equivalent of object-oriented design patterns—architectural building blocks that turn up over and over again in code. They differ from object-oriented patterns in two main ways:

- they are formally, and thus precisely, defined; and
- they are extremely (extremely) general.

This generality means they can be difficult to understand. *Everyone* finds abstraction difficult. However, it is generality that allows concepts like monads to be applied in such a wide variety of situations.

In this book we aim to show the concepts in a number of different ways, to help you build a mental model of how they work and where they are appropriate. We have extended case studies, a simple graphical notation, many smaller examples, and of course the mathematical definitions. Between them we hope you'll find something that works for you.

Ok, let's get started!

Versions

This book is written for Scala 2.12.3 and Cats 1.0.0. Here is a minimal `build.sbt` containing the relevant dependencies and settings[1]:

```
scalaVersion := "2.12.3"

libraryDependencies +=
  "org.typelevel" %% "cats-core" % "1.0.0"

scalacOptions ++= Seq(
  "-Xfatal-warnings",
  "-Ypartial-unification"
)
```

Template Projects

For convenience, we have created a Giter8 template to get you started. To clone the template type the following:

```
$ sbt new underscoreio/cats-seed.g8
```

This will generate a sandbox project with Cats as a dependency. See the generated `README.md` for instructions on how to run the sample code and/or start an interactive Scala console.

The `cats-seed` template is as minimal as it gets. If you'd prefer a more batteries-included starting point, check out Typelevel's `sbt-catalysts` template:

```
$ sbt new typelevel/sbt-catalysts.g8
```

This will generate a project with a suite of library dependencies and compiler plugins, together with templates for unit tests and tut-enabled documentation. See the project pages for catalysts and sbt-catalysts for more information.

[1] We assume you are using SBT 0.13.13 or newer.

Conventions Used in This Book

This book contains a lot of technical information and program code. We use the following typographical conventions to reduce ambiguity and highlight important concepts:

Typographical Conventions

New terms and phrases are introduced in *italics*. After their initial introduction they are written in normal roman font.

Terms from program code, filenames, and file contents, are written in `monospace font`. Note that we do not distinguish between singular and plural forms. For example, we might write `String` or `Strings` to refer to `java.lang.String`.

References to external resources are written as hyperlinks. References to API documentation are written using a combination of hyperlinks and monospace font, for example: `scala.Option`.

Source Code

Source code blocks are written as follows. Syntax is highlighted appropriately where applicable:

```
object MyApp extends App {
  println("Hello world!") // Print a fine message to the user!
}
```

Most code passes through tut to ensure it compiles. tut uses the Scala console behind the scenes, so we sometimes show console-style output as comments:

```
"Hello Cats!".toUpperCase
// res0: String = HELLO CATS!
```

Callout Boxes

We use two types of *callout box* to highlight particular content:

> Tip callouts indicate handy summaries, recipes, or best practices.

> Advanced callouts provide additional information on corner cases or underlying mechanisms. Feel free to skip these on your first read-through—come back to them later for extra information.

Acknowledgements

We'd like to thank our colleagues at Underscore, our friends at Typelevel, and everyone who helped contribute to this book. Special thanks to Jenny Clements for her fantastic artwork and Richard Dallaway for his proof reading expertise. Here is an alphabetical list of contributors:

Alessandro Marrella, Cody Koeninger, Connie Chen, Conor Fennell, Dani Rey, Daniela Sfregola, Danielle Ashley, David Castillo, David Piggott, Dennis Hunziker, Deokhwan Kim, Edd Steel, Evgeny Veretennikov, Francis Devereux, Ghislain Vaillant, Gregor Ihmor, Henk-Jan Meijer, Janne Pelkonen, Jason Scott, Javier Arrieta, Jenny Clements, Jérémie Jost, Joachim Hofer, Jonathon Ferguson, Lance Paine, Leif Wickland, ltbs, Marc Prud'hommeaux, Martin Carolan, Mr-SD, Narayan Iyer, Niccolo' Paravanti, niqdev, Noor Nashid, Pablo Francisco Pérez Hidalgo, Pawel Jurczenko, Phil Derome, Philip Schwarz, Riccardo Sirigu, Richard Dallaway, Rodney Jacobsen, Rodrigo B. de Oliveira, Seoh Char, Sergio Magnacco, Tim McIver, Toby Weston, Victor Osolovskiy, and Yinka Erinle.

If you spot an error or potential improvement, please raise an issue or submit a PR on the book's Github page.

Backers

We'd also like to extend very special thanks to our backers—fine people who helped fund the development of the book by buying a copy before we released

it as open source. This book wouldn't exist without you:

A battle-hardened technologist, Aaron Pritzlaff, Abhishek Srivastava, Aleksey "Daron" Terekhin, Algolia, Allen George (@allenageorge), Andrew Johnson, Andrew Kerr, Andy Dwelly, Anler, anthony@dribble.ai, Aravindh Sridaran, Araxis Ltd, ArtemK, Arthur Kushka (@arhelmus), Artur Zhurat, Arturas Smorgun, Attila Mravik, Axel Gschaider, Bamboo Le, bamine, Barry Kern, Ben Darfler (@bdarfler), Ben Letton, Benjamin Neil, Benoit Hericher, Bernt Andreas Langøien, Bill Leck, Blaze K, Boniface Kabaso, Brian Wongchaowart, Bryan Dragon, @cannedprimates, Ceschiatti (@6qat), Chris Gojlo, Chris Phelps, @CliffRedmond, Cody Koeninger, Constantin Gonciulea, Dadepo Aderemi, Damir Vandic, Damon Rolfs, Dan Todor, Daniel Arndt, Daniela Sfregola, David Greco, David Poltorak, Dennis Hunziker, Dennis Vriend, Derek Morr, Dimitrios Liapis, Don McNamara, Doug Clinton, Doug Lindholm (dlindhol), Edgar Mueller, Edward J Renauer Jr, Emiliano Martinez, esthom, Etienne Peiniau, Fede Silva, Filipe Azevedo, Franck Rasolo, Gary Coady, George Ball, Gerald Loeffler, Integrational, Giles Taylor, Guilherme Dantas (@gamsd), Harish Hurchurn, Hisham Ismail, Iurii Susuk, Ivan (SkyWriter) Kasatenko, Ivano Pagano, Jacob Baumbach, James Morris, Jan Vincent Liwanag, Javier Gonzalez, Jeff Gentry, Joel Chovanec, Jon Bates, Jorge Aliss (@jaliss), Juan Macias (@1macias1), Juan Ortega, Juan Pablo Romero Méndez, Jungsun Kim, Kaushik Chakraborty (@kaychaks), Keith Mannock, Ken Hoffman, Kevin Esler, Kevin Kyyro, kgillies, Klaus Rehm, Kostas Skourtis, Lance Linder, Liang, Guang Hua, Loïc Girault, Luke Tebbs, Makis A, Malcolm Robbins, Mansur Ashraf (@mansur_ashraf), Marcel Lüthi, Marek Prochera @hicolour, Marianudo (Mariano Navas), Mark Eibes, Mark van Rensburg, Martijn Blankestijn, Martin Studer, Matthew Edwards, Matthew Pflueger, mauropalsgraaf, mbarak, Mehitabel, Michael Pigg, Mikael Moghadam, Mike Gehard (@mikegehard), MonadicBind, arjun.mukherjee@gmail.com, Stephen Arbogast, Narayan Iyer, @natewave, Netanel Rabinowitz, Nick Peterson, Nicolas Sitbon, Oier Blasco Linares, Oliver Daff, Oliver Schrenk, Olly Shaw, P Villela, pandaforme, Patrick Garrity, Pawel Wlodarski from JUG Lodz, @peel, Peter Perhac, Phil Glover, Philipp Leser-Wolf, Rachel Bowyer, Radu Gancea (@radusw), Rajit Singh, Ramin Alidousti, Raymond Tay, Riccardo Sirigu, Richard (Yin-Wu) Chuo, Rob Vermazeren, Robert "Kemichal" Andersson, Robin Taylor (@badgermind), Rongcui Dong, Rui Morais, Rupert Bates, Rustem Suniev, Sanjiv Sahayam,

Shane Delmore, Stefan Plantikow, Sundy Wiliam Yaputra, Tal Pressman, Tamas Neltz, theLXK, Tim Pigden, Tobias Lutz, Tom Duhourq, @tomzalt, Utz Westermann, Vadym Shalts, Val Akkapeddi, Vasanth Loka, Vladimir Bacvanski, Vladimir Bystrov aka udav_pit, William Benton, Wojciech Langiewicz, Yann Ollivier (@ya2o), Yoshiro Naito, zero323, and zeronone.

Part I

Theory

Chapter 1

Introduction

Cats contains a wide variety of functional programming tools and allows developers to pick and choose the ones we want to use. The majority of these tools are delivered in the form of *type classes* that we can apply to existing Scala types.

Type classes are a programming pattern originating in Haskell[1]. They allow us to extend existing libraries with new functionality, without using traditional inheritance, and without altering the original library source code.

In this chapter we will refresh our memory of type classes from Underscore's Essential Scala book, and take a first look at the Cats codebase. We will look at two example type classes—`Show` and `Eq`—using them to identify patterns that lay the foundations for the rest of the book.

We'll finish by tying type classes back into algebraic data types, pattern matching, value classes, and type aliases, presenting a structured approach to functional programming in Scala.

[1]The word "class" doesn't strictly mean `class` in the Scala or Java sense.

1.1 Anatomy of a Type Class

There are three important components to the type class pattern: the *type class* itself, *instances* for particular types, and the *interface* methods that we expose to users.

1.1.1 The Type Class

A *type class* is an interface or API that represents some functionality we want to implement. In Cats a type class is represented by a trait with at least one type parameter. For example, we can represent generic "serialize to JSON" behaviour as follows:

```
// Define a very simple JSON AST
sealed trait Json
final case class JsObject(get: Map[String, Json]) extends Json
final case class JsString(get: String) extends Json
final case class JsNumber(get: Double) extends Json
case object JsNull extends Json

// The "serialize to JSON" behaviour is encoded in this trait
trait JsonWriter[A] {
  def write(value: A): Json
}
```

`JsonWriter` is our type class in this example, with `Json` and its subtypes providing supporting code.

1.1.2 Type Class Instances

The *instances* of a type class provide implementations for the types we care about, including types from the Scala standard library and types from our domain model.

In Scala we define instances by creating concrete implementations of the type class and tagging them with the `implicit` keyword:

```
final case class Person(name: String, email: String)

object JsonWriterInstances {
  implicit val stringWriter: JsonWriter[String] =
    new JsonWriter[String] {
      def write(value: String): Json =
        JsString(value)
    }

  implicit val personWriter: JsonWriter[Person] =
    new JsonWriter[Person] {
      def write(value: Person): Json =
        JsObject(Map(
          "name" -> JsString(value.name),
          "email" -> JsString(value.email)
        ))
    }

  // etc...
}
```

1.1.3 Type Class Interfaces

A type class *interface* is any functionality we expose to users. Interfaces are generic methods that accept instances of the type class as implicit parameters.

There are two common ways of specifying an interface: *Interface Objects* and *Interface Syntax*.

Interface Objects

The simplest way of creating an interface is to place methods in a singleton object:

```
object Json {
  def toJson[A](value: A)(implicit w: JsonWriter[A]): Json =
    w.write(value)
}
```

To use this object, we import any type class instances we care about and call the relevant method:

```
import JsonWriterInstances._

Json.toJson(Person("Dave", "dave@example.com"))
// res4: Json = JsObject(Map(name -> JsString(Dave), email -> JsString
     (dave@example.com)))
```

The compiler spots that we've called the `toJson` method without providing the implicit parameters. It tries to fix this by searching for type class instances of the relevant types and inserting them at the call site:

```
Json.toJson(Person("Dave", "dave@example.com"))(personWriter)
```

Interface Syntax

We can alternatively use *extension methods* to extend existing types with interface methods[2]. Cats refers to this as *"syntax"* for the type class:

```
object JsonSyntax {
  implicit class JsonWriterOps[A](value: A) {
    def toJson(implicit w: JsonWriter[A]): Json =
      w.write(value)
  }
}
```

We use interface syntax by importing it alongside the instances for the types we need:

```
import JsonWriterInstances._
import JsonSyntax._

Person("Dave", "dave@example.com").toJson
// res6: Json = JsObject(Map(name -> JsString(Dave), email -> JsString
     (dave@example.com)))
```

Again, the compiler searches for candidates for the implicit parameters and fills them in for us:

[2]You may occasionally see extension methods referred to as "type enrichment" or "pimping". These are older terms that we don't use anymore.

```
Person("Dave", "dave@example.com").toJson(personWriter)
```

The *implicitly* Method

The Scala standard library provides a generic type class interface called `implicitly`. Its definition is very simple:

```
def implicitly[A](implicit value: A): A =
  value
```

We can use `implicitly` to summon any value from implicit scope. We provide the type we want and `implicitly` does the rest:

```
import JsonWriterInstances._
// import JsonWriterInstances._

implicitly[JsonWriter[String]]
// res8: JsonWriter[String] = JsonWriterInstances$$anon$1@38ee55c4
```

Most type classes in Cats provide other means to summon instances. However, `implicitly` is a good fallback for debugging purposes. We can insert a call to `implicitly` within the general flow of our code to ensure the compiler can find an instance of a type class and ensure that there are no ambiguous implicit errors.

1.2 Working with Implicits

Working with type classes in Scala means working with implicit values and implicit parameters. There are a few rules we need to know to do this effectively.

1.2.1 Packaging Implicits

In a curious quirk of the language, any definitions marked `implicit` in Scala must be placed inside an object or trait rather than at the top level. In the example above we packaged our type class instances in an object called `Json-`

`WriterInstances`. We could equally have placed them in a companion object to `JsonWriter`. Placing instances in a companion object to the type class has special significance in Scala because it plays into something called *implicit scope*.

1.2.2 Implicit Scope

As we saw above, the compiler searches for candidate type class instances by type. For example, in the following expression it will look for an instance of type `JsonWriter[String]`:

```
Json.toJson("A string!")
```

The compiler searches for candidate instances in the *implicit scope* at the call site, which roughly consists of:

- local or inherited definitions;
- imported definitions;
- definitions in the companion object of the type class or the parameter type (in this case `JsonWriter` or `String`).

Definitions are only included in implicit scope if they are tagged with the `implicit` keyword. Furthermore, if the compiler sees multiple candidate definitions, it fails with an *ambiguous implicit values* error:

```
implicit val writer1: JsonWriter[String] =
  JsonWriterInstances.stringWriter

implicit val writer2: JsonWriter[String] =
  JsonWriterInstances.stringWriter

Json.toJson("A string")
// <console>:23: error: ambiguous implicit values:
//  both value stringWriter in object JsonWriterInstances of type =>
     JsonWriter[String]
//  and value writer1 of type => JsonWriter[String]
```

```
//  match expected type JsonWriter[String]
//         Json.toJson("A string")
//                     ^
```

The precise rules of implicit resolution are more complex than this, but the complexity is largely irrelevant for this book[3]. For our purposes, we can package type class instances in roughly four ways:

1. by placing them in an object such as `JsonWriterInstances`;
2. by placing them in a trait;
3. by placing them in the companion object of the type class;
4. by placing them in the companion object of the parameter type.

With option 1 we bring instances into scope by `importing` them. With option 2 we bring them into scope with inheritance. With options 3 and 4, instances are *always* in implicit scope, regardless of where we try to use them.

1.2.3 Recursive Implicit Resolution

The power of type classes and implicits lies in the compiler's ability to *combine* implicit definitions when searching for candidate instances.

Earlier we insinuated that all type class instances are `implicit vals`. This was a simplification. We can actually define instances in two ways:

1. by defining concrete instances as `implicit vals` of the required type[4];
2. by defining `implicit` methods to construct instances from other type class instances.

Why would we construct instances from other instances? As a motivational example, consider defining a `JsonWriter` for `Options`. We would need a `JsonWriter[Option[A]]` for every `A` we care about in our application. We could try to brute force the problem by creating a library of `implicit vals`:

[3] If you're interested in the finer rules of implicit resolution in Scala, start by taking a look at this Stack Overflow post on implicit scope and this blog post on implicit priority.

[4] `implicit objects` are treated the same way.

```
implicit val optionIntWriter: JsonWriter[Option[Int]] =
  ???

implicit val optionPersonWriter: JsonWriter[Option[Person]] =
  ???

// and so on...
```

However, this approach clearly doesn't scale. We end up requiring two `implicit vals` for every type `A` in our application: one for `A` and one for `Option[A]`.

Fortunately, we can abstract the code for handling `Option[A]` into a common constructor based on the instance for `A`:

- if the option is `Some(aValue)`, write `aValue` using the writer for `A`;
- if the option is `None`, return `JsNull`.

Here is the same code written out as an `implicit def`:

```
implicit def optionWriter[A]
    (implicit writer: JsonWriter[A]): JsonWriter[Option[A]] =
  new JsonWriter[Option[A]] {
    def write(option: Option[A]): Json =
      option match {
        case Some(aValue) => writer.write(aValue)
        case None         => JsNull
      }
  }
```

This method *constructs* a `JsonWriter` for `Option[A]` by relying on an implicit parameter to fill in the `A`-specific functionality. When the compiler sees an expression like this:

```
Json.toJson(Option("A string"))
```

it searches for an implicit `JsonWriter[Option[String]]`. It finds the implicit method for `JsonWriter[Option[A]]`:

```
Json.toJson(Option("A string"))(optionWriter[String])
```

and recursively searches for a `JsonWriter[String]` to use as the parameter to `optionWriter`:

```
Json.toJson(Option("A string"))(optionWriter(stringWriter))
```

In this way, implicit resolution becomes a search through the space of possible combinations of implicit definitions, to find a combination that summons a type class instance of the correct overall type.

> *Implicit Conversions*
>
> When you create a type class instance constructor using an `implicit def`, be sure to mark the parameters to the method as `implicit` parameters. Without this keyword, the compiler won't be able to fill in the parameters during implicit resolution.
>
> `implicit` methods with non-`implicit` parameters form a different Scala pattern called an *implicit conversion*. This is also different from the previous section on `Interface Syntax`, because in that case the `JsonWriter` is an implicit class with `extension methods`. Implicit conversion is an older programming pattern that is frowned upon in modern Scala code. Fortunately, the compiler will warn you when you do this. You have to manually enable implicit conversions by importing `scala.language.implicitConversions` in your file:

```
implicit def optionWriter[A]
    (writer: JsonWriter[A]): JsonWriter[Option[A]] =
  ???
// <console>:18: warning: implicit conversion method
     optionWriter should be enabled
// by making the implicit value scala.language.
     implicitConversions visible.
// This can be achieved by adding the import clause 'import
     scala.language.implicitConversions'
// or by setting the compiler option -language:
     implicitConversions.
// See the Scaladoc for value scala.language.implicitConversions
      for a discussion
// why the feature should be explicitly enabled.
//        implicit def optionWriter[A]
//                     ^
// error: No warnings can be incurred under -Xfatal-warnings.
```

1.3 Exercise: Printable Library

Scala provides a `toString` method to let us convert any value to a `String`. However, this method comes with a few disadvantages: it is implemented for *every* type in the language, many implementations are of limited use, and we can't opt-in to specific implementations for specific types.

Let's define a `Printable` type class to work around these problems:

1. Define a type class `Printable[A]` containing a single method `format`. `format` should accept a value of type `A` and return a `String`.

2. Create an object `PrintableInstances` containing instances of `Printable` for `String` and `Int`.

3. Define an object `Printable` with two generic interface methods:

   `format` accepts a value of type `A` and a `Printable` of the corresponding type. It uses the relevant `Printable` to convert the `A` to a `String`.

   `print` accepts the same parameters as `format` and returns `Unit`. It prints the `A` value to the console using `println`.

See the solution

Using the Library

The code above forms a general purpose printing library that we can use in multiple applications. Let's define an "application" now that uses the library.

First we'll define a data type to represent a well-known type of furry animal:

```
final case class Cat(name: String, age: Int, color: String)
```

Next we'll create an implementation of `Printable` for `Cat` that returns content in the following format:

```
NAME is a AGE year-old COLOR cat.
```

Finally, use the type class on the console or in a short demo app: create a `Cat` and print it to the console:

```
// Define a cat:
val cat = Cat(/* ... */)

// Print the cat!
```

See the solution

Better Syntax

Let's make our printing library easier to use by defining some extension methods to provide better syntax:

1. Create an object called `PrintableSyntax`.
2. Inside `PrintableSyntax` define an `implicit class PrintableOps[A]` to wrap up a value of type `A`.
3. In `PrintableOps` define the following methods:
    - `format` accepts an implicit `Printable[A]` and returns a `String` representation of the wrapped `A`;

- `print` accepts an implicit `Printable[A]` and returns `Unit`. It prints the wrapped `A` to the console.

4. Use the extension methods to print the example `Cat` you created in the previous exercise.

See the solution

1.4 Meet Cats

In the previous section we saw how to implement type classes in Scala. In this section we will look at how type classes are implemented in Cats.

Cats is written using a modular structure that allows us to choose which type classes, instances, and interface methods we want to use. Let's take a first look using `cats.Show` as an example.

`Show` is Cats' equivalent of the `Printable` type class we defined in the last section. It provides a mechanism for producing developer-friendly console output without using `toString`. Here's an abbreviated definition:

```
package cats

trait Show[A] {
  def show(value: A): String
}
```

1.4.1 Importing Type Classes

The type classes in Cats are defined in the `cats` package. We can import `Show` directly from this package:

```
import cats.Show
```

The companion object of every Cats type class has an `apply` method that locates an instance for any type we specify:

```
val showInt = Show.apply[Int]
// <console>:13: error: could not find implicit value for parameter
     instance: cats.Show[Int]
//        val showInt = Show.apply[Int]
//                                 ^
```

Oops—that didn't work! The `apply` method uses *implicits* to look up individual instances, so we'll have to bring some instances into scope.

1.4.2 Importing Default Instances

The `cats.instances` package provides default instances for a wide variety of types. We can import these as shown in the table below. Each import provides instances of all Cats' type classes for a specific parameter type:

- `cats.instances.int` provides instances for `Int`
- `cats.instances.string` provides instances for `String`
- `cats.instances.list` provides instances for `List`
- `cats.instances.option` provides instances for `Option`
- `cats.instances.all` provides all instances that are shipped out of the box with Cats

See the `cats.instances` package for a complete list of available imports.

Let's import the instances of `Show` for `Int` and `String`:

```
import cats.instances.int._    // for Show
import cats.instances.string._ // for Show

val showInt:    Show[Int]    = Show.apply[Int]
val showString: Show[String] = Show.apply[String]
```

That's better! We now have access to two instances of `Show`, and can use them to print `Int`s and `String`s:

```
val intAsString: String =
  showInt.show(123)
// intAsString: String = 123

val stringAsString: String =
  showString.show("abc")
// stringAsString: String = abc
```

1.4.3 Importing Interface Syntax

We can make Show easier to use by importing the *interface syntax* from `cats.syntax.show`. This adds an extension method called `show` to any type for which we have an instance of `Show` in scope:

```
import cats.syntax.show._ // for show

val shownInt = 123.show
// shownInt: String = 123

val shownString = "abc".show
// shownString: String = abc
```

Cats provides separate syntax imports for each type class. We will introduce these as we encounter them in later sections and chapters.

1.4.4 Importing All The Things!

In this book we will use specific imports to show you exactly which instances and syntax you need in each example. However, this can be time consuming for many use cases. You should feel free to take one of the following shortcuts to simplify your imports:

- `import cats._` imports all of Cats' type classes in one go;
- `import cats.instances.all._` imports all of the type class instances for the standard library in one go;

- `import cats.syntax.all._` imports all of the syntax in one go;
- `import cats.implicits._` imports all of the standard type class instances *and* all of the syntax in one go.

Most people start their files with the following imports, reverting to more specific imports only if they encounter naming conflicts or problems with ambiguous implicits:

```
import cats._
import cats.implicits._
```

1.4.5 Defining Custom Instances

We can define an instance of `Show` simply by implementing the trait for a given type:

```
import java.util.Date

implicit val dateShow: Show[Date] =
  new Show[Date] {
    def show(date: Date): String =
      s"${date.getTime}ms since the epoch."
  }
```

However, Cats also provides a couple of convenient methods to simplify the process. There are two construction methods on the companion object of `Show` that we can use to define instances for our own types:

```
object Show {
  // Convert a function to a `Show` instance:
  def show[A](f: A => String): Show[A] =
    ???

  // Create a `Show` instance from a `toString` method:
  def fromToString[A]: Show[A] =
```

```
    ???
}
```

These allow us to quickly construct instances with less ceremony than defining them from scratch:

```
implicit val dateShow: Show[Date] =
  Show.show(date => s"${date.getTime}ms since the epoch.")
```

As you can see, the code using construction methods is much terser than the code without. Many type classes in Cats provide helper methods like these for constructing instances, either from scratch or by transforming existing instances for other types.

1.4.6 Exercise: Cat Show

Re-implement the `Cat` application from the previous section using Show instead of `Printable`.

See the solution

1.5 Example: Eq

We will finish off this chapter by looking at another useful type class: `cats.Eq`. Eq is designed to support *type-safe equality* and address annoyances using Scala's built-in == operator.

Almost every Scala developer has written code like this before:

```
List(1, 2, 3).map(Option(_)).filter(item => item == 1)
// res0: List[Option[Int]] = List()
```

Ok, many of you won't have made such a simple mistake as this, but the principle is sound. The predicate in the `filter` clause always returns `false` because it is comparing an `Int` to an `Option[Int]`.

This is programmer error—we should have compared `item` to `Some(1)` instead of 1. However, it's not technically a type error because == works for any pair of objects, no matter what types we compare. `Eq` is designed to add some type safety to equality checks and work around this problem.

1.5.1 Equality, Liberty, and Fraternity

We can use `Eq` to define type-safe equality between instances of any given type:

```
package cats

trait Eq[A] {
  def eqv(a: A, b: A): Boolean
  // other concrete methods based on eqv...
}
```

The interface syntax, defined in `cats.syntax.eq`, provides two methods for performing equality checks provided there is an instance `Eq[A]` in scope:

- `===` compares two objects for equality;
- `=!=` compares two objects for inequality.

1.5.2 Comparing Ints

Let's look at a few examples. First we import the type class:

```
import cats.Eq
```

Now let's grab an instance for `Int`:

```
import cats.instances.int._ // for Eq

val eqInt = Eq[Int]
```

We can use `eqInt` directly to test for equality:

```
eqInt.eqv(123, 123)
// res2: Boolean = true

eqInt.eqv(123, 234)
// res3: Boolean = false
```

Unlike Scala's == method, if we try to compare objects of different types using eqv we get a compile error:

```
eqInt.eqv(123, "234")
// <console>:18: error: type mismatch;
//  found   : String("234")
//  required: Int
//        eqInt.eqv(123, "234")
//                       ^
```

We can also import the interface syntax in `cats.syntax.eq` to use the === and =!= methods:

```
import cats.syntax.eq._ // for === and =!=

123 === 123
// res5: Boolean = true

123 =!= 234
// res6: Boolean = true
```

Again, comparing values of different types causes a compiler error:

```
123 === "123"
// <console>:20: error: type mismatch;
//  found   : String("123")
//  required: Int
//        123 === "123"
//                ^
```

1.5.3 Comparing Options

Now for a more interesting example—`Option[Int]`. To compare values of type `Option[Int]` we need to import instances of `Eq` for `Option` as well as

`Int`:

```
import cats.instances.int._    // for Eq
import cats.instances.option._ // for Eq
```

Now we can try some comparisons:

```
Some(1) === None
// <console>:26: error: value === is not a member of Some[Int]
//        Some(1) === None
//                ^
```

We have received an error here because the types don't quite match up. We have `Eq` instances in scope for `Int` and `Option[Int]` but the values we are comparing are of type `Some[Int]`. To fix the issue we have to re-type the arguments as `Option[Int]`:

```
(Some(1) : Option[Int]) === (None : Option[Int])
// res9: Boolean = false
```

We can do this in a friendlier fashion using the `Option.apply` and `Option.empty` methods from the standard library:

```
Option(1) === Option.empty[Int]
// res10: Boolean = false
```

or using special syntax from `cats.syntax.option`:

```
import cats.syntax.option._ // for some and none

1.some === none[Int]
// res11: Boolean = false

1.some =!= none[Int]
// res12: Boolean = true
```

1.5.4 Comparing Custom Types

We can define our own instances of Eq using the `Eq.instance` method, which accepts a function of type `(A, A) => Boolean` and returns an `Eq[A]`:

```
import java.util.Date
import cats.instances.long._ // for Eq

implicit val dateEq: Eq[Date] =
  Eq.instance[Date] { (date1, date2) =>
    date1.getTime === date2.getTime
  }

val x = new Date() // now
val y = new Date() // a bit later than now

x === x
// res13: Boolean = true

x === y
// res14: Boolean = false
```

1.5.5 Exercise: Equality, Liberty, and Felinity

Implement an instance of Eq for our running `Cat` example:

```
final case class Cat(name: String, age: Int, color: String)
```

Use this to compare the following pairs of objects for equality and inequality:

```
val cat1 = Cat("Garfield",   38, "orange and black")
val cat2 = Cat("Heathcliff", 33, "orange and black")

val optionCat1 = Option(cat1)
val optionCat2 = Option.empty[Cat]
```

See the solution

1.6 Controlling Instance Selection

When working with type classes we must consider two issues that control instance selection:

- What is the relationship between an instance defined on a type and its subtypes?

  For example, if we define a `JsonWriter[Option[Int]]`, will the expression `Json.toJson(Some(1))` select this instance? (Remember that `Some` is a subtype of `Option`).

- How do we choose between type class instances when there are many available?

  What if we define two `JsonWriters` for `Person`? When we write `Json.toJson(aPerson)`, which instance is selected?

1.6.1 Variance

When we define type classes we can add variance annotations to the type parameter to affect the variance of the type class and the compiler's ability to select instances during implicit resolution.

To recap Essential Scala, variance relates to subtypes. We say that `B` is a subtype of `A` if we can use a value of type `B` anywhere we expect a value of type `A`.

Co- and contravariance annotations arise when working with type constructors. For example, we denote covariance with a + symbol:

```
trait F[+A] // the "+" means "covariant"
```

Covariance

Covariance means that the type `F[B]` is a subtype of the type `F[A]` if `B` is a subtype of `A`. This is useful for modelling many types, including collections like `List` and `Option`:

```
trait List[+A]
trait Option[+A]
```

The covariance of Scala collections allows us to substitute collections of one type for another in our code. For example, we can use a `List[Circle]` anywhere we expect a `List[Shape]` because `Circle` is a subtype of `Shape`:

```
sealed trait Shape
case class Circle(radius: Double) extends Shape

val circles: List[Circle] = ???
val shapes: List[Shape] = circles
```

What about contravariance? We write contravariant type constructors with a `-` symbol like this:

```
trait F[-A]
```

Contravariance

Confusingly, contravariance means that the type `F[B]` is a subtype of `F[A]` if `A` is a subtype of `B`. This is useful for modelling types that represent processes, like our `JsonWriter` type class above:

```
trait JsonWriter[-A] {
  def write(value: A): Json
}
// defined trait JsonWriter
```

Let's unpack this a bit further. Remember that variance is all about the ability to substitute one value for another. Consider a scenario where we have two values, one of type `Shape` and one of type `Circle`, and two `JsonWriters`, one for `Shape` and one for `Circle`:

```
val shape: Shape = ???
val circle: Circle = ???

val shapeWriter: JsonWriter[Shape] = ???
```

```
val circleWriter: JsonWriter[Circle] = ???

def format[A](value: A, writer: JsonWriter[A]): Json =
  writer.write(value)
```

Now ask yourself the question: "Which combinations of value and writer can I pass to `format`?" We can combine `circle` with either writer because all `Circles` are `Shapes`. Conversely, we can't combine `shape` with `circleWriter` because not all `Shapes` are `Circles`.

This relationship is what we formally model using contravariance. `JsonWriter[Shape]` is a subtype of `JsonWriter[Circle]` because `Circle` is a subtype of `Shape`. This means we can use `shapeWriter` anywhere we expect to see a `JsonWriter[Circle]`.

Invariance

Invariance is actually the easiest situation to describe. It's what we get when we don't write a `+` or `-` in a type constructor:

```
trait F[A]
```

This means the types `F[A]` and `F[B]` are never subtypes of one another, no matter what the relationship between `A` and `B`. This is the default semantics for Scala type constructors.

When the compiler searches for an implicit it looks for one matching the type *or subtype*. Thus we can use variance annotations to control type class instance selection to some extent.

There are two issues that tend to arise. Let's imagine we have an algebraic data type like:

```
sealed trait A
final case object B extends A
final case object C extends A
```

The issues are:

1. Will an instance defined on a supertype be selected if one is available? For example, can we define an instance for `A` and have it work for values of type `B` and `C`?

2. Will an instance for a subtype be selected in preference to that of a supertype. For instance, if we define an instance for `A` and `B`, and we have a value of type `B`, will the instance for `B` be selected in preference to `A`?

It turns out we can't have both at once. The three choices give us behaviour as follows:

Type Class Variance	Invariant	Covariant	Contravariant
Supertype instance used?	No	No	Yes
More specific type preferred?	No	Yes	No

It's clear there is no perfect system. Cats generally prefers to use invariant type classes. This allows us to specify more specific instances for subtypes if we want. It does mean that if we have, for example, a value of type `Some[Int]`, our type class instance for `Option` will not be used. We can solve this problem with a type annotation like `Some(1) : Option[Int]` or by using "smart constructors" like the `Option.apply`, `Option.empty`, `some`, and `none` methods we saw in Section 1.5.3.

1.7 Summary

In this chapter we took a first look at type classes. We implemented our own `Printable` type class using plain Scala before looking at two examples from Cats—`Show` and `Eq`.

We have now seen the general patterns in Cats type classes:

- The type classes themselves are generic traits in the `cats` package.

- Each type class has a companion object with, an `apply` method for materializing instances, one or more *construction* methods for creating instances, and a collection of other relevant helper methods.
- Default instances are provided via objects in the `cats.instances` package, and are organized by parameter type rather than by type class.
- Many type classes have *syntax* provided via the `cats.syntax` package.

In the remaining chapters of Part I we will look at several broad and powerful type classes—`Semigroup`, `Monoid`, `Functor`, `Monad`, `Semigroupal`, `Applicative`, `Traverse`, and more. In each case we will learn what functionality the type class provides, the formal rules it follows, and how it is implemented in Cats. Many of these type classes are more abstract than `Show` or `Eq`. While this makes them harder to learn, it makes them far more useful for solving general problems in our code.

Chapter 2

Monoids and Semigroups

In this section we explore our first type classes, **monoid** and **semigroup**. These allow us to add or combine values. There are instances for `Int`s, `String`s, `List`s, `Option`s, and many more. Let's start by looking at a few simple types and operations to see what common principles we can extract.

Integer addition

Addition of `Int`s is a binary operation that is *closed*, meaning that adding two `Int`s always produces another `Int`:

```
2 + 1
// res0: Int = 3
```

There is also the *identity* element `0` with the property that `a + 0 == 0 + a == a` for any `Int` `a`:

```
2 + 0
// res1: Int = 2

0 + 2
// res2: Int = 2
```

There are also other properties of addition. For instance, it doesn't matter in

what order we add elements because we always get the same result. This is a property known as *associativity*:

```
(1 + 2) + 3
// res3: Int = 6

1 + (2 + 3)
// res4: Int = 6
```

Integer multiplication

The same properties for addition also apply for multiplication, provided we use `1` as the identity instead of `0`:

```
1 * 3
// res5: Int = 3

3 * 1
// res6: Int = 3
```

Multiplication, like addition, is associative:

```
(1 * 2) * 3
// res7: Int = 6

1 * (2 * 3)
// res8: Int = 6
```

String and sequence concatenation

We can also add `Strings`, using string concatenation as our binary operator:

```
"One" ++ "two"
// res9: String = Onetwo
```

and the empty string as the identity:

```
"" ++ "Hello"
// res10: String = Hello

"Hello" ++ ""
// res11: String = Hello
```

Once again, concatenation is associative:

```
("One" ++ "Two") ++ "Three"
// res12: String = OneTwoThree

"One" ++ ("Two" ++ "Three")
// res13: String = OneTwoThree
```

Note that we used ++ above instead of the more usual + to suggest a parallel with sequences. We can do the same with other types of sequence, using concatenation as the binary operator and the empty sequence as our identity.

2.1 Definition of a Monoid

We've seen a number of "addition" scenarios above each with an associative binary addition and an identity element. It will be no surprise to learn that this is a monoid. Formally, a monoid for a type `A` is:

- an operation `combine` with type `(A, A) => A`
- an element `empty` of type `A`

This definition translates nicely into Scala code. Here is a simplified version of the definition from Cats:

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

In addition to providing the `combine` and `empty` operations, monoids must formally obey several *laws*. For all values `x`, `y`, and `z`, in `A`, `combine` must be associative and `empty` must be an identity element:

```
def associativeLaw[A](x: A, y: A, z: A)
      (implicit m: Monoid[A]): Boolean = {
  m.combine(x, m.combine(y, z)) ==
    m.combine(m.combine(x, y), z)
}

def identityLaw[A](x: A)
      (implicit m: Monoid[A]): Boolean = {
  (m.combine(x, m.empty) == x) &&
    (m.combine(m.empty, x) == x)
}
```

Integer subtraction, for example, is not a monoid because subtraction is not associative:

```
(1 - 2) - 3
// res15: Int = -4

1 - (2 - 3)
// res16: Int = 2
```

In practice we only need to think about laws when we are writing our own `Monoid` instances. Unlawful instances are dangerous because they can yield unpredictable results when used with the rest of Cats' machinery. Most of the time we can rely on the instances provided by Cats and assume the library authors know what they're doing.

2.2 Definition of a Semigroup

A semigroup is just the `combine` part of a monoid. While many semigroups are also monoids, there are some data types for which we cannot define an `empty` element. For example, we have just seen that sequence concatenation and integer addition are monoids. However, if we restrict ourselves to non-empty sequences and positive integers, we are no longer able to define a sensible `empty` element. Cats has a `NonEmptyList` data type that has an implementation of `Semigroup` but no implementation of `Monoid`.

A more accurate (though still simplified) definition of Cats' `Monoid` is:

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}
```

We'll see this kind of inheritance often when discussing type classes. It provides modularity and allows us to re-use behaviour. If we define a `Monoid` for a type `A`, we get a `Semigroup` for free. Similarly, if a method requires a parameter of type `Semigroup[B]`, we can pass a `Monoid[B]` instead.

2.3 Exercise: The Truth About Monoids

We've seen a few examples of monoids but there are plenty more to be found. Consider `Boolean`. How many monoids can you define for this type? For each monoid, define the `combine` and `empty` operations and convince yourself that the monoid laws hold. Use the following definitions as a starting point:

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}

object Monoid {
  def apply[A](implicit monoid: Monoid[A]) =
    monoid
}
```

See the solution

2.4 Exercise: All Set for Monoids

What monoids and semigroups are there for sets?

See the solution

2.5 Monoids in Cats

Now we've seen what monoids are, let's look at their implementation in Cats. Once again we'll look at the three main aspects of the implementation: the *type class*, the *instances*, and the *interface*.

2.5.1 The Monoid Type Class

The monoid type class is `cats.kernel.Monoid`, which is aliased as `cats.Monoid`. `Monoid` extends `cats.kernel.Semigroup`, which is aliased as `cats.Semigroup`. When using Cats we normally import type classes from the `cats` package:

```
import cats.Monoid
import cats.Semigroup
```

> *Cats Kernel?*
>
> Cats Kernel is a subproject of Cats providing a small set of typeclasses for libraries that don't require the full Cats toolbox. While these core type classes are technically defined in the `cats.kernel` package, they are all aliased to the `cats` package so we rarely need to be aware of the distinction.
>
> The Cats Kernel type classes covered in this book are `Eq`, `Semigroup`, and `Monoid`. All the other type classes we cover are part of the main Cats project and are defined directly in the `cats` package.

2.5.2 Monoid Instances

`Monoid` follows the standard Cats pattern for the user interface: the companion object has an `apply` method that returns the type class instance for a particular type. For example, if we want the monoid instance for `String`, and we have the correct implicits in scope, we can write the following:

```
import cats.Monoid
import cats.instances.string._ // for Monoid

Monoid[String].combine("Hi ", "there")
// res0: String = Hi there

Monoid[String].empty
// res1: String = ""
```

which is equivalent to:

```
Monoid.apply[String].combine("Hi ", "there")
// res2: String = Hi there

Monoid.apply[String].empty
// res3: String = ""
```

As we know, `Monoid` extends `Semigroup`. If we don't need `empty` we can equivalently write:

```
import cats.Semigroup

Semigroup[String].combine("Hi ", "there")
// res4: String = Hi there
```

The type class instances for `Monoid` are organised under `cats.instances` in the standard way described in Chapter 1. For example, if we want to pull in instances for `Int` we import from `cats.instances.int`:

```
import cats.Monoid
import cats.instances.int._ // for Monoid

Monoid[Int].combine(32, 10)
// res5: Int = 42
```

Similarly, we can assemble a `Monoid[Option[Int]]` using instances from `cats.instances.int` and `cats.instances.option`:

```
import cats.Monoid
import cats.instances.int._    // for Monoid
import cats.instances.option._ // for Monoid

val a = Option(22)
// a: Option[Int] = Some(22)

val b = Option(20)
// b: Option[Int] = Some(20)

Monoid[Option[Int]].combine(a, b)
// res6: Option[Int] = Some(42)
```

Refer back to Chapter 1 for a more comprehensive list of imports.

2.5.3 Monoid Syntax

Cats provides syntax for the `combine` method in the form of the `|+|` operator. Because `combine` technically comes from `Semigroup`, we access the syntax by importing from `cats.syntax.semigroup`:

```
import cats.instances.string._ // for Monoid
import cats.syntax.semigroup._ // for |+|

val stringResult = "Hi " |+| "there" |+| Monoid[String].empty
// stringResult: String = Hi there

import cats.instances.int._ // for Monoid

val intResult = 1 |+| 2 |+| Monoid[Int].empty
```

```
// intResult: Int = 3
```

2.5.4 Exercise: Adding All The Things

The cutting edge *SuperAdder v3.5a-32* is the world's first choice for adding together numbers. The main function in the program has signature `def add(items: List[Int]): Int`. In a tragic accident this code is deleted! Rewrite the method and save the day!

See the solution

Well done! SuperAdder's market share continues to grow, and now there is demand for additional functionality. People now want to add `List[Option[Int]]`. Change `add` so this is possible. The SuperAdder code base is of the highest quality, so make sure there is no code duplication!

See the solution

SuperAdder is entering the POS (point-of-sale, not the other POS) market. Now we want to add up `Orders`:

```
case class Order(totalCost: Double, quantity: Double)
```

We need to release this code really soon so we can't make any modifications to `add`. Make it so!

See the solution

2.6 Applications of Monoids

We now know what a monoid is—an abstraction of the concept of adding or combining—but where is it useful? Here are a few big ideas where monoids play a major role. These are explored in more detail in case studies later in the book.

2.6.1 Big Data

In big data applications like Spark and Hadoop we distribute data analysis over many machines, giving fault tolerance and scalability. This means each machine will return results over a portion of the data, and we must then combine these results to get our final result. In the vast majority of cases this can be viewed as a monoid.

If we want to calculate how many total visitors a web site has received, that means calculating an `Int` on each portion of the data. We know the monoid instance of `Int` is addition, which is the right way to combine partial results.

If we want to find out how many unique visitors a website has received, that's equivalent to building a `Set[User]` on each portion of the data. We know the monoid instance for `Set` is the set union, which is the right way to combine partial results.

If we want to calculate 99% and 95% response times from our server logs, we can use a data structure called a `QTree` for which there is a monoid.

Hopefully you get the idea. Almost every analysis that we might want to do over a large data set is a monoid, and therefore we can build an expressive and powerful analytics system around this idea. This is exactly what Twitter's Algebird and Summingbird projects have done. We explore this idea further in the map-reduce case study.

2.6.2 Distributed Systems

In a distributed system, different machines may end up with different views of data. For example, one machine may receive an update that other machines did not receive. We would like to reconcile these different views, so every machine has the same data if no more updates arrive. This is called *eventual consistency*.

A particular class of data types support this reconciliation. These data types are called commutative replicated data types (CRDTs). The key operation is the ability to merge two data instances, with a result that captures all the information in both instances. This operation relies on having a monoid instance. We explore this idea further in the CRDT case study.

2.6.3 Monoids in the Small

The two examples above are cases where monoids inform the entire system architecture. There are also many cases where having a monoid around makes it easier to write a small code fragment. We'll see lots of examples in the case studies in this book.

2.7 Summary

We hit a big milestone in this chapter—we covered our first type classes with fancy functional programming names:

- a `Semigroup` represents an addition or combination operation;
- a `Monoid` extends a `Semigroup` by adding an identity or "zero" element.

We can use `Semigroups` and `Monoids` by importing three things: the type classes themselves, the instances for the types we care about, and the semigroup syntax to give us the `|+|` operator:

```
import cats.Monoid
import cats.instances.string._ // for Monoid
import cats.syntax.semigroup._ // for |+|

"Scala" |+| " with " |+| "Cats"
// res0: String = Scala with Cats
```

With the correct instances in scope, we can set about adding anything we want:

```
import cats.instances.int._    // for Monoid
import cats.instances.option._ // for Monoid

Option(1) |+| Option(2)
// res1: Option[Int] = Some(3)

import cats.instances.map._ // for Monoid
```

```
val map1 = Map("a" -> 1, "b" -> 2)
val map2 = Map("b" -> 3, "d" -> 4)

map1 |+| map2
// res3: Map[String,Int] = Map(b -> 5, d -> 4, a -> 1)

import cats.instances.tuple._  // for Monoid


val tuple1 = ("hello", 123)
val tuple2 = ("world", 321)

tuple1 |+| tuple2
// res6: (String, Int) = (helloworld,444)
```

We can also write generic code that works with any type for which we have an instance of `Monoid`:

```
def addAll[A](values: List[A])
      (implicit monoid: Monoid[A]): A =
  values.foldRight(monoid.empty)(_ |+| _)

addAll(List(1, 2, 3))
// res7: Int = 6

addAll(List(None, Some(1), Some(2)))
// res8: Option[Int] = Some(3)
```

`Monoids` are a great gateway to Cats. They're easy to understand and simple to use. However, they're just the tip of the iceberg in terms of the abstractions Cats enables us to make. In the next chapter we'll look at *functors*, the type class personification of the beloved `map` method. That's where the fun really begins!

Chapter 3

Functors

In this chapter we will investigate **functors**, an abstraction that allows us to represent sequences of operations within a context such as a `List`, an `Option`, or any one of a thousand other possibilities. Functors on their own aren't so useful, but special cases of functors such as **monads** and **applicative functors** are some of the most commonly used abstractions in Cats.

3.1 Examples of Functors

Informally, a functor is anything with a `map` method. You probably know lots of types that have this: `Option`, `List`, and `Either`, to name a few.

We typically first encounter `map` when iterating over `Lists`. However, to understand functors we need to think of the method in another way. Rather than traversing the list, we should think of it as transforming all of the values inside in one go. We specify the function to apply, and `map` ensures it is applied to every item. The values change but the structure of the list remains the same:

```
List(1, 2, 3).map(n => n + 1)
// res0: List[Int] = List(2, 3, 4)
```

Similarly, when we `map` over an `Option`, we transform the contents but leave

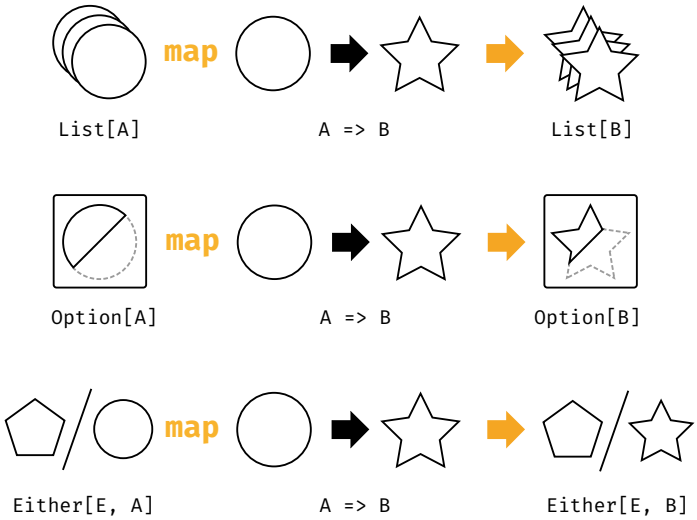


Figure 3.1: Type chart: mapping over List, Option, and Either

the `Some` or `None` context unchanged. The same principle applies to `Either` with its `Left` and `Right` contexts. This general notion of transformation, along with the common pattern of type signatures shown in Figure 3.1, is what connects the behaviour of `map` across different data types.

Because `map` leaves the structure of the context unchanged, we can call it repeatedly to sequence multiple computations on the contents of an initial data structure:

```
List(1, 2, 3).
  map(n => n + 1).
  map(n => n * 2).
  map(n => n + "!")
// res1: List[String] = List(4!, 6!, 8!)
```

We should think of `map` not as an iteration pattern, but as a way of sequencing computations on values ignoring some complication dictated by the relevant data type:

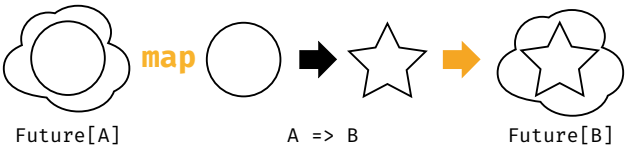


Figure 3.2: Type chart: mapping over a Future

- `Option`—the value may or may not be present;
- `Either`—there may be a value or an error;
- `List`—there may be zero or more values.

3.2 More Examples of Functors

The `map` methods of `List`, `Option`, and `Either` apply functions eagerly. However, the idea of sequencing computations is more general than this. Let's investigate the behaviour of some other functors that apply the pattern in different ways.

Futures

`Future` is a functor that sequences asynchronous computations by queueing them and applying them as their predecessors complete. The type signature of its `map` method, shown in Figure 3.2, has the same shape as the signatures above. However, the behaviour is very different.

When we work with a `Future` we have no guarantees about its internal state. The wrapped computation may be ongoing, complete, or rejected. If the `Future` is complete, our mapping function can be called immediately. If not, some underlying thread pool queues the function call and comes back to it later. We don't know *when* our functions will be called, but we do know *what order* they will be called in. In this way, `Future` provides the same sequencing behaviour seen in `List`, `Option`, and `Either`:

```
import scala.concurrent.{Future, Await}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._

val future: Future[String] =
  Future(123).
    map(n => n + 1).
    map(n => n * 2).
    map(n => n + "!")

Await.result(future, 1.second)
// res3: String = 248!
```

Futures and Referential Transparency

Note that Scala's `Futures` aren't a great example of pure functional programming because they aren't *referentially transparent*. `Future` always computes and caches a result and there's no way for us to tweak this behaviour. This means we can get unpredictable results when we use `Future` to wrap side-effecting computations. For example:

```
import scala.util.Random

val future1 = {
  // Initialize Random with a fixed seed:
  val r = new Random(0L)

  // nextInt has the side-effect of moving to
  // the next random number in the sequence:
  val x = Future(r.nextInt)

  for {
    a <- x
    b <- x
  } yield (a, b)
}

val future2 = {
  val r = new Random(0L)

  for {
    a <- Future(r.nextInt)
    b <- Future(r.nextInt)
  } yield (a, b)
}

val result1 = Await.result(future1, 1.second)
// result1: (Int, Int) = (-1155484576,-1155484576)

val result2 = Await.result(future2, 1.second)
// result2: (Int, Int) = (-1155484576,-723955400)
```

Ideally we would like `result1` and `result2` to contain the same value. However, the computation for `future1` calls `nextInt` once and the computation for `future2` calls it twice. Because `nextInt` returns a different result every time we get a different result in each case.

This kind of discrepancy makes it hard to reason about programs involving `Futures` and side-effects. There also are other problematic aspects of `Future's` behaviour, such as the way it always starts computations immediately rather than allowing the user to dictate when the program should run. For more information see this excellent Reddit answer by

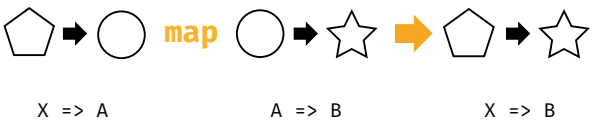


Figure 3.3: Type chart: mapping over a Function1

> Rob Norris.

If `Future` isn't referentially transparent, perhaps we should look at another similar data-type that is. You should recognise this one...

Functions (?!)

It turns out that single argument functions are also functors. To see this we have to tweak the types a little. A function `A => B` has two type parameters: the parameter type `A` and the result type `B`. To coerce them to the correct shape we can fix the parameter type and let the result type vary:

- start with `X => A`;
- supply a function `A => B`;
- get back `X => B`.

If we alias `X => A` as `MyFunc[A]`, we see the same pattern of types we saw with the other examples in this chapter. We also see this in Figure 3.3:

- start with `MyFunc[A]`;
- supply a function `A => B`;
- get back `MyFunc[B]`.

In other words, "mapping" over a `Function1` is function composition:

```
import cats.instances.function._ // for Functor
import cats.syntax.functor._     // for map

val func1: Int => Double =
  (x: Int) => x.toDouble
```

```
val func2: Double => Double =
  (y: Double) => y * 2

(func1 map func2)(1)     // composition using map
// res7: Double = 2.0

(func1 andThen func2)(1) // composition using andThen
// res8: Double = 2.0

func2(func1(1))          // composition written out by hand
// res9: Double = 2.0
```

How does this relate to our general pattern of sequencing operations? If we think about it, function composition *is* sequencing. We start with a function that performs a single operation and every time we use `map` we append another operation to the chain. Calling `map` doesn't actually *run* any of the operations, but if we can pass an argument to the final function all of the operations are run in sequence. We can think of this as lazily queueing up operations similar to `Future`:

```
val func =
  ((x: Int) => x.toDouble).
    map(x => x + 1).
    map(x => x * 2).
    map(x => x + "!")

func(123)
// res10: String = 248.0!
```

> *Partial Unification*
>
> For the above examples to work we need to add the following compiler option to `build.sbt`:
>
> ```
> scalacOptions += "-Ypartial-unification"
> ```
>
> otherwise we'll get a compiler error:

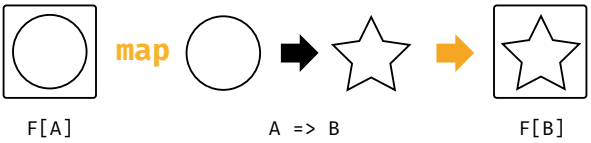


Figure 3.4: Type chart: generalised functor map

```
func1.map(func2)
// <console>: error: value map is not a member of Int => Double
//        func1.map(func2)
                  ^
```

We'll look at why this happens in detail in Section 3.8.

3.3 Definition of a Functor

Every example we've looked at so far is a functor: a class that encapsulates sequencing computations. Formally, a functor is a type `F[A]` with an operation `map` with type `(A => B) => F[B]`. The general type chart is shown in Figure 3.4.

Cats encodes `Functor` as a type class, `cats.Functor`, so the method looks a little different. It accepts the initial `F[A]` as a parameter alongside the transformation function. Here's a simplified version of the definition:

```
package cats

import scala.language.higherKinds

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

If you haven't seen syntax like `F[_]` before, it's time to take a brief detour to discuss *type constructors* and *higher kinded types*. We'll explain that `scala.language` import as well.

Functor Laws

Functors guarantee the same semantics whether we sequence many small operations one by one, or combine them into a larger function before `mapping`. To ensure this is the case the following laws must hold:

Identity: calling `map` with the identity function is the same as doing nothing:

```
fa.map(a => a) == fa
```

Composition: `mapping` with two functions `f` and `g` is the same as `mapping` with `f` and then `mapping` with `g`:

```
fa.map(g(f(_))) == fa.map(f).map(g)
```

3.4 Aside: Higher Kinds and Type Constructors

Kinds are like types for types. They describe the number of "holes" in a type. We distinguish between regular types that have no holes and "type constructors" that have holes we can fill to produce types.

For example, `List` is a type constructor with one hole. We fill that hole by specifying a parameter to produce a regular type like `List[Int]` or `List[A]`. The trick is not to confuse type constructors with generic types. `List` is a type constructor, `List[A]` is a type:

```
List    // type constructor, takes one parameter
List[A] // type, produced using a type parameter
```

There's a close analogy here with functions and values. Functions are "value constructors"—they produce values when we supply parameters:

```
math.abs    // function, takes one parameter
math.abs(x) // value, produced using a value parameter
```

In Scala we declare type constructors using underscores. Once we've declared them, however, we refer to them as simple identifiers:

```
// Declare F using underscores:
def myMethod[F[_]] = {

  // Reference F without underscores:
  val functor = Functor.apply[F]

  // ...
}
```

This is analogous to specifying a function's parameters in its definition and omitting them when referring to it:

```
// Declare f specifying parameters:
val f = (x: Int) => x * 2

// Reference f without parameters:
val f2 = f andThen f
```

Armed with this knowledge of type constructors, we can see that the Cats definition of `Functor` allows us to create instances for any single-parameter type constructor, such as `List`, `Option`, `Future`, or a type alias such as `MyFunc`.

> *Language Feature Imports*
>
> Higher kinded types are considered an advanced language feature in Scala. Whenever we declare a type constructor with `A[_]` syntax, we need to "enable" the higher kinded type language feature to suppress warnings from the compiler. We can either do this with a "language import" as above:
>
> ```
> import scala.language.higherKinds
> ```
>
> or by adding the following to `scalacOptions` in `build.sbt`:

```
scalacOptions += "-language:higherKinds"
```

We'll use the language import in this book to ensure we are as explicit as possible. In practice, however, we find the `scalacOptions` flag to be simpler and less verbose.

3.5 Functors in Cats

Let's look at the implementation of functors in Cats. We'll examine the aspects we did for monoids: the *type class*, the *instances*, and the *syntax*.

3.5.1 The Functor Type Class

The functor type class is `cats.Functor`. We obtain instances using the standard `Functor.apply` method on the companion object. As usual, default instances are arranged by type in the `cats.instances` package:

```
import scala.language.higherKinds
import cats.Functor
import cats.instances.list._   // for Functor
import cats.instances.option._ // for Functor

val list1 = List(1, 2, 3)
// list1: List[Int] = List(1, 2, 3)

val list2 = Functor[List].map(list1)(_ * 2)
// list2: List[Int] = List(2, 4, 6)

val option1 = Option(123)
// option1: Option[Int] = Some(123)

val option2 = Functor[Option].map(option1)(_.toString)
// option2: Option[String] = Some(123)
```

`Functor` also provides the `lift` method, which converts a function of type `A => B` to one that operates over a functor and has type `F[A] => F[B]`:

```
val func = (x: Int) => x + 1
// func: Int => Int = <function1>

val liftedFunc = Functor[Option].lift(func)
// liftedFunc: Option[Int] => Option[Int] = cats.Functor$$Lambda$11698
     /1371437204@27fbe7ae

liftedFunc(Option(1))
// res0: Option[Int] = Some(2)
```

3.5.2 Functor Syntax

The main method provided by the syntax for `Functor` is `map`. It's difficult to demonstrate this with `Options` and `Lists` as they have their own built-in `map` methods and the Scala compiler will always prefer a built-in method over an extension method. We'll work around this with two examples.

First let's look at mapping over functions. Scala's `Function1` type doesn't have a `map` method (it's called `andThen` instead) so there are no naming conflicts:

```
import cats.instances.function._ // for Functor
import cats.syntax.functor._     // for map

val func1 = (a: Int) => a + 1
val func2 = (a: Int) => a * 2
val func3 = (a: Int) => a + "!"
val func4 = func1.map(func2).map(func3)

func4(123)
// res1: String = 248!
```

Let's look at another example. This time we'll abstract over functors so we're not working with any particular concrete type. We can write a method that applies an equation to a number no matter what functor context it's in:

```
def doMath[F[_]](start: F[Int])
    (implicit functor: Functor[F]): F[Int] =
  start.map(n => n + 1 * 2)
```

```
import cats.instances.option._ // for Functor
import cats.instances.list._   // for Functor

doMath(Option(20))
// res3: Option[Int] = Some(22)

doMath(List(1, 2, 3))
// res4: List[Int] = List(3, 4, 5)
```

To illustrate how this works, let's take a look at the definition of the `map` method in `cats.syntax.functor`. Here's a simplified version of the code:

```
implicit class FunctorOps[F[_], A](src: F[A]) {
  def map[B](func: A => B)
      (implicit functor: Functor[F]): F[B] =
    functor.map(src)(func)
}
```

The compiler can use this extension method to insert a `map` method wherever no built-in `map` is available:

```
foo.map(value => value + 1)
```

Assuming `foo` has no built-in `map` method, the compiler detects the potential error and wraps the expression in a `FunctorOps` to fix the code:

```
new FunctorOps(foo).map(value => value + 1)
```

The `map` method of `FunctorOps` requires an implicit `Functor` as a parameter. This means this code will only compile if we have a `Functor` for `expr1` in scope. If we don't, we get a compiler error:

```
final case class Box[A](value: A)

val box = Box[Int](123)

box.map(value => value + 1)
// <console>:34: error: value map is not a member of Box[Int]
```

```
//        box.map(value => value + 1)
//            ^
```

3.5.3 Instances for Custom Types

We can define a functor simply by defining its map method. Here's an example of a `Functor` for `Option`, even though such a thing already exists in `cats.instances`. The implementation is trivial—we simply call `Option`'s `map` method:

```
implicit val optionFunctor: Functor[Option] =
  new Functor[Option] {
    def map[A, B](value: Option[A])(func: A => B): Option[B] =
      value.map(func)
  }
```

Sometimes we need to inject dependencies into our instances. For example, if we had to define a custom `Functor` for `Future` (another hypothetical example—Cats provides one in `cats.instances.future`) we would need to account for the implicit `ExecutionContext` parameter on `future.map`. We can't add extra parameters to `functor.map` so we have to account for the dependency when we create the instance:

```
import scala.concurrent.{Future, ExecutionContext}

implicit def futureFunctor
    (implicit ec: ExecutionContext): Functor[Future] =
  new Functor[Future] {
    def map[A, B](value: Future[A])(func: A => B): Future[B] =
      value.map(func)
  }
```

Whenever we summon a `Functor` for `Future`, either directly using `Functor.apply` or indirectly via the `map` extension method, the compiler will locate `futureFunctor` by implicit resolution and recursively search for an `ExecutionContext` at the call site. This is what the expansion might look like:

```
// We write this:
Functor[Future]

// The compiler expands to this first:
Functor[Future](futureFunctor)

// And then to this:
Functor[Future](futureFunctor(executionContext))
```

3.5.4 Exercise: Branching out with Functors

Write a `Functor` for the following binary tree data type. Verify that the code works as expected on instances of `Branch` and `Leaf`:

```
sealed trait Tree[+A]

final case class Branch[A](left: Tree[A], right: Tree[A])
  extends Tree[A]

final case class Leaf[A](value: A) extends Tree[A]
```

See the solution

3.6 *Contravariant* and Invariant Functors

As we have seen, we can think of `Functor's map` method as "appending" a transformation to a chain. We're now going to look at two other type classes, one representing *prepending* operations to a chain, and one representing building a *bidirectional* chain of operations. These are called *contravariant* and *invariant functors* respectively.

> *This Section is Optional!*
>
> You don't need to know about contravariant and invariant functors to understand monads, which are the most important pattern in this book and the focus of the next chapter. However, contravariant and invariant

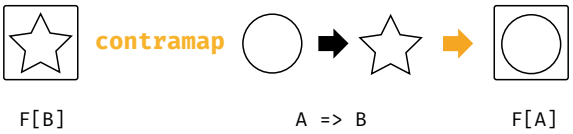


Figure 3.5: Type chart: the contramap method

> do come in handy in our discussion of `Semigroupal` and `Applicative` in Chapter 6.
>
> If you want to move on to monads now, feel free to skip straight to Chapter 4. Come back here before you read Chapter 6.

3.6.1 Contravariant Functors and the *contramap* Method

The first of our type classes, the *contravariant functor*, provides an operation called `contramap` that represents "prepending" an operation to a chain. The general type signature is shown in Figure 3.5.

The `contramap` method only makes sense for data types that represent *transformations*. For example, we can't define `contramap` for an `Option` because there is no way of feeding a value in an `Option[B]` backwards through a function `A => B`. However, we can define `contramap` for the `Printable` type class we discussed in Chapter 1:

```
trait Printable[A] {
  def format(value: A): String
}
```

A `Printable[A]` represents a transformation from `A` to `String`. Its `contramap` method accepts a function `func` of type `B => A` and creates a new `Printable[B]`:

```
trait Printable[A] {
  def format(value: A): String
```

```
  def contramap[B](func: B => A): Printable[B] =
    ???
}

def format[A](value: A)(implicit p: Printable[A]): String =
  p.format(value)
```

3.6.1.1 Exercise: Showing off with Contramap

Implement the `contramap` method for `Printable` above. Start with the following code template and replace the `???` with a working method body:

```
trait Printable[A] {
  def format(value: A): String

  def contramap[B](func: B => A): Printable[B] =
    new Printable[B] {
      def format(value: B): String =
        ???
    }
}
```

If you get stuck, think about the types. You need to turn `value`, which is of type `B`, into a `String`. What functions and methods do you have available and in what order do they need to be combined?

See the solution

For testing purposes, let's define some instances of `Printable` for `String` and `Boolean`:

```
implicit val stringPrintable: Printable[String] =
  new Printable[String] {
    def format(value: String): String =
      "\"" + value + "\""
  }

implicit val booleanPrintable: Printable[Boolean] =
  new Printable[Boolean] {
```

```
    def format(value: Boolean): String =
      if(value) "yes" else "no"
  }

format("hello")
// res3: String = "hello"

format(true)
// res4: String = yes
```

Now define an instance of `Printable` for the following `Box` case class. You'll need to write this as an `implicit def` as described in Section 1.2.3:

```
final case class Box[A](value: A)
```

Rather than writing out the complete definition from scratch (`new Printable[Box]` etc...), create your instance from an existing instance using `contramap`.

See the solution

Your instance should work as follows:

```
format(Box("hello world"))
// res5: String = "hello world"

format(Box(true))
// res6: String = yes
```

If we don't have a `Printable` for the type inside the `Box`, calls to `format` should fail to compile:

```
format(Box(123))
// <console>:21: error: could not find implicit value for parameter p:
      Printable[Box[Int]]
//        format(Box(123))
//               ^
```

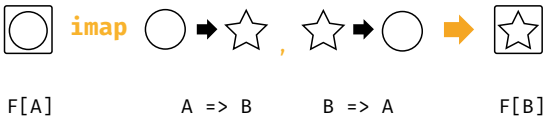


Figure 3.6: Type chart: the imap method

3.6.2 Invariant functors and the *imap* method

Invariant functors implement a method called `imap` that is informally equivalent to a combination of `map` and `contramap`. If `map` generates new type class instances by appending a function to a chain, and `contramap` generates them by prepending an operation to a chain, `imap` generates them via a pair of bidirectional transformations.

The most intuitive examples of this are a type class that represents encoding and decoding as some data type, such as Play JSON's `Format` and scodec's `Codec`. We can build our own `Codec` by enhancing `Printable` to support encoding and decoding to/from a `String`:

```
trait Codec[A] {
  def encode(value: A): String
  def decode(value: String): A
  def imap[B](dec: A => B, enc: B => A): Codec[B] = ???
}

def encode[A](value: A)(implicit c: Codec[A]): String =
  c.encode(value)

def decode[A](value: String)(implicit c: Codec[A]): A =
  c.decode(value)
```

The type chart for `imap` is shown in Figure 3.6. If we have a `Codec[A]` and a pair of functions `A => B` and `B => A`, the `imap` method creates a `Codec[B]`:

As an example use case, imagine we have a basic `Codec[String]`, whose `encode` and `decode` methods are both a no-op:

```
implicit val stringCodec: Codec[String] =
  new Codec[String] {
    def encode(value: String): String = value
    def decode(value: String): String = value
  }
```

We can construct many useful `Codecs` for other types by building off of `stringCodec` using `imap`:

```
implicit val intCodec: Codec[Int] =
  stringCodec.imap(_.toInt, _.toString)

implicit val booleanCodec: Codec[Boolean] =
  stringCodec.imap(_.toBoolean, _.toString)
```

> *Coping with Failure*
>
> Note that the `decode` method of our `Codec` type class doesn't account for failures. If we want to model more sophisticated relationships we can move beyond functors to look at *lenses* and *optics*.
>
> Optics are beyond the scope of this book. However, Julien Truffaut's library Monocle provides a great starting point for further investigation.

3.6.2.1 Transformative Thinking with *imap*

Implement the `imap` method for `Codec` above.

See the solution

Demonstrate your `imap` method works by creating a `Codec` for `Double`.

See the solution

Finally, implement a `Codec` for the following `Box` type:

```
case class Box[A](value: A)
```

See the solution

Your instances should work as follows:

```
encode(123.4)
// res0: String = 123.4

decode[Double]("123.4")
// res1: Double = 123.4

encode(Box(123.4))
// res2: String = 123.4

decode[Box[Double]]("123.4")
// res3: Box[Double] = Box(123.4)
```

What's With the Names?

What's the relationship between the terms "contravariance", "invariance", and "covariance" and these different kinds of functor?

If you recall from Section 1.6.1, variance affects subtyping, which is essentially our ability to use a value of one type in place of a value of another type without breaking the code.

Subtyping can be viewed as a conversion. If `B` is a subtype of `A`, we can always convert a `B` to an `A`.

Equivalently we could say that `B` is a subtype of `A` if there exists a function `A => B`. A standard covariant functor captures exactly this. If `F` is a covariant functor, wherever we have an `F[A]` and a conversion `A => B` we can always convert to an `F[B]`.

A contravariant functor captures the opposite case. If `F` is a contravariant functor, whenever we have a `F[A]` and a conversion `B => A` we can convert to an `F[B]`.

Finally, invariant functors capture the case where we can convert from `F[A]` to `F[B]` via a function `A => B` and vice versa via a function `B => A`.

3.7 *Contravariant* and Invariant in Cats

Let's look at the implementation of contravariant and invariant functors in Cats, provided by the `cats.Contravariant` and `cats.Invariant` type classes. Here's a simplified version of the code:

```
trait Contravariant[F[_]] {
  def contramap[A, B](fa: F[A])(f: B => A): F[B]
}

trait Invariant[F[_]] {
  def imap[A, B](fa: F[A])(f: A => B)(g: B => A): F[B]
}
```

3.7.1 Contravariant in Cats

We can summon instances of `Contravariant` using the `Contravariant.apply` method. Cats provides instances for data types that consume parameters, including `Eq`, `Show`, and `Function1`. Here's an example:

```
import cats.Contravariant
import cats.Show
import cats.instances.string._

val showString = Show[String]

val showSymbol = Contravariant[Show].
  contramap(showString)((sym: Symbol) => s"'${sym.name}")

showSymbol.show('dave)
// res2: String = 'dave
```

More conveniently, we can use `cats.syntax.contravariant`, which provides a `contramap` extension method:

```
import cats.syntax.contravariant._ // for contramap

showString.contramap[Symbol](_.name).show('dave)
```

```
// res3: String = dave
```

3.7.2 Invariant in Cats

Among other types, Cats provides an instance of `Invariant` for `Monoid`. This is a little different from the `Codec` example we introduced in Section 3.6.2. If you recall, this is what `Monoid` looks like:

```
package cats

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

Imagine we want to produce a `Monoid` for Scala's `Symbol` type. Cats doesn't provide a `Monoid` for `Symbol` but it does provide a `Monoid` for a similar type: `String`. We can write our new semigroup with an `empty` method that relies on the empty `String`, and a `combine` method that works as follows:

1. accept two `Symbols` as parameters;
2. convert the `Symbols` to `Strings`;
3. combine the `Strings` using `Monoid[String]`;
4. convert the result back to a `Symbol`.

We can implement `combine` using `imap`, passing functions of type `String => Symbol` and `Symbol => String` as parameters. Here' the code, written out using the `imap` extension method provided by `cats.syntax.invariant`:

```
import cats.Monoid
import cats.instances.string._ // for Monoid
import cats.syntax.invariant._ // for imap
import cats.syntax.semigroup._ // for |+|

implicit val symbolMonoid: Monoid[Symbol] =
  Monoid[String].imap(Symbol.apply)(_.name)
```

```
Monoid[Symbol].empty
// res5: Symbol = '

'a |+| 'few |+| 'words
// res6: Symbol = 'afewwords
```

3.8 Aside: Partial Unification

In Section 3.2 we saw a curious compiler error. The following code compiled perfectly if we had the `-Ypartial-unification` compiler flag enabled:

```
import cats.Functor
import cats.instances.function._ // for Functor
import cats.syntax.functor._     // for map

val func1 = (x: Int)    => x.toDouble
val func2 = (y: Double) => y * 2

val func3 = func1.map(func2)
// func3: Int => Double = scala.runtime.AbstractFunction1$$Lambda$7404
     /290370740@246b5bc6
```

but failed if the flag was missing:

```
val func3 = func1.map(func2)
// <console>: error: value map is not a member of Int => Double
//        val func3 = func1.map(func2)
                             ^
```

Obviously "partial unification" is some kind of optional compiler behaviour, without which our code will not compile. We should take a moment to describe this behaviour and discuss some gotchas and workarounds.

3.8.1 Unifying Type Constructors

In order to compile an expression like `func1.map(func2)` above, the compiler has to search for a `Functor` for `Function1`. However, `Functor` accepts a type constructor with one parameter:

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

and `Function1` has two type parameters (the function argument and the result type):

```
trait Function1[-A, +B] {
  def apply(arg: A): B
}
```

The compiler has to fix one of the two parameters of `Function1` to create a type constructor of the correct kind to pass to `Functor`. It has two options to choose from:

```
type F[A] = Int => A
type F[A] = A => Double
```

We know that the former of these is the correct choice. However, earlier versions of the Scala compiler were not able to make this inference. This infamous limitation, known as SI-2712, prevented the compiler from "unifying" type constructors of different arities. This compiler limitation is now fixed, although we have to enable the fix via a compiler flag in `build.sbt`:

```
scalacOptions += "-Ypartial-unification"
```

3.8.2 Left-to-Right Elimination

The partial unification in the Scala compiler works by fixing type parameters from left to right. In the above example, the compiler fixes the `Int` in `Int => Double` and looks for a `Functor` for functions of type `Int => ?`:

```
type F[A] = Int => A

val functor = Functor[F]
```

This left-to-right elimination works for a wide variety of common scenarios, including `Functors` for types such as `Function1` and `Either`:

```
import cats.instances.either._ // for Functor

val either: Either[String, Int] = Right(123)
// either: Either[String,Int] = Right(123)

either.map(_ + 1)
// res2: scala.util.Either[String,Int] = Right(124)
```

However, there are situations where left-to-right elimination is not the correct choice. One example is the `Or` type in Scalactic, which is a conventionally left-biased equivalent of `Either`:

```
type PossibleResult = ActualResult Or Error
```

Another example is the `Contravariant` functor for `Function1`.

While the covariant `Functor` for `Function1` implements `andThen`-style left-to-right function composition, the `Contravariant` functor implements `compose`-style right-to-left composition. In other words, the following expressions are all equivalent:

```
val func3a: Int => Double =
  a => func2(func1(a))

val func3b: Int => Double =
  func2.compose(func1)

// Hypothetical example. This won't actually compile:
val func3c: Int => Double =
  func2.contramap(func1)
```

If we try this for real, however, our code won't compile:

```
import cats.syntax.contravariant._ // for contramap

val func3c = func2.contramap(func1)
```

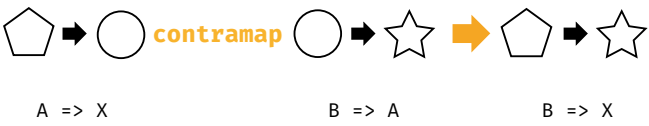


Figure 3.7: Type chart: contramapping over a Function1

```
// <console>:27: error: value contramap is not a member of Double =>
     Double
//        val func3c = func2.contramap(func1)
//                           ^
```

The problem here is that the `Contravariant` for `Function1` fixes the return type and leaves the parameter type varying, requiring the compiler to eliminate type parameters from right to left, as shown below and in Figure 3.7:

```
type F[A] = A => Double
```

The compiler fails simply because of its left-to-right bias. We can prove this by creating a type alias that flips the parameters on Function1:

```
type <=[B, A] = A => B

type F[A] = Double <= A
```

If we re-type `func2` as an instance of <=, we reset the required order of elimination and we can call `contramap` as desired:

```
val func2b: Double <= Double = func2

val func3c = func2b.contramap(func1)
// func3c: Double <= Int = scala.runtime.
     AbstractFunction1$$Lambda$7404/290370740@2d89ff6b
```

The difference between `func2` and `func2b` is purely syntactic—both refer to the same value and the type aliases are otherwise completely compatible. Incredibly, however, this simple rephrasing is enough to give the compiler the hint it needs to solve the problem.

It is rare that we have to do this kind of right-to-left elimination. Most multi-parameter type constructors are designed to be right-biased, requiring the left-to-right elimination that is supported by the compiler out of the box. However, it is useful to know about `-Ypartial-unification` and this quirk of elimination order in case you ever come across an odd scenario like the one above.

3.9 Summary

Functors represent sequencing behaviours. We covered three types of functor in this chapter:

- Regular covariant `Functors`, with their `map` method, represent the ability to apply functions to a value in some context. Successive calls to `map` apply these functions in *sequence*, each accepting the result of its predecessor as a parameter.
- `Contravariant` functors, with their `contramap` method, represent the ability to "prepend" functions to a function-like context. Successive calls to `contramap` sequence these functions in the opposite order to `map`.
- `Invariant` functors, with their `imap` method, represent bidirectional transformations.

Regular `Functors` are by far the most common of these type classes, but even then it is rare to use them on their own. Functors form a foundational building block of several more interesting abstractions that we use all the time. In the following chapters we will look at two of these abstractions: *monads* and *applicative functors*.

Functors for collections are extremely important, as they transform each element independently of the rest. This allows us to parallelise or distribute transformations on large collections, a technique leveraged heavily in "map-reduce" frameworks like Hadoop. We will investigate this approach in more detail in the Map-reduce case study later in the book.

The `Contravariant` and `Invariant` type classes are less widely applicable but are still useful for building data types that represent transformations. We will revisit them to discuss the `Semigroupal` type class later in Chapter 6.

Chapter 4

Monads

Monads are one of the most common abstractions in Scala. Many Scala programmers quickly become intuitively familiar with monads, even if we don't know them by name.

Informally, a monad is anything with a constructor and a `flatMap` method. All of the functors we saw in the last chapter are also monads, including `Option`, `List`, and `Future`. We even have special syntax to support monads: for comprehensions. However, despite the ubiquity of the concept, the Scala standard library lacks a concrete type to encompass "things that can be `flatMapped`". This type class is one of the benefits brought to us by Cats.

In this chapter we will take a deep dive into monads. We will start by motivating them with a few examples. We'll proceed to their formal definition and their implementation in Cats. Finally, we'll tour some interesting monads that you may not have seen, providing introductions and examples of their use.

4.1 What is a Monad?

This is the question that has been posed in a thousand blog posts, with explanations and analogies involving concepts as diverse as cats, Mexican food, space suits full of toxic waste, and monoids in the category of endofunctors

(whatever that means). We're going to solve the problem of explaining monads once and for all by stating very simply:

> A monad is a mechanism for *sequencing computations*.

That was easy! Problem solved, right? But then again, last chapter we said functors were a control mechanism for exactly the same thing. Ok, maybe we need some more discussion...

In Section 3.1 we said that functors allow us to sequence computations ignoring some complication. However, functors are limited in that they only allow this complication to occur once at the beginning of the sequence. They don't account further complications at each step in the sequence.

This is where monads come in. A monad's `flatMap` method allows us to specify what happens next, taking into account an intermediate complication. The `flatMap` method of `Option` takes intermediate `Options` into account. The `flatMap` method of `List` handles intermediate `Lists`. And so on. In each case, the function passed to `flatMap` specifies the application-specific part of the computation, and `flatMap` itself takes care of the complication allowing us to `flatMap` again. Let's ground things by looking at some examples.

Options

`Option` allows us to sequence computations that may or may not return values. Here are some examples:

```
def parseInt(str: String): Option[Int] =
  scala.util.Try(str.toInt).toOption

def divide(a: Int, b: Int): Option[Int] =
  if(b == 0) None else Some(a / b)
```

Each of these methods may "fail" by returning `None`. The `flatMap` method allows us to ignore this when we sequence operations:

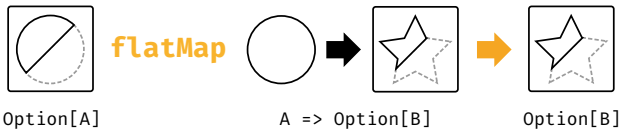


Figure 4.1: Type chart: flatMap for Option

```
def stringDivideBy(aStr: String, bStr: String): Option[Int] =
  parseInt(aStr).flatMap { aNum =>
    parseInt(bStr).flatMap { bNum =>
      divide(aNum, bNum)
    }
  }
```

We know the semantics well:

- the first call to `parseInt` returns a `None` or a `Some`;
- if it returns a `Some`, the `flatMap` method calls our function and passes us the integer `aNum`;
- the second call to `parseInt` returns a `None` or a `Some`;
- if it returns a `Some`, the `flatMap` method calls our function and passes us `bNum`;
- the call to `divide` returns a `None` or a `Some`, which is our result.

At each step, `flatMap` chooses whether to call our function, and our function generates the next computation in the sequence. This is shown in Figure 4.1.

The result of the computation is an `Option`, allowing us to call `flatMap` again and so the sequence continues. This results in the fail-fast error handling behaviour that we know and love, where a `None` at any step results in a `None` overall:

```
stringDivideBy("6", "2")
// res1: Option[Int] = Some(3)

stringDivideBy("6", "0")
// res2: Option[Int] = None
```

```
stringDivideBy("6", "foo")
// res3: Option[Int] = None

stringDivideBy("bar", "2")
// res4: Option[Int] = None
```

Every monad is also a functor (see below for proof), so we can rely on both `flatMap` and `map` to sequence computations that do and don't introduce a new monad. Plus, if we have both `flatMap` and `map` we can use for comprehensions to clarify the sequencing behaviour:

```
def stringDivideBy(aStr: String, bStr: String): Option[Int] =
  for {
    aNum <- parseInt(aStr)
    bNum <- parseInt(bStr)
    ans  <- divide(aNum, bNum)
  } yield ans
```

Lists

When we first encounter `flatMap` as budding Scala developers, we tend to think of it as a pattern for iterating over `Lists`. This is reinforced by the syntax of for comprehensions, which look very much like imperative for loops:

```
for {
  x <- (1 to 3).toList
  y <- (4 to 5).toList
} yield (x, y)
// res5: List[(Int, Int)] = List((1,4), (1,5), (2,4), (2,5), (3,4),
    (3,5))
```

However, there is another mental model we can apply that highlights the monadic behaviour of `List`. If we think of `Lists` as sets of intermediate results, `flatMap` becomes a construct that calculates permutations and combinations.

For example, in the for comprehension above there are three possible values of `x` and two possible values of `y`. This means there are six possible values

of `(x, y)`. `flatMap` is generating these combinations from our code, which states the sequence of operations:

- get x
- get y
- create a tuple `(x, y)`

Futures

`Future` is a monad that sequences computations without worrying that they are asynchronous:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._

def doSomethingLongRunning: Future[Int] = ???
def doSomethingElseLongRunning: Future[Int] = ???

def doSomethingVeryLongRunning: Future[Int] =
  for {
    result1 <- doSomethingLongRunning
    result2 <- doSomethingElseLongRunning
  } yield result1 + result2
```

Again, we specify the code to run at each step, and `flatMap` takes care of all the horrifying underlying complexities of thread pools and schedulers.

If you've made extensive use of `Future`, you'll know that the code above is running each operation *in sequence*. This becomes clearer if we expand out the for comprehension to show the nested calls to `flatMap`:

```
def doSomethingVeryLongRunning: Future[Int] =
  doSomethingLongRunning.flatMap { result1 =>
    doSomethingElseLongRunning.map { result2 =>
      result1 + result2
    }
  }
```

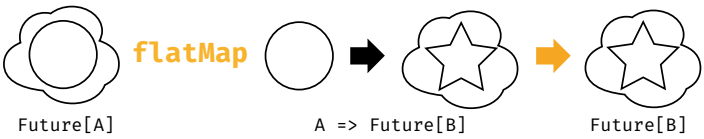


Figure 4.2: Type chart: flatMap for Future

Each `Future` in our sequence is created by a function that receives the result from a previous `Future`. In other words, each step in our computation can only start once the previous step is finished. This is born out by the type chart for `flatMap` in Figure 4.2, which shows the function parameter of type `A => Future[B]`.

We *can* run futures in parallel, of course, but that is another story and shall be told another time. Monads are all about sequencing.

4.1.1 Definition of a Monad

While we have only talked about `flatMap` above, monadic behaviour is formally captured in two operations:

- `pure`, of type `A => F[A]`;
- `flatMap`[1], of type `(F[A], A => F[B]) => F[B]`.

`pure` abstracts over constructors, providing a way to create a new monadic context from a plain value. `flatMap` provides the sequencing step we have already discussed, extracting the value from a context and generating the next context in the sequence. Here is a simplified version of the `Monad` type class in Cats:

[1] In some libraries and languages, notably Scalaz and Haskell, `pure` is referred to as `point` or `return` and `flatMap` is referred to as `bind` or `>>=`. This is purely a difference in terminology. We'll use the term `flatMap` for compatibility with Cats and the Scala standard library.

```
import scala.language.higherKinds

trait Monad[F[_]] {
  def pure[A](value: A): F[A]

  def flatMap[A, B](value: F[A])(func: A => F[B]): F[B]
}
```

Monad Laws

`pure` and `flatMap` must obey a set of laws that allow us to sequence operations freely without unintended glitches and side-effects:

Left identity: calling `pure` and transforming the result with `func` is the same as calling `func`:

```
pure(a).flatMap(func) == func(a)
```

Right identity: passing `pure` to `flatMap` is the same as doing nothing:

```
m.flatMap(pure) == m
```

Associativity: `flatMapping` over two functions `f` and `g` is the same as `flatMapping` over `f` and then `flatMapping` over `g`:

```
m.flatMap(f).flatMap(g) == m.flatMap(x => f(x).flatMap(g))
```

4.1.2 Exercise: Getting Func-y

Every monad is also a functor. We can define `map` in the same way for every monad using the existing methods, `flatMap` and `pure`:

```
import scala.language.higherKinds

trait Monad[F[_]] {
```

```
  def pure[A](a: A): F[A]

  def flatMap[A, B](value: F[A])(func: A => F[B]): F[B]

  def map[A, B](value: F[A])(func: A => B): F[B] =
    ???
}
```

Try defining `map` yourself now.

See the solution

4.2 Monads in Cats

It's time to give monads our standard Cats treatment. As usual we'll look at the type class, instances, and syntax.

4.2.1 The Monad Type Class

The monad type class is `cats.Monad`. `Monad` extends two other type classes: `FlatMap`, which provides the `flatMap` method, and `Applicative`, which provides `pure`. `Applicative` also extends `Functor`, which gives every `Monad` a `map` method as we saw in the exercise above. We'll discuss `Applicatives` in Chapter 6.

Here are some examples using `pure` and `flatMap`, and `map` directly:

```
import cats.Monad
import cats.instances.option._ // for Monad
import cats.instances.list._   // for Monad

val opt1 = Monad[Option].pure(3)
// opt1: Option[Int] = Some(3)

val opt2 = Monad[Option].flatMap(opt1)(a => Some(a + 2))
// opt2: Option[Int] = Some(5)

val opt3 = Monad[Option].map(opt2)(a => 100 * a)
```

```
// opt3: Option[Int] = Some(500)

val list1 = Monad[List].pure(3)
// list1: List[Int] = List(3)

val list2 = Monad[List].
  flatMap(List(1, 2, 3))(a => List(a, a*10))
// list2: List[Int] = List(1, 10, 2, 20, 3, 30)

val list3 = Monad[List].map(list2)(a => a + 123)
// list3: List[Int] = List(124, 133, 125, 143, 126, 153)
```

`Monad` provides many other methods, including all of the methods from `Functor`. See the scaladoc for more information.

4.2.2 Default Instances

Cats provides instances for all the monads in the standard library (`Option`, `List`, `Vector` and so on) via `cats.instances`:

```
import cats.instances.option._ // for Monad

Monad[Option].flatMap(Option(1))(a => Option(a*2))
// res0: Option[Int] = Some(2)

import cats.instances.list._ // for Monad

Monad[List].flatMap(List(1, 2, 3))(a => List(a, a*10))
// res1: List[Int] = List(1, 10, 2, 20, 3, 30)

import cats.instances.vector._ // for Monad

Monad[Vector].flatMap(Vector(1, 2, 3))(a => Vector(a, a*10))
// res2: Vector[Int] = Vector(1, 10, 2, 20, 3, 30)
```

Cats also provides a `Monad` for `Future`. Unlike the methods on the `Future` class itself, the `pure` and `flatMap` methods on the monad can't accept implicit `ExecutionContext` parameters (because the parameters aren't part of the definitions in the `Monad` trait). To work around this, Cats requires us to have an `ExecutionContext` in scope when we summon a `Monad` for `Future`:

```
import cats.instances.future._ // for Monad
import scala.concurrent._
import scala.concurrent.duration._

val fm = Monad[Future]
// <console>:37: error: could not find implicit value for parameter
     instance: cats.Monad[scala.concurrent.Future]
//        val fm = Monad[Future]
//                      ^
```

Bringing the `ExecutionContext` into scope fixes the implicit resolution required to summon the instance:

```
import scala.concurrent.ExecutionContext.Implicits.global

val fm = Monad[Future]
// fm: cats.Monad[scala.concurrent.Future] = cats.instances.
     FutureInstances$$anon$1@71738c4a
```

The `Monad` instance uses the captured `ExecutionContext` for subsequent calls to `pure` and `flatMap`:

```
val future = fm.flatMap(fm.pure(1))(x => fm.pure(x + 2))

Await.result(future, 1.second)
// res3: Int = 3
```

In addition to the above, Cats provides a host of new monads that we don't have in the standard library. We'll familiarise ourselves with some of these in a moment.

4.2.3 Monad Syntax

The syntax for monads comes from three places:

- `cats.syntax.flatMap` provides syntax for `flatMap`;
- `cats.syntax.functor` provides syntax for `map`;

- `cats.syntax.applicative` provides syntax for `pure`.

In practice it's often easier to import everything in one go from `cats.implicits`. However, we'll use the individual imports here for clarity.

We can use `pure` to construct instances of a monad. We'll often need to specify the type parameter to disambiguate the particular instance we want.

```
import cats.instances.option._   // for Monad
import cats.instances.list._     // for Monad
import cats.syntax.applicative._ // for pure

1.pure[Option]
// res4: Option[Int] = Some(1)

1.pure[List]
// res5: List[Int] = List(1)
```

It's difficult to demonstrate the `flatMap` and `map` methods directly on Scala monads like `Option` and `List`, because they define their own explicit versions of those methods. Instead we'll write a generic function that performs a calculation on parameters that come wrapped in a monad of the user's choice:

```
import cats.Monad
import cats.syntax.functor._ // for map
import cats.syntax.flatMap._ // for flatMap
import scala.language.higherKinds

def sumSquare[F[_]: Monad](a: F[Int], b: F[Int]): F[Int] =
  a.flatMap(x => b.map(y => x*x + y*y))

import cats.instances.option._ // for Monad
import cats.instances.list._   // for Monad

sumSquare(Option(3), Option(4))
// res8: Option[Int] = Some(25)

sumSquare(List(1, 2, 3), List(4, 5))
// res9: List[Int] = List(17, 26, 20, 29, 25, 34)
```

We can rewrite this code using for comprehensions. The compiler will "do the right thing" by rewriting our comprehension in terms of `flatMap` and `map` and inserting the correct implicit conversions to use our `Monad`:

```
def sumSquare[F[_]: Monad](a: F[Int], b: F[Int]): F[Int] =
  for {
    x <- a
    y <- b
  } yield x*x + y*y

sumSquare(Option(3), Option(4))
// res10: Option[Int] = Some(25)

sumSquare(List(1, 2, 3), List(4, 5))
// res11: List[Int] = List(17, 26, 20, 29, 25, 34)
```

That's more or less everything we need to know about the generalities of monads in Cats. Now let's take a look at some useful monad instances that we haven't seen in the Scala standard library.

4.3 The Identity Monad

In the previous section we demonstrated Cats' `flatMap` and `map` syntax by writing a method that abstracted over different monads:

```
import scala.language.higherKinds
import cats.Monad
import cats.syntax.functor._ // for map
import cats.syntax.flatMap._ // for flatMap

def sumSquare[F[_]: Monad](a: F[Int], b: F[Int]): F[Int] =
  for {
    x <- a
    y <- b
  } yield x*x + y*y
```

This method works well on `Options` and `Lists` but we can't call it passing in plain values:

```
sumSquare(3, 4)
// <console>:22: error: no type parameters for method sumSquare: (a: F
    [Int], b: F[Int])(implicit evidence$1: cats.Monad[F])F[Int] exist
     so that it can be applied to arguments (Int, Int)
//  --- because ---
// argument expression's type is not compatible with formal parameter
    type;
//  found   : Int
//  required: ?F[Int]
//        sumSquare(3, 4)
//        ^
// <console>:22: error: type mismatch;
//  found   : Int(3)
//  required: F[Int]
//        sumSquare(3, 4)
//                  ^
// <console>:22: error: type mismatch;
//  found   : Int(4)
//  required: F[Int]
//        sumSquare(3, 4)
//                     ^
```

It would be incredibly useful if we could use `sumSquare` with parameters that were either in a monad or not in a monad at all. This would allow us to abstract over monadic and non-monadic code. Fortunately, Cats provides the `Id` type to bridge the gap:

```
import cats.Id

sumSquare(3 : Id[Int], 4 : Id[Int])
// res2: cats.Id[Int] = 25
```

`Id` allows us to call our monadic method using plain values. However, the exact semantics are difficult to understand. We cast the parameters to `sumSquare` as `Id[Int]` and received an `Id[Int]` back as a result!

What's going on? Here is the definition of `Id` to explain:

```
package cats

type Id[A] = A
```

`Id` is actually a type alias that turns an atomic type into a single-parameter type constructor. We can cast any value of any type to a corresponding `Id`:

```
"Dave" : Id[String]
// res3: cats.Id[String] = Dave

123 : Id[Int]
// res4: cats.Id[Int] = 123

List(1, 2, 3) : Id[List[Int]]
// res5: cats.Id[List[Int]] = List(1, 2, 3)
```

Cats provides instances of various type classes for `Id`, including `Functor` and `Monad`. These let us call `map`, `flatMap`, and `pure` passing in plain values:

```
val a = Monad[Id].pure(3)
// a: cats.Id[Int] = 3

val b = Monad[Id].flatMap(a)(_ + 1)
// b: cats.Id[Int] = 4

import cats.syntax.functor._ // for map
import cats.syntax.flatMap._ // for flatMap

for {
  x <- a
  y <- b
} yield x + y
// res6: cats.Id[Int] = 7
```

The ability to abstract over monadic and non-monadic code is extremely powerful. For example, we can run code asynchronously in production using `Future` and synchronously in test using `Id`. We'll see this in our first case study in Chapter 8.

4.3.1 Exercise: Monadic Secret Identities

Implement `pure`, `map`, and `flatMap` for `Id`! What interesting discoveries do you uncover about the implementation?

See the solution

4.4 Either

Let's look at another useful monad: the `Either` type from the Scala standard library. In Scala 2.11 and earlier, many people didn't consider `Either` a monad because it didn't have `map` and `flatMap` methods. In Scala 2.12, however, `Either` became *right biased*.

4.4.1 Left and Right Bias

In Scala 2.11, `Either` had no default `map` or `flatMap` method. This made the Scala 2.11 version of `Either` inconvenient to use in for comprehensions. We had to insert calls to `.right` in every generator clause:

```
val either1: Either[String, Int] = Right(10)
val either2: Either[String, Int] = Right(32)

for {
  a <- either1.right
  b <- either2.right
} yield a + b
// res0: scala.util.Either[String,Int] = Right(42)
```

In Scala 2.12, `Either` was redesigned. The modern `Either` makes the decision that the right side represents the success case and thus supports `map` and `flatMap` directly. This makes for comprehensions much more pleasant:

```
for {
  a <- either1
  b <- either2
```

```
} yield a + b
// res1: scala.util.Either[String,Int] = Right(42)
```

Cats back-ports this behaviour to Scala 2.11 via the `cats.syntax.either` import, allowing us to use right-biased `Either` in all supported versions of Scala. In Scala 2.12+ we can either omit this import or leave it in place without breaking anything:

```
import cats.syntax.either._ // for map and flatMap

for {
  a <- either1
  b <- either2
} yield a + b
```

4.4.2 Creating Instances

In addition to creating instances of `Left` and `Right` directly, we can also import the `asLeft` and `asRight` extension methods from `cats.syntax.either`:

```
import cats.syntax.either._ // for asRight

val a = 3.asRight[String]
// a: Either[String,Int] = Right(3)

val b = 4.asRight[String]
// b: Either[String,Int] = Right(4)

for {
  x <- a
  y <- b
} yield x*x + y*y
// res4: scala.util.Either[String,Int] = Right(25)
```

These "smart constructors" have advantages over `Left.apply` and `Right.apply` because they return results of type `Either` instead of `Left` and `Right`. This helps avoid type inference bugs caused by over-narrowing, like the bug in the example below:

```
def countPositive(nums: List[Int]) =
  nums.foldLeft(Right(0)) { (accumulator, num) =>
    if(num > 0) {
      accumulator.map(_ + 1)
    } else {
      Left("Negative. Stopping!")
    }
  }
// <console>:21: error: type mismatch;
//  found   : scala.util.Either[Nothing,Int]
//  required: scala.util.Right[Nothing,Int]
//              accumulator.map(_ + 1)
//                          ^
// <console>:23: error: type mismatch;
//  found   : scala.util.Left[String,Nothing]
//  required: scala.util.Right[Nothing,Int]
//              Left("Negative. Stopping!")
//                  ^
```

This code fails to compile for two reasons:

1. the compiler infers the type of the accumulator as `Right` instead of `Either`;
2. we didn't specify type parameters for `Right.apply` so the compiler infers the left parameter as `Nothing`.

Switching to `asRight` avoids both of these problems. `asRight` has a return type of `Either`, and allows us to completely specify the type with only one type parameter:

```
def countPositive(nums: List[Int]) =
  nums.foldLeft(0.asRight[String]) { (accumulator, num) =>
    if(num > 0) {
      accumulator.map(_ + 1)
    } else {
      Left("Negative. Stopping!")
    }
  }

countPositive(List(1, 2, 3))
```

```
// res5: Either[String,Int] = Right(3)

countPositive(List(1, -2, 3))
// res6: Either[String,Int] = Left(Negative. Stopping!)
```

`cats.syntax.either` adds some useful extension methods to the `Either` companion object. The `catchOnly` and `catchNonFatal` methods are great for capturing `Exceptions` as instances of `Either`:

```
Either.catchOnly[NumberFormatException]("foo".toInt)
// res7: Either[NumberFormatException,Int] = Left(java.lang.
     NumberFormatException: For input string: "foo")

Either.catchNonFatal(sys.error("Badness"))
// res8: Either[Throwable,Nothing] = Left(java.lang.RuntimeException:
     Badness)
```

There are also methods for creating an `Either` from other data types:

```
Either.fromTry(scala.util.Try("foo".toInt))
// res9: Either[Throwable,Int] = Left(java.lang.NumberFormatException:
      For input string: "foo")

Either.fromOption[String, Int](None, "Badness")
// res10: Either[String,Int] = Left(Badness)
```

4.4.3 Transforming Eithers

`cats.syntax.either` also adds some useful methods for instances of `Either`. We can use `orElse` and `getOrElse` to extract values from the right side or return a default:

```
import cats.syntax.either._

"Error".asLeft[Int].getOrElse(0)
// res11: Int = 0
```

```
"Error".asLeft[Int].orElse(2.asRight[String])
// res12: Either[String,Int] = Right(2)
```

The `ensure` method allows us to check whether the right-hand value satisfies a predicate:

```
-1.asRight[String].ensure("Must be non-negative!")(_ > 0)
// res13: Either[String,Int] = Left(Must be non-negative!)
```

The `recover` and `recoverWith` methods provide similar error handling to their namesakes on `Future`:

```
"error".asLeft[Int].recover {
  case str: String => -1
}
// res14: Either[String,Int] = Right(-1)

"error".asLeft[Int].recoverWith {
  case str: String => Right(-1)
}
// res15: Either[String,Int] = Right(-1)
```

There are `leftMap` and `bimap` methods to complement `map`:

```
"foo".asLeft[Int].leftMap(_.reverse)
// res16: Either[String,Int] = Left(oof)

6.asRight[String].bimap(_.reverse, _ * 7)
// res17: Either[String,Int] = Right(42)

"bar".asLeft[Int].bimap(_.reverse, _ * 7)
// res18: Either[String,Int] = Left(rab)
```

The `swap` method lets us exchange left for right:

```
123.asRight[String]
// res19: Either[String,Int] = Right(123)
```

```
123.asRight[String].swap
// res20: scala.util.Either[Int,String] = Left(123)
```

Finally, Cats adds a host of conversion methods: `toOption`, `toList`, `toTry`, `toValidated`, and so on.

4.4.4 Error Handling

`Either` is typically used to implement fail-fast error handling. We sequence computations using `flatMap` as usual. If one computation fails, the remaining computations are not run:

```
for {
  a <- 1.asRight[String]
  b <- 0.asRight[String]
  c <- if(b == 0) "DIV0".asLeft[Int]
       else (a / b).asRight[String]
} yield c * 100
// res21: scala.util.Either[String,Int] = Left(DIV0)
```

When using `Either` for error handling, we need to determine what type we want to use to represent errors. We could use `Throwable` for this:

```
type Result[A] = Either[Throwable, A]
```

This gives us similar semantics to `scala.util.Try`. The problem, however, is that `Throwable` is an extremely broad type. We have (almost) no idea about what type of error occurred.

Another approach is to define an algebraic data type to represent errors that may occur in our program:

```
sealed trait LoginError extends Product with Serializable

final case class UserNotFound(username: String)
  extends LoginError

final case class PasswordIncorrect(username: String)
```

```
  extends LoginError

case object UnexpectedError extends LoginError

case class User(username: String, password: String)

type LoginResult = Either[LoginError, User]
```

This approach solves the problems we saw with `Throwable`. It gives us a fixed set of expected error types and a catch-all for anything else that we didn't expect. We also get the safety of exhaustivity checking on any pattern matching we do:

```
// Choose error-handling behaviour based on type:
def handleError(error: LoginError): Unit =
  error match {
    case UserNotFound(u) =>
      println(s"User not found: $u")

    case PasswordIncorrect(u) =>
      println(s"Password incorrect: $u")

    case UnexpectedError =>
      println(s"Unexpected error")
  }

val result1: LoginResult = User("dave", "passw0rd").asRight
// result1: LoginResult = Right(User(dave,passw0rd))

val result2: LoginResult = UserNotFound("dave").asLeft
// result2: LoginResult = Left(UserNotFound(dave))

result1.fold(handleError, println)
// User(dave,passw0rd)

result2.fold(handleError, println)
// User not found: dave
```

4.4.5 Exercise: What is Best?

Is the error handling strategy in the previous examples well suited for all purposes? What other features might we want from error handling?

See the solution

4.5 Aside: Error Handling and MonadError

Cats provides an additional type class called `MonadError` that abstracts over `Either`-like data types that are used for error handling. `MonadError` provides extra operations for raising and handling errors.

> *This Section is Optional!*
>
> You won't need to use `MonadError` unless you need to abstract over error handling monads. For example, you can use `MonadError` to abstract over `Future` and `Try`, or over `Either` and `EitherT` (which we will meet in Chapter 5).
>
> If you don't need this kind of abstraction right now, feel free to skip onwards to Section 4.6.

4.5.1 The MonadError Type Class

Here is a simplified version of the definition of `MonadError`:

```
package cats

trait MonadError[F[_], E] extends Monad[F] {
  // Lift an error into the `F` context:
  def raiseError[A](e: E): F[A]

  // Handle an error, potentially recovering from it:
  def handleError[A](fa: F[A])(f: E => A): F[A]

  // Test an instance of `F`,
```

```
  // failing if the predicate is not satisfied:
  def ensure[A](fa: F[A])(e: E)(f: A => Boolean): F[A]
}
```

`MonadError` is defined in terms of two type parameters:

- F is the type of the monad;
- E is the type of error contained within F.

To demonstrate how these parameters fit together, here's an example where we instantiate the type class for `Either`:

```
import cats.MonadError
import cats.instances.either._ // for MonadError

type ErrorOr[A] = Either[String, A]

val monadError = MonadError[ErrorOr, String]
```

> *ApplicativeError*
>
> In reality, `MonadError` extends another type class called `ApplicativeError`. However, we won't encounter `Applicatives` until Chapter 6. The semantics are the same for each type class so we can ignore this detail for now.

4.5.2 Raising and Handling Errors

The two most important methods of `MonadError` are `raiseError` and `handleError`. `raiseError` is like the `pure` method for `Monad` except that it creates an instance representing a failure:

```
val success = monadError.pure(42)
// success: ErrorOr[Int] = Right(42)
```

```
val failure = monadError.raiseError("Badness")
// failure: ErrorOr[Nothing] = Left(Badness)
```

`handleError` is the complement of `raiseError`. It allows us to consume an error and (possibly) turn it into a success, similar to the `recover` method of `Future`:

```
monadError.handleError(failure) {
  case "Badness" =>
    monadError.pure("It's ok")

  case other =>
    monadError.raiseError("It's not ok")
}
// res2: ErrorOr[ErrorOr[String]] = Right(Right(It's ok))
```

There is also a third useful method called `ensure` that implements `filter`-like behaviour. We test the value of a successful monad with a predicate and specify an error to raise if the predicate returns `false`:

```
import cats.syntax.either._ // for asRight

monadError.ensure(success)("Number too low!")(_ > 1000)
// res3: ErrorOr[Int] = Left(Number too low!)
```

Cats provides syntax for `raiseError` and `handleError` via `cats.syntax.applicativeError` and `ensure` via `cats.syntax.monadError`:

```
import cats.syntax.applicative._      // for pure
import cats.syntax.applicativeError._ // for raiseError etc
import cats.syntax.monadError._       // for ensure

val success = 42.pure[ErrorOr]
// success: ErrorOr[Int] = Right(42)

val failure = "Badness".raiseError[ErrorOr, Int]
// failure: ErrorOr[Int] = Left(Badness)
```

```
success.ensure("Number to low!")(_ > 1000)
// res4: Either[String,Int] = Left(Number to low!)
```

There are other useful variants of these methods. See the source of `cats.MonadError` and `cats.ApplicativeError` for more information.

4.5.3 Instances of MonadError

Cats provides instances of `MonadError` for numerous data types including `Either`, `Future`, and `Try`. The instance for `Either` is customisable to any error type, whereas the instances for `Future` and `Try` always represent errors as `Throwables`:

```
import scala.util.Try
import cats.instances.try_._ // for MonadError

val exn: Throwable =
  new RuntimeException("It's all gone wrong")

exn.raiseError[Try, Int]
// res6: scala.util.Try[Int] = Failure(java.lang.RuntimeException: It'
     s all gone wrong)
```

4.5.4 Exercise: Abstracting

4.6 The Eval Monad

`cats.Eval` is a monad that allows us to abstract over different *models of evaluation*. We typically hear of two such models: *eager* and *lazy*. `Eval` throws in a further distinction of whether or not a result is *memoized*.

4.6.1 Eager, Lazy, Memoized, Oh My!

What do these terms mean?

Eager computations happen immediately whereas *lazy* computations happen on access. *Memoized* computations are run once on first access, after which the results are cached.

For example, Scala `vals` are eager and memoized. We can see this using a computation with a visible side-effect. In the following example, the code to compute the value of x happens at the definition site rather than on access (eager). Accessing x recalls the stored value without re-running the code (memoized).

```
val x = {
  println("Computing X")
  math.random
}
// Computing X
// x: Double = 0.0657586956104027

x // first access
// res0: Double = 0.0657586956104027

x // second access
// res1: Double = 0.0657586956104027
```

By contrast, `def`s are lazy and not memoized. The code to compute y below is not run until we access it (lazy), and is re-run on every access (not memoized):

```
def y = {
  println("Computing Y")
  math.random
}
// y: Double

y // first access
// Computing Y
// res2: Double = 0.9184384488125138

y // second access
// Computing Y
// res3: Double = 0.20807113447602488
```

Last but not least, `lazy vals` are lazy and memoized. The code to compute

z below is not run until we access it for the first time (lazy). The result is then cached and re-used on subsequent accesses (memoized):

```
lazy val z = {
  println("Computing Z")
  math.random
}
// z: Double = <lazy>

z // first access
// Computing Z
// res4: Double = 0.1783014120350146

z // second access
// res5: Double = 0.1783014120350146
```

4.6.2 Eval's Models of Evaluation

`Eval` has three subtypes: `Now`, `Later`, and `Always`. We construct these with three constructor methods, which create instances of the three classes and return them typed as `Eval`:

```
import cats.Eval

val now = Eval.now(math.random + 1000)
// now: cats.Eval[Double] = Now(1000.885603643474)

val later = Eval.later(math.random + 2000)
// later: cats.Eval[Double] = cats.Later@679671c

val always = Eval.always(math.random + 3000)
// always: cats.Eval[Double] = cats.Always@396fe27e
```

We can extract the result of an `Eval` using its `value` method:

```
now.value
// res6: Double = 1000.885603643474

later.value
```

```
// res7: Double = 2000.1770874422618

always.value
// res8: Double = 3000.637554292833
```

Each type of `Eval` calculates its result using one of the evaluation models defined above. `Eval.now` captures a value *right now*. Its semantics are similar to a `val`—eager and memoized:

```
val x = Eval.now {
  println("Computing X")
  math.random
}
// Computing X
// x: cats.Eval[Double] = Now(0.08016953141772554)

x.value // first access
// res9: Double = 0.08016953141772554

x.value // second access
// res10: Double = 0.08016953141772554
```

`Eval.always` captures a lazy computation, similar to a `def`:

```
val y = Eval.always {
  println("Computing Y")
  math.random
}
// y: cats.Eval[Double] = cats.Always@471ed97c

y.value // first access
// Computing Y
// res11: Double = 0.9455576109936167

y.value // second access
// Computing Y
// res12: Double = 0.5996336572386713
```

Finally, `Eval.later` captures a lazy, memoized computation, similar to a `lazy val`:

```
val z = Eval.later {
  println("Computing Z")
  math.random
}
// z: cats.Eval[Double] = cats.Later@78b2f4e7

z.value // first access
// Computing Z
// res13: Double = 0.3353381222323517

z.value // second access
// res14: Double = 0.3353381222323517
```

The three behaviours are summarized below:

Scala	Cats	Properties
`val`	`Now`	eager, memoized
`lazy val`	`Later`	lazy, memoized
`def`	`Always`	lazy, not memoized

4.6.3 Eval as a Monad

Like all monads, `Eval`'s `map` and `flatMap` methods add computations to a chain. In this case, however, the chain is stored explicitly as a list of functions. The functions aren't run until we call `Eval`'s `value` method to request a result:

```
val greeting = Eval.
  always { println("Step 1"); "Hello" }.
  map { str => println("Step 2"); s"$str world" }
// greeting: cats.Eval[String] = cats.Eval$$anon$8@157f7b8c

greeting.value
// Step 1
// Step 2
// res15: String = Hello world
```

Note that, while the semantics of the originating `Eval` instances are main-

tained, mapping functions are always called lazily on demand (`def` semantics):

```
val ans = for {
  a <- Eval.now { println("Calculating A"); 40 }
  b <- Eval.always { println("Calculating B"); 2 }
} yield {
  println("Adding A and B")
  a + b
}
// Calculating A
// ans: cats.Eval[Int] = cats.Eval$$anon$8@37c1363d

ans.value // first access
// Calculating B
// Adding A and B
// res16: Int = 42

ans.value // second access
// Calculating B
// Adding A and B
// res17: Int = 42
```

`Eval` has a `memoize` method that allows us to memoize a chain of computations. The result of the chain up to the call to `memoize` is cached, whereas calculations after the call retain their original semantics:

```
val saying = Eval.
  always { println("Step 1"); "The cat" }.
  map { str => println("Step 2"); s"$str sat on" }.
  memoize.
  map { str => println("Step 3"); s"$str the mat" }
// saying: cats.Eval[String] = cats.Eval$$anon$8@2196a9a1

saying.value // first access
// Step 1
// Step 2
// Step 3
// res18: String = The cat sat on the mat

saying.value // second access
// Step 3
```

```
// res19: String = The cat sat on the mat
```

4.6.4 Trampolining and *Eval.defer*

One useful property of `Eval` is that its `map` and `flatMap` methods are *trampolined*. This means we can nest calls to `map` and `flatMap` arbitrarily without consuming stack frames. We call this property *"stack safety"*.

For example, consider this function for calculating factorials:

```
def factorial(n: BigInt): BigInt =
  if(n == 1) n else n * factorial(n - 1)
```

It is relatively easy to make this method stack overflow:

```
factorial(50000)
// java.lang.StackOverflowError
//   ...
```

We can rewrite the method using `Eval` to make it stack safe:

```
def factorial(n: BigInt): Eval[BigInt] =
  if(n == 1) {
    Eval.now(n)
  } else {
    factorial(n - 1).map(_ * n)
  }

factorial(50000).value
// java.lang.StackOverflowError
//   ...
```

Oops! That didn't work—our stack still blew up! This is because we're still making all the recursive calls to `factorial` before we start working with `Eval`'s `map` method. We can work around this using `Eval.defer`, which takes an existing instance of `Eval` and defers its evaluation. The `defer` method is trampolined like `map` and `flatMap`, so we can use it as a quick way to make an existing operation stack safe:

```
def factorial(n: BigInt): Eval[BigInt] =
  if(n == 1) {
    Eval.now(n)
  } else {
    Eval.defer(factorial(n - 1).map(_ * n))
  }

factorial(50000).value
// res20: BigInt =
     33473205095971448369154760940714864779127732238104548077301003219901680221443656
```

`Eval` is a useful tool to enforce stack safety when working on very large computations and data structures. However, we must bear in mind that trampolining is not free. It avoids consuming stack by creating a chain of function objects on the heap. There are still limits on how deeply we can nest computations, but they are bounded by the size of the heap rather than the stack.

4.6.5 Exercise: Safer Folding using Eval

The naive implementation of `foldRight` below is not stack safe. Make it so using `Eval`:

```
def foldRight[A, B](as: List[A], acc: B)(fn: (A, B) => B): B =
  as match {
    case head :: tail =>
      fn(head, foldRight(tail, acc)(fn))
    case Nil =>
      acc
  }
```

See the solution

4.7 The Writer Monad

`cats.data.Writer` is a monad that lets us carry a log along with a computation. We can use it to record messages, errors, or additional data about a computation, and extract the log alongside the final result.

One common use for `Writers` is recording sequences of steps in multi-threaded computations where standard imperative logging techniques can result in interleaved messages from different contexts. With `Writer` the log for the computation is tied to the result, so we can run concurrent computations without mixing logs.

> *Cats Data Types*
>
> `Writer` is the first data type we've seen from the `cats.data` package. This package provides instances of various type classes that produce useful semantics. Other examples from `cats.data` include the monad transformers that we will see in the next chapter, and the `Validated` type we will encounter in Chapter 6.

4.7.1 Creating and Unpacking Writers

A `Writer[W, A]` carries two values: a *log* of type `W` and a *result* of type `A`. We can create a `Writer` from values of each type as follows:

```
import cats.data.Writer
import cats.instances.vector._ // for Monoid

Writer(Vector(
  "It was the best of times",
  "it was the worst of times"
), 1859)
// res0: cats.data.WriterT[cats.Id,scala.collection.immutable.Vector[
    String],Int] = WriterT((Vector(It was the best of times, it was
    the worst of times),1859))
```

Notice that the type reported on the console is actually `WriterT[Id, Vector[String], Int]` instead of `Writer[Vector[String], Int]` as we might expect. In the spirit of code reuse, Cats implements `Writer` in terms of another type, `WriterT`. `WriterT` is an example of a new concept called a *monad transformer*, which we will cover in the next chapter.

Let's try to ignore this detail for now. `Writer` is a type alias for `WriterT`, so we can read types like `WriterT[Id, W, A]` as `Writer[W, A]`:

```
type Writer[W, A] = WriterT[Id, W, A]
```

For convenience, Cats provides a way of creating `Writers` specifying only the log or the result. If we only have a result we can use the standard `pure` syntax. To do this we must have a `Monoid[W]` in scope so Cats knows how to produce an empty log:

```
import cats.instances.vector._   // for Monoid
import cats.syntax.applicative._ // for pure

type Logged[A] = Writer[Vector[String], A]

123.pure[Logged]
// res2: Logged[Int] = WriterT((Vector(),123))
```

If we have a log and no result we can create a `Writer[Unit]` using the `tell` syntax from `cats.syntax.writer`:

```
import cats.syntax.writer._ // for tell

Vector("msg1", "msg2", "msg3").tell
// res3: cats.data.Writer[scala.collection.immutable.Vector[String],
     Unit] = WriterT((Vector(msg1, msg2, msg3),()))
```

If we have both a result and a log, we can either use `Writer.apply` or we can use the `writer` syntax from `cats.syntax.writer`:

```
import cats.syntax.writer._ // for writer

val a = Writer(Vector("msg1", "msg2", "msg3"), 123)
// a: cats.data.WriterT[cats.Id,scala.collection.immutable.Vector[
     String],Int] = WriterT((Vector(msg1, msg2, msg3),123))

val b = 123.writer(Vector("msg1", "msg2", "msg3"))
// b: cats.data.Writer[scala.collection.immutable.Vector[String],Int]
     = WriterT((Vector(msg1, msg2, msg3),123))
```

We can extract the result and log from a `Writer` using the `value` and `written` methods respectively:

```
val aResult: Int =
  a.value
// aResult: Int = 123

val aLog: Vector[String] =
  a.written
// aLog: Vector[String] = Vector(msg1, msg2, msg3)
```

We can extract both values at the same time using the `run` method:

```
val (log, result) = b.run
// log: scala.collection.immutable.Vector[String] = Vector(msg1, msg2,
      msg3)
// result: Int = 123
```

4.7.2 Composing and Transforming Writers

The log in a `Writer` is preserved when we `map` or `flatMap` over it. `flatMap` appends the logs from the source `Writer` and the result of the user's sequencing function. For this reason it's good practice to use a log type that has an efficient append and concatenate operations, such as a `Vector`:

```
val writer1 = for {
  a <- 10.pure[Logged]
  _ <- Vector("a", "b", "c").tell
  b <- 32.writer(Vector("x", "y", "z"))
} yield a + b
// writer1: cats.data.WriterT[cats.Id,Vector[String],Int] = WriterT((
     Vector(a, b, c, x, y, z),42))

writer1.run
// res4: cats.Id[(Vector[String], Int)] = (Vector(a, b, c, x, y, z)
     ,42)
```

In addition to transforming the result with `map` and `flatMap`, we can transform the log in a `Writer` with the `mapWritten` method:

```
val writer2 = writer1.mapWritten(_.map(_.toUpperCase))
// writer2: cats.data.WriterT[cats.Id,scala.collection.immutable.
     Vector[String],Int] = WriterT((Vector(A, B, C, X, Y, Z),42))

writer2.run
// res5: cats.Id[(scala.collection.immutable.Vector[String], Int)] = (
     Vector(A, B, C, X, Y, Z),42)
```

We can transform both log and result simultaneously using `bimap` or `mapBoth`. `bimap` takes two function parameters, one for the log and one for the result. `mapBoth` takes a single function that accepts two parameters:

```
val writer3 = writer1.bimap(
  log => log.map(_.toUpperCase),
  res => res * 100
)
// writer3: cats.data.WriterT[cats.Id,scala.collection.immutable.
     Vector[String],Int] = WriterT((Vector(A, B, C, X, Y, Z),4200))

writer3.run
// res6: cats.Id[(scala.collection.immutable.Vector[String], Int)] = (
     Vector(A, B, C, X, Y, Z),4200)

val writer4 = writer1.mapBoth { (log, res) =>
  val log2 = log.map(_ + "!")
  val res2 = res * 1000
  (log2, res2)
}
// writer4: cats.data.WriterT[cats.Id,scala.collection.immutable.
     Vector[String],Int] = WriterT((Vector(a!, b!, c!, x!, y!, z!)
     ,42000))

writer4.run
// res7: cats.Id[(scala.collection.immutable.Vector[String], Int)] = (
     Vector(a!, b!, c!, x!, y!, z!),42000)
```

Finally, we can clear the log with the `reset` method and swap log and result with the `swap` method:

```
val writer5 = writer1.reset
// writer5: cats.data.WriterT[cats.Id,Vector[String],Int] = WriterT((
    Vector(),42))

writer5.run
// res8: cats.Id[(Vector[String], Int)] = (Vector(),42)

val writer6 = writer1.swap
// writer6: cats.data.WriterT[cats.Id,Int,Vector[String]] = WriterT
    ((42,Vector(a, b, c, x, y, z)))

writer6.run
// res9: cats.Id[(Int, Vector[String])] = (42,Vector(a, b, c, x, y, z)
    )
```

4.7.3 Exercise: Show Your Working

`Writers` are useful for logging operations in multi-threaded environments. Let's confirm this by computing (and logging) some factorials.

The `factorial` function below computes a factorial and prints out the intermediate steps as it runs. The `slowly` helper function ensures this takes a while to run, even on the very small examples below:

```
def slowly[A](body: => A) =
  try body finally Thread.sleep(100)

def factorial(n: Int): Int = {
  val ans = slowly(if(n == 0) 1 else n * factorial(n - 1))
  println(s"fact $n $ans")
  ans
}
```

Here's the output—a sequence of monotonically increasing values:

```
factorial(5)
// fact 0 1
// fact 1 1
// fact 2 2
```

```
// fact 3 6
// fact 4 24
// fact 5 120
// res11: Int = 120
```

If we start several factorials in parallel, the log messages can become interleaved on standard out. This makes it difficult to see which messages come from which computation:

```
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._

Await.result(Future.sequence(Vector(
  Future(factorial(3)),
  Future(factorial(3))
)), 5.seconds)
// fact 0 1
// fact 0 1
// fact 1 1
// fact 1 1
// fact 2 2
// fact 2 2
// fact 3 6
// fact 3 6
// res14: scala.collection.immutable.Vector[Int] =
//   Vector(120, 120)
```

Rewrite `factorial` so it captures the log messages in a `Writer`. Demonstrate that this allows us to reliably separate the logs for concurrent computations.

See the solution

4.8 The Reader Monad

`cats.data.Reader` is a monad that allows us to sequence operations that depend on some input. Instances of `Reader` wrap up functions of one argument, providing us with useful methods for composing them.

One common use for `Readers` is dependency injection. If we have a number of operations that all depend on some external configuration, we can chain them together using a `Reader` to produce one large operation that accepts the configuration as a parameter and runs our program in the order specified.

4.8.1 Creating and Unpacking Readers

We can create a `Reader[A, B]` from a function `A => B` using the `Reader.apply` constructor:

```
import cats.data.Reader

case class Cat(name: String, favoriteFood: String)
// defined class Cat

val catName: Reader[Cat, String] =
  Reader(cat => cat.name)
// catName: cats.data.Reader[Cat,String] = Kleisli(<function1>)
```

We can extract the function again using the `Reader's` `run` method and call it using `apply` as usual:

```
catName.run(Cat("Garfield", "lasagne"))
// res0: cats.Id[String] = Garfield
```

So far so simple, but what advantage do `Readers` give us over the raw functions?

4.8.2 Composing Readers

The power of `Readers` comes from their `map` and `flatMap` methods, which represent different kinds of function composition. We typically create a set of `Readers` that accept the same type of configuration, combine them with `map` and `flatMap`, and then call `run` to inject the config at the end.

The `map` method simply extends the computation in the `Reader` by passing its result through a function:

```
val greetKitty: Reader[Cat, String] =
  catName.map(name => s"Hello ${name}")

greetKitty.run(Cat("Heathcliff", "junk food"))
// res1: cats.Id[String] = Hello Heathcliff
```

The `flatMap` method is more interesting. It allows us to combine readers that depend on the same input type. To illustrate this, let's extend our greeting example to also feed the cat:

```
val feedKitty: Reader[Cat, String] =
  Reader(cat => s"Have a nice bowl of ${cat.favoriteFood}")

val greetAndFeed: Reader[Cat, String] =
  for {
    greet <- greetKitty
    feed  <- feedKitty
  } yield s"$greet. $feed."

greetAndFeed(Cat("Garfield", "lasagne"))
// res3: cats.Id[String] = Hello Garfield. Have a nice bowl of lasagne
     .

greetAndFeed(Cat("Heathcliff", "junk food"))
// res4: cats.Id[String] = Hello Heathcliff. Have a nice bowl of junk
     food.
```

4.8.3 Exercise: Hacking on Readers

The classic use of `Reader`s is to build programs that accept a configuration as a parameter. Let's ground this with a complete example of a simple login system. Our configuration will consist of two databases: a list of valid users and a list of their passwords:

```
case class Db(
  usernames: Map[Int, String],
  passwords: Map[String, String]
)
```

Start by creating a type alias `DbReader` for a `Reader` that consumes a `Db` as input. This will make the rest of our code shorter.

See the solution

Now create methods that generate `DbReaders` to look up the username for an `Int` user ID, and look up the password for a `String` username. The type signatures should be as follows:

```
def findUsername(userId: Int): DbReader[Option[String]] =
  ???

def checkPassword(
      username: String,
      password: String): DbReader[Boolean] =
  ???
```

See the solution

Finally create a `checkLogin` method to check the password for a given user ID. The type signature should be as follows:

```
def checkLogin(
      userId: Int,
      password: String): DbReader[Boolean] =
  ???
```

See the solution

You should be able to use `checkLogin` as follows:

```
val users = Map(
  1 -> "dade",
  2 -> "kate",
  3 -> "margo"
)

val passwords = Map(
  "dade"  -> "zerocool",
  "kate"  -> "acidburn",
  "margo" -> "secret"
```

```
)

val db = Db(users, passwords)

checkLogin(1, "zerocool").run(db)
// res10: cats.Id[Boolean] = true

checkLogin(4, "davinci").run(db)
// res11: cats.Id[Boolean] = false
```

4.8.4 When to Use Readers?

`Readers` provide a tool for doing dependency injection. We write steps of our program as instances of `Reader`, chain them together with `map` and `flatMap`, and build a function that accepts the dependency as input.

There are many ways of implementing dependency injection in Scala, from simple techniques like methods with multiple parameter lists, through implicit parameters and type classes, to complex techniques like the cake pattern and DI frameworks.

`Readers` are most useful in situations where:

- we are constructing a batch program that can easily be represented by a function;
- we need to defer injection of a known parameter or set of parameters;
- we want to be able to test parts of the program in isolation.

By representing the steps of our program as `Readers` we can test them as easily as pure functions, plus we gain access to the `map` and `flatMap` combinators.

For more advanced problems where we have lots of dependencies, or where a program isn't easily represented as a pure function, other dependency injection techniques tend to be more appropriate.

> *Kleisli Arrows*
>
> You may have noticed from console output that `Reader` is implemented in terms of another type called `Kleisli`. *Kleisli arrows* provide a more general form of `Reader` that generalise over the type constructor of the result type. We will encounter Kleislis again in Chapter 5.

4.9 The State Monad

`cats.data.State` allows us to pass additional state around as part of a computation. We define `State` instances representing atomic state operations and thread them together using `map` and `flatMap`. In this way we can model mutable state in a purely functional way, without using mutation.

4.9.1 Creating and Unpacking State

Boiled down to their simplest form, instances of `State[S, A]` represent functions of type `S => (S, A)`. `S` is the type of the state and `A` is the type of the result.

```
import cats.data.State

val a = State[Int, String] { state =>
  (state, s"The state is $state")
}
// a: cats.data.State[Int,String] = cats.data.IndexedStateT@12c18313
```

In other words, an instance of `State` is a function that does two things:

- transforms an input state to an output state;
- computes a result.

We can "run" our monad by supplying an initial state. `State` provides three methods—`run`, `runS`, and `runA`—that return different combinations of state and result. Each method returns an instance of `Eval`, which `State` uses to

maintain stack safety. We call the `value` method as usual to extract the actual result:

```
// Get the state and the result:
val (state, result) = a.run(10).value
// state: Int = 10
// result: String = The state is 10

// Get the state, ignore the result:
val state = a.runS(10).value
// state: Int = 10

// Get the result, ignore the state:
val result = a.runA(10).value
// result: String = The state is 10
```

4.9.2 Composing and Transforming State

As we've seen with `Reader` and `Writer`, the power of the `State` monad comes from combining instances. The `map` and `flatMap` methods thread the state from one instance to another. Each individual instance represents an atomic state transformation, and their combination represents a complete sequence of changes:

```
val step1 = State[Int, String] { num =>
  val ans = num + 1
  (ans, s"Result of step1: $ans")
}
// step1: cats.data.State[Int,String] = cats.data.
    IndexedStateT@7c6e31c4

val step2 = State[Int, String] { num =>
  val ans = num * 2
  (ans, s"Result of step2: $ans")
}
// step2: cats.data.State[Int,String] = cats.data.
    IndexedStateT@7428b330

val both = for {
```

```
  a <- step1
  b <- step2
} yield (a, b)
// both: cats.data.IndexedStateT[cats.Eval,Int,Int,(String, String)] =
      cats.data.IndexedStateT@716401f3

val (state, result) = both.run(20).value
// state: Int = 42
// result: (String, String) = (Result of step1: 21,Result of step2:
     42)
```

As you can see, in this example the final state is the result of applying both transformations in sequence. State is threaded from step to step even though we don't interact with it in the for comprehension.

The general model for using the `State` monad is to represent each step of a computation as an instance and compose the steps using the standard monad operators. Cats provides several convenience constructors for creating primitive steps:

- `get` extracts the state as the result;
- `set` updates the state and returns unit as the result;
- `pure` ignores the state and returns a supplied result;
- `inspect` extracts the state via a transformation function;
- `modify` updates the state using an update function.

```
val getDemo = State.get[Int]
// getDemo: cats.data.State[Int,Int] = cats.data.
     IndexedStateT@4df6ba6a

getDemo.run(10).value
// res3: (Int, Int) = (10,10)

val setDemo = State.set[Int](30)
// setDemo: cats.data.State[Int,Unit] = cats.data.
     IndexedStateT@4620d0ef

setDemo.run(10).value
// res4: (Int, Unit) = (30,())
```

```
val pureDemo = State.pure[Int, String]("Result")
// pureDemo: cats.data.State[Int,String] = cats.data.
     IndexedStateT@988d7b2

pureDemo.run(10).value
// res5: (Int, String) = (10,Result)

val inspectDemo = State.inspect[Int, String](_ + "!")
// inspectDemo: cats.data.State[Int,String] = cats.data.
     IndexedStateT@13734a20

inspectDemo.run(10).value
// res6: (Int, String) = (10,10!)

val modifyDemo = State.modify[Int](_ + 1)
// modifyDemo: cats.data.State[Int,Unit] = cats.data.
     IndexedStateT@79493b6e

modifyDemo.run(10).value
// res7: (Int, Unit) = (11,())
```

We can assemble these building blocks using a for comprehension. We typically ignore the result of intermediate stages that only represent transformations on the state:

```
import State._

val program: State[Int, (Int, Int, Int)] = for {
  a <- get[Int]
  _ <- set[Int](a + 1)
  b <- get[Int]
  _ <- modify[Int](_ + 1)
  c <- inspect[Int, Int](_ * 1000)
} yield (a, b, c)
// program: cats.data.State[Int,(Int, Int, Int)] = cats.data.
     IndexedStateT@b8a0617

val (state, result) = program.run(1).value
// state: Int = 3
// result: (Int, Int, Int) = (1,2,3000)
```

4.9.3 Exercise: Post-Order Calculator

The `State` monad allows us to implement simple interpreters for complex expressions, passing the values of mutable registers along with the result. We can see a simple example of this by implementing a calculator for post-order integer arithmetic expressions.

In case you haven't heard of post-order expressions before (don't worry if you haven't), they are a mathematical notation where we write the operator *after* its operands. So, for example, instead of writing `1 + 2` we would write:

```
1 2 +
```

Although post-order expressions are difficult for humans to read, they are easy to evaluate in code. All we need to do is traverse the symbols from left to right, carrying a *stack* of operands with us as we go:

- when we see a number, we push it onto the stack;
- when we see an operator, we pop two operands off the stack, operate on them, and push the result in their place.

This allows us to evaluate complex expressions without using parentheses. For example, we can evaluate `(1 + 2) * 3)` as follows:

```
1 2 + 3 * // see 1, push onto stack
2 + 3 *   // see 2, push onto stack
+ 3 *     // see +, pop 1 and 2 off of stack,
          //        push (1 + 2) = 3 in their place
3 3 *     // see 3, push onto stack
3 *       // see 3, push onto stack
*         // see *, pop 3 and 3 off of stack,
          //        push (3 * 3) = 9 in their place
```

Let's write an interpreter for these expressions. We can parse each symbol into a `State` instance representing a transformation on the stack and an intermediate result. The `State` instances can be threaded together using `flatMap` to produce an interpreter for any sequence of symbols.

Start by writing a function `evalOne` that parses a single symbol into an instance of `State`. Use the code below as a template. Don't worry about error handling for now—if the stack is in the wrong configuration, it's OK to throw an exception.

```
import cats.data.State

type CalcState[A] = State[List[Int], A]

def evalOne(sym: String): CalcState[Int] = ???
```

If this seems difficult, think about the basic form of the `State` instances you're returning. Each instance represents a functional transformation from a stack to a pair of a stack and a result. You can ignore any wider context and focus on just that one step:

```
State[List[Int], Int] { oldStack =>
  val newStack = someTransformation(oldStack)
  val result   = someCalculation
  (newStack, result)
}
```

Feel free to write your `Stack` instances in this form or as sequences of the convenience constructors we saw above.

See the solution

`evalOne` allows us to evaluate single-symbol expressions as follows. We call `runA` supplying `Nil` as an initial stack, and call `value` to unpack the resulting `Eval` instance:

```
evalOne("42").runA(Nil).value
// res3: Int = 42
```

We can represent more complex programs using `evalOne`, `map`, and `flatMap`. Note that most of the work is happening on the stack, so we ignore the results of the intermediate steps for `evalOne("1")` and `evalOne("2")`:

```
val program = for {
  _   <- evalOne("1")
  _   <- evalOne("2")
  ans <- evalOne("+")
} yield ans
// program: cats.data.IndexedStateT[cats.Eval,List[Int],List[Int],Int]
      = cats.data.IndexedStateT@7983547b

program.runA(Nil).value
// res4: Int = 3
```

Generalise this example by writing an `evalAll` method that computes the result of a `List[String]`. Use `evalOne` to process each symbol, and thread the resulting `State` monads together using `flatMap`. Your function should have the following signature:

```
def evalAll(input: List[String]): CalcState[Int] =
  ???
```

See the solution

We can use `evalAll` to conveniently evaluate multi-stage expressions:

```
val program = evalAll(List("1", "2", "+", "3", "*"))
// program: CalcState[Int] = cats.data.IndexedStateT@f0a1bee

program.runA(Nil).value
// res6: Int = 9
```

Because `evalOne` and `evalAll` both return instances of `State`, we can thread these results together using `flatMap`. `evalOne` produces a simple stack transformation and `evalAll` produces a complex one, but they're both pure functions and we can use them in any order as many times as we like:

```
val program = for {
  _   <- evalAll(List("1", "2", "+"))
  _   <- evalAll(List("3", "4", "+"))
  ans <- evalOne("*")
} yield ans
```

```
// program: cats.data.IndexedStateT[cats.Eval,List[Int],List[Int],Int]
      = cats.data.IndexedStateT@18dd0fa3

program.runA(Nil).value
// res7: Int = 21
```

Complete the exercise by implementing an `evalInput` function that splits an input `String` into symbols, calls `evalAll`, and runs the result with an initial stack.

See the solution

4.10 Defining Custom Monads

We can define a `Monad` for a custom type by providing implementations of three methods: `flatMap`, `pure`, and a method we haven't seen yet called `tailRecM`. Here is an implementation of `Monad` for `Option` as an example:

```
import cats.Monad
import scala.annotation.tailrec

val optionMonad = new Monad[Option] {
  def flatMap[A, B](opt: Option[A])
      (fn: A => Option[B]): Option[B] =
    opt flatMap fn

  def pure[A](opt: A): Option[A] =
    Some(opt)

  @tailrec
  def tailRecM[A, B](a: A)
      (fn: A => Option[Either[A, B]]): Option[B] =
    fn(a) match {
      case None           => None
      case Some(Left(a1)) => tailRecM(a1)(fn)
      case Some(Right(b)) => Some(b)
    }
}
```

The `tailRecM` method is an optimisation used in Cats to limit the amount of stack space consumed by nested calls to `flatMap`. The technique comes from a 2015 paper by PureScript creator Phil Freeman. The method should recursively call itself until the result of `fn` returns a `Right`.

If we can make `tailRecM` tail-recursive, Cats is able to guarantee stack safety in recursive situations such as folding over large lists (see Section 7.1). If we can't make `tailRecM` tail-recursive, Cats cannot make these guarantees and extreme use cases may result in `StackOverflowErrors`. All of the built-in monads in Cats have tail-recursive implementations of `tailRecM`, although writing one for custom monads can be a challenge… as we shall see.

4.10.1 Exercise: Branching out Further with Monads

Let's write a `Monad` for our `Tree` data type from last chapter. Here's the type again:

```
sealed trait Tree[+A]

final case class Branch[A](left: Tree[A], right: Tree[A])
  extends Tree[A]

final case class Leaf[A](value: A) extends Tree[A]

def branch[A](left: Tree[A], right: Tree[A]): Tree[A] =
  Branch(left, right)

def leaf[A](value: A): Tree[A] =
  Leaf(value)
```

Verify that the code works on instances of `Branch` and `Leaf`, and that the `Monad` provides `Functor`-like behaviour for free.

Also verify that having a `Monad` in scope allows us to use for comprehensions, despite the fact that we haven't directly implemented `flatMap` or `map` on `Tree`.

Don't feel you have to make `tailRecM` tail-recursive. Doing so is quite difficult. We've included both tail-recursive and non-tail-recursive implementations in the solutions so you can check your work.

See the solution

4.11 Summary

In this chapter we've seen monads up-close. We saw that `flatMap` can be viewed as an operator for sequencing computations, dictating the order in which operations must happen. From this viewpoint, `Option` represents a computation that can fail without an error message, `Either` represents computations that can fail with a message, `List` represents multiple possible results, and `Future` represents a computation that may produce a value at some point in the future.

We've also seen some of the custom types and data structures that Cats provides, including `Id`, `Reader`, `Writer`, and `State`. These cover a wide range of use cases.

Finally, in the unlikely event that we have to implement a custom monad, we've learned about defining our own instance using `tailRecM`. `tailRecM` is an odd wrinkle that is a concession to building a functional programming library that is stack-safe by default. We don't need to understand `tailRecM` to understand monads, but having it around gives us benefits of which we can be grateful when writing monadic code.

Chapter 5

Monad Transformers

Monads are like burritos, which means that once you acquire a taste, you'll find yourself returning to them again and again. This is not without issues. As burritos can bloat the waist, monads can bloat the code base through nested for-comprehensions.

Imagine we are interacting with a database. We want to look up a user record. The user may or may not be present, so we return an `Option[User]`. Our communication with the database could fail for many reasons (network issues, authentication problems, and so on), so this result is wrapped up in an `Either`, giving us a final result of `Either[Error, Option[User]]`.

To use this value we must nest `flatMap` calls (or equivalently, for-comprehensions):

```
def lookupUserName(id: Long): Either[Error, Option[String]] =
  for {
    optUser <- lookupUser(id)
  } yield {
    for { user <- optUser } yield user.name
  }
```

This quickly becomes very tedious.

5.1 Exercise: Composing Monads

A question arises. Given two arbitrary monads, can we combine them in some way to make a single monad? That is, do monads *compose*? We can try to write the code but we soon hit problems:

```
import cats.Monad
import cats.syntax.applicative._ // for pure
import cats.syntax.flatMap._     // for flatMap
import scala.language.higherKinds

// Hypothetical example. This won't actually compile:
def compose[M1[_]: Monad, M2[_]: Monad] = {
  type Composed[A] = M1[M2[A]]

  new Monad[Composed] {
    def pure[A](a: A): Composed[A] =
      a.pure[M2].pure[M1]

    def flatMap[A, B](fa: Composed[A])
        (f: A => Composed[B]): Composed[B] =
      // Problem! How do we write flatMap?
      ???
  }
}
```

It is impossible to write a general definition of `flatMap` without knowing something about `M1` or `M2`. However, if we *do* know something about one or other monad, we can typically complete this code. For example, if we fix `M2` above to be `Option`, a definition of `flatMap` comes to light:

```
def flatMap[A, B](fa: Composed[A])
    (f: A => Composed[B]): Composed[B] =
  fa.flatMap(_.fold(None.pure[M1])(f))
```

Notice that the definition above makes use of `None`—an `Option`-specific concept that doesn't appear in the general `Monad` interface. We need this extra detail to combine `Option` with other monads. Similarly, there are things about other monads that help us write composed `flatMap` methods for them. This

is the idea behind monad transformers: Cats defines transformers for a variety of monads, each providing the extra knowledge we need to compose that monad with others. Let's look at some examples.

5.2 A Transformative Example

Cats provides transformers for many monads, each named with a T suffix: `EitherT` composes `Either` with other monads, `OptionT` composes `Option`, and so on.

Here's an example that uses `OptionT` to compose `List` and `Option`. We can use `OptionT[List, A]`, aliased to `ListOption[A]` for convenience, to transform a `List[Option[A]]` into a single monad:

```
import cats.data.OptionT

type ListOption[A] = OptionT[List, A]
```

Note how we build `ListOption` from the inside out: we pass `List`, the type of the outer monad, as a parameter to `OptionT`, the transformer for the inner monad.

We can create instances of `ListOption` using the `OptionT` constructor, or more conveniently using `pure`:

```
import cats.Monad
import cats.instances.list._     // for Monad
import cats.syntax.applicative._ // for pure

val result1: ListOption[Int] = OptionT(List(Option(10)))
// result1: ListOption[Int] = OptionT(List(Some(10)))

val result2: ListOption[Int] = 32.pure[ListOption]
// result2: ListOption[Int] = OptionT(List(Some(32)))
```

The `map` and `flatMap` methods combine the corresponding methods of `List` and `Option` into single operations:

```
result1.flatMap { (x: Int) =>
  result2.map { (y: Int) =>
    x + y
  }
}
// res1: cats.data.OptionT[List,Int] = OptionT(List(Some(42)))
```

This is the basis of all monad transformers. The combined `map` and `flatMap` methods allow us to use both component monads without having to recursively unpack and repack values at each stage in the computation. Now let's look at the API in more depth.

> *Complexity of Imports*
>
> The imports in the code samples above hint at how everything bolts together.
>
> We import `cats.syntax.applicative` to get the `pure` syntax. `pure` requires an implicit parameter of type `Applicative[ListOption]`. We haven't met `Applicatives` yet, but all `Monads` are also `Applicatives` so we can ignore that difference for now.
>
> In order to generate our `Applicative[ListOption]` we need instances of `Applicative` for `List` and `OptionT`. `OptionT` is a Cats data type so its instance is provided by its companion object. The instance for `List` comes from `cats.instances.list`.
>
> Notice we're not importing `cats.syntax.functor` or `cats.syntax.flatMap`. This is because `OptionT` is a concrete data type with its own explicit `map` and `flatMap` methods. It wouldn't cause problems if we imported the syntax—the compiler would ignore it in favour of the explicit methods.
>
> Remember that we're subjecting ourselves to these shenanigans because we're stubbornly refusing to use the universal Cats import, `cats.implicits`. If we did use that import, all of the instances and syntax we needed would be in scope and everything would just work.

5.3 Monad Transformers in Cats

Each monad transformer is a data type, defined in `cats.data`, that allows us to *wrap* stacks of monads to produce new monads. We use the monads we've built via the `Monad` type class. The main concepts we have to cover to understand monad transformers are:

- the available transformer classes;
- how to build stacks of monads using transformers;
- how to construct instances of a monad stack; and
- how to pull apart a stack to access the wrapped monads.

5.3.1 The Monad Transformer Classes

By convention, in Cats a monad `Foo` will have a transformer class called `FooT`. In fact, many monads in Cats are defined by combining a monad transformer with the `Id` monad. Concretely, some of the available instances are:

- `cats.data.OptionT` for `Option`;
- `cats.data.EitherT` for `Either`;
- `cats.data.ReaderT` for `Reader`;
- `cats.data.WriterT` for `Writer`;
- `cats.data.StateT` for `State`;
- `cats.data.IdT` for the `Id` monad.

> *Kleisli Arrows*
>
> In Section 4.8 we mentioned that the `Reader` monad was a specialisation of a more general concept called a "kleisli arrow", represented in Cats as `cats.data.Kleisli`.
>
> We can now reveal that `Kleisli` and `ReaderT` are, in fact, the same thing! `ReaderT` is actually a type alias for `Kleisli`. Hence, we were creating `Readers` last chapter and seeing `Kleislis` on the console.

5.3.2 Building Monad Stacks

All of these monad transformers follow the same convention. The transformer itself represents the *inner* monad in a stack, while the first type parameter specifies the outer monad. The remaining type parameters are the types we've used to form the corresponding monads.

For example, our `ListOption` type above is an alias for `OptionT[List, A]` but the result is effectively a `List[Option[A]]`. In other words, we build monad stacks from the inside out:

```
type ListOption[A] = OptionT[List, A]
```

Many monads and all transformers have at least two type parameters, so we often have to define type aliases for intermediate stages.

For example, suppose we want to wrap `Either` around `Option`. `Option` is the innermost type so we want to use the `OptionT` monad transformer. We need to use `Either` as the first type parameter. However, `Either` itself has two type parameters and monads only have one. We need a type alias to convert the type constructor to the correct shape:

```
// Alias Either to a type constructor with one parameter:
type ErrorOr[A] = Either[String, A]

// Build our final monad stack using OptionT:
type ErrorOrOption[A] = OptionT[ErrorOr, A]
```

`ErrorOrOption` is a monad, just like `ListOption`. We can use `pure`, `map`, and `flatMap` as usual to create and transform instances:

```
import cats.instances.either._ // for Monad

val a = 10.pure[ErrorOrOption]
// a: ErrorOrOption[Int] = OptionT(Right(Some(10)))

val b = 32.pure[ErrorOrOption]
// b: ErrorOrOption[Int] = OptionT(Right(Some(32)))
```

```
val c = a.flatMap(x => b.map(y => x + y))
// c: cats.data.OptionT[ErrorOr,Int] = OptionT(Right(Some(42)))
```

Things become even more confusing when we want to stack three or more monads.

For example, let's create a `Future` of an `Either` of `Option`. Once again we build this from the inside out with an `OptionT` of an `EitherT` of `Future`. However, we can't define this in one line because `EitherT` has three type parameters:

```
case class EitherT[F[_], E, A](stack: F[Either[E, A]]) {
  // etc...
}
```

The three type parameters are as follows:

- `F[_]` is the outer monad in the stack (`Either` is the inner);
- `E` is the error type for the `Either`;
- `A` is the result type for the `Either`.

This time we create an alias for `EitherT` that fixes `Future` and `Error` and allows `A` to vary:

```
import scala.concurrent.Future
import cats.data.{EitherT, OptionT}

type FutureEither[A] = EitherT[Future, String, A]

type FutureEitherOption[A] = OptionT[FutureEither, A]
```

Our mammoth stack now composes three monads and our `map` and `flatMap` methods cut through three layers of abstraction:

```
import cats.instances.future._ // for Monad
import scala.concurrent.Await
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
```

```
val futureEitherOr: FutureEitherOption[Int] =
  for {
    a <- 10.pure[FutureEitherOption]
    b <- 32.pure[FutureEitherOption]
  } yield a + b
```

Kind Projector

If you frequently find yourself defining multiple type aliases when building monad stacks, you may want to try the Kind Projector compiler plugin. Kind Projector enhances Scala's type syntax to make it easier to define partially applied type constructors. For example:

```
import cats.instances.option._ // for Monad
// import cats.instances.option._

123.pure[EitherT[Option, String, ?]]
// res7: cats.data.EitherT[Option,String,Int] = EitherT(Some(
    Right(123)))
```

Kind Projector can't simplify all type declarations down to a single line, but it can reduce the number of intermediate type definitions needed to keep our code readable.

5.3.3 Constructing and Unpacking Instances

As we saw above, we can create transformed monad stacks using the relevant monad transformer's `apply` method or the usual `pure` syntax[1]:

```
// Create using apply:
val errorStack1 = OptionT[ErrorOr, Int](Right(Some(10)))
// errorStack1: cats.data.OptionT[ErrorOr,Int] = OptionT(Right(Some
    (10)))
```

[1]Cats provides an instance of `MonadError` for `EitherT`, allowing us to create instances using `raiseError` as well as `pure`.

```
// Create using pure:
val errorStack2 = 32.pure[ErrorOrOption]
// errorStack2: ErrorOrOption[Int] = OptionT(Right(Some(32)))
```

Once we've finished with a monad transformer stack, we can unpack it using its `value` method. This returns the untransformed stack. We can then manipulate the individual monads in the usual way:

```
// Extracting the untransformed monad stack:
errorStack1.value
// res11: ErrorOr[Option[Int]] = Right(Some(10))

// Mapping over the Either in the stack:
errorStack2.value.map(_.getOrElse(-1))
// res13: scala.util.Either[String,Int] = Right(32)
```

Each call to `value` unpacks a single monad transformer. We may need more than one call to completely unpack a large stack. For example, to `Await` the `FutureEitherOption` stack above, we need to call `value` twice:

```
futureEitherOr
// res14: FutureEitherOption[Int] = OptionT(EitherT(Future(Success(
    Right(Some(42))))))

val intermediate = futureEitherOr.value
// intermediate: FutureEither[Option[Int]] = EitherT(Future(Success(
    Right(Some(42)))))

val stack = intermediate.value
// stack: scala.concurrent.Future[Either[String,Option[Int]]] = Future
    (Success(Right(Some(42))))

Await.result(stack, 1.second)
// res15: Either[String,Option[Int]] = Right(Some(42))
```

5.3.4 Default Instances

Many monads in Cats are defined using the corresponding transformer and the `Id` monad. This is reassuring as it confirms that the APIs for monads and

transformers are identical. `Reader`, `Writer`, and `State` are all defined in this way:

```
type Reader[E, A] = ReaderT[Id, E, A] // = Kleisli[Id, E, A]
type Writer[W, A] = WriterT[Id, W, A]
type State[S, A]  = StateT[Id, S, A]
```

In other cases monad transformers are defined separately to their corresponding monads. In these cases, the methods of the transformer tend to mirror the methods on the monad. For example, `OptionT` defines `getOrElse`, and `EitherT` defines `fold`, `bimap`, `swap`, and other useful methods.

5.3.5 Usage Patterns

Widespread use of monad transformers is sometimes difficult because they fuse monads together in predefined ways. Without careful thought, we can end up having to unpack and repack monads in different configurations to operate on them in different contexts.

We can cope with this in multiple ways. One approach involves creating a single "super stack" and sticking to it throughout our code base. This works if the code is simple and largely uniform in nature. For example, in a web application, we could decide that all request handlers are asynchronous and all can fail with the same set of HTTP error codes. We could design a custom ADT representing the errors and use a fusion `Future` and `Either` everywhere in our code:

```
sealed abstract class HttpError
final case class NotFound(item: String) extends HttpError
final case class BadRequest(msg: String) extends HttpError
// etc...

type FutureEither[A] = EitherT[Future, HttpError, A]
```

The "super stack" approach starts to fail in larger, more heterogeneous code bases where different stacks make sense in different contexts. Another design pattern that makes more sense in these contexts uses monad transformers

as local "glue code". We expose untransformed stacks at module boundaries, transform them to operate on them locally, and untransform them before passing them on. This allows each module of code to make its own decisions about which transformers to use:

```
import cats.data.Writer

type Logged[A] = Writer[List[String], A]

// Methods generally return untransformed stacks:
def parseNumber(str: String): Logged[Option[Int]] =
  util.Try(str.toInt).toOption match {
    case Some(num) => Writer(List(s"Read $str"), Some(num))
    case None      => Writer(List(s"Failed on $str"), None)
  }

// Consumers use monad transformers locally to simplify composition:
def addAll(a: String, b: String, c: String): Logged[Option[Int]] = {
  import cats.data.OptionT

  val result = for {
    a <- OptionT(parseNumber(a))
    b <- OptionT(parseNumber(b))
    c <- OptionT(parseNumber(c))
  } yield a + b + c

  result.value
}

// This approach doesn't force OptionT on other users' code:
val result1 = addAll("1", "2", "3")
// result1: Logged[Option[Int]] = WriterT((List(Read 1, Read 2, Read
    3),Some(6)))

val result2 = addAll("1", "a", "3")
// result2: Logged[Option[Int]] = WriterT((List(Read 1, Failed on a),
    None))
```

Unfortunately, there aren't one-size-fits-all approaches to working with monad transformers. The best approach for you may depend on a lot of factors: the size and experience of your team, the complexity of your code base, and so on. You may need to experiment and gather feedback from colleagues

to determine whether monad transformers are a good fit.

5.4 Exercise: Monads: Transform and Roll Out

The Autobots, well-known robots in disguise, frequently send messages during battle requesting the power levels of their team mates. This helps them coordinate strategies and launch devastating attacks. The message sending method looks like this:

```
def getPowerLevel(autobot: String): Response[Int] =
  ???
```

Transmissions take time in Earth's viscous atmosphere, and messages are occasionally lost due to satellite malfunction or sabotage by pesky Decepticons[2]. `Responses` are therefore represented as a stack of monads:

```
type Response[A] = Future[Either[String, A]]
// defined type alias Response
```

Optimus Prime is getting tired of the nested for comprehensions in his neural matrix. Help him by rewriting `Response` using a monad transformer.

See the solution

Now test the code by implementing `getPowerLevel` to retrieve data from a set of imaginary allies. Here's the data we'll use:

```
val powerLevels = Map(
  "Jazz"      -> 6,
  "Bumblebee" -> 8,
  "Hot Rod"   -> 10
)
```

If an Autobot isn't in the `powerLevels` map, return an error message reporting that they were unreachable. Include the `name` in the message for good effect.

[2]It is a well known fact that Autobot neural nets are implemented in Scala. Decepticon brains are, of course, dynamically typed.

See the solution

Two autobots can perform a special move if their combined power level is greater than 15. Write a second method, `canSpecialMove`, that accepts the names of two allies and checks whether a special move is possible. If either ally is unavailable, fail with an appropriate error message:

```
def canSpecialMove(ally1: String, ally2: String): Response[Boolean] =
  ???
```

See the solution

Finally, write a method `tacticalReport` that takes two ally names and prints a message saying whether they can perform a special move:

```
def tacticalReport(ally1: String, ally2: String): String =
  ???
```

See the solution

You should be able to use `report` as follows:

```
tacticalReport("Jazz", "Bumblebee")
// res28: String = Jazz and Bumblebee need a recharge.

tacticalReport("Bumblebee", "Hot Rod")
// res29: String = Bumblebee and Hot Rod are ready to roll out!

tacticalReport("Jazz", "Ironhide")
// res30: String = Comms error: Ironhide unreachable
```

5.5 Summary

In this chapter we introduced monad transformers, which eliminate the need for nested for comprehensions and pattern matching when working with "stacks" of nested monads.

Each monad transformer, such as `FutureT`, `OptionT` or `EitherT`, provides the code needed to merge its related monad with other monads. The transformer is a data structure that wraps a monad stack, equipping it with `map` and `flatMap` methods that unpack and repack the whole stack.

The type signatures of monad transformers are written from the inside out, so an `EitherT[Option, String, A]` is a wrapper for an `Option[Either[String, A]]`. It is often useful to use type aliases when writing transformer types for deeply nested monads.

With this look at monad transformers, we have now covered everything we need to know about monads and the sequencing of computations using `flatMap`. In the next chapter we will switch tack and discuss two new type classes, `Semigroupal` and `Applicative`, that support new kinds of operation such as `zipping` independent values within a context.

Chapter 6

Semigroupal and Applicative

In previous chapters we saw how functors and monads let us sequence operations using `map` and `flatMap`. While functors and monads are both immensely useful abstractions, there are certain types of program flow that they cannot represent.

One such example is form validation. When we validate a form we want to return *all* the errors to the user, not stop on the first error we encounter. If we model this with a monad like `Either`, we fail fast and lose errors. For example, the code below fails on the first call to `parseInt` and doesn't go any further:

```
import cats.syntax.either._ // for catchOnly

def parseInt(str: String): Either[String, Int] =
  Either.catchOnly[NumberFormatException](str.toInt).
    leftMap(_ => s"Couldn't read $str")

for {
  a <- parseInt("a")
  b <- parseInt("b")
  c <- parseInt("c")
} yield (a + b + c)
// res1: scala.util.Either[String,Int] = Left(Couldn't read a)
```

Another example is the concurrent evaluation of `Futures`. If we have several

long-running independent tasks, it makes sense to execute them concurrently. However, monadic comprehension only allows us to run them in sequence. `map` and `flatMap` aren't quite capable of capturing what we want because they make the assumption that each computation is *dependent* on the previous one:

```
// context2 is dependent on value1:
context1.flatMap(value1 => context2)
```

The calls to `parseInt` and `Future.apply` above are *independent* of one another, but `map` and `flatMap` can't exploit this. We need a weaker construct—one that doesn't guarantee sequencing—to achieve the result we want. In this chapter we will look at two type classes that support this pattern:

- `Semigroupal` encompasses the notion of composing pairs of contexts. Cats provides a `cats.syntax.apply` module that makes use of `Semigroupal` and `Functor` to allow users to sequence functions with multiple arguments.
- `Applicative` extends `Semigroupal` and `Functor`. It provides a way of applying functions to parameters within a context. `Applicative` is the source of the `pure` method we introduced in Chapter 4.

Applicatives are often formulated in terms of function application, instead of the semigroupal formulation that is emphasised in Cats. This alternative formulation provides a link to other libraries and languages such as Scalaz and Haskell. We'll take a look at different formulations of Applicative, as well as the relationships between `Semigroupal`, `Functor`, `Applicative`, and `Monad`, towards the end of the chapter.

6.1 Semigroupal

`cats.Semigroupal` is a type class that allows us to combine contexts[1]. If

[1] It is also the winner of Underscore's 2017 award for the most difficult functional programming term to work into a coherent English sentence.

we have two objects of type `F[A]` and `F[B]`, a `Semigroupal[F]` allows us to combine them to form an `F[(A, B)]`. Its definition in Cats is:

```
trait Semigroupal[F[_]] {
  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)]
}
```

As we discussed at the beginning of this chapter, the parameters `fa` and `fb` are independent of one another: we can compute them in either order before passing them to `product`. This is in contrast to `flatMap`, which imposes a strict order on its parameters. This gives us more freedom when defining instances of `Semigroupal` than we get when defining `Monads`.

6.1.1 Joining Two Contexts

While `Semigroup` allows us to join values, `Semigroupal` allows us to join contexts. Let's join some `Options` as an example:

```
import cats.Semigroupal
import cats.instances.option._ // for Semigroupal

Semigroupal[Option].product(Some(123), Some("abc"))
// res0: Option[(Int, String)] = Some((123,abc))
```

If both parameters are instances of `Some`, we end up with a tuple of the values within. If either parameter evaluates to `None`, the entire result is `None`:

```
Semigroupal[Option].product(None, Some("abc"))
// res1: Option[(Nothing, String)] = None

Semigroupal[Option].product(Some(123), None)
// res2: Option[(Int, Nothing)] = None
```

6.1.2 Joining Three or More Contexts

The companion object for `Semigroupal` defines a set of methods on top of `product`. For example, the methods `tuple2` through `tuple22` generalise

`product` to different arities:

```
import cats.instances.option._ // for Semigroupal

Semigroupal.tuple3(Option(1), Option(2), Option(3))
// res3: Option[(Int, Int, Int)] = Some((1,2,3))

Semigroupal.tuple3(Option(1), Option(2), Option.empty[Int])
// res4: Option[(Int, Int, Int)] = None
```

The methods `map2` through `map22` apply a user-specified function to the values inside 2 to 22 contexts:

```
Semigroupal.map3(Option(1), Option(2), Option(3))(_ + _ + _)
// res5: Option[Int] = Some(6)

Semigroupal.map2(Option(1), Option.empty[Int])(_ + _)
// res6: Option[Int] = None
```

There are also methods `contramap2` through `contramap22` and `imap2` through `imap22`, that require instances of `Contravariant` and `Invariant` respectively.

6.2 Apply Syntax

Cats provides a convenient *apply syntax* that provides a shorthand for the methods described above. We import the syntax from `cats.syntax.apply`. Here's an example:

```
import cats.instances.option._ // for Semigroupal
import cats.syntax.apply._     // for tupled and mapN
```

The `tupled` method is implicitly added to the tuple of `Options`. It uses the `Semigroupal` for `Option` to zip the values inside the `Options`, creating a single `Option` of a tuple:

```
(Option(123), Option("abc")).tupled
// res7: Option[(Int, String)] = Some((123,abc))
```

We can use the same trick on tuples of up to 22 values. Cats defines a separate `tupled` method for each arity:

```
(Option(123), Option("abc"), Option(true)).tupled
// res8: Option[(Int, String, Boolean)] = Some((123,abc,true))
```

In addition to `tupled`, Cats' apply syntax provides a method called `mapN` that accepts an implicit `Functor` and a function of the correct arity to combine the values:

```
case class Cat(name: String, born: Int, color: String)

(
  Option("Garfield"),
  Option(1978),
  Option("Orange & black")
).mapN(Cat.apply)
// res9: Option[Cat] = Some(Cat(Garfield,1978,Orange & black))
```

Internally `mapN` uses the `Semigroupal` to extract the values from the `Option` and the `Functor` to apply the values to the function.

It's nice to see that this syntax is type checked. If we supply a function that accepts the wrong number or types of parameters, we get a compile error:

```
val add: (Int, Int) => Int = (a, b) => a + b
// add: (Int, Int) => Int = <function2>

(Option(1), Option(2), Option(3)).mapN(add)
// <console>:27: error: type mismatch;
//  found   : (Int, Int) => Int
//  required: (Int, Int, Int) => ?
//        (Option(1), Option(2), Option(3)).mapN(add)
//                                               ^

(Option("cats"), Option(true)).mapN(add)
// <console>:27: error: type mismatch;
```

```
//  found   : (Int, Int) => Int
//  required: (String, Boolean) => ?
//        (Option("cats"), Option(true)).mapN(add)
//                                            ^
```

6.2.1 Fancy Functors and Apply Syntax

Apply syntax also has `contramapN` and `imapN` methods that accept Contravariant and Invariant functors. For example, we can combine `Monoids` using `Invariant`. Here's an example:

```
import cats.Monoid
import cats.instances.int._        // for Monoid
import cats.instances.invariant._  // for Semigroupal
import cats.instances.list._       // for Monoid
import cats.instances.string._     // for Monoid
import cats.syntax.apply._         // for imapN

case class Cat(
  name: String,
  yearOfBirth: Int,
  favoriteFoods: List[String]
)

val tupleToCat: (String, Int, List[String]) => Cat =
  Cat.apply _

val catToTuple: Cat => (String, Int, List[String]) =
  cat => (cat.name, cat.yearOfBirth, cat.favoriteFoods)

implicit val catMonoid: Monoid[Cat] = (
  Monoid[String],
  Monoid[Int],
  Monoid[List[String]]
).imapN(tupleToCat)(catToTuple)
```

Our `Monoid` allows us to create "empty" `Cats`, and add `Cats` together using the syntax from Chapter 2:

```
import cats.syntax.semigroup._ // for |+|

val garfield   = Cat("Garfield", 1978, List("Lasagne"))
val heathcliff = Cat("Heathcliff", 1988, List("Junk Food"))

garfield |+| heathcliff
// res17: Cat = Cat(GarfieldHeathcliff,3966,List(Lasagne, Junk Food))
```

6.3 Semigroupal Applied to Different Types

`Semigroupal` doesn't always provide the behaviour we expect, particularly for types that also have instances of `Monad`. We have seen the behaviour of the `Semigroupal` for `Option`. Let's look at some examples for other types.

Future

The semantics for `Future` provide parallel as opposed to sequential execution:

```
import cats.Semigroupal
import cats.instances.future._ // for Semigroupal
import scala.concurrent._
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.language.higherKinds

val futurePair = Semigroupal[Future].
  product(Future("Hello"), Future(123))

Await.result(futurePair, 1.second)
// res1: (String, Int) = (Hello,123)
```

The two `Futures` start executing the moment we create them, so they are already calculating results by the time we call `product`. We can use apply syntax to zip fixed numbers of `Futures`:

```
import cats.syntax.apply._ // for mapN

case class Cat(
  name: String,
```

```
  yearOfBirth: Int,
  favoriteFoods: List[String]
)

val futureCat = (
  Future("Garfield"),
  Future(1978),
  Future(List("Lasagne"))
).mapN(Cat.apply)

Await.result(futureCat, 1.second)
// res4: Cat = Cat(Garfield,1978,List(Lasagne))
```

List

Combining `Lists` with `Semigroupal` produces some potentially unexpected results. We might expect code like the following to *zip* the lists, but we actually get the cartesian product of their elements:

```
import cats.Semigroupal
import cats.instances.list._ // for Semigroupal

Semigroupal[List].product(List(1, 2), List(3, 4))
// res5: List[(Int, Int)] = List((1,3), (1,4), (2,3), (2,4))
```

This is perhaps surprising. Zipping lists tends to be a more common operation. We'll see why we get this behaviour in a moment.

Either

We opened this chapter with a discussion of fail-fast versus accumulating error-handling. We might expect `product` applied to `Either` to accumulate errors instead of fail fast. Again, perhaps surprisingly, we find that `product` implements the same fail-fast behaviour as `flatMap`:

```
import cats.instances.either._ // for Semigroupal

type ErrorOr[A] = Either[Vector[String], A]

Semigroupal[ErrorOr].product(
  Left(Vector("Error 1")),
```

```
  Left(Vector("Error 2"))
)
// res7: ErrorOr[(Nothing, Nothing)] = Left(Vector(Error 1))
```

In this example `product` sees the first failure and stops, even though it is possible to examine the second parameter and see that it is also a failure.

6.3.1 Semigroupal Applied to Monads

The reason for the surprising results for `List` and `Either` is that they are both monads. To ensure consistent semantics, Cats' `Monad` (which extends `Semigroupal`) provides a standard definition of `product` in terms of `map` and `flatMap`. This gives what we might think of as unexpected and less useful behaviour for a number of data types. The consistency of semantics is important for higher level abstractions, but we don't know about those yet.

Even our results for `Future` are a trick of the light. `flatMap` provides sequential ordering, so `product` provides the same. The parallel execution we observe occurs because our constituent `Futures` start running before we call `product`. This is equivalent to the classic create-then-flatMap pattern:

```
val a = Future("Future 1")
val b = Future("Future 2")

for {
  x <- a
  y <- b
} yield (x, y)
```

So why bother with `Semigroupal` at all? The answer is that we can create useful data types that have instances of `Semigroupal` (and `Applicative`) but not `Monad`. This frees us to implement `product` in different ways. We'll examine this further in a moment when we look at an alternative data type for error handling.

6.3.1.1 Exercise: The Product of Monads

Implement `product` in terms of `flatMap`:

```
import cats.Monad

def product[M[_]: Monad, A, B](x: M[A], y: M[B]): M[(A, B)] =
  ???
```

See the solution

6.4 Validated

By now we are familiar with the fail-fast error handling behaviour of `Either`. Furthermore, because `Either` is a monad, we know that the semantics of `product` are the same as those for `flatMap`. In fact, it is impossible for us to design a monadic data type that implements error accumulating semantics without breaking the consistency of these two methods.

Fortunately, Cats provides a data type called `Validated` that has an instance of `Semigroupal` but *no* instance of `Monad`. The implementation of `product` is therefore free to accumulate errors:

```
import cats.Semigroupal
import cats.data.Validated
import cats.instances.list._ // for Monoid

type AllErrorsOr[A] = Validated[List[String], A]

Semigroupal[AllErrorsOr].product(
  Validated.invalid(List("Error 1")),
  Validated.invalid(List("Error 2"))
)
// res1: AllErrorsOr[(Nothing, Nothing)] = Invalid(List(Error 1, Error
      2))
```

`Validated` complements `Either` nicely. Between the two we have support for both of the common types of error handling: fail-fast and accumulating.

6.4.1 Creating Instances of Validated

`Validated` has two subtypes, `Validated.Valid` and `Validated.Invalid`, that correspond loosely to `Right` and `Left`. There are a lot of ways to create instances of these types. We can create them directly using their `apply` methods:

```
val v = Validated.Valid(123)
// v: cats.data.Validated.Valid[Int] = Valid(123)

val i = Validated.Invalid(List("Badness"))
// i: cats.data.Validated.Invalid[List[String]] = Invalid(List(Badness
    ))
```

However, it is often easier to use the `valid` and `invalid` smart constructors, which widen the return type to `Validated`:

```
val v = Validated.valid[List[String], Int](123)
// v: cats.data.Validated[List[String],Int] = Valid(123)

val i = Validated.invalid[List[String], Int](List("Badness"))
// i: cats.data.Validated[List[String],Int] = Invalid(List(Badness))
```

As a third option we can import the `valid` and `invalid` extension methods from `cats.syntax.validated`:

```
import cats.syntax.validated._ // for valid and invalid

123.valid[List[String]]
// res2: cats.data.Validated[List[String],Int] = Valid(123)

List("Badness").invalid[Int]
// res3: cats.data.Validated[List[String],Int] = Invalid(List(Badness)
    )
```

As a fourth option we can use `pure` and `raiseError` from `cats.syntax.applicative` and `cats.syntax.applicativeError` respectively:

```
import cats.syntax.applicative._      // for pure
import cats.syntax.applicativeError._ // for raiseError

type ErrorsOr[A] = Validated[List[String], A]

123.pure[ErrorsOr]
// res5: ErrorsOr[Int] = Valid(123)

List("Badness").raiseError[ErrorsOr, Int]
// res6: ErrorsOr[Int] = Invalid(List(Badness))
```

Finally, there are helper methods to create instances of `Validated` from different sources. We can create them from `Exceptions`, as well as instances of `Try`, `Either`, and `Option`:

```
Validated.catchOnly[NumberFormatException]("foo".toInt)
// res7: cats.data.Validated[NumberFormatException,Int] = Invalid(java
    .lang.NumberFormatException: For input string: "foo")

Validated.catchNonFatal(sys.error("Badness"))
// res8: cats.data.Validated[Throwable,Nothing] = Invalid(java.lang.
    RuntimeException: Badness)

Validated.fromTry(scala.util.Try("foo".toInt))
// res9: cats.data.Validated[Throwable,Int] = Invalid(java.lang.
    NumberFormatException: For input string: "foo")

Validated.fromEither[String, Int](Left("Badness"))
// res10: cats.data.Validated[String,Int] = Invalid(Badness)

Validated.fromOption[String, Int](None, "Badness")
// res11: cats.data.Validated[String,Int] = Invalid(Badness)
```

6.4.2 Combining Instances of Validated

We can combine instances of `Validated` using any of the methods or syntax described for `Semigroupal` above.

All of these techniques require an instance of `Semigroupal` to be in scope. As with `Either`, we need to fix the error type to create a type constructor with the correct number of parameters for `Semigroupal`:

```
type AllErrorsOr[A] = Validated[String, A]
```

`Validated` accumulates errors using a `Semigroup`, so we need one of those in scope to summon the `Semigroupal`. If no `Semigroup` is visible at the call site, we get an annoyingly unhelpful compilation error:

```
Semigroupal[AllErrorsOr]
// <console>:28: error: could not find implicit value for parameter
     instance: cats.Semigroupal[AllErrorsOr]
//        Semigroupal[AllErrorsOr]
//                    ^
```

Once we import a `Semigroup` for the error type, everything works as expected:

```
import cats.instances.string._ // for Semigroup

Semigroupal[AllErrorsOr]
// res13: cats.Semigroupal[AllErrorsOr] = cats.data.
     ValidatedInstances$$anon$1@5e3850f5
```

As long as the compiler has all the implicits in scope to summon a `Semigroupal` of the correct type, we can use apply syntax or any of the other `Semigroupal` methods to accumulate errors as we like:

```
import cats.syntax.apply._ // for tupled

(
  "Error 1".invalid[Int],
  "Error 2".invalid[Int]
).tupled
// res14: cats.data.Validated[String,(Int, Int)] = Invalid(Error 1
     Error 2)
```

As you can see, `String` isn't an ideal type for accumulating errors. We commonly use `Lists` or `Vectors` instead:

```
import cats.instances.vector._ // for Semigroupal

(
  Vector(404).invalid[Int],
  Vector(500).invalid[Int]
).tupled
// res15: cats.data.Validated[scala.collection.immutable.Vector[Int],(
    Int, Int)] = Invalid(Vector(404, 500))
```

The `cats.data` package also provides the `NonEmptyList` and `NonEmptyVector` types that prevent us failing without at least one error:

```
import cats.data.NonEmptyVector

(
  NonEmptyVector.of("Error 1").invalid[Int],
  NonEmptyVector.of("Error 2").invalid[Int]
).tupled
// res16: cats.data.Validated[cats.data.NonEmptyVector[String],(Int,
    Int)] = Invalid(NonEmptyVector(Error 1, Error 2))
```

6.4.3 Methods of Validated

`Validated` comes with a suite of methods that closely resemble those available for `Either`, including the methods from `cats.syntax.either`. We can use `map`, `leftMap`, and `bimap` to transform the values inside the valid and invalid sides:

```
123.valid.map(_ * 100)
// res17: cats.data.Validated[Nothing,Int] = Valid(12300)

"?".invalid.leftMap(_.toString)
// res18: cats.data.Validated[String,Nothing] = Invalid(?)

123.valid[String].bimap(_ + "!", _ * 100)
// res19: cats.data.Validated[String,Int] = Valid(12300)

"?".invalid[Int].bimap(_ + "!", _ * 100)
// res20: cats.data.Validated[String,Int] = Invalid(?!)
```

We can't `flatMap` because `Validated` isn't a monad. However, Cats does provide a stand-in for `flatMap` called `andThen`. The type signature of `andThen` is identical to that of `flatMap`, but it has a different name because it is not a lawful implementation with respect to the monad laws:

```
32.valid.andThen { a =>
  10.valid.map { b =>
    a + b
  }
}
// res21: cats.data.Validated[Nothing,Int] = Valid(42)
```

If we want to do more than just `flatMap`, we can convert back and forth between `Validated` and `Either` using the `toEither` and `toValidated` methods. Note that `toValidated` comes from [`cats.syntax.either`]:

```
import cats.syntax.either._ // for toValidated
// import cats.syntax.either._

"Badness".invalid[Int]
// res22: cats.data.Validated[String,Int] = Invalid(Badness)

"Badness".invalid[Int].toEither
// res23: Either[String,Int] = Left(Badness)

"Badness".invalid[Int].toEither.toValidated
// res24: cats.data.Validated[String,Int] = Invalid(Badness)
```

As with `Either`, we can use the `ensure` method to fail with a specified error if a predicate does not hold:

```
// 123.valid[String].ensure("Negative!")(_ > 0)
```

Finally, we can call `getOrElse` or `fold` to extract values from the `Valid` and `Invalid` cases:

```
"fail".invalid[Int].getOrElse(0)
// res26: Int = 0

"fail".invalid[Int].fold(_ + "!!!", _.toString)
// res27: String = fail!!!
```

6.4.4 Exercise: Form Validation

Let's get used to `Validated` by implementing a simple HTML registration form. We receive request data from the client in a `Map[String, String]` and we want to parse it to create a `User` object:

```
case class User(name: String, age: Int)
```

Our goal is to implement code that parses the incoming data enforcing the following rules:

- the name and age must be specified;
- the name must not be blank;
- the age must be a valid non-negative integer.

If all the rules pass our parser we should return a `User`. If any rules fail we should return a `List` of the error messages.

To implement this example we'll need to combine rules in sequence and in parallel. We'll use `Either` to combine computations in sequence using fail-fast semantics, and `Validated` to combine them in parallel using accumulating semantics.

Let's start with some sequential combination. We'll define two methods to read the `"name"` and `"age"` fields:

- `readName` will take a `Map[String, String]` parameter, extract the `"name"` field, check the relevant validation rules, and return an `Either[List[String], String]`.

- `readAge` will take a `Map[String, String]` parameter, extract the `"age"` field, check the relevant validation rules, and return an `Either[List[String], Int]`.

We'll build these methods up from smaller building blocks. Start by defining a method `getValue` that reads a `String` from the `Map` given a field name.

See the solution

Next define a method `parseInt` that consumes a `String` and parses it as an `Int`.

See the solution

Next implement the validation checks: `nonBlank` to check `Strings`, and `nonNegative` to check `Ints`.

See the solution

Now combine `getValue`, `parseInt`, `nonBlank` and `nonNegative` to create `readName` and `readAge`:

See the solution

Finally, use a `Semigroupal` to combine the results of `readName` and `readAge` to produce a `User`. Make sure you switch from `Either` to `Validated` to accumulate errors.

See the solution

6.5 Apply and Applicative

Semigroupals aren't mentioned frequently in the wider functional programming literature. They provide a subset of the functionality of a related type class called an *applicative functor* ("applicative" for short).

`Semigroupal` and `Applicative` effectively provide alternative encodings of the same notion of joining contexts. Both encodings are introduced in the same 2008 paper by Conor McBride and Ross Paterson[2].

[2]Semigroupal is referred to as "monoidal" in the paper.

Cats models applicatives using two type classes. The first, `cats.Apply`, extends `Semigroupal` and `Functor` and adds an `ap` method that applies a parameter to a function within a context. The second, `cats.Applicative`, extends `Apply`, adds the `pure` method introduced in Chapter 4. Here's a simplified definition in code:

```
trait Apply[F[_]] extends Semigroupal[F] with Functor[F] {
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]

  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)] =
    ap(map(fa)(a => (b: B) => (a, b)))(fb)
}

trait Applicative[F[_]] extends Apply[F] {
  def pure[A](a: A): F[A]
}
```

Breaking this down, the `ap` method applies a parameter `fa` to a function `ff` within a context `F[_]`. The `product` method from `Semigroupal` is defined in terms of `ap` and `map`.

Don't worry too much about the implementation of `product`—it's difficult to read and the details aren't particularly important. The main point is that there is a tight relationship between `product`, `ap`, and `map` that allows any one of them to be defined in terms of the other two.

`Applicative` also introduces the `pure` method. This is the same `pure` we saw in `Monad`. It constructs a new applicative instance from an unwrapped value. In this sense, `Applicative` is related to `Apply` as `Monoid` is related to `Semigroup`.

6.5.1 The Hierarchy of Sequencing Type Classes

With the introduction of `Apply` and `Applicative`, we can zoom out and see a whole family of type classes that concern themselves with sequencing computations in different ways. Figure 6.1 shows the relationship between the type classes covered in this book[3].

[3] See Rob Norris' infographic for a the complete picture.

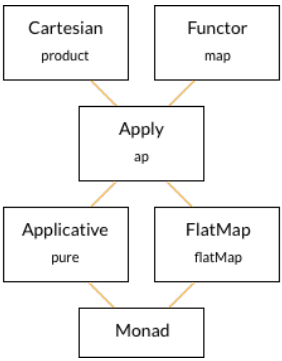


Figure 6.1: Monad type class hierarchy

Each type class in the hierarchy represents a particular set of sequencing semantics, introduces a set of characteristic methods, and defines the functionality of its supertypes in terms of them:

- every monad is an applicative;
- every applicative a semigroupal;
- and so on.

Because of the lawful nature of the relationships between the type classes, the inheritance relationships are constant across all instances of a type class. `Apply` defines `product` in terms of `ap` and `map`; `Monad` defines `product`, `ap`, and `map`, in terms of `pure` and `flatMap`.

To illustrate this let's consider two hypothetical data types:

- `Foo` is a monad. It has an instance of the `Monad` type class that implements `pure` and `flatMap` and inherits standard definitions of `product`, `map`, and `ap`;

- `Bar` is an applicative functor. It has an instance of `Applicative` that implements `pure` and `ap` and inherits standard definitions of `product` and `map`.

What can we say about these two data types without knowing more about their implementation?

We know strictly more about `Foo` than `Bar`: `Monad` is a subtype of `Applicative`, so we can guarantee properties of `Foo` (namely `flatMap`) that we cannot guarantee with `Bar`. Conversely, we know that `Bar` may have a wider range of behaviours than `Foo`. It has fewer laws to obey (no `flatMap`), so it can implement behaviours that `Foo` cannot.

This demonstrates the classic trade-off of power (in the mathematical sense) versus constraint. The more constraints we place on a data type, the more guarantees we have about its behaviour, but the fewer behaviours we can model.

Monads happen to be a sweet spot in this trade-off. They are flexible enough to model a wide range of behaviours and restrictive enough to give strong guarantees about those behaviours. However, there are situations where monads aren't the right tool for the job. Sometimes we want Thai food, and burritos just won't satisfy.

Whereas monads impose a strict *sequencing* on the computations they model, applicatives and semigroupals impose no such restriction. This puts them in a different sweet spot in the hierarchy. We can use them to represent classes of parallel / independent computations that monads cannot.

We choose our semantics by choosing our data structures. If we choose a monad, we get strict sequencing. If we choose an applicative, we lose the ability to `flatMap`. This is the trade-off enforced by the consistency laws. So choose your types carefully!

6.6 Summary

While monads and functors are the most widely used sequencing data types we've covered in this book, semigroupals and applicatives are the most general. These type classes provide a generic mechanism to combine values and apply functions within a context, from which we can fashion monads and a variety of other combinators.

`Semigroupal` and `Applicative` are most commonly used as a means of combining independent values such as the results of validation rules. Cats provides the `Validated` type for this specific purpose, along with apply syntax as a convenient way to express the combination of rules.

We have almost covered all of the functional programming concepts on our agenda for this book. The next chapter covers `Traverse` and `Foldable`, two powerful type classes for converting between data types. After that we'll look at several case studies that bring together all of the concepts from Part I.

Chapter 7

Foldable and Traverse

In this chapter we'll look at two type classes that capture iteration over collections:

- `Foldable` abstracts the familiar `foldLeft` and `foldRight` operations;
- `Traverse` is a higher-level abstraction that uses `Applicatives` to iterate with less pain than folding.

We'll start by looking at `Foldable`, and then examine cases where folding becomes complex and `Traverse` becomes convenient.

7.1 Foldable

The `Foldable` type class captures the `foldLeft` and `foldRight` methods we're used to in sequences like `Lists`, `Vectors`, and `Streams`. Using `Foldable`, we can write generic folds that work with a variety of sequence types. We can also invent new sequences and plug them into our code. `Foldable` gives us great use cases for `Monoids` and the `Eval` monad.

7.1.1 Folds and Folding

Let's start with a quick recap of the general concept of folding. We supply an *accumulator* value and a *binary function* to combine it with each item in the sequence:

```
def show[A](list: List[A]): String =
  list.foldLeft("nil")((accum, item) => s"$item then $accum")

show(Nil)
// res0: String = nil

show(List(1, 2, 3))
// res1: String = 3 then 2 then 1 then nil
```

The `foldLeft` method works recursively down the sequence. Our binary function is called repeatedly for each item, the result of each call becoming the accumulator for the next. When we reach the end of the sequence, the final accumulator becomes our final result.

Depending on the operation we're performing, the order in which we fold may be important. Because of this there are two standard variants of fold:

- `foldLeft` traverses from "left" to "right" (start to finish);
- `foldRight` traverses from "right" to "left" (finish to start).

Figure 7.1 illustrates each direction.

`foldLeft` and `foldRight` are equivalent if our binary operation is associative. For example, we can sum a `List[Int]` by folding in either direction, using `0` as our accumulator and addition as our operation:

```
List(1, 2, 3).foldLeft(0)(_ + _)
// res2: Int = 6

List(1, 2, 3).foldRight(0)(_ + _)
// res3: Int = 6
```

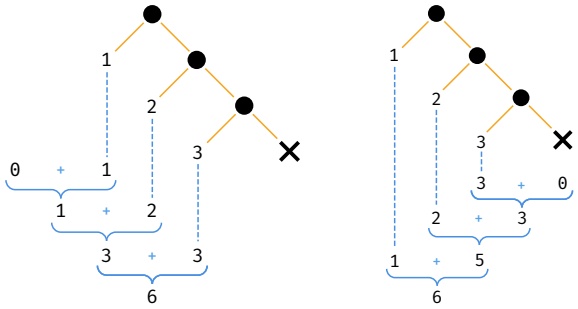


Figure 7.1: Illustration of foldLeft and foldRight

If we provide a non-associative operator the order of evaluation makes a difference. For example, if we fold using subtraction, we get different results in each direction:

```
List(1, 2, 3).foldLeft(0)(_ - _)
// res4: Int = -6

List(1, 2, 3).foldRight(0)(_ - _)
// res5: Int = 2
```

7.1.2 Exercise: Reflecting on Folds

Try using `foldLeft` and `foldRight` with an empty list as the accumulator and `::` as the binary operator. What results do you get in each case?

See the solution

7.1.3 Exercise: Scaf-fold-ing Other Methods

`foldLeft` and `foldRight` are very general methods. We can use them to implement many of the other high-level sequence operations we know. Prove this to yourself by implementing substitutes for `List`'s `map`, `flatMap`, `filter`, and `sum` methods in terms of `foldRight`.

See the solution

7.1.4 Foldable in Cats

Cats' `Foldable` abstracts `foldLeft` and `foldRight` into a type class. Instances of `Foldable` define these two methods and inherit a host of derived methods. Cats provides out-of-the-box instances of `Foldable` for a handful of Scala data types: `List`, `Vector`, `Stream`, and `Option`.

We can summon instances as usual using `Foldable.apply` and call their implementations of `foldLeft` directly. Here is an example using `List`:

```
import cats.Foldable
import cats.instances.list._ // for Foldable

val ints = List(1, 2, 3)

Foldable[List].foldLeft(ints, 0)(_ + _)
// res1: Int = 6
```

Other sequences like `Vector` and `Stream` work in the same way. Here is an example using `Option`, which is treated like a sequence of zero or one elements:

```
import cats.instances.option._ // for Foldable

val maybeInt = Option(123)

Foldable[Option].foldLeft(maybeInt, 10)(_ * _)
// res3: Int = 1230
```

7.1.4.1 Folding Right

`Foldable` defines `foldRight` differently to `foldLeft`, in terms of the `Eval` monad:

```
def foldRight[A, B](fa: F[A], lb: Eval[B])
                    (f: (A, Eval[B]) => Eval[B]): Eval[B]
```

Using `Eval` means folding is always *stack safe*, even when the collection's default definition of `foldRight` is not. For example, the default implementation

of `foldRight` for `Stream` is not stack safe. The longer the stream, the larger the stack requirements for the fold. A sufficiently large stream will trigger a `StackOverflowError`:

```
import cats.Eval
import cats.Foldable

def bigData = (1 to 100000).toStream

bigData.foldRight(0L)(_ + _)
// java.lang.StackOverflowError ...
```

Using `Foldable` forces us to use stack safe operations, which fixes the overflow exception:

```
import cats.instances.stream._ // for Foldable

val eval: Eval[Long] =
  Foldable[Stream].
    foldRight(bigData, Eval.now(0L)) { (num, eval) =>
      eval.map(_ + num)
    }

eval.value
// res7: Long = 5000050000
```

> *Stack Safety in the Standard Library*
>
> Stack safety isn't typically an issue when using the standard library. The most commonly used collection types, such as `List` and `Vector`, provide stack safe implementations of `foldRight`:
>
> ```
> (1 to 100000).toList.foldRight(0L)(_ + _)
> // res8: Long = 5000050000
>
> (1 to 100000).toVector.foldRight(0L)(_ + _)
> // res9: Long = 5000050000
> ```
>
> We've called out `Stream` because it is an exception to this rule. What-

> ever data type we're using, though, it's useful to know that `Eval` has our back.

7.1.4.2 Folding with Monoids

`Foldable` provides us with a host of useful methods defined on top of `foldLeft`. Many of these are facsimiles of familiar methods from the standard library: `find`, `exists`, `forall`, `toList`, `isEmpty`, `nonEmpty`, and so on:

```
Foldable[Option].nonEmpty(Option(42))
// res10: Boolean = true

Foldable[List].find(List(1, 2, 3))(_ % 2 == 0)
// res11: Option[Int] = Some(2)
```

In addition to these familiar methods, Cats provides two methods that make use of `Monoids`:

- `combineAll` (and its alias `fold`) combines all elements in the sequence using their `Monoid`;
- `foldMap` maps a user-supplied function over the sequence and combines the results using a `Monoid`.

For example, we can use `combineAll` to sum over a `List[Int]`:

```
import cats.instances.int._ // for Monoid

Foldable[List].combineAll(List(1, 2, 3))
// res12: Int = 6
```

Alternatively, we can use `foldMap` to convert each `Int` to a `String` and concatenate them:

```
import cats.instances.string._ // for Monoid

Foldable[List].foldMap(List(1, 2, 3))(_.toString)
// res13: String = 123
```

Finally, we can compose `Foldables` to support deep traversal of nested sequences:

```
import cats.instances.vector._ // for Monoid

val ints = List(Vector(1, 2, 3), Vector(4, 5, 6))

(Foldable[List] compose Foldable[Vector]).combineAll(ints)
// res15: Int = 21
```

7.1.4.3 Syntax for Foldable

Every method in `Foldable` is available in syntax form via `cats.syntax.foldable`. In each case, the first argument to the method on `Foldable` becomes the receiver of the method call:

```
import cats.syntax.foldable._ // for combineAll and foldMap

List(1, 2, 3).combineAll
// res16: Int = 6

List(1, 2, 3).foldMap(_.toString)
// res17: String = 123
```

> *Explicits over Implicits*
>
> Remember that Scala will only use an instance of `Foldable` if the method isn't explicitly available on the receiver. For example, the following code will use the version of `foldLeft` defined on `List`:
>
> ```
> List(1, 2, 3).foldLeft(0)(_ + _)
> // res18: Int = 6
> ```

whereas the following generic code will use `Foldable`:

```
import scala.language.higherKinds

def sum[F[_]: Foldable](values: F[Int]): Int =
  values.foldLeft(0)(_ + _)
// sum: [F[_]](values: F[Int])(implicit evidence$1: cats.
    Foldable[F])Int
```

We typically don't need to worry about this distinction. It's a feature! We call the method we want and the compiler uses a `Foldable` when needed to ensure our code works as expected. If we need a stack-safe implementation of `foldRight`, using `Eval` as the accumulator is enough to force the compiler to select the method from Cats.

7.2 Traverse

`foldLeft` and `foldRight` are flexible iteration methods but they require us to do a lot of work to define accumulators and combinator functions. The `Traverse` type class is a higher level tool that leverages `Applicatives` to provide a more convenient, more lawful, pattern for iteration.

7.2.1 Traversing with Futures

We can demonstrate `Traverse` using the `Future.traverse` and `Future.sequence` methods in the Scala standard library. These methods provide `Future`-specific implementations of the traverse pattern. As an example, suppose we have a list of server hostnames and a method to poll a host for its uptime:

```
import scala.concurrent._
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

val hostnames = List(
```

```
  "alpha.example.com",
  "beta.example.com",
  "gamma.demo.com"
)

def getUptime(hostname: String): Future[Int] =
  Future(hostname.length * 60) // just for demonstration
```

Now, suppose we want to poll all of the hosts and collect all of their uptimes. We can't simply `map` over `hostnames` because the result—a `List[Future[Int]]`—would contain more than one `Future`. We need to reduce the results to a single `Future` to get something we can block on. Let's start by doing this manually using a fold:

```
val allUptimes: Future[List[Int]] =
  hostnames.foldLeft(Future(List.empty[Int])) {
    (accum, host) =>
      val uptime = getUptime(host)
      for {
        accum  <- accum
        uptime <- uptime
      } yield accum :+ uptime
  }

Await.result(allUptimes, 1.second)
// res2: List[Int] = List(1020, 960, 840)
```

Intuitively, we iterate over `hostnames`, call `func` for each item, and combine the results into a list. This sounds simple, but the code is fairly unwieldy because of the need to create and combine `Futures` at every iteration. We can improve on things greatly using `Future.traverse`, which is tailor-made for this pattern:

```
val allUptimes: Future[List[Int]] =
  Future.traverse(hostnames)(getUptime)

Await.result(allUptimes, 1.second)
// res3: List[Int] = List(1020, 960, 840)
```

This is much clearer and more concise—let's see how it works. If we ignore

distractions like `CanBuildFrom` and `ExecutionContext`, the implementation of `Future.traverse` in the standard library looks like this:

```
def traverse[A, B](values: List[A])
    (func: A => Future[B]): Future[List[B]] =
  values.foldLeft(Future(List.empty[A])) { (accum, host) =>
    val item = func(host)
    for {
      accum <- accum
      item  <- item
    } yield accum :+ item
  }
```

This is essentially the same as our example code above. `Future.traverse` is abstracting away the pain of folding and defining accumulators and combination functions. It gives us a clean high-level interface to do what we want:

- start with a `List[A]`;
- provide a function `A => Future[B]`;
- end up with a `Future[List[B]]`.

The standard library also provides another method, `Future.sequence`, that assumes we're starting with a `List[Future[B]]` and don't need to provide an identity function:

```
object Future {
  def sequence[B](futures: List[Future[B]]): Future[List[B]] =
    traverse(futures)(identity)

  // etc...
}
```

In this case the intuitive understanding is even simpler:

- start with a `List[Future[A]]`;
- end up with a `Future[List[A]]`.

`Future.traverse` and `Future.sequence` solve a very specific problem: they allow us to iterate over a sequence of `Futures` and accumulate a result. The simplified examples above only work with `Lists`, but the real `Future.traverse` and `Future.sequence` work with any standard Scala collection.

Cats' `Traverse` type class generalises these patterns to work with any type of `Applicative`: `Future`, `Option`, `Validated`, and so on. We'll approach `Traverse` in the next sections in two steps: first we'll generalise over the `Applicative`, then we'll generalise over the sequence type. We'll end up with an extremely valuable tool that trivialises many operations involving sequences and other data types.

7.2.2 Traversing with Applicatives

If we squint, we'll see that we can rewrite `traverse` in terms of an `Applicative`. Our accumulator from the example above:

```
Future(List.empty[Int])
```

is equivalent to `Applicative.pure`:

```
import cats.Applicative
import cats.instances.future._   // for Applicative
import cats.syntax.applicative._ // for pure

List.empty[Int].pure[Future]
```

Our combinator, which used to be this:

```
def oldCombine(
  accum : Future[List[Int]],
  host  : String
): Future[List[Int]] = {
  val uptime = getUptime(host)
  for {
    accum  <- accum
```

```
    uptime <- uptime
  } yield accum :+ uptime
}
```

is now equivalent to `Semigroupal.combine`:

```
import cats.syntax.apply._ // for mapN

// Combining accumulator and hostname using an Applicative:
def newCombine(accum: Future[List[Int]],
      host: String): Future[List[Int]] =
  (accum, getUptime(host)).mapN(_ :+ _)
```

By substituting these snippets back into the definition of `traverse` we can generalise it to to work with any `Applicative`:

```
import scala.language.higherKinds

def listTraverse[F[_]: Applicative, A, B]
      (list: List[A])(func: A => F[B]): F[List[B]] =
  list.foldLeft(List.empty[B].pure[F]) { (accum, item) =>
    (accum, func(item)).mapN(_ :+ _)
  }

def listSequence[F[_]: Applicative, B]
      (list: List[F[B]]): F[List[B]] =
  listTraverse(list)(identity)
```

We can use `listTraverse` to re-implement our uptime example:

```
val totalUptime = listTraverse(hostnames)(getUptime)

Await.result(totalUptime, 1.second)
// res11: List[Int] = List(1020, 960, 840)
```

or we can use it with with other `Applicative` data types as shown in the following exercises.

7.2.2.1 Exercise: Traversing with Vectors

What is the result of the following?

```
import cats.instances.vector._ // for Applicative

listSequence(List(Vector(1, 2), Vector(3, 4)))
```

See the solution

What about a list of three parameters?

```
listSequence(List(Vector(1, 2), Vector(3, 4), Vector(5, 6)))
```

See the solution

7.2.2.2 Exercise: Traversing with Options

Here's an example that uses `Options`:

```
import cats.instances.option._ // for Applicative

def process(inputs: List[Int]) =
  listTraverse(inputs)(n => if(n % 2 == 0) Some(n) else None)
```

What is the return type of this method? What does it produce for the following inputs?

```
process(List(2, 4, 6))
process(List(1, 2, 3))
```

See the solution

7.2.2.3 Exercise: Traversing with Validated

Finally, here is an example that uses `Validated`:

```
import cats.data.Validated
import cats.instances.list._ // for Monoid

type ErrorsOr[A] = Validated[List[String], A]

def process(inputs: List[Int]): ErrorsOr[List[Int]] =
  listTraverse(inputs) { n =>
    if(n % 2 == 0) {
      Validated.valid(n)
    } else {
      Validated.invalid(List(s"$n is not even"))
    }
  }
```

What does this method produce for the following inputs?

```
process(List(2, 4, 6))
process(List(1, 2, 3))
```

See the solution

7.2.3 Traverse in Cats

Our `listTraverse` and `listSequence` methods work with any type of `Applicative`, but they only work with one type of sequence: `List`. We can generalise over different sequence types using a type class, which brings us to Cats' `Traverse`. Here's the abbreviated definition:

```
package cats

trait Traverse[F[_]] {
  def traverse[G[_]: Applicative, A, B]
      (inputs: F[A])(func: A => G[B]): G[F[B]]

  def sequence[G[_]: Applicative, B]
      (inputs: F[G[B]]): G[F[B]] =
    traverse(inputs)(identity)
}
```

Cats provides instances of `Traverse` for `List`, `Vector`, `Stream`, `Option`, `Either`, and a variety of other types. We can summon instances as usual using `Traverse.apply` and use the `traverse` and `sequence` methods as described in the previous section:

```
import cats.Traverse
import cats.instances.future._ // for Applicative
import cats.instances.list._   // for Traverse

val totalUptime: Future[List[Int]] =
  Traverse[List].traverse(hostnames)(getUptime)

Await.result(totalUptime, 1.second)
// res1: List[Int] = List(1020, 960, 840)

val numbers = List(Future(1), Future(2), Future(3))

val numbers2: Future[List[Int]] =
  Traverse[List].sequence(numbers)

Await.result(numbers2, 1.second)
// res3: List[Int] = List(1, 2, 3)
```

There are also syntax versions of the methods, imported via `cats.syntax.traverse`:

```
import cats.syntax.traverse._ // for sequence and traverse

Await.result(hostnames.traverse(getUptime), 1.second)
// res4: List[Int] = List(1020, 960, 840)

Await.result(numbers.sequence, 1.second)
// res5: List[Int] = List(1, 2, 3)
```

As you can see, this is much more compact and readable than the `foldLeft` code we started with earlier this chapter!

7.3 Summary

In this chapter we were introduced to `Foldable` and `Traverse`, two type classes for iterating over sequences.

`Foldable` abstracts the `foldLeft` and `foldRight` methods we know from collections in the standard library. It adds stack-safe implementations of these methods to a handful of extra data types, and defines a host of situationally useful additions. That said, `Foldable` doesn't introduce much that we didn't already know.

The real power comes from `Traverse`, which abstracts and generalises the `traverse` and `sequence` methods we know from `Future`. Using these methods we can turn an `F[G[A]]` into a `G[F[A]]` for any `F` with an instance of `Traverse` and any `G` with an instance of `Applicative`. In terms of the reduction we get in lines of code, `Traverse` is one of the most powerful patterns in this book. We can reduce `folds` of many lines down to a single `foo.traverse`.

...and with that, we've finished all of the theory in this book. There's plenty more to come, though, as we put everything we've learned into practice in a series of in-depth case studies in Part II!

Part II

Case Studies

Chapter 8

Case Study: Testing Asynchronous Code

We'll start with a straightforward case study: how to simplify unit tests for asynchronous code by making them synchronous.

Let's return to the example from Chapter 7 where we're measuring the uptime on a set of servers. We'll flesh out the code into a more complete structure. There will be two components. The first is an `UptimeClient` that polls remote servers for their uptime:

```
import scala.concurrent.Future

trait UptimeClient {
  def getUptime(hostname: String): Future[Int]
}
```

We'll also have an `UptimeService` that maintains a list of servers and allows the user to poll them for their total uptime:

```
import cats.instances.future._ // for Applicative
import cats.instances.list._   // for Traverse
import cats.syntax.traverse._  // for traverse
```

```
import scala.concurrent.ExecutionContext.Implicits.global

class UptimeService(client: UptimeClient) {
  def getTotalUptime(hostnames: List[String]): Future[Int] =
    hostnames.traverse(client.getUptime).map(_.sum)
}
```

We've modelled `UptimeClient` as a trait because we're going to want to stub it out in unit tests. For example, we can write a test client that allows us to provide dummy data rather than calling out to actual servers:

```
class TestUptimeClient(hosts: Map[String, Int]) extends UptimeClient {
  def getUptime(hostname: String): Future[Int] =
    Future.successful(hosts.getOrElse(hostname, 0))
}
```

Now, suppose we're writing unit tests for `UptimeService`. We want to test its ability to sum values, regardless of where it is getting them from. Here's an example:

```
def testTotalUptime() = {
  val hosts    = Map("host1" -> 10, "host2" -> 6)
  val client   = new TestUptimeClient(hosts)
  val service  = new UptimeService(client)
  val actual   = service.getTotalUptime(hosts.keys.toList)
  val expected = hosts.values.sum
  assert(actual == expected)
}
// <console>:31: warning: scala.concurrent.Future[Int] and Int are
     unrelated: they will most likely never compare equal
//          assert(actual == expected)
//                         ^
// error: No warnings can be incurred under -Xfatal-warnings.
```

The code doesn't compile because we've made a classic error[1]. We forgot that our application code is asynchronous. Our `actual` result is of type `Future[Int]` and out `expected` result is of type `Int`. We can't compare them directly!

[1]Technically this is a *warning* not an error. It has been promoted to an error in our case because we're using the `-Xfatal-warnings` flag on `scalac`.

There are a couple of ways to solve this problem. We could alter our test code to accommodate the asynchronousness. However, there is another alternative. Let's make our service code synchronous so our test works without modification!

8.1 Abstracting over Type Constructors

We need to implement two versions of `UptimeClient`: an asynchronous one for use in production and a synchronous one for use in our unit tests:

```
trait RealUptimeClient extends UptimeClient {
  def getUptime(hostname: String): Future[Int]
}

trait TestUptimeClient extends UptimeClient {
  def getUptime(hostname: String): Int
}
```

The question is: what result type should we give to the abstract method in `UptimeClient`? We need to abstract over `Future[Int]` and `Int`:

```
trait UptimeClient {
  def getUptime(hostname: String): ???
}
```

At first this may seem difficult. We want to retain the `Int` part from each type but "throw away" the `Future` part in the test code. Fortunately, Cats provides a solution in terms of the *identity type*, `Id`, that we discussed way back in Section 4.3. `Id` allows us to "wrap" types in a type constructor without changing their meaning:

```
package cats

type Id[A] = A
```

`Id` allows us to abstract over the return types in `UptimeClient`. Implement this now:

- write a trait definition for `UptimeClient` that accepts a type constructor `F[_]` as a parameter;
- extend it with two traits, `RealUptimeClient` and `TestUptimeClient`, that bind `F` to `Future` and `Id` respectively;
- write out the method signature for `getUptime` in each case to verify that it compiles.

See the solution

You should now be able to flesh your definition of `TestUptimeClient` out into a full class based on a `Map[String, Int]` as before.

See the solution

8.2 Abstracting over Monads

Let's turn our attention to `UptimeService`. We need to rewrite it to abstract over the two types of `UptimeClient`. We'll do this in two stages: first we'll rewrite the class and method signatures, then the method bodies. Starting with the method signatures:

- comment out the body of `getTotalUptime` (replace it with `???` to make everything compile);
- add a type parameter `F[_]` to `UptimeService` and pass it on to `UptimeClient`.

See the solution

Now uncomment the body of `getTotalUptime`. You should get a compilation error similar to the following:

```
// <console>:28: error: could not find implicit value for
//               evidence parameter of type cats.Applicative[F]
//            hostnames.traverse(client.getUptime).map(_.sum)
//                               ^
```

The problem here is that `traverse` only works on sequences of values that have an `Applicative`. In our original code we were traversing a `List[Future[Int]]`. There is an applicative for `Future` so that was fine. In this version we are traversing a `List[F[Int]]`. We need to *prove* to the compiler that `F` has an `Applicative`. Do this by adding an implicit constructor parameter to `UptimeService`.

See the solution

Finally, let's turn our attention to our unit tests. Our test code now works as intended without any modification. We create an instance of `TestUptimeClient` and wrap it in an `UptimeService`. This effectively binds `F` to `Id`, allowing the rest of the code to operate synchronously without worrying about monads or applicatives:

```
def testTotalUptime() = {
  val hosts    = Map("host1" -> 10, "host2" -> 6)
  val client   = new TestUptimeClient(hosts)
  val service  = new UptimeService(client)
  val actual   = service.getTotalUptime(hosts.keys.toList)
  val expected = hosts.values.sum
  assert(actual == expected)
}

testTotalUptime()
```

8.3 Summary

This case study provides an example of how Cats can help us abstract over different computational scenarios. We used the `Applicative` type class to abstract over asynchronous and synchronous code. Leaning on a functional abstraction allows us to specify the sequence of computations we want to perform without worrying about the details of the implementation.

Back in Figure 6.1, we showed a "stack" of computational type classes that are meant for exactly this kind of abstraction. Type classes like `Functor`, `Applicative`, `Monad`, and `Traverse` provide abstract implementations of patterns such as mapping, zipping, sequencing, and iteration. The mathematical laws on those types ensure that they work together with a consistent set of semantics.

We used `Applicative` in this case study because it was the least powerful type class that did what we needed. If we had required `flatMap`, we could have swapped out `Applicative` for `Monad`. If we had needed to abstract over different sequence types, we could have used `Traverse`. There are also type classes like `ApplicativeError` and `MonadError` that help model failures as well as successful computations.

Let's move on now to a more complex case study where type classes will help us produce something more interesting: a map-reduce-style framework for parallel processing.

Chapter 9

Case Study: Map-Reduce

In this case study we're going to implement a simple-but-powerful parallel processing framework using `Monoids`, `Functors`, and a host of other goodies.

If you have used Hadoop or otherwise worked in "big data" you will have heard of MapReduce, which is a programming model for doing parallel data processing across clusters of machines (aka "nodes"). As the name suggests, the model is built around a *map* phase, which is the same `map` function we know from Scala and the `Functor` type class, and a *reduce* phase, which we usually call `fold`[1] in Scala.

9.1 Parallelizing *map* and *fold*

Recall the general signature for `map` is to apply a function `A => B` to a `F[A]`, returning a `F[B]`:

`map` transforms each individual element in a sequence independently. We can easily parallelize `map` because there are no dependencies between the transformations applied to different elements (the type signature of the function `A => B` shows us this, assuming we don't use side-effects not reflected in the types).

[1] In Hadoop there is also a shuffle phase that we will ignore here.

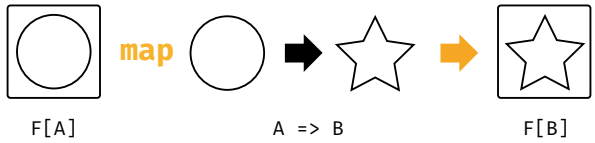


Figure 9.1: Type chart: functor map

Figure 9.2: Type chart: fold

What about `fold`? We can implement this step with an instance of `Foldable`. Not every functor also has an instance of foldable but we can implement a map-reduce system on top of any data type that has both of these type classes. Our reduction step becomes a `foldLeft` over the results of the distributed `map`.

By distributing the reduce step we lose control over the order of traversal. Our overall reduction may not be entirely left-to-right—we may reduce left-to-right across several subsequences and then combine the results. To ensure correctness we need a reduction operation that is *associative*:

```
reduce(a1, reduce(a2, a3)) == reduce(reduce(a1, a2), a3)
```

If we have associativity, we can arbitrarily distribute work between our nodes provided the subsequences at every node stay in the same order as the initial dataset.

Our fold operation requires us to seed the computation with an element of type B. Since fold may be split into an arbitrary number of parallel steps, the seed should not affect the result of the computation. This naturally requires the seed to be an *identity* element:

```
reduce(seed, a1) == reduce(a1, seed) == a1
```

In summary, our parallel fold will yield the correct results if:

- we require the reducer function to be associative;
- we seed the computation with the identity of this function.

What does this pattern sound like? That's right, we've come full circle back to `Monoid`, the first type class we discussed in this book. We are not the first to recognise the importance of monoids. The monoid design pattern for map-reduce jobs is at the core of recent big data systems such as Twitter's Summingbird.

In this project we're going to implement a very simple single-machine map-reduce. We'll start by implementing a method called `foldMap` to model the data-flow we need.

9.2 Implementing *foldMap*

We saw `foldMap` briefly back when we covered `Foldable`. It is one of the derived operations that sits on top of `foldLeft` and `foldRight`. However, rather than use `Foldable`, we will re-implement `foldMap` here ourselves as it will provide useful insight into the structure of map-reduce.

Start by writing out the signature of `foldMap`. It should accept the following parameters:

- a sequence of type `Vector[A]`;
- a function of type `A => B`, where there is a `Monoid` for `B`;

You will have to add implicit parameters or context bounds to complete the type signature.

See the solution

Now implement the body of `foldMap`. Use the flow chart in Figure 9.3 as a guide to the steps required:

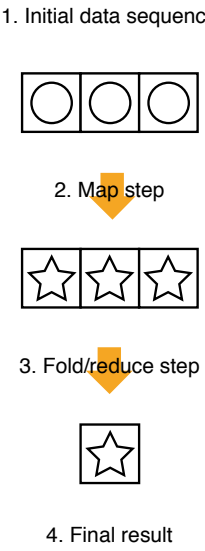


Figure 9.3: *foldMap* algorithm

1. start with a sequence of items of type `A`;
2. map over the list to produce a sequence of items of type `B`;
3. use the `Monoid` to reduce the items to a single `B`.

Here's some sample output for reference:

```
import cats.instances.int._ // for Monoid

foldMap(Vector(1, 2, 3))(identity)
// res2: Int = 6

import cats.instances.string._ // for Monoid

// Mapping to a String uses the concatenation monoid:
foldMap(Vector(1, 2, 3))(_.toString + "! ")
// res4: String = "1! 2! 3! "
```

```
// Mapping over a String to produce a String:
foldMap("Hello world!".toVector)(_.toString.toUpperCase)
// res6: String = HELLO WORLD!
```

See the solution

9.3 Parallelising *foldMap*

Now we have a working single-threaded implementation of `foldMap`, let's look at distributing work to run in parallel. We'll use our single-threaded version of `foldMap` as a building block.

We'll write a multi-CPU implementation that simulates the way we would distribute work in a map-reduce cluster as shown in Figure 9.4:

1. we start with an initial list of all the data we need to process;
2. we divide the data into batches, sending one batch to each CPU;
3. the CPUs run a batch-level map phase in parallel;
4. the CPUs run a batch-level reduce phase in parallel, producing a local result for each batch;
5. we reduce the results for each batch to a single final result.

Scala provides some simple tools to distribute work amongst threads. We could use the parallel collections library to implement a solution, but let's challenge ourselves by diving a bit deeper and implementing the algorithm ourselves using `Futures`.

9.3.1 *Futures*, Thread Pools, and ExecutionContexts

We already know a fair amount about the monadic nature of `Futures`. Let's take a moment for a quick recap, and to describe how Scala futures are scheduled behind the scenes.

`Futures` run on a thread pool, determined by an implicit `ExecutionContext` parameter. Whenever we create a `Future`, whether through a call to

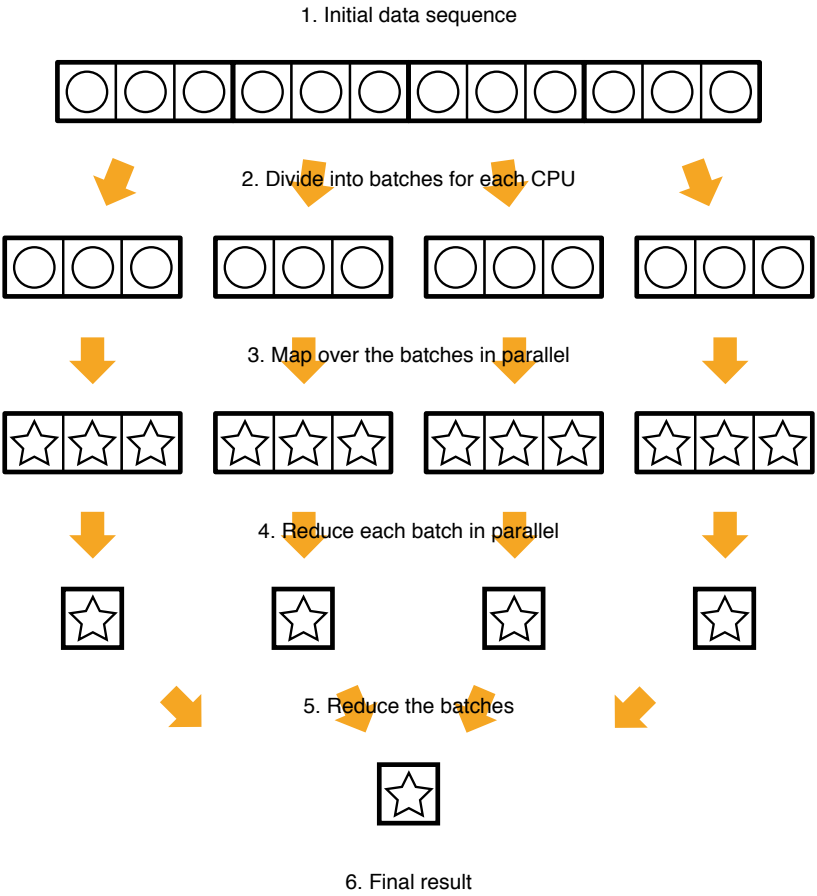


Figure 9.4: *parallelFoldMap* algorithm

`Future.apply` or some other combinator, we must have an implicit `ExecutionContext` in scope:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

val future1 = Future {
  (1 to 100).toList.foldLeft(0)(_ + _)
}
// future1: scala.concurrent.Future[Int] = Future(<not completed>)

val future2 = Future {
  (100 to 200).toList.foldLeft(0)(_ + _)
}
// future2: scala.concurrent.Future[Int] = Future(<not completed>)
```

In this example we've imported a `ExecutionContext.Implicits.global`. This default context allocates a thread pool with one thread per CPU in our machine. When we create a `Future` the `ExecutionContext` schedules it for execution. If there is a free thread in the pool, the `Future` starts executing immediately. Most modern machines have at least two CPUs, so in our example it is likely that `future1` and `future2` will execute in parellel.

Some combinators create new `Futures` that schedule work based on the results of other `Futures`. The `map` and `flatMap` methods, for example, schedule computations that run as soon as their input values are computed and a CPU is available:

```
val future3 = future1.map(_.toString)
// future3: scala.concurrent.Future[String] = Future(<not completed>)

val future4 = for {
  a <- future1
  b <- future2
} yield a + b
// future4: scala.concurrent.Future[Int] = Future(<not completed>)
```

As we saw in Section 7.2, we can convert a `List[Future[A]]` to a `Future[List[A]]` using `Future.sequence`:

```
Future.sequence(List(Future(1), Future(2), Future(3)))
// res8: scala.concurrent.Future[List[Int]] = Future(<not completed>)
```

or an instance of `Traverse`:

```
import cats.instances.future._ // for Applicative
import cats.instances.list._   // for Traverse
import cats.syntax.traverse._  // for sequence

List(Future(1), Future(2), Future(3)).sequence
// res9: scala.concurrent.Future[List[Int]] = Future(<not completed>)
```

An `ExecutionContext` is required in either case. Finally, we can use `Await.result` to block on a `Future` until a result is available:

```
import scala.concurrent._
import scala.concurrent.duration._

Await.result(Future(1), 1.second) // wait for the result
// res10: Int = 1
```

There are also `Monad` and `Monoid` implementations for `Future` available from `cats.instances.future`:

```
import cats.{Monad, Monoid}
import cats.instances.int._    // for Monoid
import cats.instances.future._ // for Monad and Monoid

Monad[Future].pure(42)

Monoid[Future[Int]].combine(Future(1), Future(2))
```

9.3.2 Dividing Work

Now we've refreshed our memory of `Futures`, let's look at how we can divide work into batches. We can query the number of available CPUs on our machine using an API call from the Java standard library:

```
Runtime.getRuntime.availableProcessors
// res15: Int = 2
```

We can partition a sequence (actually anything that implements `Vector`) using the `grouped` method. We'll use this to split off batches of work for each CPU:

```
(1 to 10).toList.grouped(3).toList
// res16: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7,
      8, 9), List(10))
```

9.3.3 Implementing *parallelFoldMap*

Implement a parallel version of `foldMap` called `parallelFoldMap`. Here is the type signature:

```
def parallelFoldMap[A, B : Monoid]
      (values: Vector[A])
      (func: A => B): Future[B] = ???
```

Use the techniques described above to split the work into batches, one batch per CPU. Process each batch in a parallel thread. Refer back to Figure 9.4 if you need to review the overall algorithm.

For bonus points, process the batches for each CPU using your implementation of `foldMap` from above.

See the solution

9.3.4 *parallelFoldMap* with more Cats

Although we implemented `foldMap` ourselves above, the method is also available as part of the `Foldable` type class we discussed in Section 7.1.

Reimplement `parallelFoldMap` using Cats' `Foldable` and `Traverseable` type classes.

See the solution

9.4 Summary

In this case study we implemented a system that imitates map-reduce as performed on a cluster. Our algorithm followed three steps:

1. batch the data and send one batch to each "node";
2. perform a local map-reduce on each batch;
3. combine the results using monoid addition.

Our toy system emulates the batching behaviour of real-world map-reduce systems such as Hadoop. However, in reality we are running all of our work on a single machine where communcation between nodes is negligible. We don't actually need to batch data to gain efficient parallel processing of a list. We can simply map using a `Functor` and reduce using a `Monoid`.

Regardless of the batching strategy, mapping and reducing with `Monoids` is a powerful and general framework that isn't limited to simple tasks like addition and string concatenation. Most of the tasks data scientists perform in their day-to-day analyses can be cast as monoids. There are monoids for all the following:

- approximate sets such as the Bloom filter;
- set cardinality estimators, such as the HyperLogLog algorithm;
- vectors and vector operations like stochastic gradient descent;
- quantile estimators such as the t-digest

to name but a few.

Chapter 10

Case Study: Data Validation

In this case study we will build a library for validation. What do we mean by validation? Almost all programs must check their input meets certain criteria. Usernames must not be blank, email addresses must be valid, and so on. This type of validation often occurs in web forms, but it could be performed on configuration files, on web service responses, and any other case where we have to deal with data that we can't guarantee is correct. Authentication, for example, is just a specialised form of validation.

We want to build a library that performs these checks. What design goals should we have? For inspiration, let's look at some examples of the types of checks we want to perform:

- A user must be over 18 years old or must have parental consent.
- A `String` ID must be parsable as a `Int` and the `Int` must correspond to a valid record ID.
- A bid in an auction must apply to one or more items and have a positive value.
- A username must contain at least four characters and all characters must be alphanumeric.

- An email address must contain a single `@` sign. Split the string at the `@`. The string to the left must not be empty. The string to the right must be at least three characters long and contain a dot.

With these examples in mind we can state some goals:

- We should be able to associate meaningful messages with each validation failure, so the user knows why their data is not valid.
- We should be able to combine small checks into larger ones. Taking the username example above, we should be able to express this by combining a check of length and a check for alphanumeric values.
- We should be able to transform data while we are checking it. There is an example above requiring we parse data, changing its type from `String` to `Int`.
- Finally, we should be able to accumulate all the failures in one go, so the user can correct all the issues before resubmitting.

These goals assume we're checking a single piece of data. We will also need to combine checks across multiple pieces of data. For a login form, for example, we'll need to combine the check results for the username and the password. This will turn out to be quite a small component of the library, so the majority of our time will focus on checking a single data item.

10.1 Sketching the Library Structure

Let's start at the bottom, checking individual pieces of data. Before we start coding let's try to develop a feel for what we'll be building. We can use a graphical notation to help us. We'll go through our goals one by one.

Providing error messages

Our first goal requires us to associate useful error messages with a check failure. The output of a check could be either the value being checked, if it passed

Figure 10.1: A validation result

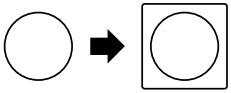


Figure 10.2: A validation check

the check, or some kind of error message. We can abstractly represent this as a value in a context, where the context is the possibility of an error message as shown in Figure 10.1.

A check itself is therefore a function that transforms a value into a value in a context as shown in Figure 10.2.

Combine checks

How do we combine smaller checks into larger ones? Is this an applicative or semigroupal as shown in Figure 10.3?

Not really. With applicative combination, both checks are applied to the same value and result in a tuple with the value repeated. What we want feels more like a monoid as shown in Figure 10.4. We can define a sensible identity—a check that always passes—and two binary combination operators—*and* and *or*:

We'll probably be using *and* and *or* about equally often with our validation

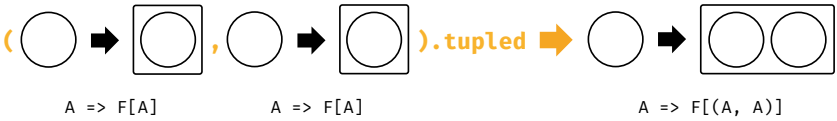


Figure 10.3: Applicative combination of checks

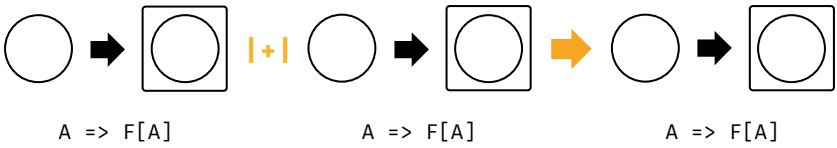


Figure 10.4: Monoid combination of checks

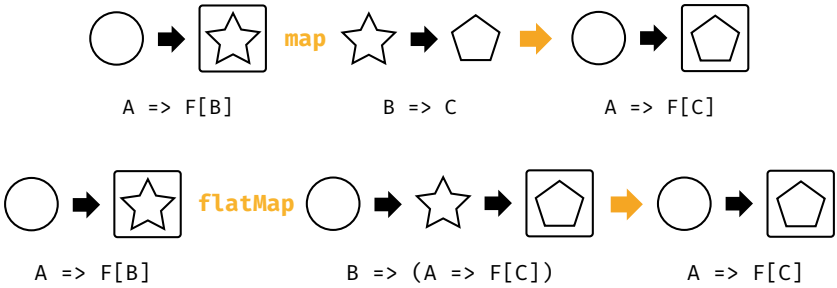


Figure 10.5: Monadic combination of checks

library and it will be annoying to continuously switch between two monoids for combining rules. We consequently won't actually use the monoid API: we'll use two separate methods, `and` and `or`, instead.

Accumulating errors as we check

Monoids also feel like a good mechanism for accumulating error messages. If we store messages as a `List` or `NonEmptyList`, we can even use a pre-existing monoid from inside Cats.

Transforming data as we check it

In addition to checking data, we also have the goal of transforming it. This seems like it should be a `map` or a `flatMap` depending on whether the transform can fail or not, so it seems we also want checks to be a monad as shown in Figure 10.5.

We've now broken down our library into familiar abstractions and are in a good position to begin development.

10.2 The Check Datatype

Our design revolves around a `Check`, which we said was a function from a value to a value in a context. As soon as you see this description you should think of something like

```
type Check[A] = A => Either[String, A]
```

Here we've represented the error message as a `String`. This is probably not the best representation. We may want to accumulate messages in a `List`, for example, or even use a different representation that allows for internationalization or standard error codes.

We could attempt to build some kind of `ErrorMessage` type that holds all the information we can think of. However, we can't predict the user's requirements. Instead let's let the user specify what they want. We can do this by adding a second type parameter to `Check`:

```
type Check[E, A] = A => Either[E, A]
```

We will probably want to add custom methods to `Check` so let's declare it as a `trait` instead of a type alias:

```
trait Check[E, A] {
  def apply(value: A): Either[E, A]

  // other methods...
}
```

As we said in Essential Scala, there are two functional programming patterns that we should consider when defining a trait:

- we can make it a typeclass, or;
- we can make it an algebraic data type (and hence seal it).

Type classes allow us to unify disparate data types with a common interface. This doesn't seem like what we're trying to do here. That leaves us with an

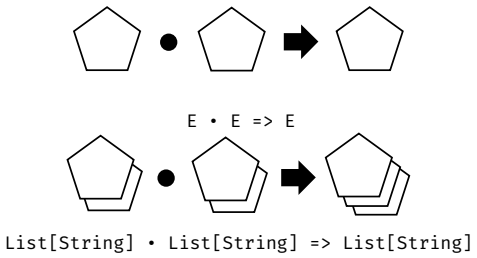


Figure 10.6: Combining error messages

algebraic data type. Let's keep that thought in mind as we explore the design a bit further.

10.3 Basic Combinators

Let's add some combinator methods to `Check`, starting with `and`. This method combines two checks into one, succeeding only if both checks succeed. Think about implementing this method now. You should hit some problems. Read on when you do!

```
trait Check[E, A] {
  def and(that: Check[E, A]): Check[E, A] =
    ???

  // other methods...
}
```

The problem is: what do you do when *both* checks fail? The correct thing to do is to return both errors, but we don't currently have any way to combine `E`s. We need a *type class* that abstracts over the concept of "accumulating" errors as shown in Figure 10.6 What type class do we know that looks like this? What method or operator should we use to implement the • operation?

See the solution

There is another semantic issue that will come up quite quickly: should `and` short-circuit if the first check fails. What do you think the most useful behaviour is?

See the solution

Use this knowledge to implement `and`. Make sure you end up with the behaviour you expect!

See the solution

Strictly speaking, `Either[E, A]` is the wrong abstraction for the output of our check. Why is this the case? What other data type could we use instead? Switch your implementation over to this new data type.

See the solution

Our implementation is looking pretty good now. Implement an `or` combinator to complement `and`.

See the solution

With `and` and `or` we can implement many of checks we'll want in practice. However, we still have a few more methods to add. We'll turn to `map` and related methods next.

10.4 Transforming Data

One of our requirements is the ability to transform data. This allows us to support additional scenarios like parsing input. In this section we'll extend our check library with this additional functionality.

The obvious starting point is `map`. When we try to implement this, we immediately run into a wall. Our current definition of `Check` requires the input and output types to be the same:

```
type Check[E, A] = A => Either[E, A]
```

When we map over a check, what type do we assign to the result? It can't be `A` and it can't be `B`. We are at an impasse:

```
def map(check: Check[E, A])(func: A => B): Check[E, ???]
```

To implement `map` we need to change the definition of `Check`. Specifically, we need to a new type variable to separate the input type from the output:

```
type Check[E, A, B] = A => Either[E, B]
```

Checks can now represent operations like parsing a `String` as an `Int`:

```
val parseInt: Check[List[String], String, Int] =
  // etc...
```

However, splitting our input and output types raises another issue. Up until now we have operated under the assumption that a `Check` always returns its input when successful. We used this in and and `or` to ignore the output of the left and right rules and simply return the original input on success:

```
(this(a), that(a)) match {
  case And(left, right) =>
    (left(a), right(a))
      .mapN((result1, result2) => Right(a))

  // etc...
}
```

In our new formulation we can't return `Right(a)` because its type is `Either[E, A]` not `Either[E, B]`. We're forced to make an arbitrary choice between returning `Right(result1)` and `Right(result2)`. The same is true of the `or` method. From this we can derive two things:

- we should strive to make the laws we adhere to explicit; and
- the code is telling us we have the wrong abstraction in `Check`.

10.4.1 Predicates

We can make progress by pulling apart the concept of a *predicate*, which can be combined using logical operations such as *and* and *or*, and the concept of a *check*, which can transform data.

What we have called `Check` so far we will call `Predicate`. For `Predicate` we can state the following *identity law* encoding the notion that a predicate always returns its input if it succeeds:

> For a predicate `p` of type `Predicate[E, A]` and elements `a1` and `a2` of type `A`, if `p(a1) == Success(a2)` then `a1 == a2`.

Making this change gives us the following code:

```
import cats.Semigroup
import cats.data.Validated
import cats.syntax.semigroup._ // for |+|
import cats.syntax.apply._     // for mapN
import cats.data.Validated._   // for Valid and Invalid

sealed trait Predicate[E, A] {
  def and(that: Predicate[E, A]): Predicate[E, A] =
    And(this, that)

  def or(that: Predicate[E, A]): Predicate[E, A] =
    Or(this, that)

  def apply(a: A)(implicit s: Semigroup[E]): Validated[E, A] =
    this match {
      case Pure(func) =>
        func(a)

      case And(left, right) =>
        (left(a), right(a)).mapN((_, _) => a)

      case Or(left, right) =>
        left(a) match {
          case Valid(a1)   => Valid(a)
          case Invalid(e1) =>
            right(a) match {
              case Valid(a2)   => Valid(a)
              case Invalid(e2) => Invalid(e1 |+| e2)
            }
        }
    }
}
```

```
final case class And[E, A](
  left: Predicate[E, A],
  right: Predicate[E, A]) extends Predicate[E, A]

final case class Or[E, A](
  left: Predicate[E, A],
  right: Predicate[E, A]) extends Predicate[E, A]

final case class Pure[E, A](
  func: A => Validated[E, A]) extends Predicate[E, A]
```

10.4.2 Checks

We'll use `Check` to represent a structure we build from a `Predicate` that also allows transformation of its input. Implement `Check` with the following interface:

```
sealed trait Check[E, A, B] {
  def apply(a: A): Validated[E, B] =
    ???

  def map[C](func: B => C): Check[E, A, C] =
    ???
}
```

See the solution

What about `flatMap`? The semantics are a bit unclear here. The method is simple enough to declare but it's not so obvious what it means or how we should implement `apply`. The general shape of `flatMap` is shown in Figure 10.7.

How do we relate F in the figure to `Check` in our code? `Check` has *three* type variables while F only has one.

To unify the types we need to fix two of the type parameters. The idiomatic choices are the error type E and the input type A. This gives us the relationships shown in Figure 10.8. In other words, the semantics of applying a `FlatMap` are:

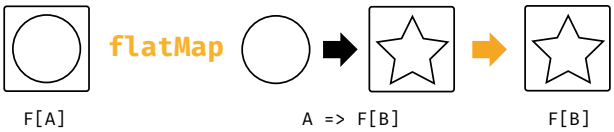


Figure 10.7: Type chart for flatMap

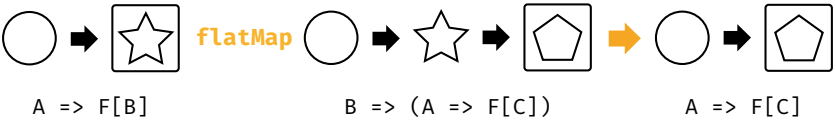


Figure 10.8: Type chart for flatMap applied to Check

- given an input of type `A`, convert to `F[B]`;
- use the output of type `B` to choose a `Check[E, A, C]`;
- return to the *original* input of type `A` and apply it to the chosen check to generate the final result of type `F[C]`.

This is quite an odd method. We can implement it, but it is hard to find a use for it. Go ahead and implement `flatMap` for `Check`, and then we'll see a more generally useful method.

See the solution

We can write a more useful combinator that chains together two `Checks`. The output of the first check is connected to the input of the second. This is analogous to function composition using `andThen`:

```
val f: A => B = ???
val g: B => C = ???
val h: A => C = f andThen g
```

A `Check` is basically a function `A => Validated[E, B]` so we can define an analagous `andThen` method:

```
trait Check[E, A, B] {
  def andThen[C](that: Check[E, B, C]): Check[E, A, C]
}
```

Implement andThen now!

See the solution

10.4.3 Recap

We now have two algebraic data types, `Predicate` and `Check`, and a host of combinators with their associated case class implementations. Look at the following solution for a complete definition of each ADT.

See the solution

We have a complete implementation of `Check` and `Predicate` that do most of what we originally set out to do. However, we are not finished yet. You have probably recognised structure in `Predicate` and `Check` that we can abstract over: `Predicate` has a monoid and `Check` has a monad. Furthermore, in implementing `Check` you might have felt the implementation doesn't do much—all we do is call through to underlying methods on `Predicate` and `Validated`.

There are a lot of ways this library could be cleaned up. However, let's implement some examples to prove to ourselves that our library really does work, and then we'll turn to improving it.

Implement checks for some of the examples given in the introduction:

- A username must contain at least four characters and consist entirely of alphanumeric characters
- An email address must contain an @ sign. Split the string at the @. The string to the left must not be empty. The string to the right must be at least three characters long and contain a dot.

You might find the following predicates useful:

```
import cats.data.{NonEmptyList, Validated}

type Errors = NonEmptyList[String]

def error(s: String): NonEmptyList[String] =
  NonEmptyList(s, Nil)

def longerThan(n: Int): Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must be longer than $n characters"),
    str => str.size > n)

val alphanumeric: Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must be all alphanumeric characters"),
    str => str.forall(_.isLetterOrDigit))

def contains(char: Char): Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must contain the character $char"),
    str => str.contains(char))

def containsOnce(char: Char): Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must contain the character $char only once"),
    str => str.filter(c => c == char).size == 1)
```

See the solution

10.5 Kleislis

We'll finish off this case study by cleaning up the implementation of `Check`. A justifiable criticism of our approach is that we've written a lot of code to do very little. A `Predicate` is essentially a function `A => Validated[E, A]`, and a `Check` is basically a wrapper that lets us compose these functions.

We can abstract `A => Validated[E, A]` to `A => F[B]`, which you'll recognise as the type of function you pass to the `flatMap` method on a monad. Imagine we have the following sequence of operations:

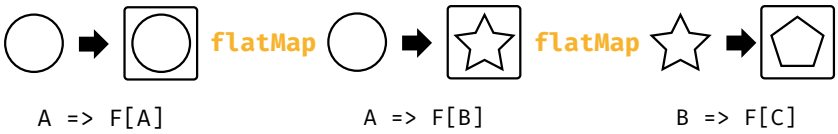


Figure 10.9: Sequencing monadic transforms

- We lift some value into a monad (by using `pure`, for example). This is a function with type `A => F[A]`.
- We then sequence some transformations on the monad using `flatMap`.

We can illustrate this as shown in Figure 10.9. We can also write out this example using the monad API as follows:

```
val aToB: A => F[B] = ???
val bToC: B => F[C] = ???

def example[A, C](a: A): F[C] =
  aToB(a).flatMap(bToC)
```

Recall that `Check` is, in the abstract, allowing us to compose functions of type `A => F[B]`. We can write the above in terms of `andThen` as:

```
val aToC = aToB andThen bToC
```

The result is a (wrapped) function `aToC` of type `A => F[C]` that we can subsequently apply to a value of type `A`.

We have achieved the same thing as the `example` method without having to reference an argument of type `A`. The `andThen` method on `Check` is analogous to function composition, but is composing function `A => F[B]` instead of `A => B`.

The abstract concept of composing functions of type `A => F[B]` has a name: a *Kleisli*.

Cats contains a data type `cats.data.Kleisli` that wraps a function just as `Check` does. `Kleisli` has all the methods of `Check` plus some additional

ones. If `Kleisli` seems familiar to you, then congratulations. You've seen through its disguise and recognised it as another concept from earlier in the book: `Kleisli` is just another name for `ReaderT`.

Here is a simple example using `Kleisli` to transform an integer into a list of integers through three steps:

```
import cats.data.Kleisli
import cats.instances.list._ // for Monad
```

These steps each transform an input `Int` into an output of type `List[Int]`:

```
val step1: Kleisli[List, Int, Int] =
  Kleisli(x => List(x + 1, x - 1))

val step2: Kleisli[List, Int, Int] =
  Kleisli(x => List(x, -x))

val step3: Kleisli[List, Int, Int] =
  Kleisli(x => List(x * 2, x / 2))
```

We can combine the steps into a single pipeline that combines the underlying `Lists` using `flatMap`:

```
val pipeline = step1 andThen step2 andThen step3
```

The result is a function that consumes a single `Int` and returns eight outputs, each produced by a different combination of transformations from `step1`, `step2`, and `step3`:

```
pipeline.run(20)
// res2: List[Int] = List(42, 10, -42, -10, 38, 9, -38, -9)
```

The only notable difference between `Kleisli` and `Check` in terms of API is that `Kleisli` renames our `apply` method to `run`.

Let's replace `Check` with `Kleisli` in our validation examples. To do so we need to make a few changes to `Predicate`. We must be able to convert

a `Predicate` to a function, as `Kleisli` only works with functions. Somewhat more subtly, when we convert a `Predicate` to a function, it should have type `A => Either[E, A]` rather than `A => Validated[E, A]` because `Kleisli` relies on the wrapped function returning a monad.

Add a method to `Predicate` called `run` that returns a function of the correct type. Leave the rest of the code in `Predicate` the same.

See the solution

Now rewrite our username and email validation example in terms of `Kleisli` and `Predicate`. Here are few tips in case you get stuck:

First, remember that the `run` method on `Predicate` takes an implicit parameter. If you call `aPredicate.run(a)` it will try to pass the implicit parameter explicitly. If you want to create a function from a `Predicate` and immediately apply that function, use `aPredicate.run.apply(a)`

Second, type inference can be tricky in this exercise. We found that the following definitions helped us to write code with fewer type declarations.

```
type Result[A] = Either[Errors, A]

type Check[A, B] = Kleisli[Result, A, B]

// Create a check from a function:
def check[A, B](func: A => Result[B]): Check[A, B] =
  Kleisli(func)

// Create a check from a Predicate:
def checkPred[A](pred: Predicate[Errors, A]): Check[A, A] =
  Kleisli[Result, A, A](pred.run)
```

See the solution

We have now written our code entirely in terms of `Kleisli` and `Predicate`, completely removing `Check`. This is a good first step to simplifying our library. There's still plenty more to do, but we have a sophisticated building block from Cats to work with. We'll leave further improvements up to the reader.

10.6 Summary

This case study has been an exercise in removing rather than building abstractions. We started with a fairly complex `Check` type. Once we realised we were conflating two concepts, we separated out `Predicate` leaving us with something that could be implemented with `Kleisli`.

We made several design choices above that reasonable developers may disagree with. Should the method that converts a `Predicate` to a function really be called `run` instead of, say, `toFunction`? Should `Predicate` be a subtype of `Function` to begin with? Many functional programmers prefer to avoid subtyping because it plays poorly with implicit resolution and type inference, but there could be an argument to use it here. As always the best decisions depend on the context in which the library will be used.

Chapter 11

Case Study: CRDTs

In this case study we will explore *Commutative Replicated Data Types (CRDTs)*, a family of data structures that can be used to reconcile eventually consistent data.

We'll start by describing the utility and difficulty of eventually consistent systems, then show how we can use monoids and their extensions to solve the issues that arise. Finally, we will model the solutions in Scala.

Our goal here is to focus on the implementation in Scala of a particular type of CRDT. We're not aiming at a comprehensive survey of all CRDTs. CRDTs are a fast-moving field and we advise you to read the literature to learn about more.

11.1 Eventual Consistency

As soon as a system scales beyond a single machine we have to make a fundamental choice about how we manage data.

One approach is to build a system that is *consistent*, meaning that all machines have the same view of data. For example, if a user changes their password then all machines that store a copy of that password must accept the change before we consider the operation to have completed successfully.

Consistent systems are easy to work with but they have their disadvantages. They tend to have high latency because a single change can result in many messages being sent between machines. They also tend to have relatively low uptime because outages can cut communications between machines creating a *network partition*. When there is a network partition, a consistent system may refuse further updates to prevent inconsistencies across machines.

An alternative approach is an *eventually consistent* system. This means that at any particular point in time machines are allowed to have differing views of data. However, if all machines can communicate and there are no further updates they will eventually all have the same view of data.

Eventually consistent systems require less communication between machines so latency can be lower. A partitioned machine can still accept updates and reconcile its changes when the network is fixed, so systems can also can have better uptime.

The big question is: how do we do this reconciliation between machines? CRDTs provide one approach to the problem.

11.2 The GCounter

Let's look at one particular CRDT implementation. Then we'll attempt to generalise properties to see if we can find a general pattern.

The data structure we will look at is called a *GCounter*. It is a distributed *increment-only* counter that can be used, for example, to count the number of visitors to a web site where requests are served by many web servers.

11.2.1 Simple Counters

To see why a straightforward counter won't work, imagine we have two servers storing a simple count of visitors. Let's call the machines `A` and `B`. Each machine is storing an integer counter and the counters all start at zero as shown in Figure 11.1.

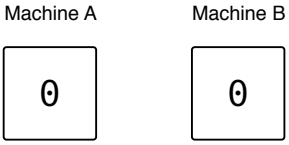


Figure 11.1: Simple counters: initial state

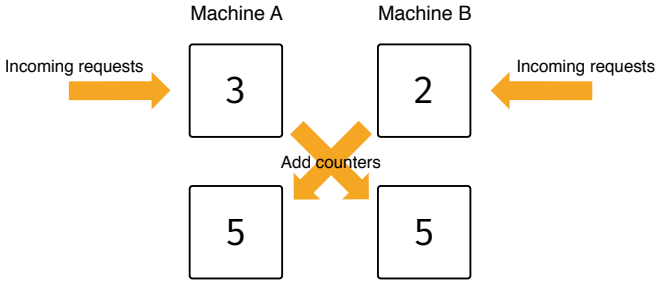


Figure 11.2: Simple counters: first round of requests and reconciliation

Now imagine we receive some web traffic. Our load balancer distributes five incoming requests to A and B, A serving three visitors and B two. The machines have inconsistent views of the system state that they need to *reconcile* to achieve consistency. One reconciliation strategy with simple counters is to exchange counts and add them as shown in Figure 11.2.

So far so good, but things will start to fall apart shortly. Suppose A serves a single visitor, which means we've seen six visitors in total. The machines attempt to reconcile state again using addition leading to the answer shown in Figure 11.3.

This is clearly wrong! The problem is that simple counters don't give us enough information about the history of interactions between the machines. Fortunately we don't need to store the *complete* history to get the correct answer—just a summary of it. Let's look at the GCounter see how it solves this problem.

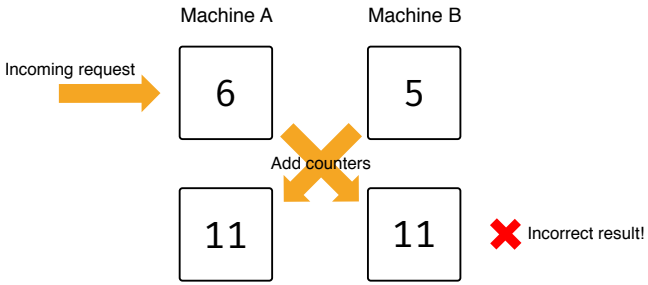


Figure 11.3: Simple counters: second round of requests and (incorrect) reconciliation

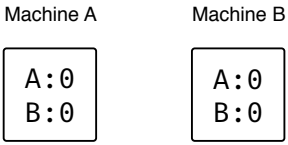


Figure 11.4: GCounter: initial state

11.2.2 GCounters

The first clever idea in the GCounter is to have each machine storing a *separate* counter for every machine it knows about (including itself). In the previous example we had two machines, A and B. In this situation both machines would store a counter for A and a counter for B as shown in Figure 11.4.

The rule with GCounters is that a given machine is only allowed to increment its own counter. If A serves three visitors and B serves two visitors the counters look as shown in Figure 11.5.

When two machines reconcile their counters the rule is to take the largest value stored for each machine. In our example, the result of the first merge will be as shown in Figure 11.6.

Subsequent incoming web requests are handled using the increment-own-counter rule and subsequent merges are handled using the take-maximum-value rule, producing the same correct values for each machine as shown in

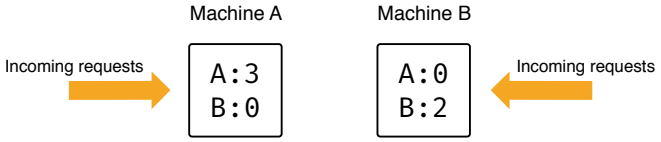


Figure 11.5: GCounter: first round of web requests

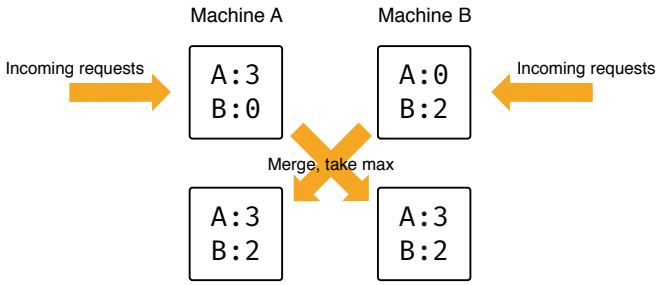


Figure 11.6: GCounter: first reconciliation

Figure 11.7.

GCounters allow each machine to keep an accurate account of the state of the whole system without storing the complete history of interactions. If a machine wants to calculate the total traffic for the whole web site, it sums up all the per-machine counters. The result is accurate or near-accurate depending on how recently we performed a reconciliation. Eventually, regardless of network outages, the system will always converge on a consistent state.

11.2.3 Exercise: GCounter Implementation

We can implement a GCounter with the following interface, where we represent machine IDs as `Strings`.

```
final case class GCounter(counters: Map[String, Int]) {
  def increment(machine: String, amount: Int) =
```

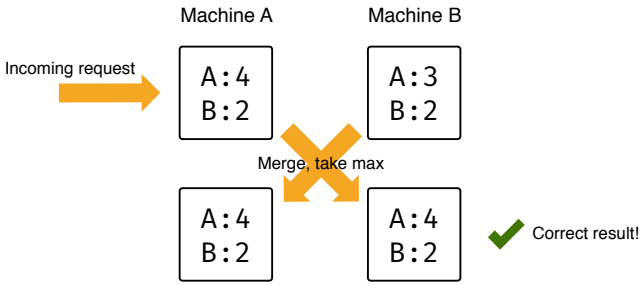


Figure 11.7: GCounter: second reconciliation

```
    ???

  def merge(that: GCounter): GCounter =
    ???

  def total: Int =
    ???
}
```

Finish the implementation!

See the solution

11.3 Generalisation

We've now created a distributed, eventually consistent, increment-only counter. This is a useful achievement but we don't want to stop here. In this section we will attempt to abstract the operations in the GCounter so it will work with more data types than just natural numbers.

The GCounter uses the following operations on natural numbers:

- addition (in `increment` and `total`);
- maximum (in `merge`);
- and the identity element 0 (in `increment` and `merge`).

You can probably guess that there's a monoid in here somewhere, but let's look in more detail at the properties we're relying on.

As a refresher, in Chapter 2 we saw that monoids must satisfy two laws. The binary operation + must be associative:

```
(a + b) + c == a + (b + c)
```

and the empty element must be an identity:

```
0 + a == a + 0 == a
```

We need an identity in `increment` to initialise the counter. We also rely on associativity to ensure the specific sequence of `merges` gives the correct value.

In `total` we implicitly rely on associativity and commutativity to ensure we get the correct value no matter what arbitrary order we choose to sum the per-machine counters. We also implicitly assume an identity, which allows us to skip machines for which we do not store a counter.

The properties of `merge` are a bit more interesting. We rely on commutativity to ensure that machine `A` merging with machine `B` yields the same result as machine `B` merging with machine `A`. We need associativity to ensure we obtain the correct result when three or more machines are merging data. We need an identity element to initialise empty counters. Finally, we need an additional property, called *idempotency*, to ensure that if two machines hold the same data in a per-machine counter, merging data will not lead to an incorrect result. Idempotent operations are ones that return the same result again and again if they are executed multiple times. Formally, a binary operation `max` is idempotent if the following relationship holds:

```
a max a = a
```

Written more compactly, we have:

Method	Identity	Commutative	Associative	Idempotent
`increment`	Y	N	Y	N
`merge`	Y	Y	Y	Y
`total`	Y	Y	Y	N

From this we can see that

- `increment` requires a monoid;
- `total` requires a commutative monoid; and
- `merge` required an idempotent commutative monoid, also called a *bounded semilattice*.

Since `increment` and `get` both use the same binary operation (addition) it's usual to require the same commutative monoid for both.

This investigation demonstrates the powers of thinking about properties or laws of abstractions. Now we have identified these properties we can substitute the natural numbers used in our GCounter with any data type with operations satisfying these properties. A simple example is a set, with the binary operation being union and the identity element the empty set. With this simple substitution of `Int` for `Set[A]` we can create a GSet type.

11.3.1 Implementation

Let's implement this generalisation in code. Remember `increment` and `total` require a commutative monoid and `merge` requires a bounded semilattice (or idempotent commutative monoid).

Cats provides a type class for both `Monoid` and `CommutativeMonoid`, but doesn't provide one for bounded semilattice[1]. That's why we're going to implement our own `BoundedSemiLattice` type class.

```
import cats.kernel.CommutativeMonoid

trait BoundedSemiLattice[A] extends CommutativeMonoid[A] {
  def combine(a1: A, a2: A): A
  def empty: A
}
```

In the implementation above, `BoundedSemiLattice[A]` extends `CommutativeMonoid[A]` because a bounded semilattice is a commutative monoid (a commutative idempotent one, to be exact).

[1] A closely related library called Spire already provides that abstractions.

11.3.2 Exercise: BoundedSemiLattice Instances

Implement `BoundedSemiLattice` type class instances for `Int`s and for `Set`s. The instance for `Int` will technically only hold for non-negative numbers, but you don't need to model non-negativity explicitly in the types.

See the solution

11.3.3 Exercise: Generic GCounter

Using `CommutativeMonoid` and `BoundedSemiLattice`, generalise `GCounter`.

When you implement this, look for opportunities to use methods and syntax on `Monoid` to simplify your implementation. This is a good example of how type class abstractions work at multiple levels in our code. We're using monoids to design a large component—our CRDTs—but they are also useful in the small, simplifying our code and making it shorter and clearer.

See the solution

11.4 Abstracting GCounter to a Type Class

We've created a generic GCounter that works with any value that has instances of `BoundedSemiLattice` and `CommutativeMonoid`. However we're still tied to a particular representation of the map from machine IDs to values. There is no need to have this restriction, and indeed it can be useful to abstract away from it. There are many key-value stores that we want to work with, from a simple `Map` to a relational database.

If we define a `GCounter` type class we can abstract over different concrete implementations. This allows us to, for example, seamlessly substitute an in-memory store for a persistent store when we want to change performance and durability tradeoffs.

There are a number of ways we can implement this. One approach is to define a `GCounter` type class with dependencies on `CommutativeMonoid` and

`BoundedSemiLattice`. We define this as a type class that takes a type constructor with *two* type parameters represent the key and value types of the map abstraction.

```
trait GCounter[F[_,_],K, V] {
  def increment(f: F[K, V])(k: K, v: V)
        (implicit m: CommutativeMonoid[V]): F[K, V]

  def merge(f1: F[K, V], f2: F[K, V])
        (implicit b: BoundedSemiLattice[V]): F[K, V]

  def total(f: F[K, V])
        (implicit m: CommutativeMonoid[V]): V
}

object GCounter {
  def apply[F[_,_], K, V]
        (implicit counter: GCounter[F, K, V]) =
    counter
}
```

Try defining an instance of this type class for `Map`. You should be able to reuse your code from the case class version of `GCounter` with some minor modifications.

See the solution

You should be able to use your instance as follows:

```
import cats.instances.int._ // for Monoid

val g1 = Map("a" -> 7, "b" -> 3)
val g2 = Map("a" -> 2, "b" -> 5)

val counter = GCounter[Map, String, Int]

val merged = counter.merge(g1, g2)
// merged: Map[String,Int] = Map(a -> 7, b -> 5)

val total  = counter.total(merged)
// total: Int = 12
```

The implementation strategy for the type class instance is a bit unsatisfying. Although the structure of the implementation will be the same for most instances we define, we won't get any code reuse.

11.5 Abstracting a Key Value Store

One solution is to capture the idea of a key-value store within a type class, and then generate `GCounter` instances for any type that has a `KeyValueStore` instance. Here's the code for such a type class:

```
trait KeyValueStore[F[_,_]] {
  def put[K, V](f: F[K, V])(k: K, v: V): F[K, V]

  def get[K, V](f: F[K, V])(k: K): Option[V]

  def getOrElse[K, V](f: F[K, V])(k: K, default: V): V =
    get(f)(k).getOrElse(default)

  def values[K, V](f: F[K, V]): List[V]
}
```

Implement your own instance for `Map`.

See the solution

With our type class in place we can implement syntax to enhance data types for which we have instances:

```
implicit class KvsOps[F[_,_], K, V](f: F[K, V]) {
  def put(key: K, value: V)
        (implicit kvs: KeyValueStore[F]): F[K, V] =
    kvs.put(f)(key, value)

  def get(key: K)(implicit kvs: KeyValueStore[F]): Option[V] =
    kvs.get(f)(key)

  def getOrElse(key: K, default: V)
        (implicit kvs: KeyValueStore[F]): V =
    kvs.getOrElse(f)(key, default)
```

```
  def values(implicit kvs: KeyValueStore[F]): List[V] =
    kvs.values(f)
}
```

Now we can generate `GCounter` instances for any data type that has instances of `KeyValueStore` and `CommutativeMonoid` using an `implicit def`:

```
implicit def gcounterInstance[F[_,_], K, V]
    (implicit kvs: KeyValueStore[F], km: CommutativeMonoid[F[K, V]]) =
  new GCounter[F, K, V] {
    def increment(f: F[K, V])(key: K, value: V)
          (implicit m: CommutativeMonoid[V]): F[K, V] = {
      val total = f.getOrElse(key, m.empty) |+| value
      f.put(key, total)
    }

    def merge(f1: F[K, V], f2: F[K, V])
          (implicit b: BoundedSemiLattice[V]): F[K, V] =
      f1 |+| f2

    def total(f: F[K, V])(implicit m: CommutativeMonoid[V]): V =
      f.values.combineAll
  }
```

The complete code for this case study is quite long, but most of it is boilerplate setting up syntax for operations on the type class. We can cut down on this using compiler plugins such as Simulacrum and Kind Projector.

11.6 Summary

In this case study we've seen how we can use type classes to model a simple CRDT, the GCounter, in Scala. Our implementation gives us a lot of flexibility and code reuse: we aren't tied to the data type we "count", nor to the data type that maps machine IDs to counters.

The focus in this case study has been on using the tools that Scala provides, not on exploring CRDTs. There are many other CRDTs, some of which operate

in a similar manner to the GCounter, and some of which have very different implementations. A fairly recent survey gives a good overview of many of the basic CRDTs. However this is an active area of research and we encourage you to read the recent publications in the field if CRDTs and eventually consistency interest you.

Part III

Solutions to Exercises

Appendix A

Solutions for: Introduction

A.1 Printable Library

These steps define the three main components of our type class. First we define `Printable`—the *type class* itself:

```
trait Printable[A] {
  def format(value: A): String
}
```

Then we define some default *instances* of `Printable` and package them in `PrintableInstances`:

```
object PrintableInstances {
  implicit val stringPrintable = new Printable[String] {
    def format(input: String) = input
  }

  implicit val intPrintable = new Printable[Int] {
    def format(input: Int) = input.toString
  }
}
```

Finally we define an *interface* object, `Printable`:

```
object Printable {
  def format[A](input: A)(implicit p: Printable[A]): String =
    p.format(input)

  def print[A](input: A)(implicit p: Printable[A]): Unit =
    println(format(input))
}
```

Return to the exercise

A.2 Printable Library Part 2

This is a standard use of the type class pattern. First we define a set of custom data types for our application:

```
final case class Cat(name: String, age: Int, color: String)
```

Then we define type class instances for the types we care about. These either go into the companion object of `Cat` or a separate object to act as a namespace:

```
import PrintableInstances._

implicit val catPrintable = new Printable[Cat] {
  def format(cat: Cat) = {
    val name  = Printable.format(cat.name)
    val age   = Printable.format(cat.age)
    val color = Printable.format(cat.color)
    s"$name is a $age year-old $color cat."
  }
}
```

Finally, we use the type class by bringing the relevant instances into scope and using interface object/syntax. If we defined the instances in companion objects Scala brings them into scope for us automatically. Otherwise we use an `import` to access them:

```
val cat = Cat("Garfield", 38, "ginger and black")
// cat: Cat = Cat(Garfield,38,ginger and black)

Printable.print(cat)
// Garfield is a 38 year-old ginger and black cat.
```

Return to the exercise

A.3 Printable Library Part 3

First we define an `implicit class` containing our extension methods:

```
object PrintableSyntax {
  implicit class PrintableOps[A](value: A) {
    def format(implicit p: Printable[A]): String =
      Printable.format(value)

    def print(implicit p: Printable[A]): Unit =
      Printable.print(value)
  }
}
```

With `PrintableOps` in scope, we can call the imaginary `print` and `format` methods on any value for which Scala can locate an implicit instance of `Printable`:

```
import PrintableSyntax._

Cat("Garfield", 38, "ginger and black").print
// Garfield is a 38 year-old ginger and black cat.
```

We get a compile error if we haven't defined an instance of `Printable` for the relevant type:

```
import java.util.Date

new Date().print
// <console>:23: error: could not find implicit value for parameter p:
```

```
      Printable[java.util.Date]
//        new Date().print
//                  ^
```

Return to the exercise

A.4 Cat Show

First let's import everything we need from Cats: the `Show` type class, the instances for `Int` and `String`, and the interface syntax:

```
import cats.Show
import cats.instances.int._    // for Show
import cats.instances.string._ // for Show
import cats.syntax.show._      // for show
```

Our definition of `Cat` remains the same:

```
final case class Cat(name: String, age: Int, color: String)
```

In the companion object we replace our `Printable` with an instance of `Show` using one of the definition helpers discussed above:

```
implicit val catShow = Show.show[Cat] { cat =>
  val name  = cat.name.show
  val age   = cat.age.show
  val color = cat.color.show
  s"$name is a $age year-old $color cat."
}
```

Finally, we use the `Show` interface syntax to print our instance of `Cat`:

```
println(Cat("Garfield", 38, "ginger and black").show)
// Garfield is a 38 year-old ginger and black cat.
```

Return to the exercise

A.5 Equality, Liberty, and Felinity

First we need our Cats imports. In this exercise we'll be using the Eq type class and the Eq interface syntax. We'll bring instances of Eq into scope as we need them below:

```
import cats.Eq
import cats.syntax.eq._ // for ===
```

Our `Cat` class is the same as ever:

```
final case class Cat(name: String, age: Int, color: String)
```

We bring the Eq instances for `Int` and `String` into scope for the implementation of `Eq[Cat]`:

```
import cats.instances.int._    // for Eq
import cats.instances.string._ // for Eq

implicit val catEqual: Eq[Cat] =
  Eq.instance[Cat] { (cat1, cat2) =>
    (cat1.name  === cat2.name ) &&
    (cat1.age   === cat2.age  ) &&
    (cat1.color === cat2.color)
  }
```

Finally, we test things out in a sample application:

```
val cat1 = Cat("Garfield",   38, "orange and black")
// cat1: Cat = Cat(Garfield,38,orange and black)

val cat2 = Cat("Heathcliff", 32, "orange and black")
// cat2: Cat = Cat(Heathcliff,32,orange and black)

cat1 === cat2
// res17: Boolean = false

cat1 =!= cat2
// res18: Boolean = true
```

```
import cats.instances.option._ // for Eq

val optionCat1 = Option(cat1)
// optionCat1: Option[Cat] = Some(Cat(Garfield,38,orange and black))

val optionCat2 = Option.empty[Cat]
// optionCat2: Option[Cat] = None

optionCat1 === optionCat2
// res19: Boolean = false

optionCat1 =!= optionCat2
// res20: Boolean = true
```

Return to the exercise

Appendix B

Solutions for: Monoids and Semigroups

B.1 The Truth About Monoids

There are four monoids for `Boolean`! First, we have *and* with operator && and identity `true`:

```
implicit val booleanAndMonoid: Monoid[Boolean] =
  new Monoid[Boolean] {
    def combine(a: Boolean, b: Boolean) = a && b
    def empty = true
  }
```

Second, we have *or* with operator `||` and identity `false`:

```
implicit val booleanOrMonoid: Monoid[Boolean] =
  new Monoid[Boolean] {
    def combine(a: Boolean, b: Boolean) = a || b
    def empty = false
  }
```

Third, we have *exclusive or* with identity `false`:

```
implicit val booleanEitherMonoid: Monoid[Boolean] =
  new Monoid[Boolean] {
    def combine(a: Boolean, b: Boolean) =
      (a && !b) || (!a && b)

    def empty = false
  }
```

Finally, we have *exclusive nor* (the negation of exclusive or) with identity `true`:

```
implicit val booleanXnorMonoid: Monoid[Boolean] =
  new Monoid[Boolean] {
    def combine(a: Boolean, b: Boolean) =
      (!a || b) && (a || !b)

    def empty = true
  }
```

Showing that the identity law holds in each case is straightforward. Similarly associativity of the `combine` operation can be shown by enumerating the cases.

Return to the exercise

B.2 All Set for Monoids

Set union forms a monoid along with the empty set:

```
implicit def setUnionMonoid[A]: Monoid[Set[A]] =
  new Monoid[Set[A]] {
    def combine(a: Set[A], b: Set[A]) = a union b
    def empty = Set.empty[A]
  }
```

We need to define `setUnionMonoid` as a method rather than a value so we can accept the type parameter `A`. The type parameter allows us to use the same definition to summon `Monoids` for `Sets` of any type of data:

```
val intSetMonoid = Monoid[Set[Int]]
val strSetMonoid = Monoid[Set[String]]

intSetMonoid.combine(Set(1, 2), Set(2, 3))
// res2: Set[Int] = Set(1, 2, 3)

strSetMonoid.combine(Set("A", "B"), Set("B", "C"))
// res3: Set[String] = Set(A, B, C)
```

Set intersection forms a semigroup, but doesn't form a monoid because it has no identity element:

```
implicit def setIntersectionSemigroup[A]: Semigroup[Set[A]] =
  new Semigroup[Set[A]] {
    def combine(a: Set[A], b: Set[A]) =
      a intersect b
  }
```

Set complement and set difference are not associative, so they cannot be considered for either monoids or semigroups. However, symmetric difference (the union less the intersection) does also form a monoid with the empty set:

```
implicit def symDiffMonoid[A]: Monoid[Set[A]] =
  new Monoid[Set[A]] {
    def combine(a: Set[A], b: Set[A]): Set[A] =
      (a diff b) union (b diff a)
    def empty: Set[A] = Set.empty
  }
```

Return to the exercise

B.3 Adding All The Things

We can write the addition as a simple `foldLeft` using `0` and the + operator:

```
def add(items: List[Int]): Int =
  items.foldLeft(0)(_ + _)
```

We can alternatively write the fold using `Monoids`, although there's not a compelling use case for this yet:

```
import cats.Monoid
import cats.instances.int._    // for Monoid
import cats.syntax.semigroup._ // for |+|

def add(items: List[Int]): Int =
  items.foldLeft(Monoid[Int].empty)(_ |+| _)
```

Return to the exercise

B.4 Adding All The Things Part 2

Now there is a use case for `Monoids`. We need a single method that adds `Ints` and instances of `Option[Int]`. We can write this as a generic method that accepts an implicit `Monoid` as a parameter:

```
import cats.Monoid
import cats.instances.int._    // for Monoid
import cats.syntax.semigroup._ // for |+|

def add[A](items: List[A])(implicit monoid: Monoid[A]): A =
  items.foldLeft(monoid.empty)(_ |+| _)
```

We can optionally use Scala's *context bound* syntax to write the same code in a friendlier way:

```
def add[A: Monoid](items: List[A]): A =
  items.foldLeft(Monoid[A].empty)(_ |+| _)
```

We can use this code to add values of type `Int` and `Option[Int]` as requested:

```
import cats.instances.int._ // for Monoid

add(List(1, 2, 3))
// res9: Int = 6

import cats.instances.option._ // for Monoid

add(List(Some(1), None, Some(2), None, Some(3)))
// res10: Option[Int] = Some(6)
```

Note that if we try to add a list consisting entirely of `Some` values, we get a compile error:

```
add(List(Some(1), Some(2), Some(3)))
// <console>:61: error: could not find implicit value for evidence
     parameter of type cats.Monoid[Some[Int]]
//        add(List(Some(1), Some(2), Some(3)))
//            ^
```

This happens because the inferred type of the list is `List[Some[Int]]`, while Cats will only generate a `Monoid` for `Option[Int]`. We'll see how to get around this in a moment.

Return to the exercise

B.5 Adding All The Things Part 3

Easy—we simply define a monoid instance for `Order`!

```
implicit val monoid: Monoid[Order] = new Monoid[Order] {
  def combine(o1: Order, o2: Order) =
    Order(
      o1.totalCost + o2.totalCost,
      o1.quantity + o2.quantity
    )

  def empty = Order(0, 0)
}
```

Return to the exercise

Appendix C

Solutions for: Functors

C.1 Branching out with Functors

The semantics are similar to writing a `Functor` for `List`. We recurse over the data structure, applying the function to every `Leaf` we find. The functor laws intuitively require us to retain the same structure with the same pattern of `Branch` and `Leaf` nodes:

```
import cats.Functor

implicit val treeFunctor: Functor[Tree] =
  new Functor[Tree] {
    def map[A, B](tree: Tree[A])(func: A => B): Tree[B] =
      tree match {
        case Branch(left, right) =>
          Branch(map(left)(func), map(right)(func))
        case Leaf(value) =>
          Leaf(func(value))
      }
  }
```

Let's use our `Functor` to transform some `Trees`:

```
Branch(Leaf(10), Leaf(20)).map(_ * 2)
// <console>:42: error: value map is not a member of wrapper.Branch[
    Int]
//       Branch(Leaf(10), Leaf(20)).map(_ * 2)
//                                  ^
```

Oops! This falls foul of the same invariance problem we discussed in Section 1.6.1. The compiler can find a `Functor` instance for `Tree` but not for `Branch` or `Leaf`. Let's add some smart constructors to compensate:

```
object Tree {
  def branch[A](left: Tree[A], right: Tree[A]): Tree[A] =
    Branch(left, right)

  def leaf[A](value: A): Tree[A] =
    Leaf(value)
}
```

Now we can use our `Functor` properly:

```
Tree.leaf(100).map(_ * 2)
// res10: wrapper.Tree[Int] = Leaf(200)

Tree.branch(Tree.leaf(10), Tree.leaf(20)).map(_ * 2)
// res11: wrapper.Tree[Int] = Branch(Leaf(20),Leaf(40))
```

Return to the exercise

C.2 Showing off with Contramap

Here's a working implementation. We call `func` to turn the `B` into an `A` and then use our original `Printable` to turn the `A` into a `String`. In a small show of sleight of hand we use a `self` alias to distinguish the outer and inner `Printables`:

```
trait Printable[A] {
  self =>

  def format(value: A): String

  def contramap[B](func: B => A): Printable[B] =
    new Printable[B] {
      def format(value: B): String =
        self.format(func(value))
    }
}

def format[A](value: A)(implicit p: Printable[A]): String =
  p.format(value)
```

Return to the exercise

C.3 Showing off with Contramap Part 2

To make the instance generic across all types of `Box`, we base it on the `Printable` for the type inside the `Box`. We can either write out the complete definition by hand:

```
implicit def boxPrintable[A](implicit p: Printable[A]) =
  new Printable[Box[A]] {
    def format(box: Box[A]): String =
      p.format(box.value)
  }
```

or use `contramap` to base the new instance on the implicit parameter:

```
implicit def boxPrintable[A](implicit p: Printable[A]) =
  p.contramap[Box[A]](_.value)
```

Using `contramap` is much simpler, and conveys the functional programming approach of building solutions by combining simple building blocks using pure functional combinators.

Return to the exercise

C.4 Transformative Thinking with imap

Here's a working implementation:

```
trait Codec[A] {
  def encode(value: A): String
  def decode(value: String): A

  def imap[B](dec: A => B, enc: B => A): Codec[B] = {
    val self = this
    new Codec[B] {
      def encode(value: B): String =
        self.encode(enc(value))

      def decode(value: String): B =
        dec(self.decode(value))
    }
  }
}
```

Return to the exercise

C.5 Transformative Thinking with imap Part 2

We can implement this using the `imap` method of `stringCodec`:

```
implicit val doubleCodec: Codec[Double] =
  stringCodec.imap[Double](_.toDouble, _.toString)
```

Return to the exercise

C.6 Transformative Thinking with imap Part 3

We need a generic `Codec` for `Box[A]` for any given `A`. We create this by calling `imap` on a `Codec[A]`, which we bring into scope using an implicit parameter:

```
implicit def boxCodec[A](implicit c: Codec[A]): Codec[Box[A]] =
  c.imap[Box[A]](Box(_), _.value)
```

Return to the exercise

Appendix D

Solutions for: Monads

D.1 Getting Func-y

At first glance this seems tricky, but if we follow the types we'll see there's only one solution. We are passed a `value` of type `F[A]`. Given the tools available there's only one thing we can do: call `flatMap`:

```
trait Monad[F[_]] {
  def pure[A](value: A): F[A]

  def flatMap[A, B](value: F[A])(func: A => F[B]): F[B]

  def map[A, B](value: F[A])(func: A => B): F[B] =
    flatMap(value)(a => ???)
}
```

We need a function of type `A => F[B]` as the second parameter. We have two function building blocks available: the `func` parameter of type `A => B` and the `pure` function of type `A => F[A]`. Combining these gives us our result:

```
trait Monad[F[_]] {
  def pure[A](value: A): F[A]
```

```
  def flatMap[A, B](value: F[A])(func: A => F[B]): F[B]

  def map[A, B](value: F[A])(func: A => B): F[B] =
    flatMap(value)(a => pure(func(a)))
}
```

Return to the exercise

D.2 Monadic Secret Identities

Let's start by defining the method signatures:

```
import cats.Id

def pure[A](value: A): Id[A] =
  ???

def map[A, B](initial: Id[A])(func: A => B): Id[B] =
  ???

def flatMap[A, B](initial: Id[A])(func: A => Id[B]): Id[B] =
  ???
```

Now let's look at each method in turn. The `pure` operation creates an `Id[A]` from an `A`. But `A` and `Id[A]` are the same type! All we have to do is return the initial value:

```
def pure[A](value: A): Id[A] =
  value

pure(123)
// res10: cats.Id[Int] = 123
```

The `map` method takes a parameter of type `Id[A]`, applies a function of type `A => B`, and returns an `Id[B]`. But `Id[A]` is simply `A` and `Id[B]` is simply `B`! All we have to do is call the function—no packing or unpacking required:

```
def map[A, B](initial: Id[A])(func: A => B): Id[B] =
  func(initial)

map(123)(_ * 2)
// res11: cats.Id[Int] = 246
```

The final punch line is that, once we strip away the `Id` type constructors, `flatMap` and `map` are actually identical:

```
def flatMap[A, B](initial: Id[A])(func: A => Id[B]): Id[B] =
  func(initial)
// flatMap: [A, B](initial: cats.Id[A])(func: A => cats.Id[B])cats.Id[
    B]

flatMap(123)(_ * 2)
// res12: cats.Id[Int] = 246
```

This ties in with our understanding of functors and monads as sequencing type classes. Each type class allows us to sequence operations ignoring some kind of complication. In the case of `Id` there is no complication, making `map` and `flatMap` the same thing.

Notice that we haven't had to write type annotations in the method bodies above. The compiler is able to interpret values of type `A` as `Id[A]` and vice versa by the context in which they are used.

The only restriction we've seen to this is that Scala cannot unify types and type constructors when searching for implicits. Hence our need to re-type `Int` as `Id[Int]` in the call to `sumSquare` at the opening of this section:

```
sumSquare(3 : Id[Int], 4 : Id[Int])
```

Return to the exercise

D.3 What is Best?

This is an open question. It's also kind of a trick question—the answer depends on the semantics we're looking for. Some points to ponder:

- Error recovery is important when processing large jobs. We don't want to run a job for a day and then find it failed on the last element.
- Error reporting is equally important. We need to know what went wrong, not just that something went wrong.
- In a number of cases, we want to collect all the errors, not just the first one we encountered. A typical example is validating a web form. It's a far better experience to report all errors to the user when they submit a form than to report them one at a time.

Return to the exercise

D.4 Safer Folding using Eval

The easiest way to fix this is to introduce a helper method called `foldRightEval`. This is essentially our original method with every occurrence of `B` replaced with `Eval[B]`, and a call to `Eval.defer` to protect the recursive call:

```
import cats.Eval

def foldRightEval[A, B](as: List[A], acc: Eval[B])
    (fn: (A, Eval[B]) => Eval[B]): Eval[B] =
  as match {
    case head :: tail =>
      Eval.defer(fn(head, foldRightEval(tail, acc)(fn)))
    case Nil =>
      acc
  }
```

We can redefine `foldRight` simply in terms of `foldRightEval` and the resulting method is stack safe:

```
def foldRight[A, B](as: List[A], acc: B)(fn: (A, B) => B): B =
  foldRightEval(as, Eval.now(acc)) { (a, b) =>
    b.map(fn(a, _))
```

```
  }.value

foldRight((1 to 100000).toList, 0L)(_ + _)
// res22: Long = 5000050000
```

Return to the exercise

D.5 Show Your Working

We'll start by defining a type alias for `Writer` so we can use it with `pure` syntax:

```
import cats.data.Writer
import cats.syntax.applicative._ // for pure

type Logged[A] = Writer[Vector[String], A]

42.pure[Logged]
// res13: Logged[Int] = WriterT((Vector(),42))
```

We'll import the `tell` syntax as well:

```
import cats.syntax.writer._ // for tell

Vector("Message").tell
// res14: cats.data.Writer[scala.collection.immutable.Vector[String],
     Unit] = WriterT((Vector(Message),()))
```

Finally, we'll import the `Semigroup` instance for `Vector`. We need this to `map` and `flatMap` over `Logged`:

```
import cats.instances.vector._ // for Monoid

41.pure[Logged].map(_ + 1)
// res15: cats.data.WriterT[cats.Id,Vector[String],Int] = WriterT((
     Vector(),42))
```

With these in scope, the definition of `factorial` becomes:

```
def factorial(n: Int): Logged[Int] =
  for {
    ans <- if(n == 0) {
             1.pure[Logged]
           } else {
             slowly(factorial(n - 1).map(_ * n))
           }
    _   <- Vector(s"fact $n $ans").tell
  } yield ans
```

When we call `factorial`, we now have to `run` the return value to extract the log and our factorial:

```
val (log, res) = factorial(5).run
// log: Vector[String] = Vector(fact 0 1, fact 1 1, fact 2 2, fact 3
     6, fact 4 24, fact 5 120)
// res: Int = 120
```

We can run several `factorials` in parallel as follows, capturing their logs independently without fear of interleaving:

```
val Vector((logA, ansA), (logB, ansB)) =
  Await.result(Future.sequence(Vector(
    Future(factorial(3).run),
    Future(factorial(5).run)
  )), 5.seconds)
// logA: Vector[String] = Vector(fact 0 1, fact 1 1, fact 2 2, fact 3
     6)
// ansA: Int = 6
// logB: Vector[String] = Vector(fact 0 1, fact 1 1, fact 2 2, fact 3
     6, fact 4 24, fact 5 120)
// ansB: Int = 120
```

Return to the exercise

D.6 Hacking on Readers

Our type alias fixes the `Db` type but leaves the result type flexible:

```
type DbReader[A] = Reader[Db, A]
```

Return to the exercise

D.7 Hacking on Readers Part 2

Remember: the idea is to leave injecting the configuration until last. This means setting up functions that accept the config as a parameter and check it against the concrete user info we have been given:

```
def findUsername(userId: Int): DbReader[Option[String]] =
  Reader(db => db.usernames.get(userId))

def checkPassword(
      username: String,
      password: String): DbReader[Boolean] =
  Reader(db => db.passwords.get(username).contains(password))
```

Return to the exercise

D.8 Hacking on Readers Part 3

As you might expect, here we use `flatMap` to chain `findUsername` and `checkPassword`. We use `pure` to lift a `Boolean` to a `DbReader[Boolean]` when the username is not found:

```
import cats.syntax.applicative._ // for pure

def checkLogin(
      userId: Int,
      password: String): DbReader[Boolean] =
  for {
    username   <- findUsername(userId)
    passwordOk <- username.map { username =>
                    checkPassword(username, password)
                  }.getOrElse {
```

```
                  false.pure[DbReader]
                }
  } yield passwordOk
```

Return to the exercise

D.9 Post-Order Calculator

The stack operation required is different for operators and operands. For clarity we'll implement `evalOne` in terms of two helper functions, one for each case:

```
def evalOne(sym: String): CalcState[Int] =
  sym match {
    case "+" => operator(_ + _)
    case "-" => operator(_ - _)
    case "*" => operator(_ * _)
    case "/" => operator(_ / _)
    case num => operand(num.toInt)
  }
```

Let's look at `operand` first. All we have to do is push a number onto the stack. We also return the operand as an intermediate result:

```
def operand(num: Int): CalcState[Int] =
  State[List[Int], Int] { stack =>
    (num :: stack, num)
  }
```

The `operator` function is a little more complex. We have to pop two operands off the stack (having the second operand at the top of the stack) and push the result in their place. The code can fail if the stack doesn't have enough operands on it, but the exercise description allows us to throw an exception in this case:

```
def operator(func: (Int, Int) => Int): CalcState[Int] =
  State[List[Int], Int] {
    case b :: a :: tail =>
      val ans = func(a, b)
      (ans :: tail, ans)

    case _ =>
      sys.error("Fail!")
  }
```

Return to the exercise

D.10 Post-Order Calculator Part 2

We implement `evalAll` by folding over the input. We start with a pure `CalcState` that returns `0` if the list is empty. We `flatMap` at each stage, ignoring the intermediate results as we saw in the example:

```
import cats.syntax.applicative._ // for pure

def evalAll(input: List[String]): CalcState[Int] =
  input.foldLeft(0.pure[CalcState]) { (a, b) =>
    a.flatMap(_ => evalOne(b))
  }
```

Return to the exercise

D.11 Post-Order Calculator Part 3

We've done all the hard work now. All we need to do is split the input into terms and call `runA` and `value` to unpack the result:

```
def evalInput(input: String): Int =
  evalAll(input.split(" ").toList).runA(Nil).value

evalInput("1 2 + 3 4 + *")
```

```
// res8: Int = 21
```

Return to the exercise

D.12 Branching out Further with Monads

The code for `flatMap` is similar to the code for `map`. Again, we recurse down the structure and use the results from `func` to build a new `Tree`.

The code for `tailRecM` is fairly complex regardless of whether we make it tail-recursive or not.

If we follow the types, the non-tail-recursive solution falls out:

```
import cats.Monad

implicit val treeMonad = new Monad[Tree] {
  def pure[A](value: A): Tree[A] =
    Leaf(value)

  def flatMap[A, B](tree: Tree[A])
      (func: A => Tree[B]): Tree[B] =
    tree match {
      case Branch(l, r) =>
        Branch(flatMap(l)(func), flatMap(r)(func))
      case Leaf(value)  =>
        func(value)
    }

 def tailRecM[A, B](a: A)
     (func: A => Tree[Either[A, B]]): Tree[B] =
   flatMap(func(a)) {
     case Left(value) =>
       tailRecM(value)(func)
     case Right(value) =>
       Leaf(value)
   }
}
```

The solution above is perfectly fine for this exercise. Its only downside is that Cats cannot make guarantees about stack safety.

The tail-recursive solution is much harder to write. We adapted this solution from this Stack Overflow post by Nazarii Bardiuk. It involves an explicit depth first traversal of the tree, maintaining an open list of nodes to visit and a `closed` list of nodes to use to reconstruct the tree:

```
import cats.Monad

implicit val treeMonad = new Monad[Tree] {
  def pure[A](value: A): Tree[A] =
    Leaf(value)

  def flatMap[A, B](tree: Tree[A])
      (func: A => Tree[B]): Tree[B] =
    tree match {
      case Branch(l, r) =>
        Branch(flatMap(l)(func), flatMap(r)(func))
      case Leaf(value)  =>
        func(value)
    }

  def tailRecM[A, B](arg: A)
      (func: A => Tree[Either[A, B]]): Tree[B] = {
    @tailrec
    def loop(
          open: List[Tree[Either[A, B]]],
          closed: List[Option[Tree[B]]]): List[Tree[B]] =
      open match {
        case Branch(l, r) :: next =>
          loop(l :: r :: next, None :: closed)

        case Leaf(Left(value)) :: next =>
          loop(func(value) :: next, closed)

        case Leaf(Right(value)) :: next =>
          loop(next, Some(pure(value)) :: closed)

        case Nil =>
          closed.foldLeft(Nil: List[Tree[B]]) { (acc, maybeTree) =>
            maybeTree.map(_ :: acc).getOrElse {
              val left :: right :: tail = acc
              branch(left, right) :: tail
            }
```

```
          }
      }

    loop(List(func(arg)), Nil).head
  }
}
```

Regardless of which version of `tailRecM` we define, we can use our `Monad` to `flatMap` and `map` on `Trees`:

```
import cats.syntax.functor._ // for map
import cats.syntax.flatMap._ // for flatMap

branch(leaf(100), leaf(200)).
  flatMap(x => branch(leaf(x - 1), leaf(x + 1)))
// res3: wrapper.Tree[Int] = Branch(Branch(Leaf(99),Leaf(101)),Branch(
    Leaf(199),Leaf(201)))
```

We can also transform `Trees` using for comprehensions:

```
for {
  a <- branch(leaf(100), leaf(200))
  b <- branch(leaf(a - 10), leaf(a + 10))
  c <- branch(leaf(b - 1), leaf(b + 1))
} yield c
// res4: wrapper.Tree[Int] = Branch(Branch(Branch(Leaf(89),Leaf(91)),
    Branch(Leaf(109),Leaf(111))),Branch(Branch(Leaf(189),Leaf(191)),
    Branch(Leaf(209),Leaf(211))))
```

The monad for `Option` provides fail-fast semantics. The monad for `List` provides concatenation semantics. What are the semantics of `flatMap` for a binary tree? Every node in the tree has the potential to be replaced with a whole subtree, producing a kind of "growing" or "feathering" behaviour, reminiscent of list concatenation along two axes.

Return to the exercise

Appendix E

Solutions for: Monad Transformers

E.1 Monads: Transform and Roll Out

This is a relatively simple combination. We want `Future` on the outside and `Either` on the inside, so we build from the inside out using an `EitherT` of `Future`:

```
import cats.data.EitherT
import scala.concurrent.Future

type Response[A] = EitherT[Future, String, A]
```

Return to the exercise

E.2 Monads: Transform and Roll Out Part 2

```
import cats.instances.future._ // for Monad
import cats.syntax.flatMap._   // for flatMap
import scala.concurrent.ExecutionContext.Implicits.global
```

```
type Response[A] = EitherT[Future, String, A]

def getPowerLevel(ally: String): Response[Int] = {
  powerLevels.get(ally) match {
    case Some(avg) => EitherT.right(Future(avg))
    case None      => EitherT.left(Future(s"$ally unreachable"))
  }
}
```

Return to the exercise

E.3 Monads: Transform and Roll Out Part 3

We request the power level from each ally and use `map` and `flatMap` to combine the results:

```
def canSpecialMove(ally1: String, ally2: String): Response[Boolean] =
  for {
    power1 <- getPowerLevel(ally1)
    power2 <- getPowerLevel(ally2)
  } yield (power1 + power2) > 15
```

Return to the exercise

E.4 Monads: Transform and Roll Out Part 4

We use the `value` method to unpack the monad stack and `Await` and `fold` to unpack the `Future` and `Either`:

```
import scala.concurrent.Await
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._

def tacticalReport(ally1: String, ally2: String): String = {
  val stack = canSpecialMove(ally1, ally2).value

  Await.result(stack, 1.second) match {
```

```
    case Left(msg) =>
      s"Comms error: $msg"
    case Right(true)  =>
      s"$ally1 and $ally2 are ready to roll out!"
    case Right(false) =>
      s"$ally1 and $ally2 need a recharge."
  }
}
```

Return to the exercise

Appendix F

Solutions for: Semigroupal and Applicative

F.1 The Product of Monads

We can implement `product` in terms of `map` and `flatMap` like so:

```
import cats.syntax.flatMap._ // for flatMap
import cats.syntax.functor._ // for map

def product[M[_]: Monad, A, B](x: M[A], y: M[B]): M[(A, B)] =
  x.flatMap(a => y.map(b => (a, b)))
```

Unsurprisingly, this code is equivalent to a for comprehension:

```
def product[M[_]: Monad, A, B](x: M[A], y: M[B]): M[(A, B)] =
  for {
    a <- x
    b <- y
  } yield (a, b)
```

The semantics of `flatMap` are what give rise to the behaviour for `List` and `Either`:

```
import cats.instances.list._ // for Semigroupal

product(List(1, 2), List(3, 4))
// res12: List[(Int, Int)] = List((1,3), (1,4), (2,3), (2,4))

type ErrorOr[A] = Either[Vector[String], A]

product[ErrorOr, Int, Int](
  Left(Vector("Error 1")),
  Left(Vector("Error 2"))
)
// res13: ErrorOr[(Int, Int)] = Left(Vector(Error 1))
```

Return to the exercise

F.2 Form Validation

We'll be using `Either` and `Validated` so we'll start with some imports:

```
import cats.data.Validated

type FormData = Map[String, String]
type FailFast[A] = Either[List[String], A]
type FailSlow[A] = Validated[List[String], A]
```

The `getValue` rule extracts a `String` from the form data. We'll be using it in sequence with rules for parsing `Int`s and checking values, so we'll define it to return an `Either`:

```
def getValue(name: String)(data: FormData): FailFast[String] =
  data.get(name).
    toRight(List(s"$name field not specified"))
```

We can create and use an instance of `getValue` as follows:

```
val getName = getValue("name") _
// getName: FormData => FailFast[String] = <function1>

getName(Map("name" -> "Dade Murphy"))
// res29: FailFast[String] = Right(Dade Murphy)
```

In the event of a missing field, our instance returns an error message containing an appropriate field name:

```
getName(Map())
// res30: FailFast[String] = Left(List(name field not specified))
```

Return to the exercise

F.3 Form Validation Part 2

We'll use `Either` again here. We use `Either.catchOnly` to consume the `NumberFormatException` from `toInt`, and we use `leftMap` to turn it into an error message:

```
import cats.syntax.either._ // for catchOnly

type NumFmtExn = NumberFormatException

def parseInt(name: String)(data: String): FailFast[Int] =
  Either.catchOnly[NumFmtExn](data.toInt).
    leftMap(_ => List(s"$name must be an integer"))
```

Note that our solution accepts an extra parameter to name the field we're parsing. This is useful for creating better error messages, but it's fine if you leave it out in your code.

If we provide valid input, `parseInt` converts it to an `Int`:

```
parseInt("age")("11")
// res33: FailFast[Int] = Right(11)
```

If we provide erroneous input, we get a useful error message:

```
parseInt("age")("foo")
// res34: FailFast[Int] = Left(List(age must be an integer))
```

Return to the exercise

F.4 Form Validation Part 3

These definitions use the same patterns as above:

```
def nonBlank(name: String)(data: String): FailFast[String] =
  Right(data).
    ensure(List(s"$name cannot be blank"))(_.nonEmpty)

def nonNegative(name: String)(data: Int): FailFast[Int] =
  Right(data).
    ensure(List(s"$name must be non-negative"))(_ >= 0)
```

Here are some examples of use:

```
nonBlank("name")("Dade Murphy")
// res36: FailFast[String] = Right(Dade Murphy)

nonBlank("name")("")
// res37: FailFast[String] = Left(List(name cannot be blank))

nonNegative("age")(11)
// res38: FailFast[Int] = Right(11)

nonNegative("age")(-1)
// res39: FailFast[Int] = Left(List(age must be non-negative))
```

Return to the exercise

F.5 Form Validation Part 4

We use `flatMap` to combine the rules sequentially:

```
def readName(data: FormData): FailFast[String] =
  getValue("name")(data).
    flatMap(nonBlank("name"))

def readAge(data: FormData): FailFast[Int] =
  getValue("age")(data).
    flatMap(nonBlank("age")).
    flatMap(parseInt("age")).
    flatMap(nonNegative("age"))
```

The rules pick up all the error cases we've seen so far:

```
readName(Map("name" -> "Dade Murphy"))
// res41: FailFast[String] = Right(Dade Murphy)

readName(Map("name" -> ""))
// res42: FailFast[String] = Left(List(name cannot be blank))

readName(Map())
// res43: FailFast[String] = Left(List(name field not specified))

readAge(Map("age" -> "11"))
// res44: FailFast[Int] = Right(11)

readAge(Map("age" -> "-1"))
// res45: FailFast[Int] = Left(List(age must be non-negative))

readAge(Map())
// res46: FailFast[Int] = Left(List(age field not specified))
```

Return to the exercise

F.6 Form Validation Part 5

We can do this by switching from `Either` to `Validated` and using apply syntax:

```
import cats.instances.list._ // for Semigroupal
import cats.syntax.apply._   // for mapN

def readUser(data: FormData): FailSlow[User] =
  (
    readName(data).toValidated,
    readAge(data).toValidated
  ).mapN(User.apply)

readUser(Map("name" -> "Dave", "age" -> "37"))
// res48: FailSlow[User] = Valid(User(Dave,37))

readUser(Map("age" -> "-1"))
// res49: FailSlow[User] = Invalid(List(name field not specified, age
     must be non-negative))
```

The need to switch back and forth between `Either` and `Validated` is annoying. The choice of whether to use `Either` or `Validated` as a default is determined by context. In application code, we typically find areas that favour accumulating semantics and areas that favour fail-fast semantics. We pick the data type that best suits our need and switch to the other as necessary in specific situations.

Return to the exercise

Appendix G

Solutions for: Foldable and Traverse

G.1 Reflecting on Folds

Folding from left to right reverses the list:

```
List(1, 2, 3).foldLeft(List.empty[Int])((a, i) => i :: a)
// res6: List[Int] = List(3, 2, 1)
```

Folding right to left copies the list, leaving the order intact:

```
List(1, 2, 3).foldRight(List.empty[Int])((i, a) => i :: a)
// res7: List[Int] = List(1, 2, 3)
```

Note that we have to carefully specify the type of the accumulator to avoid a type error. We use `List.empty[Int]` to avoid inferring the accumulator type as `Nil.type` or `List[Nothing]`:

```
List(1, 2, 3).foldRight(Nil)(_ :: _)
// <console>:13: error: type mismatch;
//  found   : List[Int]
```

```
//  required: scala.collection.immutable.Nil.type
//        List(1, 2, 3).foldRight(Nil)(_ :: _)
//                                     ^
```

Return to the exercise

G.2 Scaf-fold-ing Other Methods

Here are the solutions:

```
def map[A, B](list: List[A])(func: A => B): List[B] =
  list.foldRight(List.empty[B]) { (item, accum) =>
    func(item) :: accum
  }

map(List(1, 2, 3))(_ * 2)
// res9: List[Int] = List(2, 4, 6)

def flatMap[A, B](list: List[A])(func: A => List[B]): List[B] =
  list.foldRight(List.empty[B]) { (item, accum) =>
    func(item) ::: accum
  }

flatMap(List(1, 2, 3))(a => List(a, a * 10, a * 100))
// res10: List[Int] = List(1, 10, 100, 2, 20, 200, 3, 30, 300)

def filter[A](list: List[A])(func: A => Boolean): List[A] =
  list.foldRight(List.empty[A]) { (item, accum) =>
    if(func(item)) item :: accum else accum
  }

filter(List(1, 2, 3))(_ % 2 == 1)
// res11: List[Int] = List(1, 3)
```

We've provided two definitions of `sum`, one using `scala.math.Numeric` (which recreates the built-in functionality accurately)...

```
import scala.math.Numeric

def sumWithNumeric[A](list: List[A])
      (implicit numeric: Numeric[A]): A =
  list.foldRight(numeric.zero)(numeric.plus)

sumWithNumeric(List(1, 2, 3))
// res13: Int = 6
```

and one using `cats.Monoid` (which is more appropriate to the content of this book):

```
import cats.Monoid

def sumWithMonoid[A](list: List[A])
      (implicit monoid: Monoid[A]): A =
  list.foldRight(monoid.empty)(monoid.combine)

import cats.instances.int._ // for Monoid

sumWithMonoid(List(1, 2, 3))
// res16: Int = 6
```

Return to the exercise

G.3 Traversing with Vectors

The argument is of type `List[Vector[Int]]`, so we're using the `Applicative` for `Vector` and the return type is going to be `Vector[List[Int]]`.

`Vector` is a monad, so its semigroupal `combine` function is based on `flatMap`. We'll end up with a `Vector` of `Lists` of all the possible combinations of `List(1, 2)` and `List(3, 4)`:

```
listSequence(List(Vector(1, 2), Vector(3, 4)))
```

```
// res14: scala.collection.immutable.Vector[List[Int]] = Vector(List
    (1, 3), List(1, 4), List(2, 3), List(2, 4))
```

Return to the exercise

G.4 Traversing with Vectors Part 2

With three items in the input list, we end up with combinations of three `Int`s: one from the first item, one from the second, and one from the third:

```
listSequence(List(Vector(1, 2), Vector(3, 4), Vector(5, 6)))
// res16: scala.collection.immutable.Vector[List[Int]] = Vector(List
    (1, 3, 5), List(1, 3, 6), List(1, 4, 5), List(1, 4, 6), List(2,
    3, 5), List(2, 3, 6), List(2, 4, 5), List(2, 4, 6))
```

Return to the exercise

G.5 Traversing with Options

The arguments to `listTraverse` are of types `List[Int]` and `Int => Option[Int]`, so the return type is `Option[List[Int]]`. Again, `Option` is a monad, so the semigroupal `combine` function follows from `flatMap`. The semantics are therefore fail-fast error handling: if all inputs are even, we get a list of outputs. Otherwise we get `None`:

```
process(List(2, 4, 6))
// res20: Option[List[Int]] = Some(List(2, 4, 6))

process(List(1, 2, 3))
// res21: Option[List[Int]] = None
```

Return to the exercise

G.6 Traversing with Validated

The return type here is `ErrorsOr[List[Int]]`, which expands to `Validated[List[String], List[Int]]`. The semantics for semigroupal `combine` on validated are accumulating error handling, so the result is either a list of even `Int`s, or a list of errors detailing which `Int`s failed the test:

```
process(List(2, 4, 6))
// res26: ErrorsOr[List[Int]] = Valid(List(2, 4, 6))

process(List(1, 2, 3))
// res27: ErrorsOr[List[Int]] = Invalid(List(1 is not even, 3 is not
    even))
```

Return to the exercise

Appendix H

Solutions for: Case Study: Testing Asynchronous Code

H.1 Abstracting over Type Constructors

Here's the implementation:

```
import scala.language.higherKinds
import cats.Id

trait UptimeClient[F[_]] {
  def getUptime(hostname: String): F[Int]
}

trait RealUptimeClient extends UptimeClient[Future] {
  def getUptime(hostname: String): Future[Int]
}

trait TestUptimeClient extends UptimeClient[Id] {
  def getUptime(hostname: String): Id[Int]
}
```

Note that, because `Id[A]` is just a simple alias for `A`, we don't need to refer to the type in `TestUptimeClient` as `Id[Int]`—we can simply write `Int` instead:

```
trait TestUptimeClient extends UptimeClient[Id] {
  def getUptime(hostname: String): Int
}
```

Of course, technically speaking we don't need to redeclare `getUptime` in `RealUptimeClient` or `TestUptimeClient`. However, writing everything out helps illustrate the technique.

Return to the exercise

H.2 Abstracting over Type Constructors Part 2

The final code is similar to our original implementation of `TestUptimeClient`, except we no longer need the call to `Future.successful`:

```
class TestUptimeClient(hosts: Map[String, Int])
  extends UptimeClient[Id] {
  def getUptime(hostname: String): Int =
    hosts.getOrElse(hostname, 0)
}
```

Return to the exercise

H.3 Abstracting over Monads

The code should look like this:

```
class UptimeService[F[_]](client: UptimeClient[F]) {
  def getTotalUptime(hostnames: List[String]): F[Int] =
    ???
    // hostnames.traverse(client.getUptime).map(_.sum)
}
```

Return to the exercise

H.4 Abstracting over Monads Part 2

We can write this as an implicit parameter:

```
import cats.Applicative
import cats.syntax.functor._ // for map

class UptimeService[F[_]](client: UptimeClient[F])
    (implicit a: Applicative[F]) {

  def getTotalUptime(hostnames: List[String]): F[Int] =
    hostnames.traverse(client.getUptime).map(_.sum)
}
```

or more tersely as a context bound:

```
class UptimeService[F[_]: Applicative]
    (client: UptimeClient[F]) {

  def getTotalUptime(hostnames: List[String]): F[Int] =
    hostnames.traverse(client.getUptime).map(_.sum)
}
```

Note that we need to import `cats.syntax.functor` as well as `cats.Applicative`. This is because we're switching from using `future.map` to the Cats' generic extension method that requires an implicit `Functor` parameter.

Return to the exercise

Appendix I

Solutions for: Case Study: Map-Reduce

I.1 Implementing foldMap

```
import cats.Monoid

/** Single-threaded map-reduce function.
  * Maps `func` over `values` and reduces using a `Monoid[B]`.
  */
def foldMap[A, B: Monoid](values: Vector[A])(func: A => B): B =
  ???
```

Return to the exercise

I.2 Implementing foldMap Part 2

We have to modify the type signature to accept a `Monoid` for `B`. With that change we can use the `Monoid` `empty` and `|+|` syntax as described in Section 2.5.3:

```
import cats.Monoid
import cats.instances.int._    // for Monoid
import cats.instances.string._ // for Monoid
import cats.syntax.semigroup._ // for |+|

def foldMap[A, B : Monoid](as: Vector[A])(func: A => B): B =
  as.map(func).foldLeft(Monoid[B].empty)(_ |+| _)
```

We can make a slight alteration to this code to do everything in one step:

```
def foldMap[A, B : Monoid](as: Vector[A])(func: A => B): B =
  as.foldLeft(Monoid[B].empty)(_ |+| func(_))
```

Return to the exercise

I.3 Implementing parallelFoldMap

Here is an annotated solution that splits out each `map` and `fold` into a separate line of code:

```
import scala.concurrent.duration.Duration

def parallelFoldMap[A, B: Monoid]
      (values: Vector[A])
      (func: A => B): Future[B] = {
  // Calculate the number of items to pass to each CPU:
  val numCores  = Runtime.getRuntime.availableProcessors
  val groupSize = (1.0 * values.size / numCores).ceil.toInt

  // Create one group for each CPU:
  val groups: Iterator[Vector[A]] =
    values.grouped(groupSize)

  // Create a future to foldMap each group:
  val futures: Iterator[Future[B]] =
    groups map { group =>
      Future {
        group.foldLeft(Monoid[B].empty)(_ |+| func(_))
      }
```

```
    }

  // foldMap over the groups to calculate a final result:
  Future.sequence(futures) map { iterable =>
    iterable.foldLeft(Monoid[B].empty)(_ |+| _)
  }
}

val result: Future[Int] =
  parallelFoldMap((1 to 1000000).toVector)(identity)

Await.result(result, 1.second)
// res19: Int = 1784293664
```

We can re-use our definition of `foldMap` for a more concise solution. Note that the local maps and reduces in steps 3 and 4 of Figure 9.4 are actually equivalent to a single call to `foldMap`, shortening the entire algorithm as follows:

```
def parallelFoldMap[A, B: Monoid]
      (values: Vector[A])
      (func: A => B): Future[B] = {
  val numCores  = Runtime.getRuntime.availableProcessors
  val groupSize = (1.0 * values.size / numCores).ceil.toInt

  val groups: Iterator[Vector[A]] =
    values.grouped(groupSize)

  val futures: Iterator[Future[B]] =
    groups.map(group => Future(foldMap(group)(func)))

  Future.sequence(futures) map { iterable =>
    iterable.foldLeft(Monoid[B].empty)(_ |+| _)
  }
}

val result: Future[Int] =
  parallelFoldMap((1 to 1000000).toVector)(identity)

Await.result(result, 1.second)
// res21: Int = 1784293664
```

Return to the exercise

I.4 parallelFoldMap with more Cats

We'll restate all of the necessary imports for completeness:

```
import cats.Monoid
import cats.Foldable
import cats.Traverse

import cats.instances.int._    // for Monoid
import cats.instances.future._ // for Applicative and Monad
import cats.instances.vector._ // for Foldable and Traverse

import cats.syntax.semigroup._ // for |+|
import cats.syntax.foldable._  // for combineAll and foldMap
import cats.syntax.traverse._  // for traverse

import scala.concurrent._
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global
```

Here's the implementation of `parallelFoldMap` delegating as much of the method body to Cats as possible:

```
def parallelFoldMap[A, B: Monoid]
      (values: Vector[A])
      (func: A => B): Future[B] = {
  val numCores  = Runtime.getRuntime.availableProcessors
  val groupSize = (1.0 * values.size / numCores).ceil.toInt

  values
    .grouped(groupSize)
    .toVector
    .traverse(group => Future(group.toVector.foldMap(func)))
    .map(_.combineAll)
}

val future: Future[Int] =
  parallelFoldMap((1 to 1000).toVector)(_ * 1000)

Await.result(future, 1.second)
// res3: Int = 500500000
```

The call to `vector.grouped` returns an `Iterable[Iterator[Int]]`. We sprinkle calls to `toVector` through the code to convert the data back to a form that Cats can understand. The call to `traverse` creates a `Future[Vector[Int]]` containing one `Int` per batch. The call to `map` then combines the `match` using the `combineAll` method from `Foldable`.

Return to the exercise

Appendix J

Solutions for: Case Study: Data Validation

J.1 Basic Combinators

We need a `Semigroup` for `E`. Then we can combine values of `E` using the `combine` method or its associated `|+|` syntax:

```
import cats.Semigroup
import cats.instances.list._   // for Semigroup
import cats.syntax.semigroup._ // for |+|

val semigroup = Semigroup[List[String]]

// Combination using methods on Semigroup
semigroup.combine(List("Badness"), List("More badness"))
// res2: List[String] = List(Badness, More badness)

// Combination using Semigroup syntax
List("Oh noes") |+| List("Fail happened")
// res4: List[String] = List(Oh noes, Fail happened)
```

Note we don't need a full `Monoid` because we don't need the identity element. We should always try to keep our constraints as small as possible!

Return to the exercise

J.2 Basic Combinators Part 2

We want to report all the errors we can, so we should prefer *not* short-circuiting whenever possible.

In the case of the and method, the two checks we're combining are independent of one another. We can always run both rules and combine any errors we see.

Return to the exercise

J.3 Basic Combinators Part 3

There are at least two implementation strategies.

In the first we represent checks as functions. The `Check` data type becomes a simple wrapper for a function that provides our library of combinator methods. For the sake of disambiguation, we'll call this implementation `CheckF`:

```
import cats.Semigroup
import cats.syntax.either._    // for asLeft and asRight
import cats.syntax.semigroup._ // for |+|

final case class CheckF[E, A](func: A => Either[E, A]) {
  def apply(a: A): Either[E, A] =
    func(a)

  def and(that: CheckF[E, A])
        (implicit s: Semigroup[E]): CheckF[E, A] =
    CheckF { a =>
      (this(a), that(a)) match {
        case (Left(e1),  Left(e2))  => (e1 |+| e2).asLeft
        case (Left(e),   Right(a))  => e.asLeft
        case (Right(a),  Left(e))   => e.asLeft
        case (Right(a1), Right(a2)) => a.asRight
      }
```

```
    }
}
```

Let's test the behaviour we get. First we'll setup some checks:

```
import cats.instances.list._ // for Semigroup

val a: CheckF[List[String], Int] =
  CheckF { v =>
    if(v > 2) v.asRight
    else List("Must be > 2").asLeft
  }

val b: CheckF[List[String], Int] =
  CheckF { v =>
    if(v < -2) v.asRight
    else List("Must be < -2").asLeft
  }

val check: CheckF[List[String], Int] =
  a and b
```

Now run the check with some data:

```
check(5)
// res8: Either[List[String],Int] = Left(List(Must be < -2))

check(0)
// res9: Either[List[String],Int] = Left(List(Must be > 2, Must be <
     -2))
```

Excellent! Everything works as expected! We're running both checks and accumulating errors as required.

What happens if we try to create checks that fail with a type that we can't accumulate? For example, there is no `Semigroup` instance for `Nothing`. What happens if we create instances of `CheckF[Nothing, A]`?

```
val a: CheckF[Nothing, Int] =
  CheckF(v => v.asRight)

val b: CheckF[Nothing, Int] =
  CheckF(v => v.asRight)
```

We can create checks just fine but when we come to combine them we get an error we might expect:

```
val check = a and b
// <console>:31: error: could not find implicit value for parameter s:
     cats.Semigroup[Nothing]
//        val check = a and b
//                    ^
```

Now let's see another implementation strategy. In this approach we model checks as an algebraic data type, with an explicit data type for each combinator. We'll call this implementation `Check`:

```
sealed trait Check[E, A] {
  def and(that: Check[E, A]): Check[E, A] =
    And(this, that)

  def apply(a: A)(implicit s: Semigroup[E]): Either[E, A] =
    this match {
      case Pure(func) =>
        func(a)

      case And(left, right) =>
        (left(a), right(a)) match {
          case (Left(e1),  Left(e2))  => (e1 |+| e2).asLeft
          case (Left(e),   Right(a))  => e.asLeft
          case (Right(a),  Left(e))   => e.asLeft
          case (Right(a1), Right(a2)) => a.asRight
        }
    }
}

final case class And[E, A](
  left: Check[E, A],
```

```
  right: Check[E, A]) extends Check[E, A]

final case class Pure[E, A](
  func: A => Either[E, A]) extends Check[E, A]
```

Let's see an example:

```
val a: Check[List[String], Int] =
  Pure { v =>
    if(v > 2) v.asRight
    else List("Must be > 2").asLeft
  }

val b: Check[List[String], Int] =
  Pure { v =>
    if(v < -2) v.asRight
    else List("Must be < -2").asLeft
  }

val check: Check[List[String], Int] =
  a and b
```

While the ADT implementation is more verbose than the function wrapper implementation, it has the advantage of cleanly separating the structure of the computation (the ADT instance we create) from the process that gives it meaning (the `apply` method). From here we have a number of options:

- inspect and refactor checks after they are created;
- move the `apply` "interpreter" out into its own module;
- implement alternative interpreters providing other functionality (for example visualizing checks).

Because of its flexibility, we will use the ADT implementation for the rest of this case study.

Return to the exercise

J.4 Basic Combinators Part 4

The implementation of `apply` for `And` is using the pattern for applicative functors. `Either` has an `Applicative` instance, but it doesn't have the semantics we want. It fails fast instead of accumulating errors.

If we want to accumulate errors `Validated` is a more appropriate abstraction. As a bonus, we get more code reuse because we can lean on the applicative instance of `Validated` in the implementation of `apply`.

Here's the complete implementation:

```
import cats.Semigroup
import cats.data.Validated
import cats.syntax.semigroup._ // for |+|
import cats.syntax.apply._     // for mapN

sealed trait Check[E, A] {
  def and(that: Check[E, A]): Check[E, A] =
    And(this, that)

  def apply(a: A)(implicit s: Semigroup[E]): Validated[E, A] =
    this match {
      case Pure(func) =>
        func(a)

      case And(left, right) =>
        (left(a), right(a)).mapN((_, _) => a)
    }
}

final case class And[E, A](
  left: Check[E, A],
  right: Check[E, A]) extends Check[E, A]

final case class Pure[E, A](
  func: A => Validated[E, A]) extends Check[E, A]
```

Return to the exercise

J.5 Basic Combinators Part 5

This reuses the same technique for and. We have to do a bit more work in the `apply` method. Note that it's OK to short-circuit in this case because the choice of rules is implicit in the semantics of "or".

```
import cats.Semigroup
import cats.data.Validated
import cats.syntax.semigroup._ // for |+|
import cats.syntax.apply._     // for mapN
import cats.data.Validated._   // for Valid and Invalid

sealed trait Check[E, A] {
  def and(that: Check[E, A]): Check[E, A] =
    And(this, that)

  def or(that: Check[E, A]): Check[E, A] =
    Or(this, that)

  def apply(a: A)(implicit s: Semigroup[E]): Validated[E, A] =
    this match {
      case Pure(func) =>
        func(a)

      case And(left, right) =>
        (left(a), right(a)).mapN((_, _) => a)

      case Or(left, right) =>
        left(a) match {
          case Valid(a)    => Valid(a)
          case Invalid(e1) =>
            right(a) match {
              case Valid(a)    => Valid(a)
              case Invalid(e2) => Invalid(e1 |+| e2)
            }
        }
    }
}

final case class And[E, A](
  left: Check[E, A],
```

```
  right: Check[E, A]) extends Check[E, A]

final case class Or[E, A](
  left: Check[E, A],
  right: Check[E, A]) extends Check[E, A]

final case class Pure[E, A](
  func: A => Validated[E, A]) extends Check[E, A]
```

Return to the exercise

J.6 Checks

If you follow the same strategy as `Predicate` you should be able to create code similar to the below:

```
import cats.Semigroup
import cats.data.Validated

sealed trait Check[E, A, B] {
  def apply(in: A)(implicit s: Semigroup[E]): Validated[E, B]

  def map[C](f: B => C): Check[E, A, C] =
    Map[E, A, B, C](this, f)
}

object Check {
  def apply[E, A](pred: Predicate[E, A]): Check[E, A, A] =
    Pure(pred)
}

final case class Map[E, A, B, C](
  check: Check[E, A, B],
  func: B => C) extends Check[E, A, C] {

  def apply(in: A)(implicit s: Semigroup[E]): Validated[E, C] =
    check(in).map(func)
}

final case class Pure[E, A](
```

```
  pred: Predicate[E, A]) extends Check[E, A, A] {

  def apply(in: A)(implicit s: Semigroup[E]): Validated[E, A] =
    pred(in)
}
```

Return to the exercise

J.7 Checks Part 2

It's the same implementation strategy as before with one wrinkle: `Validated` doesn't have a `flatMap` method. To implement `flatMap` we must momentarily switch to `Either` and then switch back to `Validated`. The `withEither` method on `Validated` does exactly this. From here we can just follow the types to implement `apply`.

```
import cats.Semigroup
import cats.data.Validated

sealed trait Check[E, A, B] {
  def apply(in: A)(implicit s: Semigroup[E]): Validated[E, B]

  def flatMap[C](f: B => Check[E, A, C]) =
    FlatMap[E, A, B, C](this, f)

  // other methods...
}

final case class FlatMap[E, A, B, C](
  check: Check[E, A, B],
  func: B => Check[E, A, C]) extends Check[E, A, C] {

  def apply(a: A)(implicit s: Semigroup[E]): Validated[E, C] =
    check(a).withEither(_.flatMap(b => func(b)(a).toEither))
}

// other data types...
```

Return to the exercise

J.8 Checks Part 3

Here's a minimal definition of andThen and its corresponding AndThen class:

```
sealed trait Check[E, A, B] {
  import Check._

  def apply(in: A)(implicit s: Semigroup[E]): Validated[E, B]

  def andThen[C](that: Check[E, B, C]): Check[E, A, C] =
    AndThen[E, A, B, C](this, that)
}

final case class AndThen[E, A, B, C](
  check1: Check[E, A, B],
  check2: Check[E, B, C]) extends Check[E, A, C] {

  def apply(a: A)(implicit s: Semigroup[E]): Validated[E, C] =
    check1(a).withEither(_.flatMap(b => check2(b).toEither))
}
```

Return to the exercise

J.9 Recap

Here's our final implementaton, including some tidying and repackaging of the code:

```
import cats.Semigroup
import cats.data.Validated
import cats.data.Validated._   // for Valid and Invalid
import cats.syntax.semigroup._ // for |+|
import cats.syntax.apply._     // for mapN
import cats.syntax.validated._ // for valid and invalid
```

Here is our complete implementation of `Predicate`, including the `and` and `or` combinators and a `Predicate.apply` method to create a `Predicate` from a function:

```
sealed trait Predicate[E, A] {
  import Predicate._

  def and(that: Predicate[E, A]): Predicate[E, A] =
    And(this, that)

  def or(that: Predicate[E, A]): Predicate[E, A] =
    Or(this, that)

  def apply(a: A)(implicit s: Semigroup[E]): Validated[E, A] =
    this match {
      case Pure(func) =>
        func(a)

      case And(left, right) =>
        (left(a), right(a)).mapN((_, _) => a)

      case Or(left, right) =>
        left(a) match {
          case Valid(a1)   => Valid(a)
          case Invalid(e1) =>
            right(a) match {
              case Valid(a2)   => Valid(a)
              case Invalid(e2) => Invalid(e1 |+| e2)
            }
        }
    }
}

object Predicate {
  final case class And[E, A](
    left: Predicate[E, A],
    right: Predicate[E, A]) extends Predicate[E, A]

  final case class Or[E, A](
    left: Predicate[E, A],
    right: Predicate[E, A]) extends Predicate[E, A]

  final case class Pure[E, A](
    func: A => Validated[E, A]) extends Predicate[E, A]

  def apply[E, A](f: A => Validated[E, A]): Predicate[E, A] =
    Pure(f)
```

```
  def lift[E, A](err: E, fn: A => Boolean): Predicate[E, A] =
    Pure(a => if(fn(a)) a.valid else err.invalid)
}
```

Here is a complete implementation of `Check`. Due to a type inference bug in Scala's pattern matching, we've switched to implementing `apply` using inheritance:

```
sealed trait Check[E, A, B] {
  import Check._

  def apply(in: A)(implicit s: Semigroup[E]): Validated[E, B]

  def map[C](f: B => C): Check[E, A, C] =
    Map[E, A, B, C](this, f)

  def flatMap[C](f: B => Check[E, A, C]) =
    FlatMap[E, A, B, C](this, f)

  def andThen[C](next: Check[E, B, C]): Check[E, A, C] =
    AndThen[E, A, B, C](this, next)
}

object Check {
  final case class Map[E, A, B, C](
    check: Check[E, A, B],
    func: B => C) extends Check[E, A, C] {

    def apply(a: A)
        (implicit s: Semigroup[E]): Validated[E, C] =
      check(a) map func
  }

  final case class FlatMap[E, A, B, C](
    check: Check[E, A, B],
    func: B => Check[E, A, C]) extends Check[E, A, C] {

    def apply(a: A)
        (implicit s: Semigroup[E]): Validated[E, C] =
      check(a).withEither(_.flatMap(b => func(b)(a).toEither))
  }
```

```
  final case class AndThen[E, A, B, C](
    check: Check[E, A, B],
    next: Check[E, B, C]) extends Check[E, A, C] {

    def apply(a: A)
        (implicit s: Semigroup[E]): Validated[E, C] =
      check(a).withEither(_.flatMap(b => next(b).toEither))
  }

  final case class Pure[E, A, B](
    func: A => Validated[E, B]) extends Check[E, A, B] {

    def apply(a: A)
        (implicit s: Semigroup[E]): Validated[E, B] =
      func(a)
  }

  final case class PurePredicate[E, A](
    pred: Predicate[E, A]) extends Check[E, A, A] {

    def apply(a: A)
        (implicit s: Semigroup[E]): Validated[E, A] =
      pred(a)
  }

  def apply[E, A](pred: Predicate[E, A]): Check[E, A, A] =
    PurePredicate(pred)

  def apply[E, A, B]
      (func: A => Validated[E, B]): Check[E, A, B] =
    Pure(func)
}
```

Return to the exercise

J.10 Recap Part 2

Here's our reference solution. Implementing this required more thought than we expected. Switching between `Check` and `Predicate` at appropriate places felt a bit like guesswork till we got the rule into our heads that `Predicate`

doesn't transform its input. With this rule in mind things went fairly smoothly. In later sections we'll make some changes that make the library easier to use.

```
import cats.data.{NonEmptyList, Validated}
import cats.syntax.apply._     // for mapN
import cats.syntax.validated._ // for valid and invalid
```

Here's the implementation of `checkUsername`:

```
// A username must contain at least four characters
// and consist entirely of alphanumeric characters

val checkUsername: Check[Errors, String, String] =
  Check(longerThan(3) and alphanumeric)
```

And here's the implementation of `checkEmail`, built up from a number of smaller components:

```
// An email address must contain a single `@` sign.
// Split the string at the `@`.
// The string to the left must not be empty.
// The string to the right must be
// at least three characters long and contain a dot.

val splitEmail: Check[Errors, String, (String, String)] =
  Check(_.split('@') match {
    case Array(name, domain) =>
      (name, domain).validNel[String]

    case other =>
      "Must contain a single @ character".
        invalidNel[(String, String)]
  })

val checkLeft: Check[Errors, String, String] =
  Check(longerThan(0))

val checkRight: Check[Errors, String, String] =
  Check(longerThan(3) and contains('.'))
```

```
val joinEmail: Check[Errors, (String, String), String] =
  Check { case (l, r) =>
    (checkLeft(l), checkRight(r)).mapN(_ + "@" + _)
  }

val checkEmail: Check[Errors, String, String] =
  splitEmail andThen joinEmail
```

Finally, here's a check for a `User` that depends on `checkUsername` and `checkEmail`:

```
final case class User(username: String, email: String)

def createUser(
      username: String,
      email: String): Validated[Errors, User] =
  (checkUsername(username), checkEmail(email)).mapN(User)
```

We can check our work by creating a couple of example users:

```
createUser("Noel", "noel@underscore.io")
// res14: cats.data.Validated[wrapper.Errors,User] = Valid(User(Noel,
     noel@underscore.io))

createUser("", "dave@underscore@io")
// res15: cats.data.Validated[wrapper.Errors,User] = Invalid(
     NonEmptyList(Must be longer than 3 characters, Must contain a
     single @ character))
```

One distinct disadvantage of our example is that it doesn't tell us *where* the errors came from. We can either achieve that through judicious manipulation of error messages, or we can modify our library to track error locations as well as messages. Tracking error locations is outside the scope of this case study, so we'll leave this as an exercise to the reader.

Return to the exercise

J.11 Kleislis

Here's an abbreviated definition of `run`. Like `apply`, the method must accept an implicit `Semigroup`:

```
import cats.Semigroup
import cats.data.Validated

sealed trait Predicate[E, A] {
  def run(implicit s: Semigroup[E]): A => Either[E, A] =
    (a: A) => this(a).toEither

  def apply(a: A): Validated[E, A] =
    ??? // etc...

  // other methods...
}
```

Return to the exercise

J.12 Kleislis Part 2

Working around limitations of type inference can be quite frustrating when writing this code, Working out when to convert between `Predicates`, functions, and `Validated`, and `Either` simplifies things, but the process is still complex:

```
import cats.data.{Kleisli, NonEmptyList, Validated}
import cats.instances.either._   // for Semigroupal
import cats.instances.list._     // for Monad
```

Here is the preamble we suggested in the main text of the case study:

```
type Errors = NonEmptyList[String]

def error(s: String): NonEmptyList[String] =
  NonEmptyList(s, Nil)
```

```
type Result[A] = Either[Errors, A]

type Check[A, B] = Kleisli[Result, A, B]

def check[A, B](func: A => Result[B]): Check[A, B] =
  Kleisli(func)

def checkPred[A](pred: Predicate[Errors, A]): Check[A, A] =
  Kleisli[Result, A, A](pred.run)
```

Our base predicate definitions are essenitally unchanged:

```
def longerThan(n: Int): Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must be longer than $n characters"),
    str => str.size > n)

val alphanumeric: Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must be all alphanumeric characters"),
    str => str.forall(_.isLetterOrDigit))

def contains(char: Char): Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must contain the character $char"),
    str => str.contains(char))

def containsOnce(char: Char): Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must contain the character $char only once"),
    str => str.filter(c => c == char).size == 1)
```

Our username and email examples are slightly different in that we make use of `check()` and `checkPred()` in different situations:

```
val checkUsername: Check[String, String] =
  checkPred(longerThan(3) and alphanumeric)

val splitEmail: Check[String, (String, String)] =
  check(_.split('@') match {
```

```
    case Array(name, domain) =>
      Right((name, domain))

    case other =>
      Left(error("Must contain a single @ character"))
  })

val checkLeft: Check[String, String] =
  checkPred(longerThan(0))

val checkRight: Check[String, String] =
  checkPred(longerThan(3) and contains('.'))

val joinEmail: Check[(String, String), String] =
  check {
    case (l, r) =>
      (checkLeft(l), checkRight(r)).mapN(_ + "@" + _)
  }

val checkEmail: Check[String, String] =
  splitEmail andThen joinEmail
```

Finally, we can see that our `createUser` example works as expected using `Kleisli`:

```
final case class User(username: String, email: String)

def createUser(
      username: String,
      email: String): Either[Errors, User] = (
  checkUsername.run(username),
  checkEmail.run(email)
).mapN(User)

createUser("Noel", "noel@underscore.io")
// res16: Either[Errors,User] = Right(User(Noel,noel@underscore.io))

createUser("", "dave@underscore@io")
// res17: Either[Errors,User] = Left(NonEmptyList(Must be longer than
     3 characters))
```

Return to the exercise

Appendix K

Solutions for: Case Study: CRDTs

K.1 GCounter Implementation

Hopefully the description above was clear enough that you can get to an implementation like the one below.

```
final case class GCounter(counters: Map[String, Int]) {
  def increment(machine: String, amount: Int) = {
    val value = amount + counters.getOrElse(machine, 0)
    GCounter(counters + (machine -> value))
  }

  def merge(that: GCounter): GCounter =
    GCounter(that.counters ++ this.counters.map {
      case (k, v) =>
        k -> (v max that.counters.getOrElse(k, 0))
    })

  def total: Int =
    counters.values.sum
}
```

Return to the exercise

K.2 BoundedSemiLattice Instances

It's common to place the instances in the companion object of `BoundedSemiLattice` so they are in the implicit scope without importing them.

Implementing the instance for `Set` provides good practice with implicit methods.

```
trait BoundedSemiLattice[A] extends CommutativeMonoid[A] {
  def combine(a1: A, a2: A): A
  def empty: A
}

object BoundedSemiLattice {
  implicit val intInstance: BoundedSemiLattice[Int] =
    new BoundedSemiLattice[Int] {
      def combine(a1: Int, a2: Int): Int =
        a1 max a2

      val empty: Int =
        0
    }

  implicit def setInstance[A]: BoundedSemiLattice[Set[A]] =
    new BoundedSemiLattice[Set[A]]{
      def combine(a1: Set[A], a2: Set[A]): Set[A] =
        a1 union a2

      val empty: Set[A] =
        Set.empty[A]
    }
}
```

Return to the exercise

K.3 Generic GCounter

Here's a working implementation. Note the use of `|+|` in the definition of `merge`, which significantly simplifies the process of merging and maximising

counters:

```
import cats.instances.list._   // for Monoid
import cats.instances.map._    // for Monoid
import cats.syntax.semigroup._ // for |+|
import cats.syntax.foldable._  // for combineAll

final case class GCounter[A](counters: Map[String,A]) {
  def increment(machine: String, amount: A)
        (implicit m: CommutativeMonoid[A]): GCounter[A] = {
    val value = amount |+| counters.getOrElse(machine, m.empty)
    GCounter(counters + (machine -> value))
  }

  def merge(that: GCounter[A])
        (implicit b: BoundedSemiLattice[A]): GCounter[A] =
    GCounter(this.counters |+| that.counters)

  def total(implicit m: CommutativeMonoid[A]): A =
    this.counters.values.toList.combineAll
}
```

Return to the exercise

K.4 Abstracting GCounter to a Type Class

Here's the complete code for the instance. Write this definition in the companion object for `GCounter` to place it in glocal implicit scope:

```
import cats.instances.list._   // for Monoid
import cats.instances.map._    // for Monoid
import cats.syntax.semigroup._ // for |+|
import cats.syntax.foldable._  // for combineAll

implicit def mapInstance[K, V]: GCounter[Map, K, V] =
  new GCounter[Map, K, V] {
    def increment(map: Map[K, V])(key: K, value: V)
          (implicit m: CommutativeMonoid[V]): Map[K, V] = {
      val total = map.getOrElse(key, m.empty) |+| value
```

```
      map + (key -> total)
    }

    def merge(map1: Map[K, V], map2: Map[K, V])
          (implicit b: BoundedSemiLattice[V]): Map[K, V] =
      map1 |+| map2

    def total(map: Map[K, V])
        (implicit m: CommutativeMonoid[V]): V =
      map.values.toList.combineAll
  }
```

Return to the exercise

K.5 Abstracting a Key Value Store

Here's the code for the instance. Write the definition in the companion object for `KeyValueStore` to place it in global implicit scope:

```
implicit val mapInstance: KeyValueStore[Map] =
  new KeyValueStore[Map] {
    def put[K, V](f: Map[K, V])(k: K, v: V): Map[K, V] =
      f + (k -> v)

    def get[K, V](f: Map[K, V])(k: K): Option[V] =
      f.get(k)

    override def getOrElse[K, V](f: Map[K, V])
        (k: K, default: V): V =
      f.getOrElse(k, default)

    def values[K, V](f: Map[K, V]): List[V] =
      f.values.toList
  }
```

Return to the exercise