Advanced Scala
with Cats

Noel Welsh and Dave Gurnell

Early Access, May 2017

underscore

Advanced Scala with Cats

Early Access, May 2017

Published by Underscore Consulting LLP, Brighton, UK.

Copies of this, and related topics, can be found at http://underscore.io/training. Team discounts, when available, may also be found at that address. Contact the authors at hello@underscore.io.

Underscore is a team of developers specialising in Scala and functional programming. You can find us on the web at http://underscore.io and on Twitter at @underscoreio.

In addition to writing software, we provide other training courses, workshops, books, and mentoring to help you and your team create better software and have more fun. For more information please visit http://underscore.io/training.

Contents

Foreword

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 find 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!

Notes on the Pre-Release Edition

> This book is in *early access* status. This means there are unfinished aspects as detailed below. There may be typos and errata in the text and examples.
>
> As an early access customer you will receive a **free copy of the final text** when it is released, plus **free lifetime updates**. If you spot any mistakes or would like to provide feedback on the book, please let us know!
>
> —Dave Gurnell (dave@underscore.io) and Noel Welsh (noel@underscore.io).

Changelog

Starting from the March 2016 release, here are the major changes to the book:

- Moved the theoretical chapters from Scalaz to Cats (currently version `0.7.2`).
- Added sections on the `Reader`, `Writer`, `State`, and `Eval` monads.
- Added a chapter on monad transformers.
- Added sections on `Cartesian` to the applicatives chapter.
- Added sections on `Foldable` and `Traverse`.
- Added type chart diagrams for `Functor` and `Monad`.
- Added a new case study on asynchonous testing.
- New diagrams for the map reduce and validation case studies.

Omissions

Here are the major things missing from the book:

1. Proof reading, final tweaks.

2. Upgrade to Cats 1.0!

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, might write `String` or `Strings` to refer to `java.util.String`.

References to external resources are written as [hyperlinks][link-underscore]. 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 wrap code in an object to account for differences between the console and regular code:

```
object example {
  sealed trait Foo[A]
  final case class Bar[A](a: A) extends Foo[A]

  println(Bar("wrapping this code in an object makes sure tut
    interprets it correctly"))
}
```

Callout Boxes

We use three 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.

> Warning callouts indicate common pitfalls and gotchas. Make sure you read these to avoid problems, and come back to them if you're having trouble getting your code to run.

Acknowledgements

Thanks to everyone who helped shape this book: Richard Dallaway, Jonathon Ferguson, Cody Koeninger, Jason Scott, Toby Weston, Gregor Ihmor, Narayan Iyer, all our students who have attended a course online or in person, and everyone else who gave feedback or encouraged us to hurry up and finish!

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. 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.

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

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

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 stringJsonWriter = new JsonWriter[String] {
    def write(value: String): Json =
      JsString(value)
  }
  implicit val personJsonWriter = new JsonWriter[Person] {
    def write(value: Person): Json =
      JsObject(Map(
```

```
        "name" -> JsString(value.name),
        "email" -> JsString(value.email)
      ))
  }
  // etc...
}
```

1.1.3 Interfaces

An *interface* is any functionality we expose to users. Interfaces to type classes 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)))
```

Interface Syntax

We can alternatively use *extension methods* to extend existing types with interface methods[1]. 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 along-side the instances for the types we need:

```
import JsonWriterInstances._
import JsonSyntax._

Person("Dave", "dave@example.com").toJson
// res5: Json = JsObject(Map(name -> JsString(Dave), email ->
    JsString(dave@example.com)))
```

1.1.4 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 returns a `String`.

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

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:

1. Define a data type `Cat`:

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

2. Create an implementation of `Printable` for `Cat` that returns content in the following format:

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

3. 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 PrintOps[A]` to wrap up a value of type `A`.
3. In `PrintOps` 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

1.1.5 Take Home Points

In this section we revisited the concept of a **type class**, which allows us to add new functionality to existing types.

The Scala implementation of a type class has **three parts**:

- the *type class* itself, a generic trait;
- *instances* for each type we care about; and
- one or more generic *interface* methods.

Interface methods can be defined in *interface objects* or *interface syntax*. Implicit classes are the most common way of implementing syntax.

In the next section we will take a first look at Cats. We will examine the standard code layout Cats uses to organize its type classes, and see how to select type classes, instances, and syntax for use in our code.

1.2 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`.

Show defines one method of interest:

```
def show[A](value: A): String = ???
// show: [A](value: A)String
```

1.2.1 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.

1.2.2 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 target type:

Import	Parameter types
`cats.instances.int`	`Int`
`cats.instances.string`	`String`
`cats.instances.list`	`List`
`cats.instances.option`	`Option`
`cats.instances.map`	`Map` and subtypes
`cats.instances.all`	All instances
and so on...	See the `cats.instances` package for more

> Most people use `import cats.implicits._` to bring all instances into scope at the same time. In this book we will use specific imports to show you exactly which instances we need in each case. Don't feel you have to do this in your code.

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

```
import cats.instances.int._
import cats.instances.string._

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
```

1.2.3 Importing Interface Syntax

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

```
import cats.syntax.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.

1.2.4 Defining Custom Instances

There are two constructor methods on the companion object of `Show` that we can use to define instances for our own types:

```
// 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 allows us to quickly construct instances of `Show`.

```
import java.util.Date

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

These constructors exist for `Show` but don't make sense for all Cats type classes. We will introduce constructors for other type classes as we come to then.

1.2.5 Exercise: Cat Show

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

See the solution

1.2.6 Take Home Points

Cats type classes are defined in the `cats` package. For example, the Show type class is defined as `cats.Show`.

Default instances are defined in the `cats.instances` package. Imports are organized by parameter type (as opposed to by type class).

Interface syntax is defined in the `cats.syntax` package. There are separate syntax imports for each type class. For example, the syntax for Show is defined in `cats.syntax.show`.

1.3 Example: *Eq*

We will finish off this chapter by looking at another useful type class: `cats.Eq`.

1.3.1 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.equal`][cats.syntax.equal], provides two methods for performing type-safe equality checks provided there is an instance `Eq[A]` in scope:

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

1.3.2 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._

val eqInt = Eq[Int]
```

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

```
eqInt.eqv(123, 123)
// res1: Boolean = true

eqInt.eqv(123, 234)
// res2: 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._

123 === 123
// res4: Boolean = true

123 =!= 234
```

```
// res5: Boolean = true
```

1.3.3 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._
import cats.instances.option._
```

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 a compile error here because the Eq type class is invariant. The instances we have in scope are for `Int` and `Option[Int]`, not `Some[Int]`. To fix the issue we have to re-type the arguments as `Option[Int]`:

```
(Some(1) : Option[Int]) === (None : Option[Int])
// res7: 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]
// res8: Boolean = false
```

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

```
import cats.syntax.option._

1.some === None
// res9: Boolean = false

1.some =!= None
// res10: Boolean = true
```

1.3.4 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._

implicit val dateEqual = 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
// res11: Boolean = true

x === y
// res12: Boolean = false
```

1.3.5 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",   35, "orange and black")
val cat2 = Cat("Heathcliff", 30, "orange and black")

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

See the solution

1.3.6 Take Home Points

In this section we introduced a new type class—`cats.Eq`—that lets us perform type-safe equality checks:

- we create an instance `Eq[A]` to implement equality-testing functionality for `A`.
- `cats.syntax.eq` provides two methods of interest: `===` for testing equality and `=!=` for testing inequality.

Because `Eq[A]` is invariant in `A`, we have to be precise about the types of the values we use as arguments. We sometimes need to manually type expressions in our code to help the compiler locate the correct type class instances.

1.4 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;
  - typically, one or more additional methods for creating instances.
- Default instances are provided via 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 this book we will look at four broad and powerful type classes—`Monoids`, `Functors`, `Monads`, `Applicatives`, 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, 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 wee 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 exactly the same with other types of sequence, using concatenation as 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 these 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 for custom data types. 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 simply the `combine` part of a monoid. While many semigroups are also monoids, there are some data types for which we can-

not 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 lose access to an `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 a monoid is, 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 Obtaining Instances

`Monoid` follows the standard Cats pattern for the user interface: the companion object has an `apply` method that returns the type class instance. So if we wanted 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._

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
```

2.5.3 Default Instances

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._

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._
import cats.instances.option._

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.

2.5.4 *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.syntax.semigroup._
import cats.instances.string._

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

import cats.instances.int._

val intResult = 1 |+| 2 |+| Monoid[Int].empty
// intResult: Int = 3
```

2.5.5 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 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 `Monoid[Option[Int]]`, will the expression `Some(1) |+| Some(2)` select this instance? (Remember that `Some` is a subtype of `Option`).

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

  We've seen two monoids for `Int`: addition and zero, and multiplication and one. Similarly there are at least four monoids for `Boolean` (and, or, equal, and not equal). When we write `true |+| false`, which instance is selected?

In this section we explore how Cats answers these questions.

2.6.1 Type Class Variance

When we define type classes we can add variance annotations to the type parameter like we can for any other generic type. To quickly recap, there are three cases:

- A type with an unannotated parameter `Foo[A]` is *invariant* in `A`.

  This means there is no relationship between `Foo[B]` and `Foo[C]` no matter what the sub- or super-type relationship is between `B` and `C`.

- A type with a parameter `Foo[+A]` is *covariant* in `A`.

  If `C` is a subtype of `B`, `Foo[C]` is a subtype of `Foo[B]`.

- A type with a parameter `Foo[-A]` is *contravariant* in `A`.

  If `C` is a supertype of `B`, `Foo[C]` is a subtype of `Foo[B]`.

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?	Yes	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 monoid 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" that construct values with the type of the base trait in an algebraic data type. For example, Cats provides `some` and `none` constructors for `Option`:

```
import cats.instances.option._
import cats.syntax.option._

Some(1)
// res0: Some[Int] = Some(1)

1.some
// res1: Option[Int] = Some(1)

None
// res2: None.type = None
```

```
none[Int]
// res3: Option[Int] = None
```

2.6.2 Identically Typed Instances

The other issue is choosing between type class instances when several are available for a specific type. For example, how do we select the monoid for integer multiplication instead of the monoid for integer addition?

Cats currently has no mechanism for selecting alternative instances, though this may change in the future.

We can always define or import a type class instance into the local scope. This will take precedence over other type class instances in the implicit scope:

```
import cats.Monoid
import cats.syntax.semigroup._

implicit val multiplicationMonoid =
  new Monoid[Int] {
    def empty: Int = 1
    override def combine(x: Int, y: Int): Int = x * y
  }

3 |+| 2
// res5: Int = 6
```

2.7 Applications of Monoids

We now know what a moniod 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.

2.7.1 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.

2.7.2 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.

2.7.3 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.

2.8 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.all._
import cats.syntax.semigroup._
```

With these three things in scope, we can set about adding anything we want:

```
Option(1) |+| Option(2)
// res0: Option[Int] = Some(3)

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

map1 |+| map2
// res1: Map[String,Int] = Map(b -> 5, d -> 4, a -> 1)

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

tuple1 |+| tuple2
// res2: (String, Int) = (helloworld,444)
```

`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**. 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`, `Either`, and `Future`, to name a few.

Let's start as we did with monoids by looking at a few types and operations and seeing what general principles we can abstract.

Sequences

The `map` method is perhaps the most commonly used method on `List`. If we have a `List[A]` and a function `A => B`, `map` will create a `List[B]`.

```
List(1, 2, 3).map(x => (x % 2) == 0)
// res0: List[Boolean] = List(false, true, false)
```

There are some properties of `map` that we rely on without even thinking about them. For example, we expect the two snippets below to produce the same output:

```
List(1, 2, 3).map(_ * 2).map(_ + 4)
// res1: List[Int] = List(6, 8, 10)

List(1, 2, 3).map(x => (x * 2) + 4)
// res2: List[Int] = List(6, 8, 10)
```

In general, the `map` method for a `List` works like this: We start with a `List[A]` of length `n`, we supply a function from `A` to `B`, and we end up with a `List[B]` of length `n`. The elements are changed but the ordering and length of the list are preserved. This is illustrated in Figure 3.1.

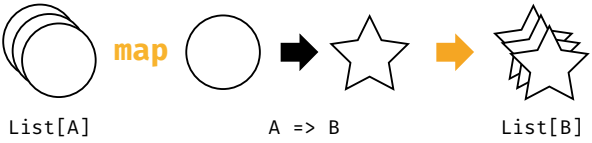

Figure 3.1: Type chart: mapping over a List

Options

We can do the same thing with an `Option`. If we have a `Option[A]` and a function `A => B`, `map` will create a `Option[B]`:

```
Option(1).map(_.toString)
// res3: Option[String] = Some(1)
```

We expect `map` on `Option` to behave in the same way as `List`:

```
Option(123).map(_ * 4).map(_ + 4)
// res4: Option[Int] = Some(496)

Option(123).map(x => (x * 2) + 4)
// res5: Option[Int] = Some(250)
```

In general, the `map` method for an `Option` works similarly to that for a `List`. We start with an `Option[A]` that is either a `Some[A]` or a `None`, we supply a function from `A` to `B`, and the result is either a `Some[B]` or a `None`. Again, the structure is preserved: if we start with a `Some` we end up with a `Some`, and a `None` always maps to a `None`. This is shown in Figure 3.2.

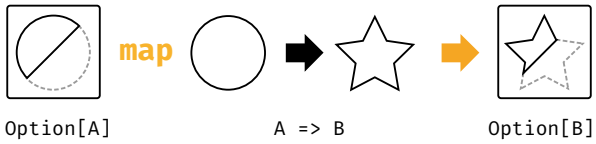

Figure 3.2: Type chart: mapping over an Option

3.2 More Examples of Functors

Let's expand how we think about `map` by taking some other examples into account:

Futures

`Future` is also a functor with a `map` method[1]. If we start with a `Future[A]` and call map supplying a function `A => B`, we end up with a `Future[B]`:

[1]Some functional purists disagree with this because the exception handling in Scala futures breaks the functor laws. We're going to ignore this detail because *real* functional programs don't do exceptions.

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

val future1 = Future("Hello world!")
// future1: scala.concurrent.Future[String] = Future(<not
    completed>)

val future2 = future1.map(_.length)
// future2: scala.concurrent.Future[Int] = Future(<not completed
    >)

Await.result(future1, 1.second)
// res6: String = Hello world!

Await.result(future2, 1.second)
// res7: Int = 12
```

The general pattern looks like Figure 3.3. Seem familiar?

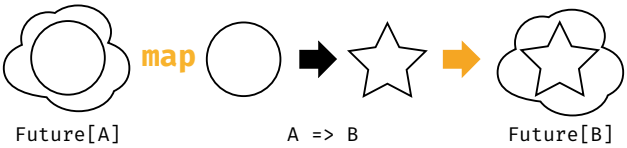

Figure 3.3: Type chart: mapping over a Future

Functions (?!)

Can we map over functions of a single argument? What would this mean?

All our examples above have had the following general shape:

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

A function with a single argument has two types: the parameter type and the result type. To get them to the same 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`.

We can see this with our trusty type chart in Figure 3.4.

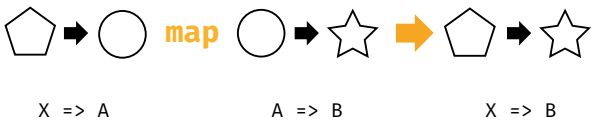

Figure 3.4: Type chart: mapping over a Function1

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

```
import cats.instances.function._
import cats.syntax.functor._

val func1 = (x: Int)    => x.toDouble
// func1: Int => Double = <function1>

val func2 = (y: Double) => y * 2
// func2: Double => Double = <function1>

val func3 = func1.map(func2)
// func3: Int => Double = scala.runtime.
    AbstractFunction1$$Lambda$9405/272413009@2deb99ee

func3(1) // function composition by calling map
// res8: Double = 2.0

func2(func1(1)) // function composition written out by hand
```

```
// res9: Double = 2.0
```

3.3 Definition of a Functor

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 3.5.

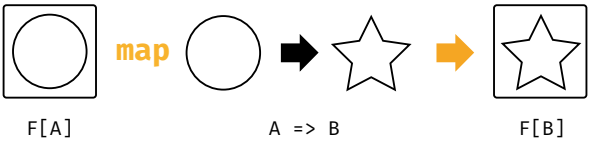

Figure 3.5: Type chart: generalised functor map

Intuitively, a functor `F[A]` represents some data (the `A` type) in a context (the `F` type). The `map` operation modifies the data within but retains the structure of the surrounding context. 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)
```

If we consider the laws in the context of the functors we've discussed above, we can see they make sense and are true. We've seen some examples of the second law already.

A simplified version of the definition from Cats is:

```
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.

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 that 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
```

> *Kind notation*
>
> We sometimes use "kind notation" to describe the shape of types and their constructors. Regular types have a kind `*`. `List` has kind `* => *` to indicate that it produces a type given a single parameter. `Either` has kind `* => * => *` because it accepts two parameters, and so on.

In Scala we declare type constructors using underscores but refer to them without:

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

  // Refer to 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

// Refer to 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`, or `Future`.

Language feature imports

Higher kinded types are considered an advanced language feature in Scala. Wherever we *declare* a type constructor with `A[_]` syntax, we need to "import" the feature to suppress compiler warnings:

```
import scala.language.higherKinds
```

3.5 Functors in Cats

Let's look at the implementation of functors in Cats. We'll follow the usual pattern of looking at the three main aspects of the implementation: the *type class*, the *instances*, and the *interface*.

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 cats.Functor
import cats.instances.list._
import cats.instances.option._

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 lifted = Functor[Option].lift(func)
// lifted: Option[Int] => Option[Int] = cats.
    Functor$$Lambda$28362/1686307543@514a39b6

lifted(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` operations. If there is a built-in method it will always be called in preference to an extension method. Instead we will use *functions* as our example:

```
import cats.instances.function._
import cats.syntax.functor._

val func1 = (a: Int) => a + 1
// func1: Int => Int = <function1>

val func2 = (a: Int) => a * 2
// func2: Int => Int = <function1>

val func3 = func1.map(func2)
```

```
// func3: Int => Int = scala.runtime.
    AbstractFunction1$$Lambda$9405/272413009@5f430fe0

func3(123)
// res1: Int = 248
```

Other methods are available but we won't discuss them here. Functors are more important to us as building blocks for later abstractions than they are as a tool for direct use.

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`:

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

The implementation is trivial—we simply call `Option`'s `map` method.

Sometimes we need to inject dependencies into our instances. For example, if we had to define a custom `Functor` for `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) =
  new Functor[Future] {
    def map[A, B](value: Future[A])(func: A => B): Future[B] =
```

```
      value.map(func)
  }
```

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

We can think of `map` as "appending" a transformation to a chain. We start with an `F[A]`, run it through a function `A => B`, and end up with an `F[B]`. We can extend the chain further by mapping again: run the `F[B]` through a function `B => C` and end up with an `F[C]`:

```
Option(1).map(_ + 2).map(_ * 3).map(_ + 100)
// res0: Option[Int] = Some(109)
```

We're now going to look at two other type classes, one that represents *prepending* operations to a chain, and one that represents building a *bidirectional* chain of operations.

One great use case for these new type classes is building libraries that transform, read, and write values. The content ties in tightly to the JSON codec case study later in the book.

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" a transformation to a chain. This is illustrated in Figure 3.6.

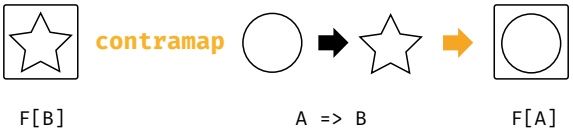

Figure 3.6: Type chart: the contramap method

We'll talk about `contramap` itself directly for now, bringing in the type class in a moment.

The `contramap` method only makes sense for certain data types. 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`.

`contramap` starts to make sense when we have a data types that represent tranformations. For example, consider the `Printable` type class we discussed in Chapter 2:

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

A `Printable[A]` represents a transformation from `A` to `String`. We can define a `contramap` method that "prepends" a transformation from another type `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)
```

This says that if `A` is `Printable`, and we can transform `B` into `A`, then `B` is also `Printable`.

3.6.1.1 Exercise: Showing off with Contramap

Implement the `contramap` method for `Printable` above.

See the solution

Let's define some basic instances of `Printable` for `String` and `Boolean`:

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

implicit val booleanPrintable =
  new Printable[Boolean] {
    def format(value: Boolean): String =
      if(value) "yes" else "no"
  }

format("hello")
// res4: String = "hello"

format(true)
```

```
// res5: String = yes
```

Define an instance of `Printable` that prints the value from this case class:

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

Rather than writing out the complete definition from scratch (new `Printable[Box]` etc...), create your instance using the `contramap` method of one of the instances above.

See the solution

Your instance should work as follows:

```
format(Box("hello world"))
// res6: String = "hello world"

format(Box(true))
// res7: String = yes
```

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

```
format(Box(123))
// <console>:19: error: could not find implicit value for
    parameter p: Printable[Box[Int]]
//        format(Box(123))
//               ^
```

3.6.2 Invariant functors and the *imap* method

The second of our type classes, the *invariant functor*, provides a method called `imap` that is informally equivalent to a combination of `map` and `contramap`. We can demonstrate this by extending `Printable` to produce a typeclass for encoding and decoding to a `String`:

```
trait Codec[A] {
  def encode(value: A): String
  def decode(value: String): Option[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]): Option[A] =
  c.decode(value)
```

The type chart for `imap` is showin in Figure 3.7.

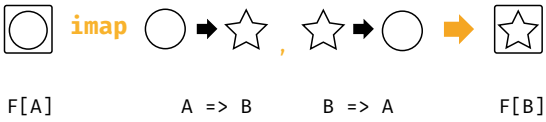

Figure 3.7: Type chart: the imap method

3.6.2.1 Transformative Thinking with Imap

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

See the solution

Here's an example `Codec` representing parsing and serializing `Int`s:

See the solution

Demonstrate your `imap` method works by creating a `Codec` for conversions between `String`s and `Boolean`s:

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

See the solution

Your instance should work as follows:

```
encode(Box(123))
// res13: String = 123

decode[Box[Int]]("123")
// res14: Option[Box[Int]] = Some(Box(123))
```

3.6.3 What's With the Name?

What's the relationship between contravariance, invariance, and covariance as we usually understand them in Scala, and the names for the functors above?

The usual meaning of these terms in Scala relates to subtypes. We say that `B` is a subtype of `A` if we can use `B` anywhere we want an `A`. Put another way, we can convert `A` into `B` and our program keeps on working.

Co- and contravariance usually arises in Scala when working with type constructors like `List` and `Option`. If we declare a type constructor `F`, and we want `F[B]` to be a subtype of `F[A]` when `B` is a subtype of `A`, we declare the type parameter to be covariant.

```
trait F[+A] // A is covariant
```

If `B` is a subtype of `A`, and we want `F[A]` to be a subtype of `F[B]`, then we declare `F` to have a contravariant type parameter.

```
trait F[-A] // A is contravariant
```

Co- and contravariant functors capture the same principle without the limitations of subtyping. As we said above subtyping can be viewed as

a conversion. B is a subtype of A if we can convert A to B. In other words there exists a function `A => B`. A covariant functor, which is what the standard `Functor` is, captures exactly this. If `F` is a (covariant) functor, whenever we have a `F[A]` and a conversion `A => B` we have a `F[B]`. A contravariant functor captures the case in the opposite direction. If `F` is a contravariant functor, whenever we have a `F[A]` and a conversion `B => A` we have a `F[B]`.

3.7 *Contravariant* and *Invariant* in Cats

Cats's `Contravariant` and `Invariant` type classes are slightly different to its other type classes: they live under `cats.functor` instead of `cats`. Here's a simplified version of the code:

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

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

  def imap[A, B](fa: F[A])(f: A => B)(fi: B => A): F[B] =
    contramap(fa)(fi)
}

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

  def imap[A, B](fa: F[A])(f: A => B)(fi: B => A): F[B] =
    map(fa)(f)
}
```

Cats treats `Functor` and `Contravariant` as specialisations of `Invariant` where one side of the bidirectional transformation is

ignored. Cats uses this to provide operations that work with any of the three types of functor[2].

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`, `Writer`, `WriterT`, and `Function1`:

```
import cats.Show
import cats.functor.Contravariant
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.instances.function._
import cats.syntax.contravariant._

val div2: Int => Double = _ / 2.0
val add1: Int => Int    = _ + 1

div2.contramap(add1)(2)
// res4: Double = 1.5
```

[2]One example is the `tupled` method provided by the cartesian builder syntax discussed in Chapter 6.

3.7.1.1 *Invariant* in Cats

Cats provides instances of `Invariant` for `Semigroup` and `Monoid`. It also provides an `imap` extension method via the `cats.syntax.invariant` import. Imagine we have a semigroup for a well known type, for example `Semigroup[String]`, and we want to convert it to another type like `Semigroup[Symbol]`. To do this we need two functions: one to convert the `Symbol` parameters to `Strings`, and one to convert the result of the `String` append back to a `Symbol`:

```
import cats.Semigroup
import cats.instances.string._ // semigroup for String
import cats.syntax.invariant._ // imap extension method

implicit val symbolSemigroup: Semigroup[Symbol] =
  Semigroup[String].imap(Symbol.apply)(_.name)

import cats.syntax.semigroup._

'a |+| 'few |+| 'words
// res7: Symbol = 'afewwords
```

3.8 Summary

We covered three types of functor in this chapter: regular covariant `Functors` with their `map` method, as well as `Contravariant` functors with their `contramap` methods, and `Invariant` functors with their `imap` methods.

Regular `Functors` are by far the most common of these type classes, but even then is rare to use them on their own. They form the 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 `Applicatives`.

The `map` method for collection types is important because each element in a collection can be transformed 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 Pygmy Hadoop case study later in the book.

The `Contravariant` and `Invariant` type classes are more situational. We won't be doing much more work with them, although we will revisit them to discuss `Cartesians`, and for the JSON Codec case study later in the book.

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 `flatMap` method. All of the functors we saw in the last chapter are also monads, including `Option`, `List`, `Either`, 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 is one of the benefits that Cats brings us.

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 heard of, 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, and monoids in the category of endofunctors (whatever they are). We're going to solve the problem of explaining monads once and for all by stating very simply:

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

That was easy! Problem solved, right? Ok, maybe we need some more discussion...

Informally, the most important feature of a monad is its `flatMap` method, which allows us to specify what happens next. This is what we mean by sequencing computations. A monad allows us to specify a sequence of operations that happen one after another. We specify the application-specific part of the computation as a function parameter, and `flatMap` runs our function and takes care of some kind of complication (conventionally referred to as an "effect"). Let's ground things by looking at some examples.

Options

`Option` is a monad that 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 computations may fail, as indicated by their `Option` return types. The `flatMap` method on `Option` allows us to sequence

these operations without having to constantly check whether they return Some or None:

```
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 `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.

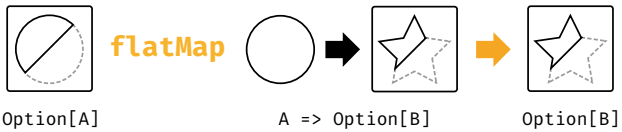

Figure 4.1: Type chart: flatMap for Option

The result of the computation is an `Option`, allowing us to call `flatMap` again and so the process 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 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:

```
def numbersBetween(min: Int, max: Int): List[Int] =
  (min to max).toList

for {
  x <- numbersBetween(1, 3)
  y <- numbersBetween(4, 5)
} 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 functions that return `Lists` as functions with multiple return values, `flatMap` becomes a construct that calculates results from permutations and combinations of intermediate values.

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 the overall expression. `flatMap` is generating these combinations from our code, which simply says "get `x` from here and `y` from over there".

The type chart in Figure 4.2 illustrates this behaviour: although the result of `flatMap` (`List[B]`) is the same type as the result of the user-supplied function, the end result is actually a larger list created from combinations of intermediate `A`s and `B`s:

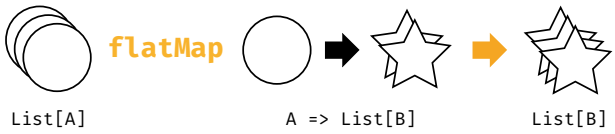

Figure 4.2: Type chart: flatMap for List

Futures

`Future` is a monad that allows us to sequence computations without worrying that they are asynchronous:

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

def getTrafficFromHost(hostname: String): Future[Int] =
```

```
  ??? // grab traffic information using a network client

def getTrafficFromAllHosts: Future[Int] =
  for {
    traffic1 <- getTrafficFromHost("host1")
    traffic2 <- getTrafficFromHost("host2")
    traffic3 <- getTrafficFromHost("host3")
  } yield traffic1 + traffic2 + traffic3
```

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 Scala's `Futures`, you'll know that the code above is fetching traffic from each server *in sequence*. This becomes clearer if we expand out the for comprehension to show the nested calls to `flatMap`:

```
def getTrafficFromAllHosts: Future[Int] =
  getTrafficFromHost("host1").flatMap { traffic1 =>
    getTrafficFromHost("host2").flatMap { traffic2 =>
      getTrafficFromHost("host3").map { traffic3 =>
        traffic1 + traffic2 + traffic3
      }
    }
  }
```

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 4.3, which shows the function parameter of type `A => Future[B]`:

In other words, the monadic behaviour of `Future` allows us to sequence asynchronous computations one after the other. We can run `Futures` in parallel, but that is another story and shall be told another time. Monads are truly all about sequencing.

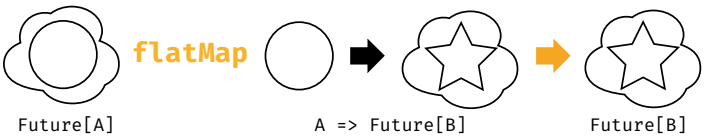

Figure 4.3: Type chart: flatMap for Future

4.1.1 Monad Definition and Laws

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

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

The `pure` operation 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 using the supplied function to generate the next context in the sequence. Here is a simplified version of the `Monad` type class in Cats:

```
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]
}
```

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

Importantly, the `pure` and `flatMap` methods must obey three laws:

Left identity: calling `pure` then transforming the result with a function `f` is the same as simply calling `f`:

```
pure(a).flatMap(f) == f(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. If `flatMap` represents sequencing a computation that introduces a new monadic context, `map` represents sequencing a computation that does not. 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]
}
```

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` so every `Monad` also has a `map` method. We'll discuss `Applicatives` in a later chapter.

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

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

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))(x => List(x, x*10))
// list2: List[Int] = List(1, 10, 2, 20, 3, 30)
```

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

`Monad` provides many other methods as well, 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._

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

import cats.instances.list._

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

import cats.instances.vector._

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

The `Monad` for `Future` doesn't accept implicit `ExecutionContext` parameters to `pure` and `flatMap` like `Future` itself does (it can't because the parameters aren't in the definitions in the `Monad` trait). To work around this, Cats requires us to have an `ExecutionContext` in scope when we summon the `Monad` for `Future`:

```
import cats.instances.future._
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]
//                      ^

import scala.concurrent.ExecutionContext.Implicits.global

val fm = Monad[Future]
// fm: cats.Monad[scala.concurrent.Future] = cats.instances.
    FutureInstances$$anon$1@1c54d4ad
```

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

```
Await.result(
  fm.flatMap(fm.pure(1)) { x =>
    fm.pure(x + 2)
  },
  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 the most important 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 particlar instance we want.

```
import cats.syntax.applicative._
import cats.instances.option._
import cats.instances.list._

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 scala.language.higherKinds
import cats.Monad
import cats.syntax.functor._
import cats.syntax.flatMap._

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

import cats.instances.option._
import cats.instances.list._

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 Scala 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[M[_] : Monad](a: M[Int], b: M[Int]): M[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.

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._
import cats.syntax.flatMap._

def sumSquare[M[_] : Monad](a: M[Int], b: M[Int]): M[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: M[Int], b: M[Int])(implicit evidence$1: cats.Monad[M])M
    [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: ?M[Int]
//        sumSquare(3, 4)
//        ^
// <console>:22: error: type mismatch;
//  found   : Int(3)
//  required: M[Int]
//        sumSquare(3, 4)
//                  ^
// <console>:22: error: type mismatch;
//  found   : Int(4)
//  required: M[Int]
//        sumSquare(3, 4)
//                     ^
```

It would be incredibly useful if we could use `sumSquare` with a combination of monadic and non-monadic parameters. 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
```

Now we can 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 `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 so on on 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.flatMap._
import cats.syntax.functor._
```

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

The main use for `Id` is to write generic methods like `sumSquare` that operate on monadic and non-monadic data types. For example, we can run code asynchronously in production using `Future` and synchronously in test using `Id`:

```
import scala.concurrent._
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global
import cats.instances.future._

// In production:
Await.result(sumSquare(Future(3), Future(4)), 1.second)
// res8: Int = 25

// In test:
sumSquare(a, b)
// res10: cats.Id[Int] = 25
```

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 monadic data type. The Scala standard library has a type `Either`. In Scala 2.11 and earlier, `Either` wasn't tech-

nically 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` was unbiased. It had no `map` or `flatMap` method:

```
// Scala 2.11 example

Right(123).flatMap(x => Right(x * 2))
// <console>:12: error: value flatMap is not a member
//   of scala.util.Right[Nothing, Int]
//        Right(123).flatMap(x => Right(x * 2))
//                   ^
```

Instead of calling `map` or `flatMap` directly, we had to decide which side we wanted to be the "correct" side by taking a left- or right-projection:

```
// Valid in Scala 2.11 and Scala 2.12

val either1: Either[String, Int] = Right(123)
// either1: Either[String,Int] = Right(123)

val either2: Either[String, Int] = Right(321)
// either2: Either[String,Int] = Right(321)

either1.right.flatMap(x => Right(x * 2))
// res2: scala.util.Either[String,Int] = Right(246)

either2.left.flatMap(x => Left(x + "!!!"))
// res3: scala.util.Either[String,Int] = Right(321)
```

This made the Scala 2.11 version of `Either` incovenient to use as a monad. If we wanted to use for comprehensions, for example, we had to insert calls to `.right` in every generator clause:

```
for {
  a <- either1.right
  b <- either2.right
} yield a + b
// res4: scala.util.Either[String,Int] = Right(444)
```

In Scala 2.12, `Either` was redesigned. The modern `Either` makes the decision that the right side is always the success case and thus supports `map` and `flatMap` directly. This turns `Either` into a monad and makes working with it much more pleasant:

```
for {
  a <- either1
  b <- either2
} yield a + b
// res5: scala.util.Either[String,Int] = Right(444)
```

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._

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
// res6: scala.util.Either[String,Int] = Right(25)
```

Smart Constructors and Avoiding Over-Narrowing

The `asLeft` and `asRight` methods have advantages over `Left.apply` and `Right.apply` in terms of type inference. The following code provides an example:

```
def countPositive(nums: List[Int]) =
  nums.foldLeft(Right(0)) { (accumulator, num) =>
    if(num > 0) {
      accumulator.map(_ + 1)
    } else {
      Left("Negative. Stopping!")
    }
  }
// <console>:18: error: type mismatch;
//  found   : scala.util.Either[Nothing,Int]
//  required: scala.util.Right[Nothing,Int]
//              accumulator.map(_ + 1)
//                                ^
// <console>:20: error: type mismatch;
//  found   : scala.util.Left[String,Nothing]
//  required: scala.util.Right[Nothing,Int]
//              Left("Negative. Stopping!")
//                   ^
```

There are two problems here, both arising because the compiler chooses the type of `accumulator` based on the first parameter list to `foldRight`:

1. the type of the accumulator ends up being `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. It as 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))
// res7: Either[String,Int] = Right(3)

countPositive(List(1, -2, 3))
// res8: Either[String,Int] = Left(Negative. Stopping!)
```

In addition to `asLeft` and `asRight`, `cats.syntax.either` also adds some useful extension methods to the `Either` companion object. The `catchOnly` and `catchNonFatal` methods are for capturing `Exceptions` in instances of `Either`:

```
Either.catchOnly[NumberFormatException]("foo".toInt)
Either.catchNonFatal(sys.error("Badness"))
```

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

```
Either.fromTry(scala.util.Try("foo".toInt))
Either.fromOption[String, Int](None, "Badness")
```

4.4.3 Transforming Eithers

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

```
import cats.syntax.either._

"Error".asLeft[Int].getOrElse(0)
// res9: Int = 0

"Error".asLeft[Int].orElse(2.asRight[String])
// res10: Either[String,Int] = Right(2)
```

The `ensure` method allows us to check whether a wrapped value satisfies a predicate:

```
-1.asRight[String].ensure("Must be non-negative!")(_ > 0)
// res11: Either[String,Int] = Left(Must be non-negative!)
```

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

```
"error".asLeft[String] recover {
  case str: String =>
    "Recovered from " + str
}
// res12: Either[String,String] = Right(Recovered from error)

"error".asLeft[String] recoverWith {
  case str: String =>
    Right("Recovered from " + str)
}
// res13: Either[String,String] = Right(Recovered from error)
```

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

```
"foo".asLeft[Int].leftMap(_.reverse)
// res14: Either[String,Int] = Left(oof)

6.asRight[String].bimap(_.reverse, _ * 7)
// res15: Either[String,Int] = Right(42)

"bar".asLeft[Int].bimap(_.reverse, _ * 7)
// res16: Either[String,Int] = Left(rab)
```

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

```
123.asRight[String]
// res17: Either[String,Int] = Right(123)

123.asRight[String].swap
// res18: 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 Fail-Fast Error Handling

`Either` is typically used to implement fail-fast error handling. We sequence a number of computations using `flatMap`, and 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
// res19: scala.util.Either[String,Int] = Left(DIV0)
```

4.4.5 Representing Errors

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 as follows:

```
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 supertype. We have (almost) no idea about what type of error occurred.

Another approach is to define an algebraic data type to represent the errors that can occur:

```
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
```

4.4.6 Exercise: What is Best?

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

See the solution

4.5 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 *memoized* and *unmemoized* to create three models of evaluation:

- *now*—evaluated once immediately (equivalent to `val`);
- *later*—evaluated once when the value is first needed (equivalent to `lazy val`);
- *always*—evaluated every time the value is needed (equivalent to `def`).

4.5.1 Eager, lazy, memoized, oh my!

What do these terms mean?

Eager computations happen immediately, whereas *lazy* computations happen on access.

For example, Scala `vals` are eager definitions. 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 eagerly at the definition site. Accessing x simply recalls the stored value without re-running the code.

```
val x = {
  println("Computing X")
  1 + 1
}
// Computing X
// x: Int = 2

x // first access
// res0: Int = 2

x // second access
// res1: Int = 2
```

By contrast, `defs` 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")
  1 + 1
}
// y: Int

y // first access
// Computing Y
// res2: Int = 2

y // second access
// Computing Y
// res3: Int = 2
```

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")
  1 + 1
}
// z: Int = <lazy>

z // first access
// Computing Z
// res4: Int = 2

z // second access
// res5: Int = 2
```

4.5.2 Eval's models of evaluation

`Eval` has three subtypes: `Eval.Now`, `Eval.Later`, and `Eval.Always`. We construct these with three constructor methods, which create instances of the three classes and return them typed as `Eval`:

```
import cats.Eval
// import cats.Eval

val now    = Eval.now(1 + 2)
// now: cats.Eval[Int] = Now(3)

val later  = Eval.later(3 + 4)
// later: cats.Eval[Int] = cats.Later@49373f5e

val always = Eval.always(5 + 6)
// always: cats.Eval[Int] = cats.Always@53dd8e4b
```

We can extract the result of an `Eval` using its `value` method:

```
now.value
// res6: Int = 3

later.value
// res7: Int = 7

always.value
// res8: Int = 11
```

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")
  1 + 1
}
// Computing X
// x: cats.Eval[Int] = Now(2)

x.value // first access
// res9: Int = 2
```

```
x.value // second access
// res10: Int = 2
```

`Eval.always` captures a lazy computation, similar to a `def`:

```
val y = Eval.always {
  println("Computing Y")
  1 + 1
}
// y: cats.Eval[Int] = cats.Always@253eef84

y.value // first access
// Computing Y
// res11: Int = 2

y.value // second access
// Computing Y
// res12: Int = 2
```

Finally, `Eval.later` captures a lazy computation and memoizes the result, similar to a `lazy val`:

```
val z = Eval.later {
  println("Computing Z")
  1 + 1
}
// z: cats.Eval[Int] = cats.Later@a81d4d5

z.value // first access
// Computing Z
// res13: Int = 2

z.value // second access
// res14: Int = 2
```

The three behaviours are summarized below:

	Eager	Lazy
Memoized	`val`, `Eval.now`	`lazy val`, `Eval.later`
Not memoized	N/A	`def`, `Eval.always`

4.5.3 Eval as a Monad

`Eval`'s `map` and `flatMap` methods add computations to a chain. This is similar to the `map` and `flatMap` methods on `scala.concurrent.Future`, except that the computations aren't run until we call `value` to obtain a result:

```
val greeting = Eval.always {
  println("Step 1")
  "Hello"
}.map { str =>
  println("Step 2")
  str + " world"
}
// greeting: cats.Eval[String] = cats.Eval$$anon$8@3c88d726

greeting.value
// Step 1
// Step 2
// res15: String = Hello world
```

Note that, while the semantics of the originating `Eval` instances are maintained, 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@9284e40

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
```

We can use `Eval`'s `memoize` method to memoize a chain of computations. Calculations before the call to `memoize` are 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@397aee40

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
```

4.5.4 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 until later. `defer` is trampolined like `Eval`'s `map` and `flatMap` methods, so we can use it as a way to quickly 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 =
    33473205095971448369154760940714864779127732238104548077301003219901680221
```

Eval is a useful tool to enforce stack 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 calls 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.

4.5.5 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

4.6 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 with 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 numerous data types: 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 that we will see in Chapter 6.

4.6.1 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._

Writer(Vector(
  "It was the best of times",
  "It was the worst of times"
), 123)
// 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),123))
```

We've used a `Vector` as the log in this example as it is a sequence structure with an efficient append operation.

Notice that the type of the writer 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 tranformer", 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.syntax.applicative._ // `pure` method

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._

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, in addition to using `Writer.apply` as we did above we can use the `writer` syntax from `cats.syntax.writer`:

```
import cats.syntax.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:

```
a.value
// res4: cats.Id[Int] = 123

a.written
// res5: cats.Id[scala.collection.immutable.Vector[String]] =
    Vector(msg1, msg2, msg3)
```

or we can extract log and result 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
```

4.6.2 Composing and Transforming Writers

The log in a `Writer` is preserved when we `map` or `flatMap` over it. `flatMap` actually 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
// res6: 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
// res7: cats.Id[(scala.collection.immutable.Vector[String], Int
    )] = (Vector(A, B, C, X, Y, Z),42)
```

We can tranform 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),
  result => result * 100
)
// writer3: cats.data.WriterT[cats.Id,scala.collection.immutable
    .Vector[String],Int] = WriterT((Vector(A, B, C, X, Y, Z)
    ,4200))
```

```
writer3.run
// res8: cats.Id[(scala.collection.immutable.Vector[String], Int
    )] = (Vector(A, B, C, X, Y, Z),4200)

val writer4 = writer1.mapBoth { (log, result) =>
  val log2    = log.map(_ + "!")
  val result2 = result * 1000
  (log2, result2)
}
// writer4: cats.data.WriterT[cats.Id,scala.collection.immutable
    .Vector[String],Int] = WriterT((Vector(a!, b!, c!, x!, y!, z
    !),42000))

writer4.run
// res9: 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
// res10: 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
// res11: cats.Id[(Int, Vector[String])] = (42,Vector(a, b, c, x
    , y, z))
```

4.6.3 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, printing out the intermediate steps in the calculation as it runs. The `slowly` helper function ensures this takes a while to run, even on the very small examples we need in this book to fit the output on the page:

```
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
// res13: 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

4.7 The *Reader* Monad

`cats.data.Reader` is a monad that allows us to compose 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 injecting configuration. If we have a number of operations that all depend on some external configuration, we can chain them together using a `Reader`. The `Reader` produces one large operation that accepts the configuration as a parameter and runs our program in the order we specified it.

4.7.1 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
```

We can create `Readers` from functions and extract the functions again. So far so simple, but what advantage do `Readers` give us over the raw functions?

4.7.2 Composing Readers

The power of `Readers` comes from their `map` and `flatMap` methods, which represent different kinds of function composition. The general pattern of usage is to 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 multiple readers that depend on the same input type. To illustrate this, let's extend our greeting example to produce a login system that checks a user's password and displays different messages depending on whether it was valid:

```
val feedKitty: Reader[Cat, String] =
  Reader(cat => s"Have a nice bowl of ${cat.favoriteFood}")

val greetAndFeed: Reader[Cat, String] =
  for {
    msg1 <- greetKitty
    msg2 <- feedKitty
  } yield s"${msg1} ${msg2}"

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
```

4.7.3 Exercise: Hacking on Readers

The classic use of `Reader`s is to build programs that accept a configuration at the end. 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 db = Db(
  Map(
    1 -> "dade",
    2 -> "kate",
    3 -> "margo"
  ),
  Map(
    "dade"  -> "zerocool",
    "kate"  -> "acidburn",
    "margo" -> "secret"
  )
)
// db: Db = Db(Map(1 -> dade, 2 -> kate, 3 -> margo),Map(dade ->
     zerocool, kate -> acidburn, margo -> secret))

checkLogin(1, "zerocool").run(db)
// res8: cats.Id[Boolean] = true

checkLogin(4, "davinci").run(db)
// res9: cats.Id[Boolean] = false
```

4.7.4 When to Use Readers?

As you can hopefully see from the exercise, `Readers` effectively provide a simple 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, 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 that 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 a our 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 in this section 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.
>
> Kleislis are beyond the scope of this book, but will be easy to pick up based on your newfound knowledge of `Reader` and the content we'll cover in the next chapter on monad transformers.

4.8 The *State* Monad

`cats.data.State` allows us to pass additional state around as part of a computation. We define `State` instances representing atomic operations on the state, and thread them together using `map` and `flatMap`. In this way we can model mutable state in a purely functional way, without using mutation.

4.8.1 Creating and Unpacking State

Boiled down to its simplest form, instances of `State[S, A]` represent functions of type `S => (S, A)`. `S` is the type of the state and 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.StateT@ad944e3
```

In other words, an instance of `State` is a combination of two things:

- a transformation from an input state to an output state;
- a computation of 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 actually 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
```

4.8.2 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. Because each primitive instance represents a transformation on the state, the combined instance represents a more complex transformation.

```
val step1 = State[Int, String] { num =>
  val ans = num + 1
  (ans, s"Result of step1: $ans")
}
// step1: cats.data.State[Int,String] = cats.data.StateT@e680f25

val step2 = State[Int, String] { num =>
  val ans = num * 2
  (ans, s"Result of step2: $ans")
}
// step2: cats.data.State[Int,String] = cats.data.StateT@1119
    acde

val both = for {
  a <- step1
  b <- step2
} yield (a, b)
// both: cats.data.StateT[cats.Eval,Int,(String, String)] = cats
    .data.StateT@2d4d4ff3

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. The 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, then, is to represent each step of a computation as an instance of `State`, 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.StateT@26c929b1

getDemo.run(10).value
// res3: (Int, Int) = (10,10)

val setDemo = State.set[Int](30)
// setDemo: cats.data.State[Int,Unit] = cats.data.StateT@1748341
    a

setDemo.run(10).value
// res4: (Int, Unit) = (30,())

val pureDemo = State.pure[Int, String]("Result")
// pureDemo: cats.data.State[Int,String] = cats.data.StateT@
    1826901

pureDemo.run(10).value
// res5: (Int, String) = (10,Result)

val inspectDemo = State.inspect[Int, String](_ + "!")
// inspectDemo: cats.data.State[Int,String] = cats.data.StateT@
    77e7bb7e

inspectDemo.run(10).value
// res6: (Int, String) = (10,10!)
```

```
val modifyDemo = State.modify[Int](_ + 1)
// modifyDemo: cats.data.State[Int,Unit] = cats.data.StateT@56
    cf32b8

modifyDemo.run(10).value
// res7: (Int, Unit) = (11,())
```

We can assemble these building blocks into useful computations. We often end up ignoring the results of intermediate stages when they 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.
    StateT@4996bc50

val (state, result) = program.run(1).value
// state: Int = 3
// result: (Int, Int, Int) = (1,2,3000)
```

4.8.3 Exercise: Post-Order Calculator

The `State` monad allows us to implement simple evaluators for complex expressions, passing the values of mutable registers along in the state component. We model the atomic operations as instances of `State`, and combine them to evaluate whole sequences of inputs. 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 using a computer program.

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
```

We can write a simple interpreter for these expressions using the `State` monad. We can parse each symbol into a `State` instance representing a context-free stack transform and intermediate result. The `State` instances can be threaded together using `flatMap` to produce an interpreter for any sequence of symbols.

Let's do this now. 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 and fail.

```
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.StateT[cats.Eval,List[Int],Int] = cats.
    data.StateT@5f0e1a6c

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] = ???
// 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.runA(Nil).value
```

Because `evalOne` and `evalAll` both return instances of `State`, we can even 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.runA(Nil).value
```

4.9 Defining Custom Monads

We can define a `Monad` for a custom type by providing implementations of thee methods: `flatMap`, `pure`, and a new method 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)
      }
  }
```

`tailRecM` is an optimisation in Cats that limits the amount of stack space used by nested calls to `flatMap`. The technique comes from a 2015 paper by PureScript creator Phil Freeman. The method should recursively call itself as long as the result of `f` returns a `Right`. If we can make `tailRecM` tail recursive, we should do so to allow Cats to perform additional internal optimisations.

4.9.1 Exercise: Branching out Further with Monads

Let's write a `Monad` for our `Tree` data type from last chapter. Here's the type again, together with the smart constructors we used to simplify type class instance selection:

```
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.

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`.

See the solution

4.10 Summary

In this chapter we've seen monads up-close. We saw that `flatMap` can be viewed as sequencing computations, giving the order in which operations must happen. In this view, `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.

In this chapter 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 uses and many problems can be solved by using one of these constructs.

Finally, if we do have to implement our own monad instance, we've have learned about `tailRecM`. This is an odd wrinkle—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 mechanical benefits that we can be grateful for 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, database 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.

A question arises. Given two monads, can we make one monad out of them in a generic way? That is, do monads *compose?* We can try to write the code but we'll soon find it impossible to implement `flatMap`:

```
// This code won't actually compile.
// It's just illustrating a point:
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] =
      // This is impossible to implement in general
      // without knowing something about M1 or M2:
      ???
  }
}
```

We can't compose monads in general. However, some monad instances can be made to compose with instance-specific glue code. For these special cases we can use *monad transformers* to compose them.

Monad transformers allow us to squash together monads, creating one monad where we previously had two or more. With this transformed monad we can avoid nested calls to `flatMap`.

5.1 A Transformative Example

Cats provides a library of such transformers: `EitherT` for composing `Either` with other monads, `OptionT` for composing `Option`, and so on.

Here's an example that uses `OptionT` to squash `List` and `Option` into a single monad. Where we might use `List[Option[A]]` we can use `ListOption[A]` to avoid nested `flatMap` calls.

```
import cats.data.OptionT

type ListOption[A] = OptionT[List, A]
```

`ListOption` is a monad that combines the properties of `List` and `Option`. Note how we build it 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 with `pure` as usual:

```
import cats.Monad
import cats.instances.list._
import cats.syntax.applicative._

val result: ListOption[Int] = 42.pure[ListOption]
// result: ListOption[Int] = OptionT(List(Some(42)))
```

The `map` and `flatMap` methods of `ListOption` combine the corresponding methods of `List` and `Option` into single operations:

```
val a = 10.pure[ListOption]
// a: ListOption[Int] = OptionT(List(Some(10)))

val b = 32.pure[ListOption]
// b: ListOption[Int] = OptionT(List(Some(32)))

a flatMap { (x: Int) =>
  b map { (y: Int) =>
    x + y
  }
}
// res1: cats.data.OptionT[List,Int] = OptionT(List(Some(42)))
```

This is the basics of using 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

Note the imports in the code samples above—they 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.

We need an `Applicative[ListOption]` to call `pure`. We have `cats.data.OptionT` in scope, which provides the implicits for `OptionT`. However, in order to generate our `Applicative[ListOption]`, the implicits for `OptionT` also require an `Applicative` for `List`. Hence the additional import 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 hurt to import the syntax—the compiler will simply ignore it in favour of the explicit methods.

Remember that we're subjecting ourselves to this shenanigans because we're stubbornly refusing to import our instances from `cats.instances.all`. If we did that, everything would just work.

5.2 Monad Transformers in Cats

Monad transformers are a little different to the other abstractions we've seen—they don't have their own type class. We use monad transformers to build monads, which we then use via the `Monad` type class. Thus the main points of interest when using monad transformers are:

- the available transformer classes;
- building stacks of monads using transformers;
- constructing instances of a monad stack; and
- pulling apart a stack to access the wrapped monads.

5.2.1 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`, `cats.data.WriterT`, and `cats.data.StateT`;
- `cats.data.IdT` for the `Id` monad.

All of these monad transformers follow the same convention: the first type parameter specifies the monad that is wrapped around the monad implied by the transformer. The remaining type parameters are the types we've used to form the corresponding monads.

> *Kleisli Arrows*
>
> Last chapter, in the section on the `Reader` monad, we mentioned that `Reader` was a specialisation of a more general concept called a "kleisli arrow" (aka `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 why we were creating `Readers` last chapter and seeing `Kleislis` on the console.

5.2.2 Building Monad Stacks

Building monad stacks is a little confusing until you know the patterns. The first type parameter to a monad transformer is the *outer* monad in the stack—the transformer itself provides the inner monad. For example, our `ListOption` type above was built using `OptionT[List, A]` but the result was effectively a `List[Option[A]]`. In other words, we build monad stacks from the inside out.

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 make everything the correct shape:

```
import cats.instances.either._

type Error = String

// Create a type alias, ErrorOr, to convert Either to
// a type constructor with a single parameter:
type ErrorOr[A] = Either[Error, A]
```

```
// Use ErrorOr as a type parameter to OptionT:
type ErrorOptionOr[A] = OptionT[ErrorOr, A]
```

`ErrorOptionOr` is a monad. We can use `pure` and `flatMap` as usual to create and transform instances:

```
val result1 = 41.pure[ErrorOptionOr]
// result1: ErrorOptionOr[Int] = OptionT(Right(Some(41)))

val result2 = result1.flatMap(x => (x + 1).pure[ErrorOptionOr])
// result2: cats.data.OptionT[ErrorOr,Int] = OptionT(Right(Some
    (42)))
```

Now let's add another monad into our stack. 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:

```
import scala.concurrent.Future
import cats.data.EitherT

type FutureEitherOption[A] = OptionT[EitherT, A]
// <console>:25: error: cats.data.EitherT takes three type
    parameters, expected: one
//       type FutureEitherOption[A] = OptionT[EitherT, A]
//                                                 ^
```

As before, we solve the problem by creating a type alias with a single parameter. 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 Error = String
```

```
type FutureEither[A] = EitherT[Future, String, A]
type FutureEitherOption[A] = OptionT[FutureEither, A]
```

Our mammoth stack composes not two but *three* monads. Our `map` and `flatMap` methods cut through three layers of abstraction:

```
import scala.concurrent.ExecutionContext.Implicits.global
import cats.instances.future._

val answer: FutureEitherOption[Int] =
  for {
    a <- 10.pure[FutureEitherOption]
    b <- 32.pure[FutureEitherOption]
  } yield a + b
// answer: FutureEitherOption[Int] = OptionT(EitherT(Future(<not
     completed>)))
```

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 partial types. For example:

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

123.pure[EitherT[Option, String, ?]]
// res9: 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 we need to keep our code readable.

5.2.3 Constructing and Unpacking Instances

As we saw above, we can use `pure` to directly inject raw values into transformed monad stacks. We can also create instances from untransformed stacks using the monad transformer's `apply` method:

```
import cats.syntax.either._ // for foo.asRight
import cats.syntax.option._ // for foo.some

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

// Create using pure:
val stack1 = 123.pure[ErrorOrOption]
// stack1: ErrorOrOption[Int] = OptionT(Right(Some(123)))

// Create using apply:
val stack2 = OptionT[ErrorOr, Int](
  123.some.asRight[String]
)
// stack2: cats.data.OptionT[ErrorOr,Int] = OptionT(Right(Some
    (123)))
```

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:

```
stack1.value
// res13: ErrorOr[Option[Int]] = Right(Some(123))

stack2.value
// res14: ErrorOr[Option[Int]] = Right(Some(123))
```

Each call to `value` unpacks a single monad transformer, so we may need more than one call to completely unpack a large stack:

```
import cats.instances.vector._
import cats.data.{Writer, EitherT, OptionT}

type Logged[A] = Writer[Vector[String], A]
type LoggedFallable[A] = EitherT[Logged, String, A]
type LoggedFallableOption[A] = OptionT[LoggedFallable, A]

val packed = 123.pure[LoggedFallableOption]
// packed: LoggedFallableOption[Int] = OptionT(EitherT(WriterT((
    Vector(),Right(Some(123))))))

val partiallyPacked = packed.value
// partiallyPacked: LoggedFallable[Option[Int]] = EitherT(
    WriterT((Vector(),Right(Some(123)))))

val completelyUnpacked = partiallyPacked.value
// completelyUnpacked: Logged[Either[String,Option[Int]]] =
    WriterT((Vector(),Right(Some(123))))
```

5.2.4 Usage Patterns

Widespread use of monad tranformers 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.

One way of avoiding this is to use monad transformers as local "glue code". 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. Here's an example:

```
type Logged[A] = Writer[List[String], A]

// Example method that returns nested monads:
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)
  }

// Example combining multiple calls to parseNumber:
def addNumbers(
  a: String,
  b: String,
  c: String
): Logged[Option[Int]] = {
  import cats.data.OptionT

  // Transform the incoming stacks to work on them:
  val result = for {
    a <- OptionT(parseNumber(a))
    b <- OptionT(parseNumber(b))
    c <- OptionT(parseNumber(c))
  } yield a + b + c

  // Return the untransformed monad stack:
  result.value
}

// This approach doesn't force OptionT on other users' code:
val result1 = addNumbers("1", "2", "3")
// result1: Logged[Option[Int]] = WriterT((List(Read 1, Read 2,
    Read 3),Some(6)))

val result2 = addNumbers("1", "a", "3")
// result2: Logged[Option[Int]] = WriterT((List(Read 1, Failed
    on a),None))
```

5.2.5 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 these 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 have separate definitions 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 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 malfunctioning satellites or Decepticon interception. `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.

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")
// res25: String = Jazz and Bumblebee need a recharge.

tacticalReport("Bumblebee", "Hot Rod")
// res26: String = Bumblebee and Hot Rod are ready to roll out!

tacticalReport("Jazz", "Ironhide")
// res27: String = Comms error: Ironhide unreachable
```

5.4 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 such as below:

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

val a = Option(1.asRight[String])
// a: Option[Either[String,Int]] = Some(Right(1))

val b = Option(1.asRight[String])
// b: Option[Either[String,Int]] = Some(Right(1))

val result = for {
  x <- a
  y <- b
} yield {
  for {
    u <- x
    v <- y
  } yield u + v
}
// result: Option[scala.util.Either[String,Int]] = Some(Right(2)
    )
```

Each monad transformer, such as `FutureT`, `OptionT` or `EitherT`, pro-

vides 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:

```
import cats.data.EitherT

val wrappedA = EitherT(a)
// wrappedA: cats.data.EitherT[Option,String,Int] = EitherT(Some
    (Right(1)))

val wrappedB = EitherT(b)
// wrappedB: cats.data.EitherT[Option,String,Int] = EitherT(Some
    (Right(1)))

import cats.instances.option._

val wrappedResult = for {
  x <- wrappedA
  y <- wrappedB
} yield x + y
// wrappedResult: cats.data.EitherT[Option,String,Int] = EitherT
    (Some(Right(2)))

val result = wrappedResult.value
// result: Option[Either[String,Int]] = Some(Right(2))
```

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, `Cartesian` and `Applicative`, that support new kinds of operation such as `zipping` independent values within a context.

Chapter 6

Cartesians and Applicatives

In previous chapters we saw how functors and monads let us transform values using `map` and `flatMap`. While functors and monads are both immensely useful abstractions, there are types of transformation that are inconvenient to represent with these methods.

One such example is form validation. When we validate a form we want to return *all* the errors to the user, not simply 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._

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. Even on a multicore CPU, the code below runs in sequence as you can see from the timestamps:

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

lazy val timestamp0 = System.currentTimeMillis

def getTimestamp: Long = {
  val timestamp = System.currentTimeMillis - timestamp0
  Thread.sleep(100)
  timestamp
}

val timestamps = for {
  a <- Future(getTimestamp)
  b <- Future(getTimestamp)
  c <- Future(getTimestamp)
} yield (a, b, c)

Await.result(timestamps, 1.second)
// res5: (Long, Long, Long) = (0,106,210)
```

`map` and `flatMap` aren't quite capable of capturing what we want here 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:

- *Cartesians* encompass the notion of "zipping" pairs of contexts. Cats provides a `CartesianBuilder` syntax that combines `Cartesians` and `Functors` to allow users to join values within a context using arbitrary functions.
- *Applicative functors*, also known as `Applicatives`, extend `Cartesian` and `Functor` and provide 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 cartesian 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 a different formulations of Applicative, as well as the relationships between `Cartesian`, `Functor`, `Applicative`, and `Monad`, towards the end of the chapter.

6.1 *Cartesian*

`Cartesian` is a type class that allows us to "zip" values within a context. If we have two objects of type `F[A]` and `F[B]`, a `Cartesian[F]` allows us to combine them to form an `F[(A, B)]`. Its definition in Cats is:

```
trait Cartesian[F[_]] {
  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)]
}
```

As we discussed above, the parameters `fa` and `fb` are independent of one another. This gives us a lot more flexibility when defining instances of `Cartesian` than we do when defining `Monads`.

6.1.1 Joining Two Contexts

Whereas `Semigroups` allow us to join values, `Cartesians` allow us to join contexts. Let's join some `Options` as an example:

```
import cats.Cartesian
import cats.instances.option._ // Cartesian for Option

Cartesian[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`:

```
Cartesian[Option].product(None, Some("abc"))
// res1: Option[(Nothing, String)] = None

Cartesian[Option].product(Some(123), None)
// res2: Option[(Int, Nothing)] = None
```

6.1.2 Joining Three or More Contexts

The companion object for `Cartesian` 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._ // Cartesian for Option

Cartesian.tuple3(Option(1), Option(2), Option(3))
// res3: Option[(Int, Int, Int)] = Some((1,2,3))

Cartesian.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:

```
Cartesian.map3(
  Option(1),
  Option(2),
  Option(3)
)(_ + _ + _)
// res5: Option[Int] = Some(6)

Cartesian.map3(
  Option(1),
  Option(2),
  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 *Cartesian Builder* Syntax

Cats provides a convenient syntax called *cartesian builder syntax*, that provides shorthand for methods like `tupleN` and `mapN`. We import the syntax from `cats.syntax.cartesian`. Here's an example:

```
import cats.instances.option._
import cats.syntax.cartesian._

(Option(123) |@| Option("abc")).tupled
// res7: Option[(Int, String)] = Some((123,abc))
```

The `|@|` operator, better known as a "tie fighter", creates a temporary "builder" object that provides several methods for combining the parameters to create useful data types. For example, the `tupled` method zips the values into a tuple:

```
val builder2 = Option(123) |@| Option("abc")

builder2.tupled
// res8: Option[(Int, String)] = Some((123,abc))
```

We can use `|@|` repeatedly to create builders for up to 22 values. Each arity of builder, from 2 to 22, defines a `tupled` method to combine the values to form a tuple of the correct size:

```
val builder3 = Option(123) |@| Option("abc") |@| Option(true)

builder3.tupled
// res9: Option[(Int, String, Boolean)] = Some((123,abc,true))

val builder5 = builder3 |@| Option(0.5) |@| Option('x')

builder5.tupled
// res10: Option[(Int, String, Boolean, Double, Char)] = Some
    ((123,abc,true,0.5,x))
```

The idiomatic way of writing builder syntax is to combine `|@|` and `tupled` in a single expression, going from single values to a tuple in one step:

```
(
  Option(1) |@|
  Option(2) |@|
  Option(3)
).tupled
// res11: Option[(Int, Int, Int)] = Some((1,2,3))
```

In addition to `tupled`, every builder has a `map` method 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 and black")
).map(Cat.apply)
// res12: Option[Cat] = Some(Cat(Garfield,1978,Orange and black)
    )
```

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)).map(add)
// <console>:27: error: type mismatch;
//  found   : (Int, Int) => Int
//  required: (Int, Int, Int) => ?
//        (Option(1) |@| Option(2) |@| Option(3)).map(add)
//                                                ^

(Option("cats") |@| Option(true)).map(add)
// <console>:27: error: type mismatch;
//  found   : (Int, Int) => Int
//  required: (String, Boolean) => ?
//        (Option("cats") |@| Option(true)).map(add)
//                                          ^
```

6.2.1 Fancy Functors and Cartesian Builder Syntax

Cartesian builders also have a `contramap` and `imap` methods that accept Contravariant and Invariant functors. For example, we can combine `Monoids` and `Semigroups` using `Invariant`. Here's an example:

```
import cats.Monoid
import cats.instances.boolean._
import cats.instances.int._
import cats.instances.list._
import cats.instances.string._
import cats.syntax.cartesian._

case class Cat(
  name: String,
  yearOfBirth: Int,
  favoriteFoods: List[String]
)

def catToTuple(cat: Cat) =
  (cat.name, cat.yearOfBirth, cat.favoriteFoods)

implicit val catMonoid = (
  Monoid[String] |@|
  Monoid[Int] |@|
  Monoid[List[String]]
).imap(Cat.apply)(catToTuple)
```

Our `Monoid` allows us to create "empty" `Cats` and add `Cats` together using the syntax from Chapter 2:

```
import cats.syntax.monoid._

Monoid[Cat].empty
// res18: Cat = Cat(,0,List())

val garfield   = Cat("Garfield", 1978, List("Lasagne"))
val heathcliff = Cat("Heathcliff", 1988, List("Junk Food"))

garfield |+| heathcliff
// res19: Cat = Cat(GarfieldHeathcliff,3966,List(Lasagne, Junk
    Food))
```

6.3 *Cartesian* Applied to Different Types

`Cartesians` don't always provide the behaviour we expect, particularly for types that also have instances of `Monad`. We have seen the behaviour of the `Cartesian` for `Option`. Let's look at some examples for other types.

6.3.1 *Cartesian* Applied to *Future*

The semantics for `Future` are pretty much what we'd expect, providing parallel as opposed to sequential execution:

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

import cats.Cartesian
import cats.instances.future._

val futurePair = Cartesian[Future].
  product(Future("Hello"), Future(123))

Await.result(futurePair, 1.second)
// res2: (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`. Cartesian builder syntax provides a concise syntax for zipping fixed numbers of `Futures`:

```
import cats.syntax.cartesian._

case class Cat(
  name: String,
  yearOfBirth: Int,
  favoriteFoods: List[String]
```

```
)

val futureCat = (
  Future("Garfield") |@|
  Future(1978)       |@|
  Future(List("Lasagne"))
).map(Cat.apply)

Await.result(futureCat, 1.second)
// res5: Cat = Cat(Garfield,1978,List(Lasagne))
```

6.3.2 *Cartesian* Applied to *List*

There is a `Cartesian` instance for `List`. What value do you think the following expression will produce?

```
import cats.Cartesian
import cats.instances.list._

Cartesian[List].product(List(1, 2), List(3, 4))
```

There are at least two reasonable answers:

1. `product` could *zip* the lists, returning `List((1, 3), (2, 4))`;
2. `product` could compute the *cartesian product*, taking every element from the first list and combining it with every element from the second returning `List((1, 3), (1, 4), (2, 3), (2, 4))`.

The name `Cartesian` is a hint as to which answer we'll get, but let's run the code to be sure:

```
Cartesian[List].product(List(1, 2), List(3, 4))
// res8: List[(Int, Int)] = List((1,3), (1,4), (2,3), (2,4))
```

We get the cartesian product! This is perhaps surprising: zipping lists tends to be a more common operation.

6.3.3 *Cartesian* Applied to *Either*

What about `Either`? We opened this chapter with a discussion of fail-fast versus accumulating error-handling. Which behaviour will `product` produce?

```
import cats.instances.either._

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

Cartesian[ErrorOr].product(
  Left(Vector("Error 1")),
  Left(Vector("Error 2"))
)
// res10: ErrorOr[(Nothing, Nothing)] = Left(Vector(Error 1))
```

Surprisingly, we still get fail-fast semantics. The `product` method sees the first failure and stops, despite knowing that the second parameter is also a failure.

6.3.4 *Cartesian* Applied to Monads

The reason for these surprising results is that, like `Option`, `List` and `Either` are both monads. To ensure consistent semantics, Cats' `Monad` (which extends `Cartesian`) provides a standard definition of `product` in terms of `map` and `flatMap`.

Try writing this implementation now:

```
import scala.language.higherKinds
import cats.Monad

def product[M[_]: Monad, A, B](
  fa: M[A],
  fb: M[B]
): M[(A, B)] = ???
```

See the solution

We can implement `product` in terms of the monad operations, and Cats enforces this implementation for all monads. This gives what we might think of as unexpected and less useful behaviour for a number of data types. The consistency of semantics is actually useful for higher level abstractions, but we don't know about those yet.

So why bother with `Cartesian` at all? The answer is that we can create useful data types that have instances of `Cartesian` (and `Applicative`) but not `Monad`. This frees us to implement `product` in different ways. Let's examing this further by looking at a new data type for error handling.

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 rules between these two methods.

Fortunately, Cats provides a data type called `Validated` that has an instance of `Cartesian` but *no* instace of `Monad`. The implementation of `product` is therefore free to accumulate errors:

```
import cats.Cartesian
import cats.data.Validated
import cats.instances.list._ // Semigroup for List

type AllErrorsOr[A] = Validated[List[String], A]

Cartesian[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`. We can create instances directly using their `apply` methods:

```
val v = Validated.Valid(123)
// v: cats.data.Validated.Valid[Int] = Valid(123)

val i = Validated.Invalid("Badness")
// i: cats.data.Validated.Invalid[String] = Invalid(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[String, Int](123)
// v: cats.data.Validated[String,Int] = Valid(123)
```

```
val i = Validated.invalid[String, Int]("Badness")
// i: cats.data.Validated[String,Int] = Invalid(Badness)
```

As a third option we can import the `valid` and `invalid` extension methods from `cats.syntax.validated`:

```
import cats.syntax.validated._

123.valid[String]
// res2: cats.data.Validated[String,Int] = Valid(123)

"Badness".invalid[Int]
// res3: cats.data.Validated[String,Int] = Invalid(Badness)
```

Finally, there are a variety of methods on `Validated` to create instances from different sources. We can create them from `Exceptions`, as well as instances of `Try`, `Either`, and `Option`:

```
Validated.catchOnly[NumberFormatException]("foo".toInt)
// res4: cats.data.Validated[NumberFormatException,Int] =
    Invalid(java.lang.NumberFormatException: For input string: "
    foo")

Validated.catchNonFatal(sys.error("Badness"))
// res5: cats.data.Validated[Throwable,Nothing] = Invalid(java.
    lang.RuntimeException: Badness)

Validated.fromTry(scala.util.Try("foo".toInt))
// res6: cats.data.Validated[Throwable,Int] = Invalid(java.lang.
    NumberFormatException: For input string: "foo")

Validated.fromEither[String, Int](Left("Badness"))
// res7: cats.data.Validated[String,Int] = Invalid(Badness)

Validated.fromOption[String, Int](None, "Badness")
// res8: 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 described above: `product`, `map2..22`, cartesian builder syntax, and so on.

All of these techniques require an appropriate `Cartesian` 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 `Cartesian`:

```
type AllErrorsOr[A] = Validated[String, A]
```

`Validated` accumulates errors using a `Semigroup`, so we need one of those in scope to summon the `Cartesian`. If we don't have one we get an annoyingly unhelpful compilation error:

```
Cartesian[AllErrorsOr]
// <console>:22: error: could not find implicit value for
    parameter instance: cats.Cartesian[AllErrorsOr]
//        Cartesian[AllErrorsOr]
//                  ^
```

Once we import a `Semigroup[String]`, everything works as expected:

```
import cats.instances.string._

Cartesian[AllErrorsOr]
// res10: cats.Cartesian[AllErrorsOr] = cats.data.
    ValidatedInstances$$anon$1@5e0fe145
```

As long as the compiler has all the implicits in scope to summon a `Cartesian` of the correct type, we can use cartesian builder syntax or any of the other `Cartesian` methods to accumulate errors as we like:

```
import cats.syntax.cartesian._

(
  "Error 1".invalid[Int] |@|
  "Error 2".invalid[Int]
).tupled
// res11: cats.data.Validated[String,(Int, Int)] = Invalid(Error
     1Error 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._

(
  Vector(404).invalid[Int] |@|
  Vector(500).invalid[Int]
).tupled
// res12: 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
// res13: 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)
// res14: cats.data.Validated[Nothing,Int] = Valid(12300)

"?".invalid.leftMap(_.toString)
// res15: cats.data.Validated[String,Nothing] = Invalid(?)

123.valid[String].bimap(_ + "!", _ * 100)
// res16: cats.data.Validated[String,Int] = Valid(12300)

"?".invalid[Int].bimap(_ + "!", _ * 100)
// res17: cats.data.Validated[String,Int] = Invalid(?!)
```

We can't flatMap because Validated isn't a monad. However, we can convert back and forth between Validated and Either using the toEither and toValidated methods. This allows us to switch error-handling semantics on the fly. Note that toValidated comes from [cats.syntax.either]:

```
import cats.syntax.either._ // toValidated method
// import cats.syntax.either._

"Badness".invalid[Int]
// res18: cats.data.Validated[String,Int] = Invalid(Badness)

"Badness".invalid[Int].toEither
// res19: Either[String,Int] = Left(Badness)

"Badness".invalid[Int].toEither.toValidated
// res20: 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)
// res22: Int = 0

"fail".invalid[Int].fold(_ + "!!!", _.toString)
// res23: 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 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 complete 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 an `Int` and parses it as a `String`.

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 `Cartesian` 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*

Cartesians 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). Cartesians and applicatives effectively provide alternative encodings of the notion of "zipping" values. Both encodings are introduced in the same 2008 paper by Conor McBride and Ross Paterson.

Cats models `Applicatives` using two type classes. The first, `Apply`, extends `Cartesian` and `Functor` and adds an `ap` method that applies a parameter to a function within a context. The second, `Applicative` extends `Apply`, adds the `pure` method introduced in Chapter 4. Here's a simplified definition in code:

```
trait Apply[F[_]] extends Cartesian[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 `Cartesian` 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 a whole family of type classes that concern themselves with sequencing computations in different ways. Figure 6.1 shows the big picture.

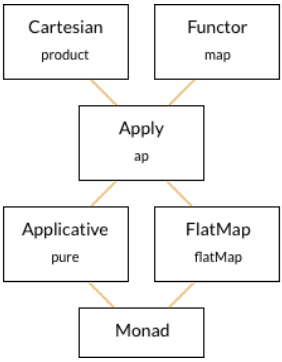

Figure 6.1: Monad type class hierarchy

Each type class in the hierarchy represents a particular set of sequencing semantics. It introduces its characteristic methods, and defines all of the functionality from its supertypes in terms of them. Every monad is an applicative, every applicative a cartesian, 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 cartesians impose no such restriction. This puts them in another 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, friend!

6.6 Summary

While monads and functors are the most widely used sequencing data types we've covered in this book, cartesians 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.

Cartesians and applicatives 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 cartesian builder 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 covered.

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 with folds.

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 on the general concept of folding. `Foldable` is a type class for folding over sequences. We supply an *accumulator* value and a *binary function* to combine it with an 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 view provided by `Foldable` is recursive. Our binary function is called repeatedly for each item in the sequence, result from each call becoming the accumulator for the next. When we reach the end of the sequence, the final accumulator becomes our 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 commutative. For example, we can sum a `List[Int]` by folding in either direction, using `0` as our accumulator and + as our operation:

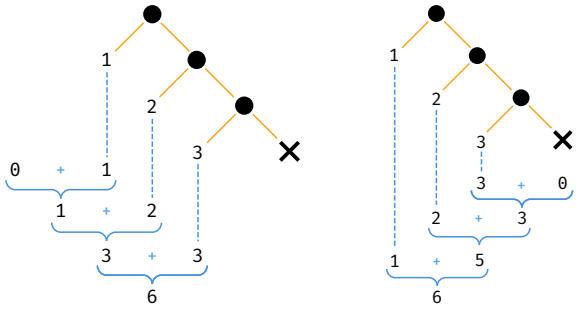

Figure 7.1: Illustration of foldLeft and foldRight

```
List(1, 2, 3).foldLeft(0)(_ + _)
// res2: Int = 6

List(1, 2, 3).foldRight(0)(_ + _)
// res3: Int = 6
```

If provide a non-commutative operator the order of evaluation makes a difference. For example, if we fold using `-`, 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 for free. Cats provides out-of-the-box instances of `Foldable` for a handful of Scala data types: `List`, `Vector`, `Stream`, `Option`, and `Map`.

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._

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 0 or 1 elements:

```
import cats.instances.option._

val maybeInt = Option(123)
```

```
Foldable[Option].foldLeft(maybeInt, 10)(_ * _)
// res3: Int = 1230
```

Finally, here is an example for `Map`. The `Foldable` instance folds over the values in the map (as opposed to its keys). `Map` has two type parameters so we have to fix the key type to summon the `Foldable`:

```
import cats.instances.map._

type StringMap[A] = Map[String, A]

val stringMap = Map("a" -> "b", "c" -> "d")

Foldable[StringMap].foldLeft(stringMap, "nil")(_ + "," + _)
// res6: String = nil,b,d
```

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 `StackOverflowException`:

```
import cats.Eval
import cats.Foldable

def bigData = (1 to 100000).toStream

bigData.foldRight(0)(_ + _)
```

```
// java.lang.StackOverflowError ...
```

Using `Foldable` forces us to use stack safe operations, which fixes the overflow exception:

```
import cats.instances.stream._

val eval = Foldable[Stream].
  foldRight(bigData, Eval.now(0)) { (num, eval) =>
    eval.map(_ + num)
  }
// eval: cats.Eval[Int] = cats.Eval$$anon$8@5fa48dad

eval.value
// res10: Int = 705082704
```

> *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(0)(_ + _)
> // res11: Int = 705082704
>
> (1 to 100000).toVector.foldRight(0)(_ + _)
> // res12: Int = 705082704
> ```
>
> We've called out `Stream` because it is an exception to this rule. Whatever data type we're using, though, it's useful to know that `Eval` has our backs.

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 facimiles of familiar methods from the

standard library: `find`, `exists`, `forall`, `toList`, `isEmpty`, `nonEmpty`, and so on:

```
Foldable[Option].nonEmpty(Option(42))
// res13: Boolean = true

Foldable[List].find(List(1, 2, 3))(_ % 2 == 0)
// res14: 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._ // Monoid for Int

Foldable[List].combineAll(List(1, 2, 3))
// res15: Int = 6
```

Alternatively, we can use `foldMap` to convert each `Int` to a `String` and concatenate them:

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

Foldable[List].foldMap(List(1, 2, 3))(_.toString)
// res16: String = 123
```

Finally, we can compose `Foldables` to support deep traversal of nested sequences:

```
import cats.instances.vector._ // Monoid of Vector

val ints = List(Vector(1, 2, 3), Vector(4, 5, 6))

(Foldable[List] compose Foldable[Vector]).combineAll(ints)
// res18: 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._

List(1, 2, 3).combineAll
// res19: Int = 6

List(1, 2, 3).foldMap(_.toString)
// res20: 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)(_ + _)
> // res21: 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`, simply 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:

```
object Future {
  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 "effect": `Future`, `Option`, `Validated`, and so on. We'll approach `Traverse` in the next sections in two steps: first we'll generalise over the effect type, 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._
import cats.syntax.applicative._

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 `Cartesian.combine`:

```
import cats.syntax.cartesian._

// Combining an accumulator and a hostname using an Applicative:
def newCombine(
  accum: Future[List[Int]],
  host: String
): Future[List[Int]] =
  (accum |@| getUptime(host)).map(_ :+ _)
```

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)).map(_ :+ _)
```

```
  }

def listSequence[F[_] : Applicative, B]
    (list: List[F[B]]): F[List[B]] =
  listTraverse(list)(identity)
```

We can use this new `listTraverse` to re-implement our uptime example:

```
Await.result(
  listTraverse(hostnames)(getUptime),
  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._

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._

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, gere's an example that uses `Validated`:

```
import cats.data.Validated
import cats.instances.list._ // Applicative[ErrorsOr] needs a
    Monoid[List]

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` effect, 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)(func)
}
```

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` as usual and use the `traverse` and `sequence` methods as described in the previous section:

```
import cats.Traverse
import cats.instances.future._
import cats.instances.list._

Await.result(
  Traverse[List].traverse(hostnames)(getUptime),
  1.second
```

```
)
// res0: List[Int] = List(1020, 960, 840)

val numbers = List(Future(1), Future(2), Future(3))

Await.result(
  Traverse[List].sequence(numbers),
  1.second
)
// res1: List[Int] = List(1, 2, 3)
```

There are also syntax versions of the methods, imported via `cats.syntax.traverse`:

```
import cats.syntax.traverse._

Await.result(hostnames.traverse(getUptime), 1.second)
// res2: List[Int] = List(1020, 960, 840)

Await.result(numbers.sequence, 1.second)
// res3: 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!

7.2.4 *Unapply*, *traverseU*, and *sequenceU*

One frequent problem people encounter when using `Traverse` is that it doesn't play well with effects with two or more type parameters. For example, suppose we have a `List` of `Either`s:

```
import cats.instances.list._
import cats.syntax.traverse._

val eithers: List[Either[String, String]] = List(
  Right("Wow!"),
  Right("Such cool!")
```

```
)
```

When we call `sequence` we get a compile error:

```
eithers.sequence
// <console>:20: error: Cannot prove that Either[String,String]
    <:< G[A].
//        eithers.sequence
//                ^
```

The reason for this failure is that the compiler can't find an implicit `Applicative`. This isn't a problem in our code—we have the correct syntax and instances in scope—it's simply a weakness of Scala's type inference that has only recently been fixed (more on the fix in a moment).

To understand what's going on, let's look again at the definition of `sequence`:

```
trait Traverse[F[_]]
  def sequence[G[_]: Applicative, B]: G[F[B]] =
    // etc...
}
```

To compile a call like `eithers.sequence`, the compiler has to find values for the type parameters `G` and `B`. The types it is attempting to unify them with are `Either[String, Int]` and `Int`, so it has to make a decision about which parameter on `Either` to fix to create a type constructor of the correct shape.

There are two possible solutions as you can see below:

```
type G[A] = Either[A, Int]
type G[A] = Either[String, A]
```

It's obvious to us which unification method to choose. However, prior to Scala 2.12, an infamous compiler limitation called SI-2712 prevented this inferrence.

To work around this issue Cats provides a utilitiy type class called `Unapply`, whose purpose is to tell the compiler which parameters to "fix" to create a unary type constructor for a given type. Cats provides instances of `Unapply` for the common binary types: `Either`, `Either`, `Validated`, and `Function1`, and so on. `Traverse` provides variants of `traverse` and `sequence` called `traverseU` and `sequenceU` that use `Unapply` to guide the compiler to the correct solution:

```
import cats.instances.either._

eithers.sequenceU
// res2: scala.util.Either[String,List[String]] = Right(List(Wow
    !, Such cool!))
```

The inner workings of `Unapply` aren't particularly important— all we need to know is that this tool is available to fix these kinds of problems.

> *Fixes to SI-2712*
>
> SI-2712 is fixed in Lightbend Scala 2.12.1 and Typelevel Scala 2.11.8. The fix allows calls to `traverse` and `sequence` to compile in a much wider set of cases, although tools like `Unapply` are still necessary in certain situations.
>
> The SI-2712 fix can be backported to Scala 2.11 and 2.10 using this compiler plugin.

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 standaed 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`.

Finally we looked at the `Unapply` type class, which works around restrictions in the compiler and allows us to use methods like `traverse` with types that have multiple type parameters. Fixes in recent releases of Scala make `Unapply` less important than it once was, but will still be a necessity in many Scala versions to come.

...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 2!

Part II

Case Studies

Chapter 8

Case Study: Testing Asynchronous Code

We'll start with a simple 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._
import cats.instances.list._
import cats.syntax.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!

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

[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`.

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 them 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 header 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 get the class and method headers compiling, then we'll turn our attention to the method bodies. Starting with the method headers:

- 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 Conclusions

This case study provides a nice introduction to 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: Pygmy Hadoop

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 tens or hundreds 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]`:

[1] In Hadoop there is also a shuffle phase that we will ignore here.

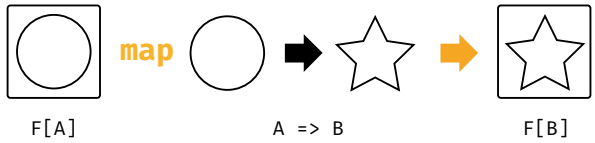

Figure 9.1: Type chart: functor map

`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).

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`.

Figure 9.2: Type chart: fold

If you remember from our discussion of `Foldable`, then depending on the reduction operation we use, the order of combination can have effect on the final result. To remain correct we need to ensure our reduction operation 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 we preserve the ordering on the sequence of elements we're processing.

Our fold operation requires us to seed the computation with an element of type B. Since our fold may be split into an arbitrary number of parallel steps, the seed should not effect 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:

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`.

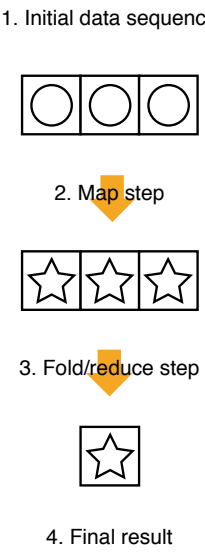

Figure 9.3: *foldMap* algorithm

Here's some sample output for reference:

```
import cats.instances.int._
```

`scala foldMap(Vector(1, 2, 3))(identity) // res1: Int = 6scala import cats.instances.string._ “‘`

```
// Mapping to a String uses the concatenation monoid:
foldMap(Vector(1, 2, 3))(_.toString + "! ")
// res3: String = "1! 2! 3! "

// Mapping over a String to produce a String:
foldMap("Hello world!".toVector)(_.toString.toUpperCase)
// res5: 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 simply 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`.

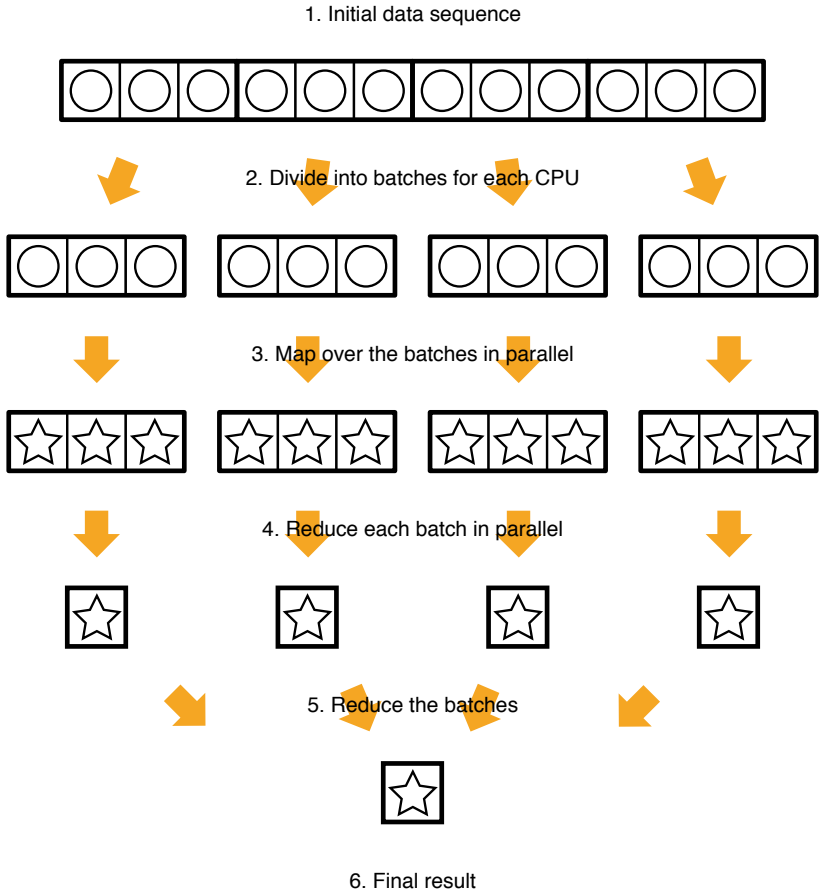

Figure 9.4: *parallelFoldMap* algorithm

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 `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)))
// res7: scala.concurrent.Future[List[Int]] = Future(Success(
    List(1, 2, 3)))
```

or an instance of `Traverse`:

```
import cats.instances.future._ // Applicative for Future
import cats.instances.list._   // Traverse for List
import cats.syntax.traverse._  // foo.sequence syntax

List(Future(1), Future(2), Future(3)).sequence
// res8: scala.concurrent.Future[List[Int]] = Future(Success(
    List(1, 2, 3)))
```

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 forever until a result
     arrives
// res9: Int = 1
```

There are also `Monad` and `Monoid` implementations for `Future` available from `cats.instances.future`:

```
import cats.Monad
import cats.instances.future._

Monad[Future].pure(42)

import cats.Monoid
import cats.instances.int._

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 = 8
```

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 monoidal addition.

9.4.1 Batching Strategies in the Real World

The main bottleneck in real map-reduce is network communication between the nodes. To counter this, systems like Hadoop provide mechanisms for pre-batching data to limit the communication required to distribute work.

Our toy system is designed to emulate this real-world batching behaviour. However, in reality we are running all of our work on a single machine where communcation between nodes is negligable. We don't actually need to pre-batch data to gain efficient parallel processing of a list. We can simply map:

```
val future1: Future[Vector[Int]] =
  (1 to 1000).toVector.
    traverse(item => Future(item + 1))
```

and reduce using a `Monoid`:

```
val future2: Future[Int] =
  future1.map(_.combineAll)

Await.result(future2, 1.second)
// res4: Int = 501500
```

9.4.2 Reduction using *Monoids*

Regardless of the batching strategy, mapping and reducing with `Monoids` is a powerful and general framework. The core idea of monoidal addition underlies Summingbird, Twitter's framework that powers all their internal data processing jobs.

Monoids are not restricted 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 hence 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 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 the check, or some kind of error message. We can abstactly represent this as a value in a context, where the context is the possibility of an error message as shown in Figure 10.1.

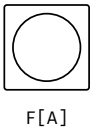

Figure 10.1: A validation result

A check itself is therefore a function that transforms a value into a value in a context as shown in Figure 10.2.

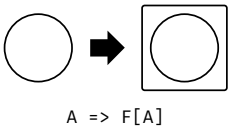

Figure 10.2: A validation check

Combine checks

How do we combine smaller checks into larger ones? Is this an applicative or cartesian as shown in Figure 10.3?

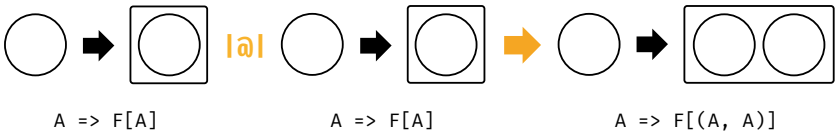

Figure 10.3: Applicative combination of checks

Not really. With a cartesian, 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*:

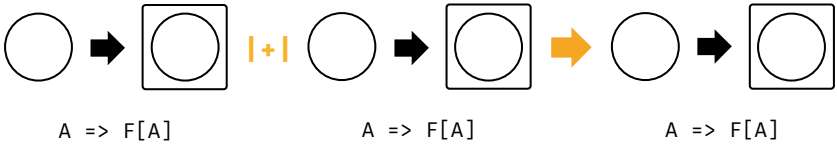

Figure 10.4: Monoidal combination of checks

We'll probably be using *and* and *or* about equally often with our validation 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.

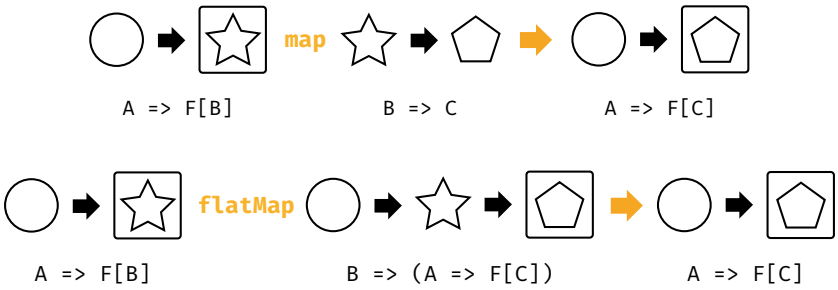

Figure 10.5: Monadic combination of checks

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...
}
```

If you think back to 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 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...
}
```

You should very quickly run into a problem: 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

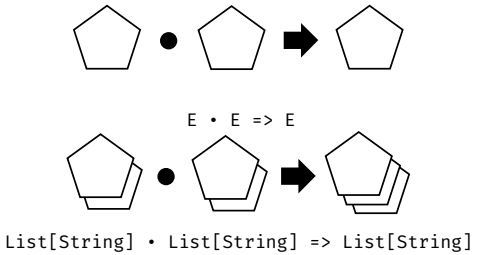

Figure 10.6: Combining error messages

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 behavior is?

See the solution

Use this knowledge to implement and. Make sure you end up with the behavior 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 compliment `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 succesful. 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))
      .map((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._ // |+| syntax
import cats.syntax.cartesian._ // |@| syntax
import cats.data.Validated._   // 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)).map((_, _) => 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. using the same ADT strategy we used for `Predicate`:

```
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 igure 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

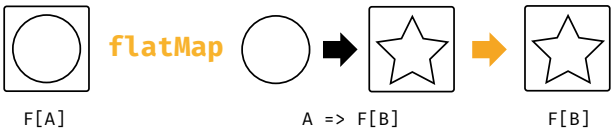

Figure 10.7: Type chart for flatMap

us the relationships shown in Figure 10.8:

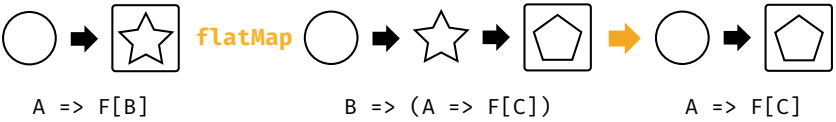

Figure 10.8: Type chart for flatMap applied to Check

In words, the semantics of applying a `FlatMap` are:

- 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. Check 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, OneAnd, Validated}
import cats.instances.list._
import cats.syntax.cartesian._
import cats.syntax.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:

- 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:

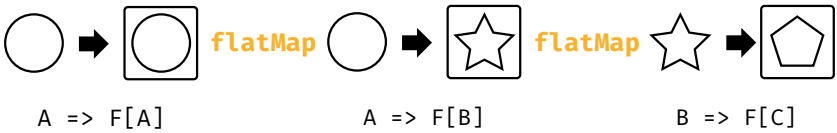

Figure 10.9: Sequencing monadic transforms

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 `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 the `Reader` monad!

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._
```

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 rmail 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 Conclusions

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`.

`Predicate` is very much like a stripped down version of the matchers found in testing libraries like ScalaTest and Specs2. One next step would be to develop a more elaborate predicate library along these lines. There are a few other directions to consider.

With the current representation of `Predicate` there is no way to implement logical negation. To implement negation we need to know the error message that a successful predicate would have returned if it had failed (so that the negation can return that message). One way to implement this is to have a predicate return a `Boolean` flag indicating success or failure and the associated message.

We could also do better in how error messages are represented. At the moment there is no indication with an error message of the structure of the predicates that failed. For example, if we represent error messsages as a `List[String]` and we get back the message:

```
List("Must be longer than 4 characters",
     "Must not contain a number")
```

does this message indicate a failing conjunction (two `and`s) or a failing disjunction (two `or`s)? We can probably guess in this case but in general we don't have sufficient information to work this out. We can solve this problem by wrapping all messages in a type as follows:

```
sealed trait Structure[E]

final case class Or[E](messages: List[Structure[E]])
  extends Structure[E]

final case class And[E](messages: List[Structure[E]])
  extends Structure[E]

final case class Not[E](messages: List[Structure[E]])
  extends Structure[E]

final case class Pure[E](message: E)
  extends Structure[E]
```

We can simplify this structure by converting all predicates into a normal form. For example, if we use disjunctive normal form the structure of the predicate will always be a disjunction (logical or) of conjunctions

(logical and). By doing so we could errors as a `List[List[Either[E, E]]]`, with the outer list representing disjunction, the inner list representing conjunction, and the `Either` representing negation.

Finally, we made several design choices that reasonable people could 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? The name `run` makes sense if you have experienced monad transformers and other similar abstractions, but is not clear if you don't have this experience. Many functional programmers come to prefer avoiding subtyping, as 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: Commutative Replicated Data Types

In this case study we explore commutative replicated data types (CRDTs), a data structure that can be used to reconcile eventually consistent data.

We 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 the issues that arise, and finally model the solutions in Scala.

Our goal here is to focus on the implementation in Scala of a particular type of CRDT. We're specifically 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. We can build a sys-

tem 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 simple to work with but they have their disadvantages. They tend to have high latency, as every change can result is many messages being sent between machines. They can also can have relatively low uptime. A network problem can cause some machines to be unable to communicate with others. This is called a network partition. When there is a network partition a consistent system may refuse further updates as allowing them could result in data becoming inconsistent between the partitioned systems.

An alternative approach is an *eventually consistent* system. This means that if all machines can communicate and there are no further updates they will evenutally all have the same view of data. However, at any particular point in time machines are allowed to have differing views of data.

Latency can be lower because eventually consistent systems require less communication between machines. A partitioned machine can still accept updates and reconcile its changes when the network is fixed, so systems can also can have better uptime. How exactly are we to do this reconciliation, though? CRDTs are 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. It can be used, for example, for counting the number of visitors to a 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 count of visitors. Let's call the machines `A` and `B`. Each machine is storing just an integer counter and the counters all start at zero.

```
A: 0
B: 0
```

`A` serves three visitors, and `B` two.

```
A: 3
B: 2
```

Now the machines want to merge their counters so they each have an up-to-date view of the total number of visitors. At this point we know the machines should add together their counters, because we know the history of their interactions. However, there is nothing in the data the machines store that records this. Nonetheless, let's use addition as our strategy for merging counters and see what happens.

```
A: 5
B: 5
```

Now `A` serves a single visitor.

```
A: 6
B: 4
```

The machines attempt to merge counters again. If they use addition as the merging algorithm they will end up with

```
A: 10
B: 10
```

This is clearly wrong! There have only been six visitors in total. Do we need to store the complete history of interactions to be able to compute the correct value? It turns out we do not, so let's look at the GCounter now to see how it solves this problem in an elegant way.

11.2.2 GCounters

The first clever idea in the GCounter is to have each machine storing a *separate* counter for every machine (including itself) that it knows about. 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`.

```
Machine A   Machine B
A: 0        A: 0
B: 0        B: 0
```

The rule with these counters is that a given machine is only allowed to increment it's own counter. If `A` serves 3 visitors and `B` serves two visitors the counters will look like

```
Machine A   Machine B
A: 3        A: 0
B: 0        B: 2
```

Now when two machines merge their counters the rule is to take the largest value stored for a given machine. Given the state above, when `A` and `B` merge counters the result will be

```
Machine A   Machine B
A: 3        A: 3
B: 2        B: 2
```

as 3 is the largest value stored for the `A` counter, and 2 is the largest value stored for the `B` counter. The combination of only allowing machines to increment their counter and choosing the maximum value on merging means we get the correct answer without storing the complete history of interactions.

If a machine wants to calculate the current value of the counter (given its current knowledge of other machines' state) it simply sums up all the per-machine counter. Given the state

```
Machine A   Machine B
A: 3        A: 3
B: 2        B: 2
```

each machine would report the current values as `3 + 2 = 5`.

11.2.3 Exercise: GCounter Implementation

We can implement a GCounter with the interface

```
final case class GCounter(counters: Map[String, Int]) {
  def increment(machine: String, amount: Int) =
    ???

  def get: Int =
    ???

  def merge(that: GCounter): GCounter =
    ???
}
```

where we represent machine IDs as `Strings`.

Finish the implementation.

See the solution

11.3 Generalisation

We've now created a distributed eventually consistent increment only counter. This is a nice achievement, but don't want to stop here. In this section we will attempt to abstract the operations used 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 `get`); - maximum (in `merge`); - and the identity element 0 (in `increment` and `merge`).

This should already make you feel there is a monoid somewhere in here, but let's look in more detail on the properties we rely on.

As a refresher, here are the properties we've of monoids we've seen earlier:

- the binary operation + is associative, meaning `(a + b) + c = a + (b + c)`;
- the identity `0` is commutative, meaning `a + 0 = 0 + a`; and
- the identity is an identity, meaning `a + 0 = a`.

In `increment`, we need an identity to initialise the counter. We also rely on associativity to ensure the specific sequence of additions we perform gives the correct value.

In `get` we implicitly rely on associativity and commutivity 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 `merge` relies on are a bit more interesting. We rely on commutivity 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. Formally, a binary operation `max` is idempotent if `a max a = a`.

Written more compactly, we have:

Method	Identity	Commutative	Associative	Idempotent
`increment`	Y	N	Y	N
`get`	Y	Y	Y	N
`merge`	Y	Y	Y	Y

From this we can see that

- `increment` requires a monoid;
- `get` 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 meeting these properties. A simple example is a set, with union being the binary operation and the identity element

the empty set. Set union is idempotent, commutative, and associative and therefore fits all our requirements to work with a GCounter. With this simple substitution of `Int` for `Set[A]` we can create a GSet type.

11.3.1 Implementation

Let us now implement this generalisation in code. Remember `increment` and `get` require a commutative monoid and `merge` requires a bounded semilattice (or idempotent commutative monoid).

Cats provides a `Monoid`, but no commutative monoid or bounded semilattice type class[1]. For simplicity of implementation we'll use `Monoid` when we really mean a commutative monoid, and require the programmer to ensure the implementation is commutative. We'll implement our own `BoundedSemiLattice` type class.

```
import cats.Monoid

trait BoundedSemiLattice[A] extends Monoid[A] {
  def combine(a1: A, a2: A): A
  def empty: A
}
```

In the implementation above, `BoundedSemiLattice[A]` extends `Monoid[A]` because a bounded semilattice is a monoid (a commutative idempotent one, to be exact).

11.3.2 Exercises

11.3.2.1 BoundedSemiLattice Instances

Implement some `BoundedSemiLattice` type class instances (e.g. for `Int` and `Set`).

[1] A closely related library called Spire provides both these abstractions.

See the solution

11.3.2.2 Generic GCounter

Using `Monoid` 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 of code. We're using monoids to design a large component—our CRDTs—but they are also useful in the small, making our code simpler.

See the solution

11.4 Abstracting GCounter to a Type Class

We've created a generic GCounter that works with any value that has (commutative) `Monoid` and `BoundedSemiLattice` type class instances. 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 might like to work with our GCounter, 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. Try your own implementation before reading on.

A simple way to achieve this is by defining a `GCounter` type class with dependencies on `Monoid` and `BoundedSemiLattice`. I defined this

type class as taking a higher-kinded type with *two* type parameters, intended to represent the key and value types of the map abstraction.

```
import scala.language.higherKinds
import cats.Monoid

trait BoundedSemiLattice[A] extends Monoid[A] {
  def combine(a1: A, a2: A): A
  def empty: A
}
object BoundedSemiLattice {
  implicit object intBoundedSemiLatticeInstance extends
    BoundedSemiLattice[Int] {
    def combine(a1: Int, a2: Int): Int =
      a1 max a2

    val empty: Int = 0
  }

  implicit def setBoundedSemiLatticeInstance[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]
    }
}

trait GCounter[F[_,_],K, V] {
  def increment(f: F[K, V])(k: K, v: V)(implicit m: Monoid[V]):
    F[K, V]
  def total(f: F[K, V])(implicit m: Monoid[V]): V
  def merge(f1: F[K, V], f2: F[K, V])(implicit b:
    BoundedSemiLattice[V]): F[K, V]
}
```

We can easily define some instances of this type class. Here's a complete example, containing a type class instance for `Map` and a simple

test.

```
import cats.syntax.semigroup._
import cats.syntax.foldable._

object GCounterExample {
  trait BoundedSemiLattice[A] extends Monoid[A] {
    def combine(a1: A, a2: A): A
    def empty: A
  }
  object BoundedSemiLattice {
    implicit object intBoundedSemiLatticeInstance extends
    BoundedSemiLattice[Int] {
      def combine(a1: Int, a2: Int): Int =
        a1 max a2

      val empty: Int = 0
    }

    implicit def setBoundedSemiLatticeInstance[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]
      }
  }

  trait GCounter[F[_,_],K, V] {
    def increment(f: F[K, V])(k: K, v: V)(implicit m: Monoid[V])
    : F[K, V]
    def total(f: F[K, V])(implicit m: Monoid[V]): V
    def merge(f1: F[K, V], f2: F[K, V])(implicit b:
    BoundedSemiLattice[V]): F[K, V]
  }
  object GCounter {
    implicit def mapGCounterInstance[K, V]: GCounter[Map, K, V]
    =
```

```
      new GCounter[Map, K, V] {
        import cats.instances.map._

        def increment(f: Map[K, V])(k: K, v: V)(implicit m:
    Monoid[V]): Map[K, V] =
          f + (k -> (f.getOrElse(k, m.empty) |+| v))

        def total(f: Map[K, V])(implicit m: Monoid[V]): V =
          f.foldMap(identity)

        def merge(f1: Map[K, V], f2: Map[K, V])(implicit b:
    BoundedSemiLattice[V]): Map[K, V] =
          f1 |+| f2
      }

    def apply[F[_,_],K, V](implicit g: GCounter[F, K, V]) = g
  }

  import cats.instances.int._

  val g1 = Map("a" -> 7, "b" -> 3)
  val g2 = Map("a" -> 2, "b" -> 5)

  println(s"Merged: ${GCounter[Map, String, Int].merge(g1,g2)}")
  println(s"Total: ${GCounter[Map, String, Int].total(g1)}")
}
```

This implementation strategy is a bit unsatisfying. Although the structure of the implementation will be the same for most of the type class instances we won't get any code reuse.

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 `KeyValueStore`, including syntax and an example instance for `Map`.

```
trait KeyValueStore[F[_,_]] {
  def +[K, V](f: F[K, V])(key: K, value: V): F[K, V]
  def get[K, V](f: F[K, V])(key: K): Option[V]
```

```
  def getOrElse[K, V](f: F[K, V])(key: K, default: V): V =
    get(f)(key).getOrElse(default)
}

object KeyValueStore {
  implicit class KeyValueStoreOps[F[_,_],K, V](f: F[K, V]) {
    def +(key: K, value: V)(implicit kv: KeyValueStore[F]): F[K,
     V] =
      kv.+(f)(key, value)

    def get(key: K)(implicit kv: KeyValueStore[F]): Option[V] =
      kv.get(f)(key)

    def getOrElse(key: K, default: V)(implicit kv: KeyValueStore
    [F]): V =
      kv.getOrElse(f)(key, default)
  }

  implicit object mapKeyValueStoreInstance extends KeyValueStore
    [Map] {
    def +[K, V](f: Map[K, V])(key: K, value: V): Map[K, V] =
      f + (key, value)

    def get[K, V](f: Map[K, V])(key: K): Option[V] =
      f.get(key)

    override def getOrElse[K, V](f: Map[K, V])(key: K, default:
    V): V =
      f.getOrElse(key, default)
  }
}
```

Now we can generate `GCounter` instances with an `implicit def`. This implementation is moderately advanced: it has a number of type class dependencies, including one on `Foldable` that uses a type lambda.

```
import cats.Foldable

implicit def keyValueInstance[F[_,_],K, V](
  implicit
  k: KeyValueStore[F],
  km: Monoid[F[K, V]],
  kf: Foldable[({type l[A]=F[K, A]})#l]
): GCounter[F, K, V] =
  new GCounter[F, K, V] {
    import KeyValueStore._  // For KeyValueStore syntax

    def increment(f: F[K, V])(key: K, value: V)(implicit m:
    Monoid[V]): F[K, V] =
      f + (key, (f.getOrElse(key, m.empty) |+| value))

    def total(f: F[K, V])(implicit m: Monoid[V]): V =
      f.foldMap(identity _)

    def merge(f1: F[K, V], f2: F[K, V])(implicit b:
    BoundedSemiLattice[V]): F[K, V] =
      f1 |+| f2
  }
```

Here's the complete code, including an example. This code is quite long but the majority of it is boilerplate. We could cut down on the boilerplate by using compiler plugins such as Simulacrum and Kind Projector.

```
object GCounterExample {
  import cats.{Monoid, Foldable}
  import cats.syntax.foldable._
  import cats.syntax.semigroup._

  import scala.language.higherKinds

  trait BoundedSemiLattice[A] extends Monoid[A] {
    def combine(a1: A, a2: A): A
    def empty: A
  }
  object BoundedSemiLattice {
```

```
  implicit object intBoundedSemiLatticeInstance extends
  BoundedSemiLattice[Int] {
    def combine(a1: Int, a2: Int): Int =
      a1 max a2

    val empty: Int = 0
  }

  implicit def setBoundedSemiLatticeInstance[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]
    }
}

trait GCounter[F[_,_],K, V] {
  def increment(f: F[K, V])(key: K, value: V)(implicit m:
  Monoid[V]): F[K, V]
  def total(f: F[K, V])(implicit m: Monoid[V]): V
  def merge(f1: F[K, V], f2: F[K, V])(implicit b:
  BoundedSemiLattice[V]): F[K, V]
}
object GCounter {
  def apply[F[_,_],K, V](implicit g: GCounter[F, K, V]) = g

  implicit class GCounterOps[F[_,_],K, V](f: F[K, V]) {
    def increment(key: K, value: V)(implicit g: GCounter[F, K,
   V], m: Monoid[V]): F[K, V] =
      g.increment(f)(key, value)

    def total(implicit g: GCounter[F, K, V], m: Monoid[V]): V
  =
      g.total(f)

    def merge(that: F[K, V])(implicit g: GCounter[F, K, V], b:
```

```
     BoundedSemiLattice[V]): F[K, V] =
        g.merge(f, that)
    }

    implicit def keyValueInstance[F[_,_],K, V](implicit k:
    KeyValueStore[F], km: Monoid[F[K, V]], kf: Foldable[({type l
    [A]=F[K, A]})#l]): GCounter[F, K, V] =
      new GCounter[F, K, V] {
        import KeyValueStore._  // For KeyValueStore syntax

        def increment(f: F[K, V])(key: K, value: V)(implicit m:
    Monoid[V]): F[K, V] =
          f + (key, (f.getOrElse(key, m.empty) |+| value))

        def total(f: F[K, V])(implicit m: Monoid[V]): V =
          f.foldMap(identity _)

        def merge(f1: F[K, V], f2: F[K, V])(implicit b:
    BoundedSemiLattice[V]): F[K, V] =
          f1 |+| f2
      }
}

trait KeyValueStore[F[_,_]] {
  def +[K, V](f: F[K, V])(key: K, value: V): F[K, V]
  def get[K, V](f: F[K, V])(key: K): Option[V]

  def getOrElse[K, V](f: F[K, V])(key: K, default: V): V =
    get(f)(key).getOrElse(default)
}

object KeyValueStore {
  implicit class KeyValueStoreOps[F[_,_],K, V](f: F[K, V]) {
    def +(key: K, value: V)(implicit kv: KeyValueStore[F]): F[
    K, V] =
      kv.+(f)(key, value)

    def get(key: K)(implicit kv: KeyValueStore[F]): Option[V]
    =
```

```
        kv.get(f)(key)

      def getOrElse(key: K, default: V)(implicit kv:
    KeyValueStore[F]): V =
        kv.getOrElse(f)(key, default)
    }

    implicit object mapKeyValueStoreInstance extends
    KeyValueStore[Map] {
      def +[K, V](f: Map[K, V])(key: K, value: V): Map[K, V] =
        f + (key, value)

      def get[K, V](f: Map[K, V])(key: K): Option[V] =
        f.get(key)

      override def getOrElse[K, V](f: Map[K, V])(key: K, default
    : V): V =
        f.getOrElse(key, default)
    }
  }

  object Example {
    import cats.instances.map._
    import cats.instances.int._

    import KeyValueStore._
    import GCounter._

    val crdt1 = Map("a" -> 1, "b" -> 3, "c" -> 5)
    val crdt2 = Map("a" -> 2, "b" -> 4, "c" -> 6)

    crdt1.increment("a", 20).merge(crdt2).total
  }
}
```

11.5 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 are not 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, and 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 then 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", 35, "ginger and black")
// cat: Cat = Cat(Garfield,35,ginger and black)

Printable.print(cat)
// Garfield is a 35 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 PrintOps[A](value: A) {
    def format(implicit p: Printable[A]): String =
      p.format(value)

    def print(implicit p: Printable[A]): Unit =
      println(p.format(value))
  }
}
```

With `PrintOps` 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", 35, "ginger and black").print
// Garfield is a 35 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>:34: 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._
import cats.instances.string._
import cats.syntax.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", 35, "ginger and black").show)
// Garfield is a 35 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._
```

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]`:

```
implicit val catEqual = Eq.instance[Cat] { (cat1, cat2) =>
  import cats.instances.int._
  import cats.instances.string._

  (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",   35, "orange and black")
// cat1: Cat = Cat(Garfield,35,orange and black)

val cat2 = Cat("Heathcliff", 30, "orange and black")
// cat2: Cat = Cat(Heathcliff,30,orange and black)

cat1 === cat2
// res14: Boolean = false

cat1 =!= cat2
// res15: Boolean = true

import cats.instances.option._

val optionCat1 = Option(cat1)
// optionCat1: Option[Cat] = Some(Cat(Garfield,35,orange and
    black))

val optionCat2 = Option.empty[Cat]
// optionCat2: Option[Cat] = None

optionCat1 === optionCat2
// res16: Boolean = false

optionCat1 =!= optionCat2
// res17: 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`. Scala's implicit resolution is fine with this—it is capable of determining the correct type parameter to create a `Monoid` of the desired type:

```
implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
  def combine(a: Int, b: Int) = a + b
  def empty = 0
}

val intSetMonoid = Monoid[Set[Int]]
// intSetMonoid: Monoid[Set[Int]] = $anon$1@5ec1a336

intSetMonoid.combine(Set(1, 2), Set(2, 3))
// res2: Set[Int] = Set(1, 2, 3)
```

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
  }
```

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.syntax.semigroup._

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.syntax.semigroup._

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._

add(List(1, 2, 3))
// res9: Int = 6

import cats.instances.option._

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>:55: 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
import cats.syntax.functor._

implicit val treeFunctor = 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>:38: error: value map is not a member of Branch[Int]
//        Branch(Leaf(10), Leaf(20)).map(_ * 2)
//                                   ^
```

Oops! This is the same invariance problem we saw with `Monoids`. The compiler can't find a `Functor` instance for `Leaf`. Let's add some smart constructors to compensate:

```
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:

```
leaf(100).map(_ * 2)
// res6: Tree[Int] = Leaf(200)

branch(leaf(10), leaf(20)).map(_ * 2)
// res7: Tree[Int] = Branch(Leaf(20),Leaf(40))
```

Return to the exercise

C.2 Showing off with Contramap

Here's a working implementation:

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

  def contramap[B](func: B => A): Printable[B] = {
    val self = this
    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`:

```
implicit def boxPrintable[A](implicit p: Printable[A]) =
  p.contramap[Box[A]](_.value)
```

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): Option[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): Option[B] =
```

```
        self.decode(value).map(dec)
    }
  }
}

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

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

Return to the exercise

C.5 Transformative Thinking with Imap Part 2

```
implicit val intCodec =
  new Codec[Int] {
    def encode(value: Int): String =
      value.toString

    def decode(value: String): Option[Int] =
      scala.util.Try(value.toInt).toOption
  }
```

Return to the exercise

C.6 Transformative Thinking with Imap Part 3

```
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. Let's start by writing the method header:

```
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] =
    ???
}
```

Now we look at the types. We've been given 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 headers:

```
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 is a constructor—it creates an `Id[A]` from an initial value of type `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)
// res14: cats.Id[Int] = 123
```

The `map` method applies a function of type `A => B` to an `Id[A]`, creating 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)
// res15: 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)
// res16: cats.Id[Int] = 246
```

Notice that we haven't had to add any casts to any of the examples in this solution. Scala is able to interpret values of type `A` as `Id[A]` and vice versa, simply by the context in which they are used.

The only restriction to this is that Scala cannot unify different shapes of type constructor when searching for implicits. Hence our need to cast to `Id[A]` 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, 0)(_ + _)
// res22: Int = 705082704
```

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._

type Logged[A] = Writer[Vector[String], A]

42.pure[Logged]
```

```
// res15: Logged[Int] = WriterT((Vector(),42))
```

We'll import the `tell` syntax as well:

```
import cats.syntax.writer._

Vector("Message").tell
// res16: 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._

41.pure[Logged].map(_ + 1)
// res17: 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
```

Now, when we call `factorial`, we have to `run` the result to extract the log and our factorial:

```
val (log, result) = 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)
// result: 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(5).run),
    Future(factorial(5).run)
  )), 5.seconds)
// logA: Vector[String] = Vector(fact 0 1, fact 1 1, fact 2 2,
    fact 3 6, fact 4 24, fact 5 120)
// ansA: Int = 120
// 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 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 a :: b :: 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._
// import cats.syntax.applicative._

def evalAll(input: List[String]): CalcState[Int] = {
  input.foldLeft(0.pure[CalcState]) { (a, b) =>
    a flatMap (_ => evalOne(b))
  }
```

Return to the exercise

D.11 Branching out Further with Monads

The code for `flatMap` is simple. It's 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 less simple. In fact, it's fairly complex! However, if we follow the types the solution falls out. Note that we can't make `tailRecM` tail recursive in this case because we have to recurse twice when processing a `Branch`. We implement the `tailRecM` method, and we don't use the `tailrec` annotation:

```
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] =
    func(arg) match {
      case Branch(l, r) =>
        Branch(
          flatMap(l) {
            case Left(l)  => tailRecM(l)(func)
            case Right(l) => pure(l)
          },
          flatMap(r) {
            case Left(r)  => tailRecM(r)(func)
            case Right(r) => pure(r)
          }
        )

      case Leaf(Left(value)) =>
        tailRecM(value)(func)

      case Leaf(Right(value)) =>
        Leaf(value)
    }
}
```

Now we can use our `Monad` to `flatMap` and `map`:

```
import cats.syntax.functor._
import cats.syntax.flatMap._

branch(leaf(100), leaf(200)).
  flatMap(x => branch(leaf(x - 1), leaf(x + 1)))
// res4: 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
// res5: 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._
import cats.syntax.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 =
  Await.result(
    canSpecialMove(ally1, ally2).value,
    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: Cartesians and Applicatives

F.1 Cartesian Applied to Monads

We can implement `product` in terms of `map` and `flatMap` like so:

```
import cats.syntax.flatMap._
import cats.syntax.functor._

def product[M[_] : Monad, A, B](fa: M[A], fb: M[B]): M[(A, B)] =
  fa.flatMap(a => fb.map(b => (a, b)))
```

Unsurprisingly, this code is equivalent to a for comprehension:

```
def product[M[_] : Monad, A, B](
  fa: M[A],
  fb: M[B]
): M[(A, B)] =
  for {
    a <- fa
    b <- fb
```

```
  } yield (a, b)
```

The semantics of `flatMap` are what give rise to the behaviour for `List` and `Either`:

```
import cats.instances.list._

product(List(1, 2), List(3, 4))
// res13: 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"))
)
// res14: ErrorOr[(Int, Int)] = Left(Vector(Error 1))
```

Even our results for `Future` are a trick of the light. `flatMap` provides sequential ordering, so `product` provides the same. The only reason we get parallel execution is 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)
```

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 ErrorsOr[A] = Either[List[String], A]
type AllErrorsOr[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): ErrorsOr[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 => ErrorsOr[String] = <function1>

getName(Map("name" -> "Dade Murphy"))
// res25: ErrorsOr[String] = Right(Dade Murphy)
```

In the event of a missing field, our instance returns an error message containing an appropriate field name:

```
getName(Map())
// res26: ErrorsOr[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:

```
type NumFmtExn = NumberFormatException

def parseInt(name: String)(data: String): ErrorsOr[Int] =
  Right(data).
    flatMap(s => Either.catchOnly[NumFmtExn](s.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")
// res28: ErrorsOr[Int] = Right(11)
```

If we provide erroneous input, we get a useful error message:

```
parseInt("age")("foo")
// res29: ErrorsOr[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): ErrorsOr[String] =
  Right(data).
    ensure(List(s"$name cannot be blank"))(_.nonEmpty)

def nonNegative(name: String)(data: Int): ErrorsOr[Int] =
  Right(data).
    ensure(List(s"$name must be non-negative"))(_ >= 0)
```

Here are some examples of use:

```
nonBlank("name")("Dade Murphy")
// res31: ErrorsOr[String] = Right(Dade Murphy)

nonBlank("name")("")
// res32: ErrorsOr[String] = Left(List(name cannot be blank))

nonNegative("age")(11)
// res33: ErrorsOr[Int] = Right(11)

nonNegative("age")(-1)
// res34: ErrorsOr[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): ErrorsOr[String] =
  getValue("name")(data).
    flatMap(nonBlank("name"))

def readAge(data: FormData): ErrorsOr[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"))
// res36: ErrorsOr[String] = Right(Dade Murphy)

readName(Map("name" -> ""))
// res37: ErrorsOr[String] = Left(List(name cannot be blank))

readName(Map())
```

```
// res38: ErrorsOr[String] = Left(List(name field not specified)
    )

readAge(Map("age" -> "11"))
// res39: ErrorsOr[Int] = Right(11)

readAge(Map("age" -> "-1"))
// res40: ErrorsOr[Int] = Left(List(age must be non-negative))

readAge(Map())
// res41: ErrorsOr[Int] = Left(List(age field not specified))
```

Return to the exercise

F.6 Form Validation Part 5

There are a couple of ways to do this, each involving switching from `Either` to `Validated`. One option is to use `product` and `map`:

```
def readUser(data: FormData): AllErrorsOr[User] =
  Cartesian[AllErrorsOr].product(
    readName(data).toValidated,
    readAge(data).toValidated
  ).map(User.tupled)
```

A more idiomatic solution is to use cartesian builder syntax:

```
import cats.syntax.cartesian._

def readUser(data: FormData): AllErrorsOr[User] =
  (
    readName(data).toValidated |@|
    readAge(data).toValidated
  ).map(User.apply)
```

Both solutions yield the same results:

```
readUser(Map("name" -> "Dave", "age" -> "37"))
// res43: AllErrorsOr[User] = Valid(User(Dave,37))

readUser(Map("age" -> "-1"))
// res44: AllErrorsOr[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._

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 cartesian `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 cartesian `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 cartesian `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._

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: Pygmy Hadoop

I.1 Implementing foldMap

```
/** Single-threaded map reduce function.
  * Maps `func` over `values`
  * and reduces using a `Monoid[B]`.
  */
def foldMap[A, B](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 2.5.4:

```
import cats.Monoid
import cats.instances.int._
import cats.instances.string._
import cats.syntax.semigroup._

def foldMap[A, B : Monoid](values: Vector[A])(func: A => B): B =
  values.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](values: Vector[A])(func: A => B): B =
  values.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)(_ |+| _)
  }
}

Await.result(parallelFoldMap((1 to 1000000).toVector)(identity),
     1.second)
// res18: 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)(_ |+| _)
  }
}

Await.result(parallelFoldMap((1 to 1000000).toVector)(identity),
```

```
      1.second)
// res19: 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.monoid._   // 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._
import cats.syntax.monoid._

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._    // asLeft and asRight syntax
import cats.syntax.semigroup._ // |+| syntax

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 behavior we get. First we'll setup some checks:

```
import cats.instances.list._ // Semigroup for List

val a: CheckF[List[String], Int] =
  CheckF { v =>
    if(v > 2) v.asRight
    else List("Must be > 2").asLeft
  }
// a: CheckF[List[String],Int] = CheckF(<function1>)

val b: CheckF[List[String], Int] =
  CheckF { v =>
    if(v < -2) v.asRight
    else List("Must be < -2").asLeft
  }
// b: CheckF[List[String],Int] = CheckF(<function1>)

val check = a and b
// check: CheckF[List[String],Int] = CheckF(<function1>)
```

Now run the check with some data:

```
check(5)
// res5: Either[List[String],Int] = Left(List(Must be < -2))

check(0)
```

```
// res6: 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 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
  }
// a: wrapper.Check[List[String],Int] = Pure(<function1>)

val b: Check[List[String], Int] =
  Pure { v =>
    if(v < -2) v.asRight
    else List("Must be < -2").asLeft
  }
// b: wrapper.Check[List[String],Int] = Pure(<function1>)

val check = a and b
// check: wrapper.Check[List[String],Int] = And(Pure(<function1
    >),Pure(<function1>))
```

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._ // |+| syntax
import cats.syntax.cartesian._ // |@| syntax

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)).map((_, _) => 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._ // |+| syntax
import cats.syntax.cartesian._ // |@| syntax
import cats.data.Validated._   // 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)).map((_, _) => 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
import cats.syntax.either._

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 corrsponding 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.syntax.semigroup._ // |+| syntax
import cats.syntax.cartesian._ // |@| syntax
import cats.syntax.validated._ // .valid and .invalid syntax
import cats.data.Validated._   // Valid and Invalid patterns
```

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)).map((_, _) => 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](error: E, func: A => Boolean): Predicate[E, A]
```

```
    =
    Pure(a => if(func(a)) a.valid else error.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, OneAnd, Validated}
import cats.instances.list._
import cats.syntax.cartesian._
import cats.syntax.validated._
```

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)).map(_+"@"+_) }

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)).map(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._
import cats.instances.function._
import cats.instances.list._
import cats.syntax.cartesian._
import cats.syntax.validated._
```

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)).map(_+"@"+_) }

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)
).map(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: Commutative Replicated Data Types

K.1 GCounter Implementation

Hopefully the description above was clear enough that you can get to an implementation like the below.

```
final case class GCounter(counters: Map[String, Int]) {
  def increment(machine: String, amount: Int) =
    GCounter(counters + (machine -> (amount + counters.getOrElse
    (machine, 0))))

  def get: Int =
    counters.values.sum

  def merge(that: GCounter): GCounter =
    GCounter(that.counters ++ {
      for((k, v) <- counters) yield {
```

```
        k -> (v max that.counters.getOrElse(k,0))
      }
    })
}
```

Return to the exercise

K.2 BoundedSemiLattice Instances

It's natural to place the instance in the companion object of `BoundedSemiLattice` so they are in the implicit scope without importing them.

Implementing the instance for `Set` is good practice with using implicit methods.

```
trait BoundedSemiLattice[A] extends Monoid[A] {
  def combine(a1: A, a2: A): A
  def empty: A
}

object BoundedSemiLattice {
  implicit object intBoundedSemiLatticeInstance extends
    BoundedSemiLattice[Int] {
    def combine(a1: Int, a2: Int): Int =
      a1 max a2

    val empty: Int = 0
  }

  implicit def setBoundedSemiLatticeInstance[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

```
import cats.syntax.semigroup._
import cats.syntax.foldable._
import cats.instances.map._

final case class GCounter[A](counters: Map[String,A]) {
  def increment(machine: String, amount: A)(implicit m: Monoid[A
    ]) =
    GCounter(counters + (machine -> (amount |+| counters.
    getOrElse(machine, m.empty))))

  def get(implicit m: Monoid[A]): A =
    this.counters.foldMap(identity)

  def merge(that: GCounter[A])(implicit b: BoundedSemiLattice[A
    ]): GCounter[A] =
    GCounter(this.counters |+| that.counters)
}
```

Return to the exercise