Functional Programming Strategies

Functional Programming Strategies

Noel Welsh

Draft built on 2026-01-21

Artwork by Jenny Clements.

Published by Inner Product Consulting Ltd, UK.

This book is dedicated to those who laid the path that I have followed, to those who will take up where I have left off, and to those who have joined me along the way.

Contents

Preface

Some twenty years ago I started my first job in the UK. This job involved a commute by train, giving me about an hour a day to read without distraction. Around about the same time I first heard about *Structure and Interpretation of Computer Programs* [1], referred to as the "wizard book" and spoken of in reverential terms. It sounded like the just the thing for a recent graduate looking to become a better developer. I purchased a copy and spent the journey reading it, doing most of the exercises in my head. *Structure and Interpretation of Computer Programs* was already an old book at this time, and it's programming style was archaic. However it's core concepts were timeless and it's fair to say it absolutely blew my mind, putting me on a path I'm still on today.

Another notable stop on this path occured some ten years ago when Dave and I started writing *Scala with Cats*. In *Scala with Cats* we attempted to explain the core type classes found in the Cats library, and their use in building software. I'm proud of the book we wrote together, but time and experience showed that type classes are only a small piece of the puzzle of building software in a functional programming style. We needed a much wider scope if we were to show people how to effectively build software with all the tools that functional programming provides. Still, writing a book is a lot of work, and we were busy with other projects, so *Scala with Cats* remained largely untouched for many years.

Around 2020 I got the itch to return to *Scala with Cats*. My initial plan was simply to update the book for Scala 3. Dave was busy with other projects so I decided to go alone. As the writing got underway I realized I really wanted to cover the additional topics I thought were missing. If *Scala with Cats* was a good book, I wanted to aim to write a great book; one that would contain almost everything I had learned about building software. The title

Scala with Cats no longer fit the content, and hence I adopted a new name for what is largely a new book. The result, *Functional Programming Strategies in Scala with Cats*, is what you are reading now. I hope you find it useful, and I hope that just maybe some young developer will find this book inspiring the same way I found *Structure and Interpretation of Computer Programs* inspiring all those years ago.

Preface from Scala with Cats

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[1].

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

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

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

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

[1] https://typelevel.org/cats

Ok, let's get started!

Versions

This book is written for Scala 3.7.3 and Cats 2.13.0. Here is a minimal `build.sbt` containing the relevant dependencies and settings[2]:

```
scalaVersion := "3.7.3"

libraryDependencies +=
  "org.typelevel" %% "cats-core" % "2.13.0"

scalacOptions ++= Seq(
  "-Xfatal-warnings"
)
```

Template Projects

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

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

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

[2]We assume you are using SBT 1.0.0 or newer

Conventions Used in This Book

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

Typographical Conventions

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

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

References to external resources are written as hyperlinks[3], which also render as footnotes for situations when you cannot conveniently follow links. References to API documentation are written using a combination of hyperlinks and monospace font, for example: `scala.Option`[4].

Source Code

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

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

[3] https://scalawithcats.com
[4] http://www.scala-lang.org/api/current/scala/Option.html

Most code passes through mdoc[5] to ensure it compiles. mdoc uses the Scala console behind the scenes, so we sometimes show console-style output as comments:

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

Callout Boxes

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

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

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

License

[5] https://scalameta.org/mdoc/
[6] http://creativecommons.org/licenses/by-sa/4.0/
[7] https://creativecommons.org/licenses/by-sa/3.0/

1. Functional Programming Strategies

This is a book on strategies for creating code in a functional programming (FP) style, seen through a Scala lens. If you understand most of the mechanics of Scala, but feel there is something missing in your understanding of how to use the language effectively, this book is for you. If you want to learn about functional programming, and are prepared to learn some Scala, this book is also for you. It covers the usual functional programming abstractions like monads and monoids, but more than that it tries to teach you how to think like a functional programmer. It's a book as much about process as it is about the code that results from process, and in particular it focuses on what I call metacognitive programming strategies.

Functional programmers love fancy words for simple ideas, so it's no surprise I'm drawn to metacognitive programming strategies. Let's unpack that phrase. Metacognition means thinking about thinking. A lot of research has shown the benefits of metacognition in learning and its importance in developing expertise (see, for example, [15,53,70]). Metacognition is not just one thing—it's not sufficient to just tell someone to think about their thinking. Rather, metacognition is a collection of different strategies, some of which are general and some of which are domain specific. From this we get the idea of metacognitive programming strategies—explicitly naming and describing different thinking strategies that proficient programmers use.

Let's think a little about metacognitive strategies you might already use when coding. My experience is that most developers

struggle to answer this. Software teams usually have many well-defined processes *around* coding, such as daily stand-ups and kanban boards. Developers have a huge amount of language specific knowledge. However, the part inbetween deciding what should be done and working code is often very fuzzy. When developers can answer this question they often mention test driven development and pair programming, and design patterns such as the builder pattern. These date from the nineties, the former from *Extreme Programming* [4] and the latter from the *Design Patterns* book [31]. In my experience they are used quite informally, if at all.

The question then becomes: what metacognitive strategies can programmers use? I believe that functional programming is particularly well suited to answer this question. One major theme in functional programming research is finding and naming useful code structures. Once we have discovered a useful abstraction we can get the programmer to ask themselves "would this abstraction solve this problem?" This is essentially what the design patterns community did but there is an important difference. The academic FP community strongly values formal models, which means that the building blocks of FP have a precision that design patterns lack. However there is more to strategies than categorizing their output. There is also the actual process of how the code comes to be. Code doesn't usually spring fully formed from our keyboard, and in the iterative refinement of code we also find structure. Here the academic FP community has explored various algorithms for deriving code. As working programmers we must usually execute these algorithms by hand, but the benefit still remains.

I believe metacognitive programming strategies are useful for both beginners and experts. For beginners we can make programming a more systematic and repeatable process. Producing code no longer requires magic in the majority of cases, but rather the application of well defined steps. For experts, the benefit is exactly the same. At least that is my experience (and I believe I've been

programming long enough to call myself an expert.) By having an explicit process I can run it exactly the same way every day, which makes my code simpler to write and read, and saves my brain cycles for more important problems. In some ways this is an attempt to bring to programming the benefit that process and standardization has brought to manufacturing, particularly the "Toyota Way". In Toyota's process individuals are expected to think about how their work is done and how it can be improved. This is, in effect, metacognition for assembly lines. This is only possible if the actual work itself does not require their full attention. The dramatic improvements in productivity and quality in car manufacturing that Toyota pioneered speak to the effectiveness of this approach. Software development is more varied than car manufacturing but we should still expect some benefit, particularly given the primitive state of our current industry.

Over the last ten or so years of programming and teaching programming I've collected a wide range of strategies. Some come from others (for example, *How to Design Programs* [27] and its many offshoots remain very influential for me) and some I've found myself. Ultimately I don't think anything here is new; rather my contribution is in collecting and presenting these strategies as one coherent whole, in a way that I hope is accessible to the working programmer.

1.1. Three Levels for Thinking About Code

Let's start thinking about thinking about programming, with a model that describes three different levels that we can use to think about code. The levels, from highest to lowest, are paradigm, theory, and craft. Each level provides guidance for the ones below.

The paradigm level refers to the programming paradigm, such as object-oriented or functional programming. You're probably familiar with these terms, but what exactly is a programming paradigm? To me, the core of a programming paradigm is a set of principles that define, usually somewhat loosely, the properties of good code. A paradigm is also, implicitly, a claim that code that follows these principles will be better than code that does not. For functional programming I believe these principles are composition and reasoning. I'll explain these shortly. Object-oriented programmers might point to, say, the SOLID principles as guiding their coding decisions.

The importance of the paradigm is that it provides criteria for choosing between different implementation strategies. There are many possible solutions for any programming problem, and we can use the principles in the paradigm to decide which approach to take. For example, if we're a functional programmer we can consider how easily we can reason about a particular implementation, or how composable it is. Without the paradigm we have no basis for making a choice.

The theory level translates the broad principles of the paradigm to specific well defined techniques that apply to many languages within the paradigm. We are still, however, at a level above the code. Design patterns are an example in the object-oriented world. Algebraic data types are an example in functional programming. Most languages that are in the functional programming paradigm, such as Haskell and O'Caml, support algebraic data types, as do many languages that straddle multiple paradigms, such as Rust, Scala, and Swift.

The theory level is where we find most of our programming strategies.

At the craft level we get to actual code, and the language specific nuance that goes into it. An example in Scala is the implementation of algebraic data types in terms of `sealed trait`

and `final case class` in Scala 2, or `enum` in Scala 3. There are many concerns at this level that are important for writing idiomatic code, such as placing constructors on companion objects in Scala, that are not relevant at the higher levels.

In the next section I'll describe the functional programming paradigm. The remainder of this book is primarily concerned with theory and craft. The theory is language agnostic but the craft is firmly in the world of Scala. Before we move onto the functional programming paradigm are two points I want to emphasize:

1. Paradigms are social constructs. They change over time. Object-oriented programming as practiced today differs from the style originally used in Simula and Smalltalk, and functional programming today is very different from the original LISP code.
2. The three level organization is just a tool for thought. The real world it is more complicated.

1.2. Functional Programming

This is a book about the techniques and practices of functional programming (FP). This naturally leads to the question: what is FP and what does it mean to write code in a functional style? It's common to view functional programming as a collection of language features, such as first class functions, or to define it as a programming style using immutable data and pure functions. (Pure functions always return the same output given the same input.) This was my view when I started down the FP route, but I now believe the true goals of FP are enabling local reasoning and composition. Language features and programming style are in service of these goals. Let me attempt to explain the meaning and value of local reasoning and composition.

1.2.1. What Functional Programming Is

I believe that functional programming is a hypothesis about software quality: that it is easier to write and maintain software that can be understood before it is run, and is built of small reusable components. The first property is known as local reasoning, and the second as composition. Let's address each in turn.

Local reasoning means we can understand pieces of code in isolation. When we see the expression `1 + 1` we know what it means regardless of the weather, the database, or the current status of our Kubernetes cluster. None of these external events can change it. This is a trivial and slightly silly example, but it illustrates the point. A goal of functional programming is to extend this ability across our code base.

It can help to understand local reasoning by looking at what it is not. Shared mutable state is out because relying on shared state means that other code can change what our code does without our knowledge. It means no global mutable configuration, as found in many web frameworks and graphics libraries for example, as any random code can change that configuration. Metaprogramming has to be carefully controlled. No monkey patching[8], for example, as again it allows other code to change our code in non-obvious ways. As we can see, adapting code to enable local reasoning can mean quite some sweeping changes. However if we work in a language that embraces functional programming this style of programming is the default.

Composition means building big things out of smaller things. Numbers are compositional. We can take any number and add one, giving us a new number. Lego is also compositional. We compose Lego by sticking it together. In the particular sense we're using composition we also require the original elements we combine

[8] https://en.wikipedia.org/wiki/Monkey_patch

don't change in any way when they are composed. When we create by `2` by adding `1` and `1` we get a new result that doesn't change what `1` means.

We can find compositional ways to model common programming tasks once we start looking for them. React components are one example familiar to many front-end developers: a component can consist of many components. HTTP routes can be modelled in a compositional way. A route is a function from an HTTP request to either a handler function or a value indicating the route did not match. We can combine routes as a logical or: try this route or, if it doesn't match, try this other route. Processing pipelines are another example that often use sequential composition: perform this pipeline stage and then this other pipeline stage.

1.2.1.1. Types

Types are not strictly part of functional programming but statically typed FP is the most popular form of FP and sufficiently important to warrant a mention. Types help compilers generate efficient code but types in FP are as much for the programmer as they are the compiler. Types express properties of programs, and the type checker automatically ensures that these properties hold. They can tell us, for example, what a function accepts and what it returns, or that a value is optional. We can also use types to express our beliefs about a program and the type checker will tell us if those beliefs are correct. For example, we can use types to tell the compiler we do not expect an error at a particular point in our code and the type checker will let us know if this is the case. In this way types are another tool for reasoning about code.

Type systems push programs towards particular designs, as to work effectively with the type checker requires designing code in a way the type checker can understand. As modern type systems come to more languages they naturally tend to shift programmers in those languages towards a FP style of coding.

1.2.2. What Functional Programming Isn't

In my view functional programming is not about immutability, or keeping to "the substitution model of evaluation", and so on. These are tools in service of the goals of enabling local reasoning and composition, but they are not the goals themselves. Code that is immutable always allows local reasoning, for example, but it is not necessary to avoid mutation to still have local reasoning. Here is an example of summing a collection of numbers.

```
def sum(numbers: List[Int]): Int = {
  var total = 0
  numbers.foreach(x => total = total + x)
  total
}
```

In the implementation we mutate `total`. This is ok though! We cannot tell from the outside that this is done, and therefore all users of `sum` can still use local reasoning. Inside `sum` we have to be careful when we reason about `total` but this block of code is small enough that it shouldn't cause any problems.

In this case we can reason about our code despite the mutation, but the Scala compiler can determine that this is ok. Scala allows mutation but it's up to us to use it appropriately. A more expressive type system, perhaps with features like Rust's, would be able to tell that `sum` doesn't allow mutation to be observed by other parts of the system[9]. Another approach, which is the one

[9]The example I gave is fairly simple. A compiler that used escape analysis[10] could recognize that no reference to `total` is possible outside `sum` and hence `sum` is pure (or referentially transparent). Escape analysis is a well studied technique. In the general case the problem is a lot harder. We'd often like to know that a value is only referenced once at various points in our program, and hence we can mutate that value without changes being observable in other parts of the program. This might be used, for example, to pass an accumulator through various processing stages. To do this requires a programming language with what is called a substructural type system[11]. Rust

taken by Haskell, is to disallow all mutation and thus guarantee it cannot cause problems.

Mutation also interferes with composition. For example, if a value relies on internal state then composing it may produce unexpected results. Consider Scala's `Iterator`. It maintains internal state that is used to generate the next value. If we have two `Iterators` we might want to combine them into one `Iterator` that yields values from the two inputs. The `zip` method does this.

This works if we pass two distinct generators to `zip`.

```
val it = Iterator(1, 2, 3, 4)

val it2 = Iterator(1, 2, 3, 4)
```

```
it.zip(it2).next()
// res0: Tuple2[Int, Int] = (1, 1)
```

However if we pass the same generator twice we get a surprising result.

```
val it3 = Iterator(1, 2, 3, 4)
```

```
it3.zip(it3).next()
// res1: Tuple2[Int, Int] = (1, 2)
```

The usual functional programming solution is to avoid mutable state but we can envisage other possibilities. For example, an effect tracking system[12] would allow us to avoid combining two generators that use the same memory region. These systems are mostly still research projects, however.

---

has such a system, with affine types. Linear types are in development for Haskell.

[10] https://en.wikipedia.org/wiki/Escape_analysis

[11] https://en.wikipedia.org/wiki/Substructural_type_system

[12] https://en.wikipedia.org/wiki/Effect_system

So in my opinion immutability (and purity, referential transparency, and no doubt more fancy words that I have forgotten) have become associated with functional programming because they guarantee local reasoning and composition, and until recently we didn't have the language tools to automatically distinguish safe uses of mutation from those that cause problems. Restricting ourselves to immutability is the easiest way to ensure the desirable properties of functional programming, but as languages evolve this might come to be regarded as a historical artifact.

1.2.3. Why It Matters

I have described local reasoning and composition but have not discussed their benefits. Why are they are desirable? The answer is that they make efficient use of knowledge. Let me expand on this.

We care about local reasoning because it allows our ability to understand code to scale with the size of the code base. We can understand module A and module B in isolation, and our understanding does not change when we bring them together in the same program. By definition if both A and B allow local reasoning there is no way that B (or any other code) can change our understanding of A, and vice versa. If we don't have local reasoning every new line of code can force us to revisit the rest of the code base to understand what has changed. This means it becomes exponentially harder to understand code as it grows in size as the number of interactions (and hence possible behaviours) grows exponentially. We can say that local reasoning is compositional. Our understanding of module A calling module B is just our understanding of A, our understanding of B, and whatever calls A makes to B.

We introduced numbers and Lego as examples of composition. They have an interesting property in common: the operations that

we can use to combine them (for example, addition, subtraction, and so on for numbers; for Lego the operation is "sticking bricks together") give us back the same kind of thing. A number multiplied by a number is a number. Two bits of Lego stuck together is still Lego. This property is called closure: when you combine things you end up with the same kind of thing. Closure means you can apply the combining operations (sometimes called combinators) an arbitrary number of times. No matter how many times you add one to a number you still have a number and can still add or subtract or multiply or…you get the idea. If we understand module A, and the combinators that A provides are closed, we can build very complex structures using A without having to learn new concepts! This is also one reason functional programmers tend to like abstractions such a monads (beyond liking fancy words): they allow us to use one mental model in lots of different contexts.

In a sense local reasoning and composition are two sides of the same coin. Local reasoning is compositional; composition allows local reasoning. Both make code easier to understand.

1.2.4. The Evidence for Functional Programming

I've made arguments in favour of functional programming and I admit I am biased—I do believe it is a better way to develop code than imperative programming. However, is there any evidence to back up my claim? There has not been much research on the effectiveness of functional programming, but there has been a reasonable amount done on static typing. I feel static typing, particularly using modern type systems, serves as a good proxy for functional programming so let's look at the evidence there.

In the corners of the Internet I frequent the common refrain is that

static typing has neglible effect on productivity[13]. I decided to look into this and was surprised that the majority of the results I found support the claim that static typing increases productivity. For example, one literature review [85] finds a majority of results in favour of static typing, and in particular finds support amongst the more recent studies. However the majority of these studies are very small and use relatively inexperienced developers—which is noted in the review by Dan Luu. My belief is that functional programming comes into its own on larger systems. Furthermore, programming languages, like all tools, require proficiency to use effectively. I'm not convinced very junior developers have sufficient skill to demonstrate a significant difference between languages.

To me the most useful evidence of the effectiveness of functional programming is that industry is adopting functional programming en masse. Consider, say, the widespread and growing adoption of Typescript and React. If we are to argue that FP as embodied by Typescript or React has no value we are also arguing that the thousands of Javascript developers who have switched to using them are deluded. At some point this argument becomes untenable.

This doesn't mean we'll all be using Haskell in five years. More likely we'll see something like the shift to object-oriented programming of the nineties: Smalltalk was the paradigmatic example of OO, but it was more familiar languages like C++ and Java that brought OO to the mainstream. In the case of FP this probably means languages like Scala, Swift, Kotlin, or Rust, and mainstream languages like Javascript and Java continuing to adopt more FP features.

[13] https://danluu.com/empirical-pl/

1.2.5. Final Words

I've given my opinion on functional programming—that the real goals are local reasoning and composition, and programming practices like immutability are in service of these. Other people may disagree with this definition, and that's ok. Words are defined by the community that uses them, and meanings change over time.

Functional programming emphasises formal reasoning, and there are some implications that I want to briefly touch on.

Firstly, I find that FP is most valuable in the large. For a small system it is possible to keep all the details in our head. It's when a program becomes too large for anyone to understand all of it that local reasoning really shows its value. This is not to say that FP should not be used for small projects, but rather that if you are, say, switching from an imperative style of programming you shouldn't expect to see the benefit when working on toy projects.

The formal models that underlie functional programming allow systematic construction of code. This is in some ways the reverse of reasoning: instead of taking code and deriving properties, we start from some properties and derive code. This sounds very academic but is in fact very practical, and how I develop most of my code.

Finally, reasoning is not the only way to understand code. It's valuable to appreciate the limitations of reasoning, other methods for gaining understanding, and using a variety of strategies depending on the situation.

Part I: Foundations

In this first part of the book we're building the foundational strategies on which the rest of the book will build and elaborate. In Chapter 2 we discuss the role of types as representing constraints, and see how we can separate representation and operations. In Chapter 3 we look at algebraic data types. Algebraic data types are our main way of modelling data, where we are concerned with what things are. We turn to codata in Chapter 4, which is the opposite, or dual, or algebraic data. Codata gives us a way to model things by what they can do. Abstracting over context, and the particular case of type classes, are the focus of Chapter 5. Type classes allow us to extend existing types with new functionality, and to abstract over types that are not related by the inheritance hierarchy. The fundamentals of interpreters are discussed in Chapter 6, and are the final chapter of this part. Interpreters give a clear distinction between description and action, and are a fundamental tool for achieving composition when working with effects.

These five strategies all describe code artifacts. For example, we can label part of code as an algebraic data type or a type class. We'll also see strategies that help us write code but don't necessarily end up directly reflected in it, such as following the types.

2. Types as Constraints

Our very first strategy is using **types as constraints**. We'll start by discussing two different ways we can think of types: by a type is, sometimes known as an **extensional** view; and by what a type can do, sometimes known as an **intensional** view. The latter view is less familiar, but is necessary to get the most from an expressive type system and is the core of the strategy. Hence we'll spend some time elaborating on this idea and discussing examples.

Once we understand the concept of types as constraints, we'll look at a Scala 3 feature, known as **opaque types**. Opaque types allow us to create a distinct type that has the same runtime representation as another type. As such, they provide a way to decouple representation from operations, and allow us to work with a purely intensional view.

2.1. Sets and Constraints

What is a type? Here we'll address this question from the programmer's perspective, but I want to note that there is a subfield within mathematics and philosophy known as type theory. There are some references in the conclusions if you want to follow that direction.

The most common view is that types define a set of values. For example, an `Int` in Scala is 32-bits, and as such defines a set of 4,294,967,296 possible values. When we define a type by enumerating all the possible values of that type, we are working with an extensional definition. This is a natural approach to take, not least because we need to tell the programming language how to represent values in memory, and the extensional view provides this.

The extensional view, however, doesn't provide any **encapsulation** or **information hiding**. Knowing the representation can be a problem when that integer represents, say, an index into an array, or an age, or a timestamp. In these cases we have access to a whole range of operations that aren't meaningful on the data. For example, neither indices nor ages can be negated, but nonetheless we can negate any index or age that is represented as an `Int`. Similarly, we can perform bitwise operations on machine integers, but this is not semantically meaningful for, say, a timestamp. Furthermore, as we'll see in Chapter 14, it can be useful to have types that have no representation, which the extensional view doesn't have much to say about.

This brings us to an alternate view of types, the intensional view. Instead of thinking of a type in terms of its representation, we can think of a type in terms of the conditions, invariants, or constraints that hold for elements of that type. This may in turn imply a set of operations that are valid on our types. So, for example, we can think of age (in years) as a non-negative integer with an increment operation, but no decrement operation (we, unfortunately, cannot get younger.) Similarly, indices are non-negative integers within the range of the array they refer to, names are non-empty strings, and email addresses are case insensitive strings with a username and domain separated by an `@`.

We might argue that our `Int` example above *is* defined by a constraint: namely it's an integer that fits into 32-bits. This is true! This constraint also implies which operations are available on `Int`. We cannot, for example, try to convert an `Int` to upper case; this is meaningless. Remember that we're taking two different views on the same concept. It's expected that we can translate between these views in many cases. The problem is the purely extensional view couples operations and representation. We cannot represent, say, a timestamp as an `Int` and not make meaningless bitwise operations available if we only have the extensional view.

Decoupling operations and representation sounds a lot like programming to an interface. Indeed this is true, and we'll look at this in much more detail in Chapter 4. In this chapter we'll look at opaque types, which directly decouple type and representation, allowing us to reuse a representation as a different type. However, before doing so I want to spend more time on the mindset shift that the intensional view promotes.

2.2. Building Constraints

Most applications work by progressively adding structure to inputs. We might receive data from, say, the network or a database. We perform some checks on that data and remove instances that are invalid. We then do some more work, which entails further checks, and so on.

For example, imagine we're implementing a sign up flow. We start by asking for a user name and email address. Basic checks could be requiring names that are not empty, and email addresses that contain an `@`. We won't even let the user submit the form if these checks fail. Once the form is submitted we'll move on to further checks. For example, we might validate email addresses by sending them a verification email.

How should we represent these multiple levels of validation? For example, how do we distinguish a string representing a name from one that is an email address? How about an unverified email from a verified one? The most common approach that I've seen, across many code bases, is to use ad hoc naming conventions. For example, we might use the name `email` and `verifiedEmail` to distinguish the different kinds of email addresses in method parameters and data structure, while still representing both as

strings.[14]

Types provide a compelling alternative to naming schemes. They enforce consistency of nomenclature, while also representing this information in a form the compiler can check. For example, if we have `EmailAddress` and `VerifiedEmailAddress` types, not only do we have standard names, but the compiler will tell us if we try to use an `EmailAddress` where a `VerifiedEmailAddress` is required, or a `String` where an `EmailAddress` is required. Furthermore, when we see an `EmailAddress` we know it's already been through some validation, so we don't needlessly repeat validation, or worse, forget to do it. This brings us to two principles:

1. Types should represent what we know about values, or in other words the invariants or constraints on values. A `String` could be any sequence of characters. A `VerifiedEmailAddress` is also a sequence of characters, but it's one that represents an email address that we have verified is active.

2. Whenever we establish an additional invariant or constraint we should change the type to reflect this additional information. So for example, an email address might start out as a `String`, become an `EmailAddress` if we have verified it looks like an email, and then become a `VerifiedEmailAddress` when we've successfully sent it a verification email and received a response.

A corollary of this approach is that we push constraints upstream. Let me explain. In a code base where validation is done on an ad-hoc basis, we often end up with methods that can fail. For example, a method to get the domain from an email, where the email is represented as a `String`, might have the signature

[14] Hungarian notation[15] is a more formal approach to this idea of encoding type information in names. Hungarian notation was popular within Microsoft and its ecosystem, but to the best of my knowledge it is no longer in common use.

[15] https://en.wikipedia.org/wiki/Hungarian_notation

```
def domain(email: String): Option[String]
```

indicating that the `String` might not be a valid email. In this case we push the error handling, reflecting the constraint that we only work with valid email addresses, onto the downstream code that deals with the result of calling this method.

When we work with types as constraints the signature becomes

```
def domain(email: EmailAddress): String
```

There is now no possibility of error, as an email address must contain a domain. However, we have pushed the constraint, obtaining an `EmailAddress`, onto the upstream code that calls this method. At some point we must have conversions that could fail, which requires error handling, but this approach pushes error handling to the edges of the program. This tends to result in a better user experience, as the user is immediately notified of problems, and also makes the code simpler to work with as overall less error handling is required.

Although this strategy is easiest to explain in the context of validation, it's not restricted to this use case. As an example, imagine writing an API for updates to a database table. Some columns allow nulls and some do not. When updating a nullable column our API could accept an `Option`, with the `None` case meaning the column is set to null. When updating a non-nullable column we could also accept an `Option`, with the `None` case meaning we retain the column's existing value. These two different meaning of the same type are a sure way to introduce errors, with users nulling out columns they intended to leave unchanged. The solution is the same: use different types for the different kinds of columns. Here the constraints are not on the values represented by the type, but on the behaviour associated with them.

2.3. Opaque Types

Let's now look at opaque types. Opaque types are a Scala 3 feature that decouple the representation of a type from the set of allowed operations on that type. In simpler words, they allow us to create a type (e.g. an `EmailAddress`) that has the same runtime representation as another type (e.g. a `String`), but is distinct from that type in all other ways.

Here's a definition of `EmailAddress` as an opaque type.

```
opaque type EmailAddress = String
```

This is enough to define the type `EmailAddress` as represented by a `String`. However, it's a useless definition as it lacks any way to construct an `EmailAddress`. To properly understand how we can define a constructor, we need to understand that opaque types divide our code base into two distinct parts: where our type is transparent, which is where we know the underlying representation, and the remainder where it is opaque. The rule is pretty simple: an opaque type is transparent within the scope in which it is defined, so within an enclosing object or class. If there is no enclosing scope, as in the example above, it is transparent only within the file in which it is defined. Everywhere else it is opaque.

Knowing this we can define a constructor. Following Scala convention we will define it as the `apply` method on the `EmailAddress` companion object.

```
opaque type EmailAddress = String
object EmailAddress {
  def apply(address: String): EmailAddress = {
    assert(
      {
        val idx = address.indexOf('@')
        idx != -1 && address.lastIndexOf('@') == idx
```

```
      },
      "Email address must contain exactly one @ symbol."
    )
    address.toLowerCase
  }
}
```

The constructor does a basic check on the input (ensuring it contains only one `@` character) and converts the input to lower case, as email addresses are case insensitive. I used an `assert` to do the check, but in a real application we'd probably want a result type that indicates something can go wrong. More on this below. Finally, notice that the constructor returns just the `address`, showing that the representation doesn't change. Here's an example, showing the result type `EmailAddress`

```
val email = EmailAddress("someone@example.com")
// email: EmailAddress = "someone@example.com"
```

This shows that an `EmailAddress` is represented as a `String`, but as far as the type system is concerned it is not a `String`. We cannot, for example, call methods defined on `String` on an instance of `EmailAddress`.[16]

```
email.toUpperCase
// Compiler says NO!
```

We can view this as an efficiency gain. Our `EmailAddress` uses exactly the same amount of memory as the underlying `String` that represents it, yet it is a different type. Alternatively, we can view it as a semantic gain. An `EmailAddress` *is* a sequence of characters, the same as a `String`, but it has additional constraints. In this case

[16]Scala usually runs on the JVM, and the JVM was not designed to support opaque types. This means there are, unfortunately, a few ways to poke holes in the abstraction boundary created by an opaque type. If we use `isInstanceOf` we can test for the underlying representation. Using the methods defined on `Object` (`Any` in Scala), namely `equals`, `hashCode`, and `toString`, also allow us to peek inside.

we verify it contains exactly one `@` character, and ensure it is case insensitive.

We've seen how to define opaque types and their constructors. What about other methods? For example, for an `EmailAddress` we might want methods to get the username and domain. We can use extension methods to do this. As with the constructor, we just need to define these extension methods in a place where the type is transparent.

```
opaque type EmailAddress = String
extension (address: EmailAddress) {
  def username: String =
    address.substring(0, address.indexOf('@'))

  def domain: String =
    address.substring(address.indexOf('@') + 1, address.size)
}
object EmailAddress {
  def apply(address: String): EmailAddress = {
    assert(
      {
        val idx = address.indexOf('@')
        idx != -1 && address.lastIndexOf('@') == idx
      },
      "Email address must contain exactly one @ symbol."
    )
    address.toLowerCase
  }
}
```

With this definition we can use the extension methods as we'd expect.

```
email.username
// res3: String = "someone"
email.domain
// res4: String = "example.com"
```

There are two other features of opaque types that we should mention:

1. they can have type parameters; and

2. they can have type bounds.

Let's see an example of these two features used together. Earlier we saw an example of using an `Option` to represent two different types of database columns: nullable columns, where `None` mean to set the column to null, and non-nullable, where `None` means to keep the existing value. We can define these as opaque types with a type parameter.

```
// null is a reserved word in Scala, so we use the name nil
// instead.
opaque type Nilable[+A] = Option[A]
object Nilable {
  def apply[A](value: A): Nilable[A] = Some(value)

  def fromOption[A](option: Option[A]): Nilable[A] = option

  val nil: Nilable[Nothing] = None
}

opaque type Default[+A] = Option[A]
object Default {
  def apply[A](value: A): Default[A] = Some(value)

  def fromOption[A](option: Option[A]): Default[A] = option

  val default: Default[Nothing] = None
}
```

This works just as we'd expect, but we users will probably want to use the `Option` API on `Nilable` and `Default`. We can avoid tediously reimplementing it as extension methods by declaring that `Nilable` and `Default` are subtypes of `Option`.

```
opaque type Nilable[+A] <: Option[A] = Option[A]
object Nilable {
  def apply[A](value: A): Nilable[A] = Some(value)

  def fromOption[A](option: Option[A]): Nilable[A] = option

  val nil: Nilable[Nothing] = None
}
```

```
opaque type Default[+A] <: Option[A] = Option[A]
object Default {
  def apply[A](value: A): Default[A] = Some(value)

  def fromOption[A](option: Option[A]): Default[A] = option

  val default: Default[Nothing] = None
}
```

The type bound `Default[+A] <: Option[A]` says that `Default` is a subtype of `Option`, and crucially this information is publically available. Therefore all of the methods on `Option` are available on `Default`.

We can verify this with a few examples.

```
Nilable(1).orElse(Nilable.nil)
// res7: Option[Int] = Some(value = 1)

Default(1).map(_ + 1)
// res8: Option[Int] = Some(value = 2)
```

Notice that the results have type `Option`, because the methods on `Option` that we call have return type `Option`. We can easily convert back to `Nilable` or `Default` as required by using the `fromOption` constructor.

2.3.1. Best Practices

We've seen all the important technical details for opaque types, so let's now discuss some of the best practices of using them.

The first point I want to address is illustrated by the constructor for `EmailAddress`. There is a constraint on the `String` input to the constructor: it must contain an `@` character. This is a constraint and we should represent this as a type! We could create another opaque type, called something like `StringWithAnAtCharacter`, but this approach leads to infinite regress. We cannot push constraints

upstream indefinitely. At some point we have to return a result that indicates the possibility of error. So our constructor would be better if it returned, say, an `Option` or `Either` to indicate that construction can fail.

There are cases where we know the constructor cannot fail, but we don't have a convenient way of proving this to the compiler. For example, if we're loading email addresses from a list that is known to be good, it would be nice to avoid having to writing useless error handling code. For this reason I recommend including a constructor that doesn't do any validation. I usually call this method `unsafeApply`, to indicate to the reader that certain checks are not being done. These changes are shown below. For simplicity I've used `Option` as the result type to indicate the possibility of failure.

```
type EmailAddress = String
object EmailAddress {
  def apply(address: String): Option[EmailAddress] = {
    val idx = address.indexOf('@')
    if idx != -1 && address.lastIndexOf('@') == idx
    then Some(address.toLowerCase)
    else None
  }

  def unsafeApply(address: String): EmailAddress = address
}
```

We'll almost certainly need to convert from our opaque type back to its underlying type at some point in our code. I've seen a few conventions for naming such a method; `value` and `get` are popular. However, I prefer a more descriptive `toType`, replacing `Type` with the concrete type name, as this extends to conversions to other types. For `EmailAddress` this means an extension method `toString`, as shown below. Notice that the method simply returns the `address` value, once showing the distinction between the type and it's representation as a value.

```
extension (address: EmailAddress) {
  def toString: String = address
}
```

2.3.2. Beyond Opaque Types

Opaque types are a lightweight way to add structure—to use types to represent constraints—to our code. However there are two cases where they aren't appropriate.

The first case is when the data requires more structure that we can represent with an opaque type. For example, a (two-dimensional) point requires two coordinates, so there is no single type that we can use[17]. In these cases we're probably looking for an algebraic data type, which is discussed in Chapter 3.

The second case is when we need to reimplement one of the methods, most commonly `toString`, that opaque types cannot override. For example, we might want to ensure that types representing personal information, such as addresses and passwords, cannot be accidentally exposed in logs. Overriding `toString` helps ensure this, but we cannot do this for opaque types.

2.4. Conclusions

In this chapter we've looked at using types to represent constraints, which allows the compiler to help us ensure these constraints are met throughout our program. We call this strategy "types as constraints". We constrasted this strategy to the better known view of types that focuses on representation. Finally, we

[17] We could use an `Array[Double]` or `Tuple2[Double, Double]`, but it's simpler to just define a class in the usual way.

saw opaque types as a lightweight tool that decouples types from their representation, allowing us to define a type that uses the same representation as some other type.

The view of types as constraints is perhaps best presented in Alexis King's blog post Parse, don't validate[18]. *Types Are Not Sets* [58] is a very early paper (typewritten in two column justified text, a truly virtuoso performance on the type writer!) that also presents the intensional view of types. I feel it ends a bit abruptly, but has the seed of many ideas that will only be fully developed much later. You can see the suggestion of opaque types as discussed in this chapter, and also module systems and existential types.

From a programming language perspective, *Types and Programming Languages* [69] is the standard reference on type systems. They define a type system as "a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases by the kinds of values they compute". The introduction provides a very nice overview of the role of type systems in programming languages, as well as pointers to the broader study of type systems in mathematics and philosophy.

Having said that types are not sets, it feels only fair to mention there are type systems that do treat types as sets. *The Design Principles of the Elixir Type System* [12] describes one such system. These type systems emphasize the extensional view, and have a very different feel to conventional type systems.

I'm very far from an expert in mathematical type theory. As such, I found *A Comparison of Type Theory with Set Theory* [46] useful to relate type theory to something I better understand, set theory.

[18] https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/

3. Algebraic Data Types

This chapter has our first example of a programming strategy: **algebraic data types**. Any data we can describe using logical ands and logical ors is an algebraic data type. Once we recognize an algebraic data type we get three things for free:

- the Scala representation of the data;
- a **structural recursion** skeleton to transform the algebraic data type into any other type; and
- a **structural corecursion** skeleton to construct the algebraic data type from any other type.

The key point is this: from an implementation independent representation of data we can automatically derive most of the interesting implementation specific parts of working with that data.

We'll start with some examples of data, from which we'll extract the common structure that motivates algebraic data types. We will then look at their representation in Scala 2 and Scala 3. Next we'll turn to structural recursion for transforming algebraic data types, followed by structural corecursion for constructing them. We'll finish by looking at the algebra of algebraic data types, which is interesting but not essential.

3.1. Building Algebraic Data Types

Let's start with some examples of data from a few different domains. These are simplified description but they are all representative of real applications.

A user in a discussion forum will typically have a screen name, an email address, and a password. Users also typically have a specific

role: normal user, moderator, or administrator, for example. From this we get the following data:

- a user is a screen name, an email address, a password, and a role; and
- a role is normal, moderator, or administrator.

A product in an e-commerce store might have a stock keeping unit (a unique identifier for each variant of a product), a name, a description, a price, and a discount.

In two-dimensional vector graphics it's typical to represent shapes as a path, which is a sequence of actions of a virtual pen. The possible actions are usually straight lines, Bezier curves, or movement that doesn't result in visible output. A straight line has an end point (the starting point is implicit), a Bezier curve has two control points and an end point, and a move has an end point.

What is common between all the examples above is that the individual elements—the atoms, if you like—are connected by either a logical and or a logical or. For example, a user is a screen name *and* an email address *and* a password *and* a role. A 2D action is a straight line *or* a Bezier curve *or* a move. This is the core of algebraic data types: an algebraic data type is data that is combined using logical ands or logical ors. Conversely, whenever we can describe data in terms of logical ands and logical ors we have an algebraic data type.

3.1.1. Sums and Products

Being functional programmers we can't let a simple concept go without attaching some fancy jargon:

- a **product type** means a logical and; and
- a **sum type** means a logical or.

So algebraic data types consist of sum and product types.

3.1.2. Closed Worlds

Algebraic data types are closed worlds, which means they cannot be extended after they have been defined. In practical terms this means we have to modify the source code where we define the algebraic data type if we want to add or remove elements.

The closed world property is important because it gives us guarantees we would not otherwise have. In particular, it allows the compiler to check that we handle all possible cases when we use an algebraic data type. This is known as **exhaustivity checking**. This is an example of how functional programming prioritizes reasoning about code—in this case automated reasoning by the compiler—over other properties such as extensibility. We'll learn more about exhaustivity checking soon.

3.2. Algebraic Data Types in Scala

Now we know what algebraic data types are, we will turn to their representation in Scala. The important point here is that the translation to Scala is entirely determined by the structure of the data; no thinking is required! This means the work is in finding the structure of the data that best represents the problem at hand. Work out the structure of the data and the code directly follows from it.

As algebraic data types are defined in terms of logical ands and logical ors, to represent algebraic data types in Scala we must know how to represent these two concepts. Scala 3 simplifies the representation of algebraic data types compared to Scala 2, so we'll look at each language version separately.

I'm assuming that you're familiar with the language features we use to represent algebraic data types in Scala, so I won't be going over them.

3.2.1. Algebraic Data Types in Scala 3

In Scala 3 a logical and (a product type) is represented by a `final case class`. If we define a product type `A` is `B` *and* `C`, the representation in Scala 3 is

```
final case class A(b: B, c: C)
```

Not everyone makes their case classes `final`, but they should. A non-`final` case class can still be extended by a class, which breaks the closed world criteria for algebraic data types.

A logical or (a sum type) is represented by an `enum`. For the sum type `A` is `B` *or* `C`, the Scala 3 representation is

```
enum A {
  case B
  case C
}
```

There are a few wrinkles to be aware of.

If we have a sum of products, such as:

- `A` is `B` or `C`; and
- `B` is `D` and `E`; and
- `C` is `F` and `G`

the representation is

```
enum A {
  case B(d: D, e: E)
  case C(f: F, g: G)
}
```

In other words we don't write `final case class` inside an `enum`. You also can't nest an `enum` inside an `enum`. Nested logical ors can be rewritten into a single logical or containing only logical ands (known as disjunctive normal form) so this is not a limitation in practice. However the Scala 2 representation is still available in Scala 3 should you want more expressivity.

3.2.2. Algebraic Data Types in Scala 2

A logical and (product type) has the same representation in Scala 2 as in Scala 3. If we define a product type `A` is `B` *and* `C`, the representation in Scala 2 is

```
final case class A(b: B, c: C)
```

A logical or (a sum type) is represented by a `sealed abstract class`. For the sum type `A` is a `B` *or* `C` the Scala 2 representation is

```
sealed abstract class A
final case class B() extends A
final case class C() extends A
```

Scala 2 has several little tricks to defining algebraic data types.

Firstly, instead of using a `sealed abstract class` you can use a `sealed trait`. There isn't much practical difference between the two. When teaching beginners I'll often use `sealed trait` to avoid having to introduce `abstract class`. I believe `sealed abstract class` has slightly better performance and Java interoperability, but I haven't tested this. I also think `sealed abstract class` is closer, semantically, to the meaning of a sum type.

For extra style points we can `extend Product with Serializable` from `sealed abstract class`. Compare the reported types below with and without this little addition.

Let's first see the code without extending `Product` and `Serializable`.

```
sealed abstract class A
final case class B() extends A
final case class C() extends A
```

```
val list = List(B(), C())
// list: List[A extends Product with Serializable] = List(B(),
C())
```

Notice how the type of `list` includes `Product` and `Serializable`.

Now we have extending `Product` and `Serializable`.

```
sealed abstract class A extends Product with Serializable
final case class B() extends A
final case class C() extends A
```

```
val list = List(B(), C())
// list: List[A] = List(B(), C())
```

Much easier to read!

You'll only see this in Scala 2. Scala 3 has the concept of **transparent traits**, which aren't reported in inferred types, so you'll see the same output in Scala 3 no matter whether you add `Product` and `Serializable` or not.

Finally, we can use a `case object` instead of a `case class` when we're defining some type that holds no data. For example, reading from a text stream, such as a terminal, can return a character or the end-of-file. We can model this as

```
sealed abstract class Result
final case class Character(value: Char) extends Result
case object Eof extends Result
```

As the end-of-file indicator `Eof` has no associated data we use a `case object`. There is no need to mark the `case object` as `final`, as objects cannot be extended.

3.2.3. Examples

Let's make the discussion above more concrete with some examples.

3.2.3.1. Role and User

In the discussion forum example, we said a role is normal, moderator, or administrator. This is a logical or, so we can directly translate it to Scala using the appropriate pattern. In Scala 3 we write

```
enum Role {
  case Normal
  case Moderator
  case Administrator
}
```

In Scala 2 we write

```
sealed abstract class Role extends Product with Serializable
case object Normal extends Role
case object Moderator extends Role
case object Administrator extends Role
```

The cases within a role don't hold any data, so we used a `case object` in the Scala 2 code.

We defined a user as a screen name, an email address, a password, and a role. In both Scala 3 and Scala 2 this becomes

```
final case class User(
  screenName: String,
```

```
  emailAddress: String,
  password: String,
  role: Role
)
```

I've used `String` to represent most of the data within a `User`, but in real code we might want to define distinct types for each field.

3.2.3.2. Paths

We defined a path as a sequence of actions of a virtual pen. The possible actions are straight lines, Bezier curves, or movement that doesn't result in visible output. A straight line has an end point (the starting point is implicit), a Bezier curve has two control points and an end point, and a move has an end point.

This has a straightforward translation to Scala. We can represent paths as the following in both Scala 3 and Scala 2.

```
final case class Path(actions: Seq[Action])
```

An action is a logical or, so we have different representations in Scala 3 and Scala 2. In Scala 3 we'd write

```
enum Action {
  case Line(end: Point)
  case Curve(cp1: Point, cp2: Point, end: Point)
  case Move(end: Point)
}
```

where `Point` is a suitable representation of a two-dimensional point.

In Scala 2 we have to go with the more verbose

```
sealed abstract class Action extends Product with Serializable
final case class Line(end: Point) extends Action
final case class Curve(cp1: Point, cp2: Point, end: Point)
```

```
  extends Action
final case class Move(end: Point) extends Action
```

3.2.4. Representing ADTs in Scala 3

We've seen that the Scala 3 representation of algebraic data types, using `enum`, is more compact than the Scala 2 representation. However the Scala 2 representation is still available. Should you ever use the Scala 2 representation in Scala 3? There are a few cases where you may want to:

- Scala 3's doesn't currently support nested `enums` (`enums` within `enums`). This may change in the future, but right now it can be more convenient to use the Scala 2 representation to express this without having to convert to disjunctive normal form.
- Scala 2's representation can express things that are almost, but not quite, algebraic data types. For example, if you define a method on an `enum` you must be able to define it for all the members of the `enum`. Sometimes you want a case of an `enum` to have methods that are only defined for that case. To implement this you'll need to use the Scala 2 representation instead.

Exercise: Tree

To gain a bit of practice defining algebraic data types, code the following description in Scala (your choice of version, or do both.)

A `Tree` with elements of type `A` is:

- a `Leaf` with a value of type `A`; or
- a `Node` with a left and right child, which are both `Trees` with elements of type `A`.

3.3. Structural Recursion

Structural recursion is our second programming strategy. Algebraic data types tell us how to create data given a certain structure. Structural recursion tells us how to transform an algebraic data types into any other type. Given an algebraic data type, the transformation can be implemented using structural recursion.

As with algebraic data types, there is distinction between the concept of structural recursion and the implementation in Scala. This is more obvious because there are two ways to implement structural recursion in Scala: via pattern matching or via dynamic dispatch. We'll look at each in turn.

3.3.1. Pattern Matching

I'm assuming you're familiar with pattern matching in Scala, so I'll only talk about how to implement structural recursion using pattern matching. Remember there are two kinds of algebraic data types: sum types (logical ors) and product types (logical ands). We have corresponding rules for structural recursion implemented using pattern matching:

1. For each branch in a sum type we have a distinct `case` in the pattern match; and
2. Each `case` corresponds to a product type with the pattern written in the usual way.

Let's see this in code, using an example ADT that includes both sum and product types:

- `A` is `B` or `C`; and
- `B` is `D` and `E`; and
- `C` is `F` and `G`

which we represent (in Scala 3) as

```
enum A {
  case B(d: D, e: E)
  case C(f: F, g: G)
}
```

Following the rules above means a structural recursion would look like

```
anA match {
  case B(d, e) => ???
  case C(f, g) => ???
}
```

The `???` bits are problem specific, and we cannot give a general solution for them. However we'll soon see strategies to help create them.

3.3.2. The Recursion in Structural Recursion

At this point you might be wondering where the recursion in structural recursion comes from. This is an additional rule for recursion: whenever the data is recursive the method is recursive in the same place.

Let's see this in action for a real data type.

We can define a list with elements of type `A` as:

- the empty list; or
- a pair containing an `A` and a tail, which is a list of `A`.

This is exactly the definition of `List` in the standard library. Notice it's an algebraic data type as it consists of sums and products. It is also recursive: in the pair case the tail is itself a list.

We can directly translate this to code, using the strategy for algebraic data types we saw previously. In Scala 3 we write

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])
}
```

Let's implement `map` for `MyList`. We start with the method skeleton specifying just the name and types.

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])

  def map[B](f: A => B): MyList[B] =
    ???
}
```

Our first step is to recognize that `map` can be written using a structural recursion. `MyList` is an algebraic data type, `map` is transforming this algebraic data type, and therefore structural recursion is applicable. We now apply the structural recursion strategy, giving us

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])

  def map[B](f: A => B): MyList[B] =
    this match {
      case Empty() => ???
      case Pair(head, tail) => ???
    }
}
```

I forgot the recursion rule! The data is recursive in the `tail` of `Pair`, so `map` is recursive there as well.

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])

  def map[B](f: A => B): MyList[B] =
```

```
    this match {
      case Empty() => ???
      case Pair(head, tail) => ??? tail.map(f)
    }
}
```

I left the `???` to indicate that we haven't finished with that case.

Now we can move on to the problem specific parts. Here we have three strategies to help us:

1. reasoning independently by case;
2. assuming the recursion is correct; and
3. following the types

The first two are specific to structural recursion, while the final one is a general strategy we can use in many situations. Let's briefly discuss each and then see how they apply to our example.

The first strategy is relatively simple: when we consider the problem specific code on the right hand side of a pattern matching `case`, we can ignore the code in any other pattern match cases. So, for example, when considering the case for `Empty` above we don't need to worry about the case for `Pair`, and vice versa.

The next strategy is a little bit more complicated, and has to do with recursion. Remember that the structural recursion strategy tells us where to place any recursive calls. This means we don't have to think through the recursion. Instead we assume the recursive call will correctly compute what it claims, and only consider how to further process the result of the recursion. The result is guaranteed to be correct so long as we get the non-recursive parts correct.

In the example above we have the recursion `tail.map(f)`. We can assume this correctly computes `map` on the tail of the list, and we only need to think about what we should do with the remaining data: the `head` and the result of the recursive call.

It's this property that allows us to consider cases independently. Recursive calls are the only thing that connect the different cases, and they are given to us by the structural recursion strategy.

Our final strategy is **following the types**. It can be used in many situations, not just structural recursion, so I consider it a separate strategy. The core idea is to use the information in the types to restrict the possible implementations. We can look at the types of inputs and outputs to help us.

Now let's use these strategies to finish the implementation of `map`. We start with

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])

  def map[B](f: A => B): MyList[B] =
    this match {
      case Empty() => ???
      case Pair(head, tail) => ??? tail.map(f)
    }
}
```

Our first strategy is to consider the cases independently. Let's start with the `Empty` case. There is no recursive call here, so reasoning about recursion doesn't come into play. Let's instead use the types. There is no input here other than the `Empty` case we have already matched, so we cannot use the input types to further restrict the code. Let's instead consider the output type. We're trying to create a `MyList[B]`. There are only two ways to create a `MyList[B]`: an `Empty` or a `Pair`. To create a `Pair` we need a `head` of type `B`, which we don't have. So we can only use `Empty`. *This is the only possible code we can write.* The types are sufficiently restrictive that we cannot write incorrect code for this case.

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])
```

```
  def map[B](f: A => B): MyList[B] =
    this match {
      case Empty() => Empty()
      case Pair(head, tail) => ??? tail.map(f)
    }
}
```

Now let's move to the `Pair` case. We can apply both the structural recursion reasoning strategy and following the types. Let's use each in turn.

The case for `Pair` is

```
case Pair(head, tail) => ??? tail.map(f)
```

Remember we can consider this independently of the other case. We assume the recursion is correct. This means we only need to think about what we should do with the `head`, and how we should combine this result with `tail.map(f)`. Let's now follow the types to finish the code. Our goal is to produce a `MyList[B]`. We already the following available:

- `tail.map(f)`, which has type `MyList[B]`;
- `head`, with type `A`;
- `f`, with type `A => B`; and
- the constructors `Empty` and `Pair`.

We could return just `Empty`, matching the case we've already written. This has the correct type but we might expect it is not the correct answer because it does not use the result of the recursion, `head`, or `f` in any way.

We could return just `tail.map(f)`. This has the correct type but we might expect it is not correct because we don't use `head` or `f` in any way.

We can call `f` on `head`, producing a value of type `B`, and then combine this value and the result of the recursive call using `Pair` to produce a `MyList[B]`. This is the correct solution.

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])

  def map[B](f: A => B): MyList[B] =
    this match {
      case Empty() => Empty()
      case Pair(head, tail) => Pair(f(head), tail.map(f))
    }
}
```

If you've followed this example you've hopefully see how we can use the three strategies to systematically find the correct implementation. Notice how we interleaved the recursion strategy and following the types to guide us to a solution for the `Pair` case. Also note how following the types alone gave us three possible implementations for the `Pair` case. In this code, and as is usually the case, the solution was the implementation that used all of the available inputs.

3.3.3. Exhaustivity Checking

Remember that algebraic data types are a closed world: they cannot be extended once defined. The Scala compiler can use this to check that we handle all possible cases in a pattern match, so long as we write the pattern match in a way the compiler can work with. This is known as exhaustivity checking.

Here's a simple example. We start by defining a straight-forward algebraic data type.

```
// Some of the possible units for lengths in CSS
enum CssLength {
```

```
  case Em(value: Double)
  case Rem(value: Double)
  case Pt(value: Double)
}
```

If we write a pattern match using the structural recursion strategy, the compiler will complain if we're missing a case.

```
import CssLength.*

CssLength.Em(2.0) match {
  case Em(value) => value
  case Rem(value) => value
}
// -- [E029] Pattern Match Exhaustivity Warning:
---------------------------------
// 1 |CssLength.Em(2.0) match {
//   |^^^^^^^^^^^^^^^^^
//   |match may not be exhaustive.
//   |
//   |It would fail on pattern case: CssLength.Pt(_)
//   |
//   | longer explanation available when compiling with `-
explain`
```

Exhaustivity checking is incredibly useful. For example, if we add or remove a case from an algebraic data type, the compiler will tell us all the pattern matches that need to be updated.

3.3.4. Dynamic Dispatch

Using dynamic dispatch to implement structural recursion is an implementation technique that may feel more natural to people with a background in object-oriented programming.

The dynamic dispatch approach consists of:

1. defining an *abstract method* at the root of the algebraic data types; and

2. implementing that abstract method at every leaf of the algebraic data type.

This implementation technique is only available if we use the Scala 2 encoding of algebraic data types.

Let's see it in the `MyList` example we just looked at. Our first step is to rewrite the definition of `MyList` to the Scala 2 style.

```
sealed abstract class MyList[A] extends Product with Serializable
final case class Empty[A]() extends MyList[A]
final case class Pair[A](head: A, tail: MyList[A]) extends
MyList[A]
```

Next we define an abstract method for `map` on `MyList`.

```
sealed abstract class MyList[A] extends Product with Serializable
{
  def map[B](f: A => B): MyList[B]
}
final case class Empty[A]() extends MyList[A]
final case class Pair[A](head: A, tail: MyList[A]) extends
MyList[A]
```

Then we implement `map` on the concrete subtypes `Empty` and `Pair`.

```
sealed abstract class MyList[A] extends Product with Serializable
{
  def map[B](f: A => B): MyList[B]
}
final case class Empty[A]() extends MyList[A] {
  def map[B](f: A => B): MyList[B] =
    Empty()
}
final case class Pair[A](head: A, tail: MyList[A]) extends
MyList[A] {
  def map[B](f: A => B): MyList[B] =
    Pair(f(head), tail.map(f))
}
```

We can use exactly the same strategies we used in the pattern matching case to create this code. The implementation technique is different but the underlying concept is the same.

Given we have two implementation strategies, which should we use? If we're using `enum` in Scala 3 we don't have a choice; we must use pattern matching. In other situations we can choose between the two. I prefer to use pattern matching when I can, as it puts the entire method definition in one place. However, Scala 2 in particular has problems inferring types in some pattern matches. In these situations we can use dynamic dispatch instead. We'll learn more about this when we look at generalized algebraic data types.

Exercise: Methods for Tree

In a previous exercise we created a `Tree` algebraic data type:

```
enum Tree[A] {
  case Leaf(value: A)
  case Node(left: Tree[A], right: Tree[A])
}
```

Or, in the Scala 2 encoding:

```
sealed abstract class Tree[A] extends Product with Serializable
final case class Leaf[A](value: A) extends Tree[A]
final case class Node[A](left: Tree[A], right: Tree[A]) extends
Tree[A]
```

Let's get some practice with structural recursion and write some methods for `Tree`. Implement

- `size`, which returns the number of values (`Leafs`) stored in the `Tree`;
- `contains`, which returns `true` if the `Tree` contains a given element of type `A`, and `false` otherwise; and
- `map`, which creates a `Tree[B]` given a function `A => B`

Use whichever you prefer of pattern matching or dynamic dispatch to implement the methods.

3.3.5. Folds as Structural Recursions

Let's finish by looking at the fold method as an abstraction over structural recursion. If you did the `Tree` exercise above, you will have noticed that we wrote the same kind of code again and again. Here are the methods we wrote. Notice the left-hand sides of the pattern matches are all the same, and the right-hand sides are very similar.

```
def size: Int =
  this match {
    case Leaf(value)       => 1
    case Node(left, right) => left.size + right.size
  }

def contains(element: A): Boolean =
  this match {
    case Leaf(value)       => element == value
    case Node(left, right) => left.contains(element) ||
right.contains(element)
  }

def map[B](f: A => B): Tree[B] =
  this match {
    case Leaf(value)       => Leaf(f(value))
    case Node(left, right) => Node(left.map(f), right.map(f))
  }
```

This is the point of structural recursion: to recognize and formalize this similarity. However, as programmers we might want to abstract over this repetition. Can we write a method that captures everything that doesn't change in a structural recursion, and allows the caller to pass arguments for everything that does change? It turns out we can. For any algebraic data type we can define at least one method, called a fold, that captures all the parts of structural recursion that don't change and allows the caller to specify all the problem specific parts.

Let's see how this is done using the example of `MyList`. Recall the definition of `MyList` is

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])
}
```

We know the structural recursion skeleton for `MyList` is

```
def doSomething[A](list: MyList[A]) =
  list match {
    case Empty()          => ???
    case Pair(head, tail) => ??? doSomething(tail)
  }
```

Implementing fold for `MyList` means defining a method

```
def fold[A, B](list: MyList[A]): B =
  list match {
    case Empty() => ???
    case Pair(head, tail) => ??? fold(tail)
  }
```

where `B` is the type the caller wants to create.

To complete `fold` we add method parameters for the problem specific (`???`) parts. In the case for `Empty`, we need a value of type `B` (notice that I'm following the types here).

```
def fold[A, B](list: MyList[A], empty: B): B =
  list match {
    case Empty() => empty
    case Pair(head, tail) => ??? fold(tail, empty)
  }
```

For the `Pair` case, we have the head of type `A` and the recursion producing a value of type `B`. This means we need a function to combine these two values.

```
def foldRight[A, B](list: MyList[A], empty: B, f: (A, B) => B): B
=
  list match {
```

```
    case Empty() => empty
    case Pair(head, tail) => f(head, foldRight(tail, empty, f))
  }
```

This is `foldRight` (and I've renamed the method to indicate this). You might have noticed there is another valid solution. Both `empty` and the recursion produce values of type `B`. If we follow the types we can come up with

```
def foldLeft[A,B](list: MyList[A], empty: B, f: (A, B) => B): B =
  list match {
    case Empty() => empty
    case Pair(head, tail) => foldLeft(tail, f(head, empty), f)
  }
```

which is `foldLeft`, the tail-recursive variant of fold for a list. (We'll talk about tail-recursion in a later chapter.)

We can follow the same process for any algebraic data type to create its folds. The rules are:

- a fold is a function from the algebraic data type and additional parameters to some generic type that I'll call `B` below for simplicity;
- the fold has one additional parameter for each case in a logical or;
- each parameter is a function, with result of type `B` and parameters that have the same type as the corresponding constructor arguments *except* recursive values are replaced with `B`; and
- if the constructor has no arguments (for example, `Empty`) we can use a value of type `B` instead of a function with no arguments.

Returning to `MyList`, it has:

- two cases, and hence two parameters to fold (other than the parameter that is the list itself);
- `Empty` is a constructor with no arguments and hence we use a parameter of type `B`; and

- `Pair` is a constructor with one parameter of type `A` and one recursive parameter, and hence the corresponding function has type `(A, B) => B`.

Exercise: Tree Fold

Implement a fold for `Tree` defined earlier. There are several different ways to traverse a tree (pre-order, post-order, and in-order). Just choose whichever seems easiest.

Exercise: Using Fold

Prove to yourself that you can replace structural recursion with calls to fold, by redefining `size`, `contains`, and `map` for `Tree` using only fold.

3.4. Structural Corecursion

Structural corecursion is the opposite—more correctly, the dual—of structural recursion. Whereas structural recursion tells us how to take apart an algebraic data type, structural corecursion tells us how to build up, or construct, an algebraic data type. Whereas we can use structural recursion whenever the input of a method or function is an algebraic data type, we can use structural corecursion whenever the output of a method or function is an algebraic data type.

> **Duality in Functional Programming**
>
> Two concepts or structures are duals if one can be translated in a one-to-one fashion to the other. Duality is

> one of the main themes of this book. By relating concepts as duals we can transfer knowledge from one domain to another.
>
> Duality is often indicated by attaching the co- prefix to one of the structures or concepts. For example, corecursion is the dual of recursion, and sum types, also known as coproducts, are the dual of product types.

Structural recursion works by considering all the possible inputs (which we usually represent as patterns), and then working out what we do with each input case. Structural corecursion works by considering all the possible outputs, which are the constructors of the algebraic data type, and then working out the conditions under which we'd call each constructor.

Let's return to the list with elements of type `A`, defined as:

- the empty list; or
- a pair containing an `A` and a tail, which is a list of `A`.

In Scala 3 we write

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])
}
```

We can use structural corecursion if we're writing a method that produces a `MyList`. A good example is `map`:

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])

  def map[B](f: A => B): MyList[B] =
    ???
}
```

The output of this method is a `MyList`, which is an algebraic data type. Since we need to construct a `MyList` we can use structural corecursion. The structural corecursion strategy says we write down all the constructors and then consider the conditions that will cause us to call each constructor. So our starting point is to just write down the two constructors, and put in dummy conditions.

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])

  def map[B](f: A => B): MyList[B] =
    if ??? then Empty()
    else Pair(???, ???)
}
```

We can also apply the recursion rule: where the data is recursive so is the method.

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])

  def map[B](f: A => B): MyList[B] =
    if ??? then Empty()
    else Pair(???, ???.map(f))
}
```

To complete the left-hand side we can use the strategies we've already seen:

- we can use structural recursion to tell us there are two possible conditions; and
- we can follow the types to align these conditions with the code we have already written.

In short order we arrive at the correct solution

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])

  def map[B](f: A => B): MyList[B] =
    this match {
      case Empty() => Empty()
      case Pair(head, tail) => Pair(f(head), tail.map(f))
    }
}
```

There are a few interesting points here. Firstly, we should acknowledge that `map` is both a structural recursion and a structural corecursion. This is not always the case. For example, `foldLeft` and `foldRight` are not structural corecursions because they are not constrained to only produce an algebraic data type. Secondly, note that when we walked through the process of creating `map` as a structural recursion we implicitly used the structural corecursion pattern, as part of following the types. We recognised that we were producing a `List`, that there were two possibilities for producing a `List`, and then worked out the correct conditions for each case. Formalizing structural corecursion as a separate strategy allows us to be more conscious of where we apply it. Finally, notice how I switched from an `if` expression to a pattern match expression as we progressed through defining `map`. This is perfectly fine. Both kinds of expression achieve the same effect. Pattern matching is a little bit safer due to exhaustivity checking. If we wanted to continue using an `if` we'd have to define a method (for example, `isEmpty`) that allows us to distinguish an `Empty` element from a `Pair`. This method would have to use pattern matching in its implementation, so avoiding pattern matching directly is just pushing it elsewhere.

3.4.1. Unfolds as Structural Corecursion

Just as we could abstract structural recursion as a fold, for any given algebraic data type we can abstract structural corecursion as an unfold. Unfolds are much less commonly used than folds, but they are still a nice tool to have.

Let's work through the process of deriving unfold, using `MyList` as our example again.

```
enum MyList[A] {
  case Empty()
  case Pair(head: A, tail: MyList[A])
}
```

The corecursion skeleton is

```
if ??? then MyList.Empty()
else MyList.Pair(???, recursion(???))
```

Our starting point is writing the skeleton for `unfold`. It's a little bit unusual in that I've added a parameter `seed`. This is the information we use to create an element. We'll need this, but we cannot derive it from our strategies, so I've added it in here as a starting assumption.

```
def unfold[A, B](seed: A): MyList[B] =
  ???
```

Now we start using our strategies to fill in the missing pieces. I'm using the corecursion skeleton and I've applied the recursion rule immediately in the code below, to save a bit of time in the derivation.

```
def unfold[A, B](seed: A): MyList[B] =
  if ??? then MyList.Empty()
  else MyList.Pair(???, unfold(seed))
```

We can abstract the condition using a function from `A => Boolean`.

```
def unfold[A, B](seed: A, stop: A => Boolean): MyList[B] =
  if stop(seed) then MyList.Empty()
  else MyList.Pair(???, unfold(seed, stop))
```

Now we need to handle the case for `Pair`. We have a value of type `A` (`seed`), so to create the `head` element of `Pair` we can ask for a function `A => B`

```
def unfold[A, B](seed: A, stop: A => Boolean, f: A => B):
MyList[B] =
  if stop(seed) then MyList.Empty()
  else MyList.Pair(f(seed), unfold(???, stop, f))
```

Finally we need to update the current value of `seed` to the next value. That's a function `A => A`.

```
def unfold[A, B](seed: A, stop: A => Boolean, f: A => B, next: A
=> A): MyList[B] =
  if stop(seed) then MyList.Empty()
  else MyList.Pair(f(seed), unfold(next(seed), stop, f, next))
```

At this point we're done. Let's see that `unfold` is useful by declaring some other methods in terms of it. We're going to declare `map`, which we've already seen is a structural corecursion, using `unfold`. We will also define `fill` and `iterate`, which are methods that construct lists and correspond to the methods with the same names on `List` in the Scala standard library.

To make this easier to work with I'm going to declare `unfold` as a method on the `MyList` companion object. I have made a slight tweak to the definition to make type inference work a bit better. In Scala, types inferred for one method parameter cannot be used for other method parameters in the same parameter list. However, types inferred for one method parameter list can be used in subsequent lists. Separating the function parameters from the `seed`

parameter means that the value inferred for `A` from `seed` can be used for inference of the function parameters' input parameters.

I have also declared some **destructor** methods, which are methods that take apart an algebraic data type. For `MyList` these are `head`, `tail`, and the predicate `isEmpty`. We'll talk more about these a bit later.

Here's our starting point.

```
enum MyList[A] {
  case Empty()
  case Pair(_head: A, _tail: MyList[A])

  def isEmpty: Boolean =
    this match {
      case Empty() => true
      case _       => false
    }

  def head: A =
    this match {
      case Pair(head, _) => head
    }

  def tail: MyList[A] =
    this match {
      case Pair(_, tail) => tail
    }
}
object MyList {
  def unfold[A, B](seed: A)(stop: A => Boolean, f: A => B, next:
A => A): MyList[B] =
    if stop(seed) then MyList.Empty()
    else MyList.Pair(f(seed), unfold(next(seed))(stop, f, next))
}
```

Now let's define the constructors `fill` and `iterate`, and `map`, in terms of `unfold`. I think the constructors are a bit simpler, so I'll do those first.

```
object MyList {
  def unfold[A, B](seed: A)(stop: A => Boolean, f: A => B, next:
```

```
A => A): MyList[B] =
    if stop(seed) then MyList.Empty()
    else MyList.Pair(f(seed), unfold(next(seed))(stop, f, next))

  def fill[A](n: Int)(elem: => A): MyList[A] =
    ???

  def iterate[A](start: A, len: Int)(f: A => A): MyList[A] =
    ???
}
```

Here I've just added the method skeletons, which are taken straight from the `List` documentation. To implement these methods we can use one of two strategies:

- reasoning about loops in the way we might in an imperative language; or
- reasoning about structural recursion over the natural numbers.

Let's talk about each in turn.

You might have noticed that the parameters to `unfold` are almost exactly those you need to create a for-loop in a language like Java. A classic for-loop, of the `for(i = 0; i < n; i++)` kind, has four components:

1. the initial value of the loop counter;
2. the stopping condition of the loop;
3. the statement that advances the counter; and
4. the body of the loop that uses the counter.

These correspond to the `seed`, `stop`, `next`, and `f` parameters of `unfold` respectively.

Loop variants and invariants are the standard way of reasoning about imperative loops. I'm not going to describe them here, as you have probably already learned how to reason about loops (though perhaps not using these terms). Instead I'm going to discuss the second reasoning strategy, which relates writing `unfold` to something we've already discussed: structural recursion.

Our first step is to note that natural numbers (the integers 0 and larger) are conceptually algebraic data types even though the implementation in Scala—using `Int`—is not. A natural number is either:

- zero; or
- 1 + a natural number.

It's the simplest possible algebraic data type that is both a sum and a product type.

Once we see this, we can use the reasoning tools for structural recursion for creating the parameters to `unfold`. Let's show how this works with `fill`. The `n` parameter tells us how many elements there are in the `List` we're creating. The `elem` parameter creates those elements, and is called once for each element. So our starting point is to consider this as a structural recursion over the natural numbers. We can take `n` as `seed`, and `stop` as the function `x => x == 0`. These are the standard conditions for a structural recursion over the natural numbers. What about `next`? Well, the definition of natural numbers tells us we should subtract one in the recursive case, so `next` becomes `x => x - 1`. We only need `f`, and that comes from the definition of how `fill` is supposed to work. We create the value from `elem`, so `f` is just `_ => elem`

```
object MyList {
  def unfold[A, B](seed: A)(stop: A => Boolean, f: A => B, next:
A => A): MyList[B] =
    if stop(seed) then MyList.Empty()
    else MyList.Pair(f(seed), unfold(next(seed))(stop, f, next))

  def fill[A](n: Int)(elem: => A): MyList[A] =
    unfold(n)(_ == 0, _ => elem, _ - 1)

  def iterate[A](start: A, len: Int)(f: A => A): MyList[A] =
    ???
}
```

We should check that our implementation works as intended. We can do this by comparing it to `List.fill`.

```
List.fill(5)(1)
// res6: List[Int] = List(1, 1, 1, 1, 1)
MyList.fill(5)(1)
// res7: MyList[Int] = MyList(1, 1, 1, 1, 1)
```

Here's a slightly more complex example, using a stateful method to create a list of ascending numbers. First we define the state and method that uses it.

```
var counter = 0
def getAndInc(): Int = {
  val temp = counter
  counter = counter + 1
  temp
}
```

Now we can create it to create lists.

```
List.fill(5)(getAndInc())
// res8: List[Int] = List(0, 1, 2, 3, 4)
counter = 0
MyList.fill(5)(getAndInc())
// res10: MyList[Int] = MyList(0, 1, 2, 3, 4)
```

Exercise: Iterate

Implement `iterate` using the same reasoning as we did for `fill`. This is slightly more complex than `fill` as we need to keep two bits of information: the value of the counter and the value of type `A`.

Exercise: Map

Once you've completed `iterate`, try to implement `map` in terms of `unfold`. You'll need to use the destructors to implement it.

Now a quick discussion on destructors. The destructors do two things:

1. distinguish the different cases within a sum type; and
2. extract elements from each product type.

So for `MyList` the minimal set of destructors is `isEmpty`, which distinguishes `Empty` from `Pair`, and `head` and `tail`. The extractors are partial functions in the conceptual, not Scala, sense; they are only defined for a particular product type and throw an exception if used on a different case. You may have also noticed that the functions we passed to `fill` are exactly the destructors for natural numbers.

The destructors are another part of the duality between structural recursion and corecursion. Structural recursion is:

- defined by pattern matching on the constructors; and
- takes apart an algebraic data type into smaller pieces.

Structural corecursion instead is:

- defined by conditions on the input, which may use destructors; and
- build up an algebraic data type from smaller pieces.

One last thing before we leave `unfold`. If we look at the usual definition of `unfold` we'll probably find the following definition.

```
def unfold[A, B](in: A)(f: A => Option[(A, B)]): List[B]
```

This is equivalent to the definition we used, but a bit more compact in terms of the interface it presents. We used a more explicit definition that makes the structure of the method clearer.

3.5. The Algebra of Algebraic Data Types

A question that sometimes comes up is where the "algebra" in algebraic data types comes from. I want to talk about this a little bit and show some of the algebraic manipulations that can be done on algebraic data types.

The term algebra is used in the sense of abstract algebra, an area of mathematics. Abstract algebra deals with algebraic structures. An algebraic structure consists of a set of values, operations on that set, and properties that those operations must maintain. An example is the set of integers, the operations addition and multiplication, and the familiar properties of these operations such as associativity, which says that $a + (b + c) = (a + b) + c$. The abstract in abstract algebra means that it doesn't deal with concrete values like integers—that would be far too easy to understand—and instead with abstractions with wacky names like semigroup, monoid, and ring. The example of integers above is an instance of a ring. We'll see a lot more of these soon enough!

Algebraic data types also correspond to the algebraic structure called a ring. A ring has two operations, which are conventionally written $+$ and $\times$. You'll perhaps guess that these correspond to sum and product types respectively, and you'd be absolutely correct. What about the properties of these operations? We'll they are similar to what we know from basic algebra:

- $+$ and $\times$ are associative, so $a + (b + c) = (a + b) + c$ and likewise for $\times$;
- $a + b = b + a$, known as commutivitiy;
- there is an identity 0 such that $a + 0 = a$;
- there is an identity 1 such that $a \times 1 = a$;
- there is distribution, so that $a \times (b + c) = (a \times b) + (a \times c)$

So far, so abstract. Let's make it concrete by looking at actual examples in Scala.

Remember the algebraic data types work with types, so the operations $+$ and $\times$ take types as parameters. So Int $\times$ String is equivalent to

```
final case class IntAndString(int: Int, string: String)
```

We can use tuples to avoid creating lots of names.

```
type IntAndString = (Int, String)
```

We can do the same thing for $+$. Int $+$ String is

```
enum IntOrString {
  case IsInt(int: Int)
  case IsString(string: String)
}
```

or just

```
type IntOrString = Either[Int, String]
```

Exercise: Identities

Can you work out which Scala type corresponds to the identity 1 for product types?

What about the Scala type corresponding to the identity 0 for sum types?

What about the distribution law? This allows us to manipulate algebraic data types to form equivalent, but perhaps more useful, representations. Consider this example of a user data type.

```
final case class Person(name: String, permissions: Permissions)
enum Permissions {
```

```
  case User
  case Moderator
}
```

Written in mathematical notation, this is

$$\text{Person} = \text{String} \times \text{Permissions}$$
$$\text{Permissions} = \text{User} + \text{Moderator}$$

Performing substitution gets us

$$\text{Person} = \text{String} \times (\text{User} + \text{Moderator})$$

Applying distribution results in

$$\text{Person} = (\text{String} \times \text{User}) + (\text{String} \times \text{Moderator})$$

which in Scala we can represent as

```
enum Person {
  case User(name: String)
  case Moderator(name: String)
}
```

Is this representation more useful? I can't say without the context of where the data is being used. However I can say that knowing this manipulation is possible, and correct, is useful.

There is a lot more that could be said about algebraic data types, but at this point I feel we're really getting into the weeds. I'll finish up with a few pointers to other interesting facts:

- Exponential types exist. They are functions! A function `A => B` is equivalent to b^a.
- Quotient types also exist, but they are a bit weird. Read up about them if you're interested.
- Another interesting algebraic manipulation is taking the derivative of an algebraic data type. This gives us a kind of iterator, known as a zipper, for that type.

3.6. Conclusions

We have covered a lot of material in this chapter. Let's recap the key points.

Algebraic data types allow us to express data types by combining existing data types with logical and and logical or. A logical and constructs a product type while a logical or constructs a sum type. Algebraic data types are the main way to represent data in Scala.

Structural recursion gives us a skeleton for transforming any given algebraic data type into any other type. Structural recursion can be abstracted into a `fold` method.

We use several reasoning principles to help us complete the problem specific parts of a structural recursion:

1. reasoning independently by case;
2. assuming recursion is correct; and
3. following the types.

Following the types is a very general strategy that is can be used in many other situations.

Structural corecursion gives us a skeleton for creating any given algebraic data type from any other type. Structural corecursion can be abstracted into an `unfold` method. When reasoning about structural corecursion we can reason as we would for an imperative loop, or, if the input is an algebraic data type, use the principles for reasoning about structural recursion.

Notice that the two main themes of functional programming—composition and reasoning—are both already apparent. Algebraic data types are compositional: we compose algebraic data types using sum and product. We've seen many reasoning principles in this chapter.

I haven't covered everything there is to know about algebraic data types; I think doing so would be a book in its own right. Below are

some references that you might find useful if you want to dig in further, as well as some biographical remarks.

Algebraic data types are standard in introductory material on functional programming. Structural recursion is certainly extremely common in functional programming, but strangely seems to rarely be explicitly defined as I've done here. I learned about both from *How to Design Programs* [27].

I'm not aware of any approachable yet thorough treatment of either algebraic data types or structural recursion. Both seem to have become assumed background of any researcher in the field of programming languages, and relatively recent work is caked in layers of mathematics and obtuse notation that I find difficult reading. The infamous *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire* [56] is an example of such work. I suspect the core ideas of both date back to at least the emergence of computability theory in the 1930s, well before any digital computers existed.

The earliest reference I've found to structural recursion is *Proving Properties of Programs by Structural Induction* [9]. Algebraic data types don't seem to have been fully developed, along with pattern matching, until NPL[19] in 1977. NPL was quickly followed by the more influential language Hope[20], which spread the concept to other programming languages.

Corecursion is a bit better documented in the contemporary literature. *How to Design Co-Programs* [35] covers the main ideas we have looked at here, while *The Under-appreciated Unfold* [32] discusses uses of `unfold`.

The Derivative of a Regular Type is its Type of One-Hole Contexts [55] describes the derivative of algebraic data types.

[19] https://en.wikipedia.org/wiki/NPL_(programming_language)
[20] https://en.wikipedia.org/wiki/Hope_(programming_language)

4. Objects as Codata

In this chapter we will look at **codata**, the dual of algebraic data types. Algebraic data types focus on how things are constructed. Codata, in contrast, focuses on how things are used. We define codata by specifying the operations that can be performed on the type. This is very similar to the use of interfaces in object-oriented programming, and this is the first reason that we are interested in codata: codata puts object-oriented programming into a coherent conceptual framework with the other strategies we are discussing.

We're not only interested in codata as a lens to view object-oriented programming. Codata also has properties that algebraic data does not. Codata allows us to create structures with an infinite number of elements, such as a list that never ends or a server loop that runs indefinitely. Codata has a different form of extensibility to algebraic data. Whereas we can easily write new functions that transform algebraic data, we cannot add new cases to the definition of an algebraic data type without changing the existing code. The reverse is true for codata. We can easily create new implementations of codata, but functions that transform codata are limited by the interface the codata defines.

In the previous chapter we saw structural recursion and structural corecursion as strategies to guide us in writing programs using algebraic data types. The same holds for codata. We can use codata forms of structural recursion and corecursion to guide us in writing programs that consume and produce codata respectively.

We'll begin our exploration of codata by more precisely defining it and seeing some examples. We'll then talk about representing codata in Scala, and the relationship to object-oriented programming. Once we can create codata, we'll see how to work with it using structural recursion and corecursion, using an example of an infinite structure. Next we will look at transforming

algebraic data to codata, and vice versa. We will finish by examining differences in extensibility.

A quick note about terminology before we proceed. We might expect to use the term algebraic codata for the dual of algebraic data, but conventionally just codata is used. I assume this is because data is usually understood to have a wider meaning than just algebraic data, but codata is not used outside of programming language theory. For simplicity and symmetry, within this chapter I'll just use the term data to refer to algebraic data types.

4.1. Data and Codata

Data describes what things are, while codata describes what things can do.

We have seen that data is defined in terms of constructors producing elements of the data type. Let's take a very simple example: a `Bool` is either `True` or `False`. We know we can represent this in Scala as

```
enum Bool {
  case True
  case False
}
```

The definition tells us there are two ways to construct an element of type `Bool`. Furthermore, if we have such an element we can tell exactly which case it is, by using a pattern match for example. Similarly, if the instances themselves hold data, as in `List` for example, we can always extract all the data within them. Again, we can use pattern matching to achieve this.

Codata, in contrast, is defined in terms of operations we can perform on the elements of the type. These operations are sometimes called **destructors** (which we've already encountered),

observations, or **eliminators**. A common example of codata is a data structure such as a set. We might define the operations on a `Set` with elements of type `A` as:

- `contains`, which takes a `Set[A]` and an element `A` and returns a `Boolean` indicating if the set contains the element;
- `insert`, which takes a `Set[A]` and an element `A` and returns a `Set[A]` containing all the elements from the original set and the new element; and
- `union`, which takes a `Set[A]` and a set `Set[A]` and returns a `Set[A]` containing all the elements of both sets.

In Scala we could implement this definition as

```
trait Set[A] {

  /** True if this set contains the given element */
  def contains(elt: A): Boolean

  /** Construct a new set containing all elements in this set and
the given element */
  def insert(elt: A): Set[A]

  /** Construct the union of this and that set */
  def union(that: Set[A]): Set[A]
}
```

This definition does not tell us anything about the internal representation of the elements in the set. It could use a hash table, a tree, or something more exotic. It does, however, tell us what we can do with the set. We know we can take the union but not the intersection, for example.

If you come from the object-oriented world you might recognize the description of codata above as programming to an interface. In some ways codata is just taking concepts from the object-oriented world and presenting them in a way that is consistent with the rest of the functional programming paradigm. However, this does not mean adopting all the features of object-oriented programming. We won't use state, which is difficult to reason

about. We also won't use implementation inheritance either, for the same reason. In our subset of object-oriented programming we'll either be defining interfaces (which may have default implementations of some methods) or final classes that implement those interfaces. Interestingly, this subset of object-oriented programming is often recommended by advocates of object-oriented programming[21].

Let's now be a little more precise in our definition of codata, which will make the duality between data and codata clearer. Remember the definition of data: it is defined in terms of sums (logical ors) and products (logical ands). We can transform any data into a sum of products, which is disjunctive normal form. Each product in the sum is a constructor, and the product itself is the parameters that the constructor accepts. Finally, we can think of constructors as functions which take some arbitrary input and produce an element of data. Our end point is a sum of functions from arbitrary input to data.

More concretely, if we are constructing an element of some data type `A` we call one of the constructors

- `A1: (B, C, ...) => A`; or
- `A2: (D, E, ...) => A`; or
- `A3: (F, G, ...) => A`; and so on.

Now we'll turn to codata. Codata is defined as a product of functions, these functions being the destructors. The input to a destructor is always an element of the codata type and possibly some other parameters. The output is usually something that is not of the codata type. Thus constructing an element of some codata type `A` means defining

- `A1: (A, B, ...) => C`; and
- `A2: (A, D, ...) => E`; and

[21]For example, *Effective Java* [6] suggests developers "minimize mutability" and "favor composition over [implementation] inheritance". Together these form the subset of object-oriented programming that we consider to be codata.

- `A3: (A, F, ...) => G`; and so on.

This hopefully makes the duality between the two clearer.

Now we understand what codata is, we will turn to representing codata in Scala.

4.2. Codata in Scala

We have already seen an example of codata, which I have repeated below.

```
trait Set[A] {

  def contains(elt: A): Boolean

  def insert(elt: A): Set[A]

  def union(that: Set[A]): Set[A]
}
```

The abstract definition of this, which is a product of functions, defines a `Set` with elements of type `A` as:

- a function `contains` taking a `Set[A]` and an element `A` and returning a `Boolean`,
- a function `insert` taking a `Set[A]` and an element `A` and returning a `Set[A]`, and
- a function `union` taking a `Set[A]` and a set `Set[A]` and returning a `Set[A]`.

Notice that the first parameter of each function is the type we are defining, `Set[A]`.

The translation to Scala is:

- the overall type becomes a `trait`; and

- each function becomes a method on that `trait`. The first parameter is the hidden `this` parameter, and other parameters become normal parameters to the method.

This gives us the Scala representation we started with.

This is only half the story for codata. We also need to actually implement the interface we've just defined. There are three approaches we can use:

1. a `final` subclass, in the case where we want to name the implementation;
2. an anonymous subclass; or
3. more rarely, an `object`.

Neither `final` nor anonymous subclasses can be further extended, meaning we cannot create deep inheritance hierarchies. This in turn avoids the difficulties that come from reasoning about deep hierarchies. Using a `class` rather than a `case class` means we don't expose implementation details like constructor arguments.

Some examples are in order. Here's a simple example of `Set`, which uses a `List` to hold the elements in the set.

```
final class ListSet[A](elements: List[A]) extends Set[A] {

  def contains(elt: A): Boolean =
    elements.contains(elt)

  def insert(elt: A): Set[A] =
    ListSet(elt :: elements)

  def union(that: Set[A]): Set[A] =
    elements.foldLeft(that) { (set, elt) => set.insert(elt) }
}
object ListSet {
  def empty[A]: Set[A] = ListSet(List.empty)
}
```

This uses the first implementation approach, a `final` subclass. Where would we use an anonymous subclass? They are most useful when implementing methods that return our codata type.

Let's take `union` as an example. It returns our codata type, `Set`, and we could implement it as shown below.

```
trait Set[A] {

  def contains(elt: A): Boolean

  def insert(elt: A): Set[A]

  def union(that: Set[A]): Set[A] = {
    val self = this
    new Set[A] {
      def contains(elt: A): Boolean =
        self.contains(elt) || that.contains(elt)

      def insert(elt: A): Set[A] =
        // Arbitrary choice to insert into self
        self.insert(elt).union(that)
    }
  }
}
```

This uses an anonymous subclass to implement `union` on the `Set` trait, and hence defines the method for all subclasses. I haven't made the method `final` so that subclasses can override it with a more efficient implementation. This does open up the danger of implementation inheritance. This is an example of where theory and craft diverge. In theory we never want implementation inheritance, but in practice it can be useful as an optimization.

It can also be useful to implement utility methods defined purely in terms of the destructors. Let's say we wanted to implement a method `containsAll` that checks if a `Set` `contains` all the elements in an `Iterable` collection.

```
def containsAll(elements: Iterable[A]): Boolean
```

We can implement this purely in terms of `contains` on `Set` and `forall` on `Iterable`.

```
trait Set[A] {

  def contains(elt: A): Boolean

  def insert(elt: A): Set[A]

  def union(that: Set[A]): Set[A]

  def containsAll(elements: Iterable[A]): Boolean =
    elements.forall(elt => this.contains(elt))
}
```

Once again we could make this a `final` method. In this case it's probably more justified as it's difficult to imagine a more efficient implementation.

Data and codata are both realized in Scala as variations of the same language features of classes and objects. This means we can define types that have properties of both data and codata. We have actually already done this. When we define data we must define names for the fields within the data, thus defining destructors. Most languages are same, not making a hard distinction between data and codata.

Part of the appeal, I think, of classes and objects is that they can express so many conceptually different abstractions with the same language constructs. This gives them a surface appearance of simplicity; it seems we need to learn only one abstraction to solve a huge of number of coding problems. However this apparent simplicity hides real complexity, as this variety of uses forces us to reverse engineer the conceptual intention from the code.

4.3. Structural Recursion and Corecursion for Codata

In this section we'll build a library for streams, also known as lazy lists. These are the codata equivalent of lists. Whereas a list must have a finite length, streams have an infinite length. We'll use this example to explore structural recursion and structural corecursion as applied to codata.

Let's start by reviewing structural recursion and corecursion. The key idea is to use the input or output type, respectively, to drive the process of writing the method. We've already seen how this works with data, where we emphasized structural recursion. With codata it's more often the case that structural corecursion is used. The steps for using structural corecursion are:

1. recognize the output of the method or function is codata;
2. write down the skeleton to construct an instance of the codata type, usually using an anonymous subclass; and
3. fill in the methods, where strategies such as structural recursion or following the types can help.

It's important that all computations are defined within the methods, and so only run when the methods are called. Once we start creating streams the importance of this will become clear.

For structural recursion the steps are:

1. recognize the input of the method or function is codata;
2. note the codata's destructors as possible sources of values in writing the method; and
3. complete the method, using strategies such as following the types or structural corecursion and the methods identified above.

Now on to creating streams. Our first step is to define our stream type. As this is codata, it is defined in terms of its destructors. The destructors that define a `Stream` of elements of type `A` are:

- a `head` of type `A`; and
- a `tail` of type `Stream[A]`.

Note these are almost the destructors of `List`. We haven't defined `isEmpty` as a destructor because our streams never end and thus this method would always return `false`[22].

We can translate this to Scala, as we've previously seen, giving us

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]
}
```

Now we can create an instance of `Stream`. Let's create a never-ending stream of ones. We will start with the skeleton below and apply strategies to complete the code.

```
val ones: Stream[Int] = ???
```

The first strategy is structural corecursion. We're returning an instance of codata, so we can insert the skeleton to construct a `Stream`.

```
val ones: Stream[Int] =
  new Stream[Int] {
    def head: Int = ???
    def tail: Stream[Int] = ???
  }
```

[22] A lot of real implementations, such as the `LazyList` in the Scala standard library, do define such a method which allows them to represent finite and infinite lists in the same structure. We're not doing this for simplicity and because we want to work with codata in its purest form.

Here I've used the anonymous subclass approach, so I can just write all the code in one place.

The next step is to fill in the method bodies. The first method, `head`, is trivial. The answer is `1` by definition.

```
val ones: Stream[Int] =
  new Stream[Int] {
    def head: Int = 1
    def tail: Stream[Int] = ???
  }
```

It's not so obvious what to do with `tail`. We want to return a `Stream[Int]` so we could apply structural corecursion again.

```
val ones: Stream[Int] =
  new Stream[Int] {
    def head: Int = 1
    def tail: Stream[Int] =
      new Stream[Int] {
        def head: Int = 1
        def tail: Stream[Int] = ???
      }
  }
```

This approach doesn't seem like it's going to work. We'll have to write this out an infinite number of times to correctly implement the method, which might be a problem.

Instead we can follow the types. We need to return a `Stream[Int]`. We have one in scope: `ones`. This is exactly the `Stream` we need to return: the infinite stream of ones!

```
val ones: Stream[Int] =
  new Stream[Int] {
    def head: Int = 1
    def tail: Stream[Int] = ones
  }
```

You might be alarmed to see the circular reference to `ones` in `tail`. This works because it is within a method, and so is only evaluated

when that method is called. This delaying of evaluation is what allows us to represent an infinite number of elements, as we only ever evaluate a finite portion of them. This is a core difference from data, which is fully evaluated when it is constructed.

Let's check that our definition of `ones` does indeed work. We can't extract all the elements from an infinite `Stream` (at least, not in finite time) so in general we'll have to resort to checking a finite sequence of elements.

```
ones.head
// res0: Int = 1
ones.tail.head
// res1: Int = 1
ones.tail.tail.head
// res2: Int = 1
```

This all looks correct. We'll often want to check our implementation in this way, so let's implement a method, `take`, to make this easier.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]

  def take(count: Int): List[A] =
    count match {
      case 0 => Nil
      case n => head :: tail.take(n - 1)
    }
}
```

We can use either the structural recursion or structural corecursion strategies for data to implement `take`. Since we've already covered these in detail I won't go through them here. The important point is that `take` only uses the destructors when interacting with the `Stream`.

Now we can more easily check our implementations are correct.

```
ones.take(5)
// res4: List[Int] = List(1, 1, 1, 1, 1)
```

For our next task we'll implement `map`. Implementing a method on `Stream` allows us to see both structural recursion and corecursion for codata in action. As usual we begin by writing out the method skeleton.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]

  def map[B](f: A => B): Stream[B] =
    ???
}
```

Now we have a choice of strategy to use. Since we haven't used structural recursion yet, let's start with that. The input is codata, a `Stream`, and the structural recursion strategy tells us we should consider using the destructors. Let's write them down to remind us of them.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]

  def map[B](f: A => B): Stream[B] = {
    this.head ???
    this.tail ???
  }
}
```

To make progress we can follow the types or use structural corecursion. Let's choose corecursion to see another example of it in use.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]
```

```
  def map[B](f: A => B): Stream[B] = {
    this.head ???
    this.tail ???

    new Stream[B] {
      def head: B = ???
      def tail: Stream[B] = ???
    }
  }
}
```

Now we've used structural recursion and structural corecursion, a bit of following the types is in order. This quickly arrives at the correct solution.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]

  def map[B](f: A => B): Stream[B] = {
    val self = this
    new Stream[B] {
      def head: B = f(self.head)
      def tail: Stream[B] = self.tail.map(f)
    }
  }
}
```

There are two important points. Firstly, notice how I gave the name `self` to `this`. This is so I can access the value inside the new `Stream` we are creating, where `this` would be bound to this new `Stream`. Next, notice that we access `self.head` and `self.tail` inside the methods on the new `Stream`. This maintains the correct semantics of only performing computation when it has been asked for. If we perform computation outside of the methods we create the possibility of infinite loops.

As our final example, let's return to constructing `Stream`, and implement the universal constructor `unfold`. We start with the skeleton for `unfold`, remembering the `seed` parameter.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]
}
object Stream {
  def unfold[A, B](seed: A): Stream[B] =
    ???
}
```

It's natural to apply structural corecursion to make progress.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]
}
object Stream {
  def unfold[A, B](seed: A): Stream[B] =
    new Stream[B]{
      def head: B = ???
      def tail: Stream[B] = ???
    }
}
```

Now we can follow the types, adding parameters as we need them. This gives us the complete method shown below.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]
}
object Stream {
  def unfold[A, B](seed: A, f: A => B, next: A => A): Stream[B] =
    new Stream[B]{
      def head: B =
        f(seed)
      def tail: Stream[B] =
        unfold(next(seed), f, next)
    }
}
```

We can use this to implement some interesting streams. Here's a stream that alternates between `1` and `-1`.

```
val alternating = Stream.unfold(
  true,
  x => if x then 1 else -1,
  x => !x
)
```

We can check it works.

```
alternating.take(5)
// res11: List[Int] = List(1, -1, 1, -1, 1)
```

Exercise: Stream Combinators

It's time for you to get some practice with structural recursion and structural corecursion using codata. Implement `filter`, `zip`, and `scanLeft` on `Stream`. They have the same semantics as the same methods on `List`, and the signatures shown below.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]

  def filter(pred: A => Boolean): Stream[A]
  def zip[B](that: Stream[B]): Stream[(A, B)]
  def scanLeft[B](zero: B)(f: (B, A) => B): Stream[B]
}
```

We can do some neat things with the methods defined above. For example, here is the stream of natural numbers.

```
val naturals = Stream.ones.scanLeft(0)((b, a) => b + a)
```

As usual, we should check it works.

```
naturals.take(5)
// res15: List[Int] = List(0, 1, 2, 3, 4)
```

We could also define `naturals` using `unfold`. More interesting is defining it in terms of itself.

```
val naturals: Stream[Int] =
  new Stream {
    def head = 1
    def tail = naturals.map(_ + 1)
  }
```

This might be confusing. If so, spend a bit of time thinking about it. It really does work!

```
naturals.take(5)
// res17: List[Int] = List(1, 2, 3, 4, 5)
```

4.3.1. Efficiency and Effects

You may have noticed that our implement recomputes values, possibly many times. A good example is the implementation of `filter`. This recalculates the `head` and `tail` on each call, which could be a very expensive operation.

```
def filter(pred: A => Boolean): Stream[A] = {
  val self = this
  new Stream[A] {
    def head: A = {
      def loop(stream: Stream[A]): A =
        if pred(stream.head) then stream.head
        else loop(stream.tail)

      loop(self)
    }

    def tail: Stream[A] = {
      def loop(stream: Stream[A]): Stream[A] =
        if pred(stream.head) then stream.tail.filter(pred)
        else loop(stream.tail)

      loop(self)
    }
```

```
  }
}
```

We know that delaying the computation until the method is called is important, because that is how we can handle infinite and self-referential data. However we don't need to redo this computation on successive calls. We can instead cache the result from the first call and use that next time. Scala makes this easy with `lazy val`, which is a `val` that is not computed until its first call. Additionally, Scala's use of the **uniform access principle** means we can implement a method with no parameters using a `lazy val`. Here's a quick example demonstrating it in use.

```
def always[A](elt: => A): Stream[A] =
  new Stream[A] {
    lazy val head: A = elt
    lazy val tail: Stream[A] = always(head)
  }

val twos = always(2)
```

As usual we should check our work.

```
twos.take(5)
// res18: List[Int] = List(2, 2, 2, 2, 2)
```

We get the same result whether we use a method or a `lazy val`, because we are assuming that we are only dealing with pure computations that have no dependency on state that might change. In this case a `lazy val` simply consumes additional space to save on time.

Recomputing a result every time it is needed is known as **call-by-name**, while caching the result the first time it is computed is known as **call-by-need**. These two different **evaluation strategies** can be applied to individual values, as we've done here, or across an entire programming. Haskell, for example, uses call-by-need; all values in Haskell are only computed the first time

they are needed. call-by-need is also commonly known as **lazy evaluation**. Another alternative, called **call-by-value**, computes results when they are defined instead of waiting until they are needed. This is the default in Scala.

We can illustrate the difference between call-by-name and call-by-need if we use an impure computation. For example, we can define a stream of random numbers. Random number generators depend on some internal state.

Here's the call-by-name implementation, using the methods we have already defined.

```
import scala.util.Random

val randoms: Stream[Double] =
  Stream.unfold(Random, r => r.nextDouble(), r => r)
```

Notice that we get *different* results each time we `take` a section of the `Stream`. We would expect these results to be the same.

```
randoms.take(5)
// res19: List[Double] = List(
//   0.9958715526508084,
//   0.9905054125578865,
//   0.7149026161241467,
//   0.41489341880053754,
//   0.8332214254923472
// )
randoms.take(5)
// res20: List[Double] = List(
//   0.018009030082648647,
//   0.19401791161758175,
//   0.4181208758067497,
//   0.5886545798244842,
//   0.5689310383336211
// )
```

Now let's define the same stream in a call-by-need style, using `lazy val`.

```
val randomsByNeed: Stream[Double] =
  new Stream[Double] {
    lazy val head: Double = Random.nextDouble()
    lazy val tail: Stream[Double] = randomsByNeed
  }
```

This time we get the *same* result when we `take` a section, and each number is the same.

```
randomsByNeed.take(5)
// res21: List[Double] = List(
//   0.4977714927617243,
//   0.4977714927617243,
//   0.4977714927617243,
//   0.4977714927617243,
//   0.4977714927617243
// )
randomsByNeed.take(5)
// res22: List[Double] = List(
//   0.4977714927617243,
//   0.4977714927617243,
//   0.4977714927617243,
//   0.4977714927617243,
//   0.4977714927617243
// )
```

If we wanted a stream that had a different random number for each element but those numbers were constant, we could redefine `unfold` to use call-by-need.

```
def unfoldByNeed[A, B](seed: A, f: A => B, next: A => A):
Stream[B] =
  new Stream[B]{
    lazy val head: B =
      f(seed)
    lazy val tail: Stream[B] =
      unfoldByNeed(next(seed), f, next)
  }
```

Now redefining `randomsByNeed` using `unfoldByNeed` gives us the result we are after. First, redefine it.

```
val randomsByNeed2 =
  unfoldByNeed(Random, r => r.nextDouble(), r => r)
```

Then check it works.

```
randomsByNeed2.take(5)
// res23: List[Double] = List(
//   0.4974976362208684,
//   0.4577160700892696,
//   0.12697756949708894,
//   0.8164660001821396,
//   0.19197356060616477
// )
randomsByNeed2.take(5)
// res24: List[Double] = List(
//   0.4974976362208684,
//   0.4577160700892696,
//   0.12697756949708894,
//   0.8164660001821396,
//   0.19197356060616477
// )
```

These subtleties are one of the reasons that functional programmers try to avoid using state as far as possible.

4.4. Relating Data and Codata

In this section we'll explore the relationship between data and codata, and in particular converting one to the other. We'll look at it in two ways: firstly a very surface-level relationship between the two, and then a deep connection via `fold`.

Remember that data is a sum of products, where the products are constructors and we can view constructors as functions. So we can view data as a sum of functions. Meanwhile, codata is a product of functions. We can easily make a direct correspondence between the functions-as-constructors and the functions in codata. What about the difference between the sum and the product that

remains. Well, when we have a product of functions we only call one at any point in our code. So the logical or is in the choice of function to call.

Let's see how this works with a familiar example of data, `List`. As an algebraic data type we can define

```
enum List[A] {
  case Pair(head: A, tail: List[A])
  case Empty()
}
```

The codata equivalent is

```
trait List[A] {
  def pair(head: A, tail: List[A]): List[A]
  def empty: List[A]
}
```

In the codata implementation we are explicitly representing the constructors as methods, and pushing the choice of constructor to the caller. In a few chapters we'll see a use for this relationship, but for now we'll leave it and move on.

The other way to view the relationship is a connection via `fold`. We've already learned how to derive the `fold` for any algebraic data type. For `Bool`, defined as

```
enum Bool {
  case True
  case False
}
```

the `fold` method is

```
enum Bool {
  case True
  case False

  def fold[A](t: A)(f: A): A =
```

```
    this match {
      case True => t
      case False => f
    }
}
```

We know that `fold` is universal: we can write any other method in terms of it. It therefore provides a universal destructor and is the key to treating data as codata. This example of `fold` is something we use all the time, except we usually call it `if`.

Here's the codata version of `Bool`, with `fold` renamed to `if`. (Note that Scala allows us to define methods with the same name as key words, in this case `if`, but we have to surround them in backticks to use them.)

```
trait Bool {
  def `if`[A](t: A)(f: A): A
}
```

Now we can define the two instances of `Bool` purely as codata.

```
val True = new Bool {
  def `if`[A](t: A)(f: A): A = t
}

val False = new Bool {
  def `if`[A](t: A)(f: A): A = f
}
```

Let's see this in use by defining `and` in terms of `if`, and then creating some examples. First the definition of `and`.

```
def and(l: Bool, r: Bool): Bool =
  new Bool {
    def `if`[A](t: A)(f: A): A =
      l.`if`(r)(False).`if`(t)(f)
  }
```

Now the examples. This is simple enough that we can try the entire truth table.

```
and(True, True).`if`("yes")("no")
// res1: String = "yes"
and(True, False).`if`("yes")("no")
// res2: String = "no"
and(False, True).`if`("yes")("no")
// res3: String = "no"
and(False, False).`if`("yes")("no")
// res4: String = "no"
```

Exercise: Or and Not

Test your understanding of `Bool` by implementing `or` and `not` in the same way we implemented `and` above.

Notice that, once again, computation only happens on demand. In this case, nothing happens until `if` is actually called. Until that point we're just building up a representation of what we want to happen. This again points to how codata can handle infinite data, by only computing the finite amount required by the actual computation.

The rules here for converting from data to codata are:

1. On the interface (`trait`) defining the codata, define a method with the same signature as `fold`.
2. Define an implementation of the interface for each product case in the data. The data's constructor arguments become constructor arguments on the codata `classes`. If there are no constructor arguments, as in `Bool`, we can define values instead of classes.
3. Each implementation implements the case of `fold` that it corresponds to.

Let's apply this to a slightly more complex example: `List`. We'll start by defining it as data and implementing `fold`. I've chosen to implement `foldRight` but `foldLeft` would be just as good.

```
enum List[A] {
  case Pair(head: A, tail: List[A])
  case Empty()

  def foldRight[B](empty: B)(f: (A, B) => B): B =
    this match {
      case Pair(head, tail) => f(head, tail.foldRight(empty)(f))
      case Empty() => empty
    }
}
```

Now let's implement it as codata. We start by defining the interface with the `fold` method. In this case I'm calling it `foldRight` as it's going to exactly mirror the `foldRight` we just defined.

```
trait List[A] {
  def foldRight[B](empty: B)(f: (A, B) => B): B
}
```

Now we define the implementations. There is one for `Pair` and one for `Empty`, which are the two cases in data definition of `List`. Notice that in this case the classes have constructor arguments, which correspond to the constructor arguments on the correspnding product types.

```
final class Pair[A](head: A, tail: List[A]) extends List[A] {
  def foldRight[B](empty: B)(f: (A, B) => B): B =
    ???
}

final class Empty[A]() extends List[A] {
  def foldRight[B](empty: B)(f: (A, B) => B): B =
    ???
}
```

I didn't implement the bodies of `foldRight` so I could show this as a separate step. The implementation here directly mirrors `foldRight` on the data implementation, and we can use the same strategies to implement the codata equivalents. That is to say, we

can use the recursion rule, reasoning by case, and following the types. I'm going to skip these details as we've already gone through them in depth. The final code is shown below.

```
final class Pair[A](head: A, tail: List[A]) extends List[A] {
  def foldRight[B](empty: B)(f: (A, B) => B): B =
    f(head, tail.foldRight(empty)(f))
}

final class Empty[A]() extends List[A] {
  def foldRight[B](empty: B)(f: (A, B) => B): B =
    empty
}
```

This code is almost the same as the dynamic dispatch implementation, which again shows the relationship between codata and object-oriented code.

The transformation from data to codata goes under several names: **refunctionalization**, **Church encoding**, and **Böhm-Berarducci encoding**. The latter two terms specifically refer to transformations into the untyped and typed lambda calculus respectively. The lambda calculus is a simple model programming language that contains only functions. We're going to take a quick detour to show that we can, indeed, encode lists using just functions. This demonstrates that objects and functions have equivalent power.

The starting point is creating a type alias `List`, which defines a list as a fold. This uses a polymorphic function type, which is new in Scala 3. Inspect the type signature and you'll see it is the same as `foldRight` above.

```
type List[A, B] = (B, (A, B) => B) => B
```

Now we can define `Pair` and `Empty` as functions. The first parameter list is the constructor arguments, and the second parameter list is the parameters for `foldRight`.

```
val Empty: [A, B] => () => List[A, B] =
  [A, B] => () => (empty, f) => empty

val Pair: [A, B] => (A, List[A, B]) => List[A, B] =
  [A, B] => (head: A, tail: List[A, B]) => (empty, f) =>
    f(head, tail(empty, f))
```

Finally, let's see an example to show it working. We will first define the list containing 1, 2, 3. Due to a restriction in polymorphic function types, I have to add the useless empty parameter.

```
val list: [B] => () => List[Int, B] =
  [B] => () => Pair(1, Pair(2, Pair(3, Empty())))
```

Now we can compute the sum and product of the elements in this list.

```
val sum = list()(0, (a, b) => a + b)
// sum: Int = 6
val product = list()(1, (a, b) => a * b)
// product: Int = 6
```

It works!

The purpose of this little demonstration is to show that functions are just objects (in the codata sense) with a single method. Scala this makes apparent, as functions *are* objects with an `apply` method.

We've seen that data can be translated to codata. The reverse is also possible: we simply tabulate the results of each possible method call. In other words, the data representation is memoisation, a lookup table, or a cache.

Although we can convert data to codata and vice versa, there are good reasons to choose one over the other. We've already seen one reason: with codata we can represent infinite structures. In this

next section we'll see another difference: the extensibility that data and codata permit.

4.5. Data and Codata Extensibility

We have seen that codata can represent types with an infinite number of elements, such as `Stream`. This is one expressive difference from data, which must always be finite. We'll now look at another, which is the type of extensibility we get from data and from codata. Together these gives use guidelines to choose between the two.

Firstly, let's define extensibility. It means the ability to add new features without modifying existing code. (If we allow modification of existing code then any extension becomes trivial.) In particular there are two dimensions along which we can extend code: adding new functions or adding new elements. We will see that data and codata have orthogonal extensibility: it's easy to add new functions to data but adding new elements is impossible without modifying existing code, while adding new elements to codata is straight-forward but adding new functions is not.

Let's start with a concrete example of both data and codata. For data we'll use the familiar `List` type.

```
enum List[A] {
  case Empty()
  case Pair(head: A, tail: List[A])
}
```

For codata, we'll use `Set` as our exemplar.

```
trait Set[A] {
  def contains(elt: A): Boolean
  def insert(elt: A): Set[A]
```

```
  def union(that: Set[A]): Set[A]
}
```

We know there are lots of methods we can define on `List`. The standard library is full of them! We also know that any method we care to write can be written using structural recursion. Finally, we can write these methods without modifying existing code.

Imagine `filter` was not defined on `List`. We can easily implement it as

```
import List.*

def filter[A](list: List[A], pred: A => Boolean): List[A] =
  list match {
    case Empty() => Empty()
    case Pair(head, tail) =>
      if pred(head) then Pair(head, filter(tail, pred))
      else filter(tail, pred)
  }
```

We could even use an extension method to make it appear as a normal method.

```
extension [A](list: List[A]) {
  def filter(pred: A => Boolean): List[A] =
    list match {
      case Empty() => Empty()
      case Pair(head, tail) =>
        if pred(head) then Pair(head, tail.filter(pred))
        else tail.filter(pred)
    }
}
```

This shows we can add new functions to data without issue.

What about adding new elements to data? Perhaps we want to add a special case to optimize single-element lists. This is impossible without changing existing code. By definition, we cannot add a new element to an `enum` without changing the `enum`. Adding such a new element would break all existing pattern matches, and so

require they all change. So in summary we can add new functions to data, but not new elements.

Now let's look at codata. This has the opposite extensibility; duality strikes again! In the codata case we can easily add new elements. We simply implement the `trait` that defines the codata interface. We saw this when we defined, for example, `ListSet`.

```
final class ListSet[A](elements: List[A]) extends Set[A] {

  def contains(elt: A): Boolean =
    elements.contains(elt)

  def insert(elt: A): Set[A] =
    ListSet(elt :: elements)

  def union(that: Set[A]): Set[A] =
    elements.foldLeft(that) { (set, elt) => set.insert(elt) }
}
object ListSet {
  def empty[A]: Set[A] = ListSet(List.empty)
}
```

What about adding new functionality? If the functionality can be defined in terms of existing functionality then we're ok. We can easily define this functionality, and we can use the extension method trick to make it appear like a built-in. However, if we want to define a function that cannot be expressed in terms of existing functions we are out of luck. Let's saw we want to define some kind of iterator over the elements of a `Set`. We might use a `LazyList`, the standard library's equivalent of `Stream` we defined earlier, because we know some sets have an infinite number of elements. Well, we can't do this without changing the definition of `Set`, which in turn breaks all existing implementations. We cannot define it in a different way because we don't know all the possible implementations of `Set`.

So in summary we can add new elements to codata, but not new functions.

If we tabulate this we clearly see that data and codata have orthogonal extensibility.

Extension	Data	Codata
Add elements	No	Yes
Add functions	Yes	No

This difference in extensibility gives us another rule for choosing between data and codata as an implementation strategy, in addition to the finite vs infinite distinction we saw earlier. If we want extensibilty of functions but not elements we should use data. If we have a fixed interface but an unknown number of possible implementations we should use codata.

You might wonder if we can have both forms of extensibility. Achieving this is called the **expression problem**. There are various ways to solve the expression problem, and we'll see one that works particularly well in Scala in Chapter 15.

Exercise: Sets

In this extended exercise we'll explore the `Set` interface we have already used in several examples, reproduced below.

```
trait Set[A] {

  /** True if this set contains the given element */
  def contains(elt: A): Boolean

  /** Construct a new set containing the given element */
  def insert(elt: A): Set[A]

  /** Construct the union of this and that set */
  def union(that: Set[A]): Set[A]
}
```

We also saw a simple implementation, storing the elements in the set in a `List`.

```
final class ListSet[A](elements: List[A]) extends Set[A] {

  def contains(elt: A): Boolean =
    elements.contains(elt)

  def insert(elt: A): Set[A] =
    ListSet(elt :: elements)

  def union(that: Set[A]): Set[A] =
    elements.foldLeft(that) { (set, elt) => set.insert(elt) }
}
object ListSet {
  def empty[A]: Set[A] = ListSet(List.empty)
}
```

The implementation for `union` is a bit unsatisfactory; it's doesn't use any of our strategies for writing code. We can implement both `union` and `insert` in a generic way that works for *all* sets (in other words, is implemented on the `Set` trait) and uses the strategies we've seen in this chapter. Go ahead and do this.

Your next challenge is to implement `Evens`, the set of all even integers, which we'll represent as a `Set[Int]`. This is an infinite set; we cannot directly enumerate all the elements in this set. (We actually could enumerate all the even elements that are 32-bit `Int`s, but we don't want to as this would use excessive amounts of space.)

We can generalize this idea to defining sets in terms of **indicator functions**, which is a function of type `A => Boolean`, returning returns true if the input belows to the set. Implement `IndicatorSet`, which is constructed with a single indicator function parameter.

4.6. Conclusions

In this chapter we've explored codata, the dual of data. Codata is defined by its interface—what we can do with it—as opposed to data, which is defined by what it is. More formally, codata is a product of destructors, where destructors are functions from the codata type (and, optionally, some other inputs) to some type. By avoiding the elements of object-oriented programming that make it hard to reason about—state and implementation inheritance—codata brings elements of object-oriented programming that accord with the other functional programming strategies. In Scala we define codata as a `trait`, and implement it as a `final class`, anonymous subclass, or an object.

We have two strategies for implementing methods using codata: structural corecursion, which we can use when the result is codata, and structural recursion, which we can use when an input is codata. Structural corecursion is usually the more useful of the two, as it gives more structure (pun intended) to the method we are implementing. The reverse is true for data.

We saw that data is connected to codata via fold: any data can instead be implemented as codata with a single destructor that is the fold for that data. The reverse is also: we can enumerate all potential pairs of inputs and outputs of destructors to represent codata as data. However this does not mean that data and codata are equivalent. We have seen many examples of codata representing infinite structures, such as sets of all even numbers and streams of all natural numbers. We have also seen that data and codata offer different forms of extensibility: data makes it easy to add new functions, but adding new elements requires changing existing code, while it is easy to add new elements to codata but we change existing code if we add new functions.

Codatatypes in ML [40] is the earliest reference to codata in programming languages that I could find. This is much more

recent than algebraic data, which I think explains why codata is relatively unknown. There are some excellent recent papers that deal with codata. I highly recommend *Codata in Action* [22], which inspired large portions of this chapter. *Exploring Codata: The Relation to Object-Orientation* [81] is also worthwhile. *How to add laziness to a strict language without even being odd* [90] is an older paper that discusses the implementation of streams, and in particular the difference between a not-quite-lazy-enough implementation they label odd and the version we saw, which they call even. These correspond to `Stream` and `LazyList` in the Scala standard library respectively. *Classical (Co)Recursion: Programming* [21] is an interesting survey of corecursion in different languages, and covers many of the same examples that I used here. Finally, if you really want to get into the weeds of the relationship between data and codata, *Beyond Church encoding: Boehm-Berarducci isomorphism of algebraic data types and polymorphic lambda-terms* [44] is for you.

5. Contextual Abstraction

All but the simplest programs depend on the **context** in which they run. The number of available CPU cores is an example of context provided by the computer. A program might adapt to this context by changing how work is distributed. Other forms of context include configuration read from files and environment variables, and (and we'll see at lot of this later) values created at compile-time, such as serialization formats, in response to the type of some method parameters.

Scala is one of the few languages that provides features for **contextual abstraction**, known as **implicits** in Scala 2 or **given instances** in Scala 3. In Scala these features are intimately related to types; types are used to select between different available given instances and drive construction of given instances at compile-time.

Most Scala programmers are less confident with the features for contextual abstraction than with other parts of the language, and they are often entirely novel to programmers coming from other languages. Hence this chapter will start by reviewing the abstractions formerly known as implicits: given instances and using clauses. We will then look at one of their major uses, **type classes**. Type classes allow us to extend existing types with new functionality, without using traditional inheritance, and without altering the original source code. Type classes are the core of Cats[23], which we will be exploring in the next part of this book.

[23] https://typelevel.org/cats/

5.1. The Mechanics of Contextual Abstraction

In section we'll go through the main Scala language features for contextual abstraction. Once we have a firm understanding of the mechanics of contextual abstraction we'll move on to their use.

The language features for contextual abstraction have changed name from Scala 2 to Scala 3, but they work in largely the same way. In the table below I show the Scala 3 features, and their Scala 2 equivalents. If you use Scala 2 you'll find that most of the code works simply by replacing `given` with `implicit val` and `using` with `implicit`.

Scala 3	Scala 2
given instance	implicit value
using clause	implicit parameter

Let's now explain how these language features work.

5.1.1. Using Clauses

We'll start with **using clauses**. A using clause is a method parameter list that starts with the `using` keyword. We use the term **context parameters** for the parameters in a using clause.

```
def double(using x: Int) = x + x
```

The `using` keyword applies to all parameters in the list, so in `add` below both `x` and `y` are context parameters.

```
def add(using x: Int, y: Int) = x + y
```

We can have normal parameter lists, and multiple using clauses, in the same method.

```
def addAll(x: Int)(using y: Int)(using z: Int): Int =
  x + y + z
```

We cannot pass parameters to a using clause in the normal way. We must proceed the parameters with the `using` keyword as shown below.

```
double(using 1)
// res0: Int = 2
add(using 1, 2)
// res1: Int = 3
addAll(1)(using 2)(using 3)
// res2: Int = 6
```

However this is not the typical way to pass parameters. In fact we don't usually explicitly pass parameters to using clause at all. We usually use given instances instead, so let's turn to them.

5.1.2. Given Instances

A given instance is a value that is defined with the `given` keyword. Here's a simple example.

```
given theMagicNumber: Int = 3
```

We can use a given instance like a normal value.

```
theMagicNumber * 2
// res3: Int = 6
```

However, it's more common to use them with a using clause. When we call a method that has a using clause, and we do not explicitly supply values for the context parameters, the compiler

will look for given instances of the required type. If it finds a given instance it will automatically use it to complete the method call.

For example, we defined `double` above with a single `Int` context parameter. The given instance we just defined, `theMagicNumber`, also has type `Int`. So if we call `double` without providing any value for the context parameter the compiler will provide the value `theMagicNumber` for us.

```
double
// res4: Int = 6
```

The same given instance will be used for multiple parameters with the same type in a using clause, as in `add` defined above.

```
add
// res5: Int = 6
```

The above are the most important points for using clauses and given instances. We'll now turn to some of the details of their semantics.

5.1.3. Given Scope and Imports

Given instances are usually not explicitly passed to using clauses. Their whole reason for existence is to get the compiler to do this for us. This could make code hard to understand, so we need to be very clear about which given instances are candidates to be supplied to a using clause. In this section we'll look at the **given scope**, which is all the places that the compiler will look for given instances, and the special syntax for importing given instances.

The first rule we should know about the given scope is that it starts at the **call site**, where the method with a using clause is called, not at the **definition site** where the method is defined. This means the following code does not compile, because the given

instance is not in scope at the call site, even though it is in scope at the definition site.

```
object A {
  given a: Int = 1
  def whichInt(using int: Int): Int = int
}

A.whichInt
// error:
// No given instance of type Int was found for parameter int of
method whichInt in object A
// A.whichInt
//   ^^^^^^^^
```

The second rule, which we have been relying on in all our examples so far, is that the given scope includes the **lexical scope** at the call site. The lexical scope is where we usually look up the values associated with names (like the names of method parameters or `val` declarations). This means the following code works, as `a` is defined in a scope that includes the call site.

```
object A {
  given a: Int = 1

  object B {
    def whichInt(using int: Int): Int = int
  }

  object C {
    B.whichInt
  }
}
```

However, if there are multiple given instances in the same scope the compiler will not arbitrarily choose one. Instead it fails with an error telling us the choice is ambiguous.

```
object A {
  given a: Int = 1
  given b: Int = 2
```

```
  def whichInt(using int: Int): Int = int

  whichInt
}
// error:
// Ambiguous given instances: both given instance a in object A
and
// given instance b in object A match type Int of parameter int
of
// method whichInt in object A
```

We can import given instances from other scopes, just like we can import normal declarations, but we must explicitly say we want to import given instances. The following code does not work because we have not explicitly imported the given instances.

```
object A {
  given a: Int = 1

  def whichInt(using int: Int): Int = int
}
object B {
  import A.*

  whichInt
}
// error:
// No given instance of type Int was found for parameter int of
method whichInt in object A
//
// Note: given instance a in object A was not considered because
it was not imported with `import given`.
//   whichInt
//             ^
```

It works when we do explicitly import them using `import A.given`.

```
object A {
  given a: Int = 1

  def whichInt(using int: Int): Int = int
```

```
}
object B {
  import A.{given, *}

  whichInt
}
```

One final wrinkle: the given scope includes the companion objects of any type involved in the type of the using clause. This is best illustrated with an example. We'll start by defining a type `Sound` that represents the sound made by its type variable `A`, and a method `soundOf` to access that sound.

```
trait Sound[A] {
  def sound: String
}

def soundOf[A](using s: Sound[A]): String =
  s.sound
```

Now we'll define some given instances. Notice that they are defined on the relevant companion objects.

```
trait Cat
object Cat {
  given catSound: Sound[Cat] with {
    def sound: String = "meow"
  }
}

trait Dog
object Dog {
  given dogSound: Sound[Dog] with {
    def sound: String = "woof"
  }
}
```

When we call `soundOf` we don't have to explicitly bring the instances into scope. They are automatically in the given scope by virtue of being defined on the companion objects of the types we use (`Cat` and `Dog`). If we had defined these instances on the `Sound`

companion object they would also be in the given scope; when looking for a `Sound[A]` both the companion objects of `Sound` and `A` are in scope.

```
soundOf[Cat]
// res12: String = "meow"
soundOf[Dog]
// res13: String = "woof"
```

We should almost always be defining given instances on companion objects. This simple organization scheme means that users do not have to explicitly import them but can easily find the implementations if they wish to inspect them.

5.1.3.1. Given Instance Priority

Notice that given instance selection is based entirely on types. We don't even pass any values to `soundOf`! This means given instances are easiest to use when there is only one instance for each type. In this case we can just put the instances on a relevant companion object and everything works out.

However, this is not always possible (though it's often an indication of a bad design if it is not). For cases where we need multiple instances for a type, we can use the instance priority rules to select between them. We'll look at the three most important rules below.

The first rule is that explicitly passing an instance takes priority over everything else.

```
given a: Int = 1
def whichInt(using int: Int): Int = int
```

```
whichInt(using 2)
// res15: Int = 2
```

The second rule is that instances in the lexical scope take priority over instances in a companion object. Here we define an instance on the `Cat` companion object.

```
trait Sound[A] {
  def sound: String
}
trait Cat
object Cat {
  given catSound: Sound[Cat] with {
    def sound: String = "meow"
  }
}

def soundOf[A](using s: Sound[A]): String =
  s.sound
```

Now we define an instance in the lexical scope, and we see it is chosen in preference to the instance on the companion object.

```
given purr: Sound[Cat] with {
  def sound: String = "purr"
}

soundOf[Cat]
// res17: String = "purr"
```

The final rule is that instances in a closer lexical scope take preference over those further away.

```
{
  given growl: Sound[Cat] with {
    def sound: String = "growl"
  }

  {
    given mew: Sound[Cat] with {
      def sound: String = "mew"
    }

    soundOf[Cat]
  }
```

```
}
// res18: String = "mew"
```

We're now seen most of the details of the workings of given instances and using clauses. This is a craft level explanation, and it naturally leads to the question: where would use these tools? This is what we'll address next, where we look at type classes and their implementation in Scala.

5.2. Anatomy of a Type Class

Let's now look at how type classes are implemented. There are three important components to a type class: the type class itself, which defines an interface; type class instances, which implement the type class for particular types; and the methods that use type classes. The table below shows the language features that correspond to each component.

Type Class Concept	Language Feature
Type class	trait
Type class instance	given instance
Type class use	using clause

Let's see how this works in detail.

5.2.1. The Type Class

A type class is an interface or API that represents some functionality we want implemented. In Scala 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
enum Json {
  case JsObject(get: Map[String, Json])
  case JsString(get: String)
  case JsNumber(get: Double)
  case JsNull
}

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

`JsonWriter` is our type class in this example, with the `Json` algebraic data type providing supporting code. When we come to implement instances of `JsonWriter`, the type parameter `A` will be the concrete type of data we are writing.

5.2.2. Type Class Instances

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

In Scala we create type class instances by defining given instances implementing the type class.

```
object JsonWriterInstances {
  given stringWriter: JsonWriter[String] with {
    def write(value: String): Json =
      Json.JsString(value)
  }

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

  given JsonWriter[Person] with
    def write(value: Person): Json =
```

```
      Json.JsObject(Map(
        "name" -> Json.JsString(value.name),
        "email" -> Json.JsString(value.email)
      ))

  // etc...
}
```

In this example we define two type class instances of `JsonWriter`, one for `String` and one for `Person`. The definition for `String` uses the syntax we saw in the previous section. The definition for `Person` uses two bits of syntax that are new in Scala 3. Firstly, writing `given JsonWriter[Person]` creates an anonymous given instance. We declare just the type and don't need to name the instance. This is fine because we don't usually need to refer to given instances by name. The second bit of syntax is the use of `with` to implement a trait directly without having to write out `new JsonWriter[Person]` and so on.

In a real implementation we'd usually want to define the instances on a companion object: the instance for `String` on the `JsonWriter` companion object (because we cannot define it on the `String` companion object) and the instance for `Person` on the `Person` companion object. I haven't done this here because I would need to redeclare `JsonWriter`, as a type and its companion object must be declared at the same time.

5.2.3. Type Class Use

A type class use is any functionality that requires a type class instance to work. In Scala this means any method that accepts instances of the type class as part of a using clause.

We're going to look at two patterns of type class usage, which we call **interface objects** and **interface syntax**. You'll find these in Cats and other libraries.

5.2.3.1. Interface Objects

The simplest way of creating an interface that uses a type class is to place methods in an object.

```
object Json {
  def toJson[A](value: A)(using 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.{*, given}
```

```
Json.toJson(Person("Dave", "dave@example.com"))
// res1: Json = JsObject(
//   get = Map(
//     "name" -> JsString(get = "Dave"),
//     "email" -> JsString(get = "dave@example.com")
//   )
// )
```

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

5.2.3.2. Interface Syntax

We can alternatively use **extension methods** to extend existing types with interface methods[24]. This is sometimes called **syntax** for the type class, which is the term used by Cats. Scala 2 has an equivalent to extension methods known as **implicit classes**.

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

Here's an example defining an extension method to add a `toJson` method to any type for which we have a `JsonWriter` instance.

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

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

```
import JsonWriterInstances.given
import JsonSyntax.*
```

```
Person("Dave", "dave@example.com").toJson
// res2: Json = JsObject(
//   get = Map(
//     "name" -> JsString(get = "Dave"),
//     "email" -> JsString(get = "dave@example.com")
//   )
// )
```

Extension Methods on Traits

In Scala 3 we can define extension methods directly on a type class trait. Since we're defining `toJson` as just calling `write` on `JsonWriter`, we could instead define `toJson` directly on `JsonWriter` and avoid creating an separate extension method.

```
trait JsonWriter[A] {
  extension (value: A) def toJson: Json
}

object JsonWriter {
```

```
  given stringWriter: JsonWriter[String] with {
    extension (value: String)
      def toJson: Json = Json.JsString(value)
  }

  // etc...
}
```

We do *not* advocate this approach, because of a limitation in how Scala searches for extension methods. The following code fails because Scala only looks within the `String` companion object for extension methods, and consequently does not find the extension method on the instance in the `JsonWriter` companion object.

```
"A string".toJson
// error:
// value toJson is not a member of String
// "A string".toJson
// ^^^^^^^^^^^^^^^^
```

This means that users will have to explicitly import at least the instances for the built-in types (for which we cannot modify the companion objects).

```
import JsonWriter.given

"A string".toJson
// res5: Json = JsString(get = "A string")
```

For consistency we recommend separating the syntax from the type class instances and always explicitly importing it, rather than requiring explicit imports for only some extension methods.

5.2.3.3. The `summon` Method

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

```
def summon[A](using value: A): A =
  value
```

We can use `summon` to summon any value in the given scope. We provide the type we want and `summon` does the rest:

```
summon[JsonWriter[String]]
// res6: stringWriter =
repl.MdocSession$MdocApp3$JsonWriter$stringWriter$@c022a05
```

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

5.3. Type Class Composition

So far we've seen type classes as a way to get the compiler to pass values to methods. This is nice but it does seem like we've introduced a lot of new concepts for a small gain. The real power of type classes lies in the compiler's ability to combine given instances to construct new given instances. This is known as **type class composition**.

Type class composition works by a feature of given instances we have not yet seen: given instances can themselves have context parameters. However, before we go into this let's see a motivational example.

Consider defining a `JsonWriter` for `Option`. We would need a `JsonWriter[Option[A]]` for every `A` we care about in our application. We could try to brute force the problem by creating a library of given instances:

```
given optionIntWriter: JsonWriter[Option[Int]] with {
  ???
}

given optionPersonWriter: JsonWriter[Option[Person]] with {
  ???
}

// and so on...
```

This approach clearly doesn't scale. We end up requiring two given instances for every type `A` in our application: one for `A` and one for `Option[A]`.

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

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

Here is the same code written out using a parameterized given instance:

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

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

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

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

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

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

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

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

5.3.1. Type Class Composition in Scala 2

In Scala 2 we can achieve the same effect with an `implicit` method with `implicit` parameters. Here's the Scala 2 equivalent of `optionWriter` above.

```
implicit def scala2OptionWriter[A]
    (implicit writer: JsonWriter[A]): JsonWriter[Option[A]] =
  new JsonWriter[Option[A]] {
    def write(option: Option[A]): Json =
      option match {
        case Some(aValue) => writer.write(aValue)
        case None         => JsNull
      }
  }
```

Make sure you make the method's parameter implicit! If you don't, you'll end up defining an **implicit conversion**. Implicit conversion is an older programming pattern that is frowned upon

in modern Scala code. Fortunately, the compiler will warn you should you do this.

5.4. What Type Classes Are

We've have now seen the mechanics of type classes: they are a specific arrangement of trait, given instances, and using clauses. This is a very craft-level explanation. Let's now raise the level of the explanation with three different views of type classes.

The first view goes back Chapter 4, where we looked at codata. The type class itself—the trait—is an example of codata with the usual advantages of codata (we can easily add implementations) and disadvantages (we cannot easily change the interface). Given instances and using clauses add the ability to chose the codata implementation based on the type of the context parameter and the instances in the given scope, and to compose instances from smaller components.

Raising the level of abstraction again, we can say that type classes allow us to implement functionality (the type class instance) separately from the type to which it applies, so that the implementation only needs to be defined at the point of the use—the call site—not at the point of declaration.

Raising the level again, we can say type classes allow us to implement **ad-hoc polymorphism**. I find it easiest to understand ad-hoc polymorphism in contrast to **parametric polymorphism**. Parametric polymorphism is what we get with type parameters, also known as generic types. It allows us to treat all types in a uniform way. For example, the following function calculates the length of any list of an arbitrary type `A`.

```
def length[A](list: List[A]): Int =
  list match {
```

```
    case Nil => 0
    case x :: xs => 1 + length(xs)
  }
```

We can implement `length` because we don't require any particular functionality from the values of type `A` that make up the elements of the list. We don't call any methods on them, and indeed we cannot call any methods on them because we don't know what concrete type `A` will be at the point where `length` is defined[25].

Ad-hoc polymorphism allows us to call methods on values with a generic type. The methods we can call are exactly those defined by the type class. For example, we can use the `Numeric` type class from the standard library to write a method that adds together elements of any type that implements that type class.

```
import scala.math.Numeric

def add[A](x: A, y: A)(using n: Numeric[A]): A = {
  n.plus(x, y)
}
```

So parametric polymorphism can be understood as meaning any type, while ad-hoc polymorphism means any type *that also implements this functionality*. In ad-hoc polymorphism there doesn't have to be any particular type relationship between the concrete types that implement the functionality of interest. This is in contast to object-oriented style polymorphism where all

[25]Parametric polymorphism represents an abstraction boundary. At the point of definition we don't know the concrete types that `A` will take; the concrete types are only known at the point of use. (Once again we see the distinction between definition site and call site.) This abstraction boundary allows a kind of reasoning known as **free theorems** [91]. For example, if we see a function with type `A => A` we know it must be the identity function. This is the only possible function with this type. Unfortunately the JVM allows us to break the abstraction boundary introduced by parametric polymorphism. We can call `equals`, `hashCode`, and a few other methods on all values, and we can inspect runtime tags that reflect some type information at run-time.

concrete types must be subtypes of the type that defines the functionality of interest.

5.5. Exercise: Display Library

Scala provides a `toString` method to let us convert any value to a `String`. This method comes with a few disadvantages:

1. It is implemented for *every* type in the language. There are situations where we don't want to be able to view data. For example, we may want to ensure we don't log sensitive information, such as passwords, in plain text.
2. We can't customize `toString` for types we don't control.

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

1. Define a type class `Display[A]` containing a single method `display`. `display` should accept a value of type `A` and return a `String`.
2. Create instances of `Display` for `String` and `Int` on the `Display` companion object.
3. On the `Display` companion object create two generic interface methods:
   - `display` accepts a value of type `A` and a `Display` of the corresponding type. It uses the relevant `Display` to convert the `A` to a `String`.
   - `print` accepts the same parameters as `display` and returns `Unit`. It prints the displayed `A` value to the console using `println`.

5.5.1. Using the Library

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

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

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

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

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

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

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

// Print the cat!
```

5.5.2. Better Syntax

Let's make our printing library easier to use by adding extension methods for its functionality:

1. Create an object `DisplaySyntax`.
2. Define `display` and `print` as extension methods on `DisplaySyntax`.
3. Use the extension methods to print the example `Cat` you created in the previous exercise.

5.6. Type Classes and Variance

In this section we'll discuss how **variance** interacts with type class instance selection. Variance is one of the darker corners of Scala's type system, so we start by reviewing it before moving on to its interaction with type classes.

5.6.1. Variance

Variance concerns the relationship between an instance defined on a type and its subtypes. For example, if we define a `JsonWriter[Option[Int]]`, will the expression `Json.toJson(Some(1))` select this instance? (Remember that `Some` is a subtype of `Option`).

We need two concepts to explain variance: type constructors, and subtyping.

Variance applies to any **type constructor**, which is the `F` in a type `F[A]`. So, for example, `List`, `Option`, and `JsonWriter` are all type constructors. A type constructor must have at least one type parameter, and may have more. So `Either`, with two type parameters, is also a type constructor.

Subtyping is a relationship between types. We say that `B` is a subtype of `A` if we can use a value of type `B` anywhere we expect a value of type `A`. We may sometimes use the shorthand `B <: A` to indicate that `B` is a subtype of `A`.

Variance concerns the subtyping relationship between types `F[A]` and `F[B]`, given a subtyping relationship between `A` and `B`. If `B` is a subtype of `A` then

1. if `F[B] <: F[A]` we say `F` is **covariant** in `A`; else
2. if `F[B] >: F[A]` we say `F` is **contravariant** in `A`; else

3. if there is no subtyping relationship between `F[B]` and `F[A]` we say `F` is **invariant** in `A`.

When we define a type constructor we can also add variance annotations to its type parameters. For example, we denote covariance with a + symbol:

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

Similarly, the - variance annotation indicate contravariance. If we don't add a variance annotation, the type parameter is invariant. Let's now look at covariance, contravariance, and invariance in detail.

5.6.2. Covariance

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

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

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

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

```
val circles: List[Circle] = List(Circle(5.0))
val shapes: List[Shape] = circles
```

Generally speaking, covariance is used for outputs: data that we can later get out of a container type such as `List`, or is otherwise returned by some method.

5.6.3. Contravariance

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

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

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

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

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

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

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

We also have a method `format` that expects a `JsonWriter` instance.

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

Now ask yourself the question: "Which combinations of value and writer can I pass to `format`?" We can `write` a `Circle` with either

writer because all `Circles` are `Shapes`. Conversely, we can't write a `Shape` with `circleWriter` because not all `Shapes` are `Circles`.

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

5.6.4. Invariance

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

```
trait F[A]
```

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

5.6.5. Variance and Instance Selection

When the compiler searches for a given instance 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:

```
enum A {
  case B
  case C
}
```

The issues are:

1. Will an instance defined on a supertype be selected if one is available? For example, can we define an instance for `A` and have it work for values of type `B` and `C`?
2. Will an instance for a subtype be selected in preference to that of a supertype. For instance, if we define an instance for `A` and `B`, and we have a value of type `B`, will the instance for `B` be selected in preference to `A`?

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

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

Let's see some examples, using the following types to show the subtyping relationship.

```
trait Animal
trait Cat extends Animal
trait DomesticShorthair extends Cat
```

Now we'll define three different type classes for the three types of variance, and define an instance of each for the `Cat` types.

```
trait Inv[A] {
  def result: String
}
object Inv {
  given Inv[Cat] with
    def result = "Invariant"

  def apply[A](using instance: Inv[A]): String =
    instance.result
```

```
}

trait Co[+A] {
  def result: String
}
object Co {
  given Co[Cat] with
    def result = "Covariant"

  def apply[A](using instance: Co[A]): String =
    instance.result
}

trait Contra[-A] {
  def result: String
}
object Contra {
  given Contra[Cat] with
    def result = "Contravariant"

  def apply[A](using instance: Contra[A]): String =
    instance.result
}
```

Now the cases that work, all of which select the `Cat` instance. For the invariant case we must ask for exactly the `Cat` type. For the covariant case we can ask for a supertype of `Cat`. For contravariance we can ask for a subtype of `Cat`.

```
Inv[Cat]
// res1: String = "Invariant"
Co[Animal]
// res2: String = "Covariant"
Co[Cat]
// res3: String = "Covariant"
Contra[DomesticShorthair]
// res4: String = "Contravariant"
Contra[Cat]
// res5: String = "Contravariant"
```

Now cases that fail. With invariance any type that is not `Cat` will fail. So the supertype fails

```
Inv[Animal]
// error:
// No given instance of type
MdocApp0.this.Inv[MdocApp0.this.Animal] was found for parameter
instance of method apply in object Inv
```

as does the subtype.

```
Inv[DomesticShorthair]
// error:
// No given instance of type
MdocApp0.this.Inv[MdocApp0.this.DomesticShorthair] was found for
parameter instance of method apply in object Inv
```

Covariance fails for any subtype of the type for which the instance is declared.

```
Co[DomesticShorthair]
// error:
// No given instance of type
MdocApp0.this.Co[MdocApp0.this.DomesticShorthair] was found for
parameter instance of method apply in object Co
```

Contravariance fails for any supertype of the type for which the instance is declared.

```
Contra[Animal]
// error:
// No given instance of type
MdocApp0.this.Contra[MdocApp0.this.Animal] was found for
parameter instance of method apply in object Contra
```

It's clear there is no perfect system. The most common choice is to use invariant type classes. This allows us to specify more specific instances for subtypes if we want. It does mean that if we have, for example, a value of type `Some[Int]`, our type class instance for `Option` will not be used. We can solve this problem with a type annotation like `Some(1) : Option[Int]` or by using "smart constructors" like `Option.apply` and `Option.empty` which always return a result of type `Option`.

5.7. Conclusions

In this chapter we took a first look at type classes. We saw the components that make up a type class:

- A `trait`, which is the type class
- Type class instances, which are given instances.
- Type class usage, which uses using clauses.

We saw that type classes can be composed from components using type class composition. This is one form of metaprogramming in Scala, where we can get the compiler to do work for us based on our program's types.

We can view type classes as marrying codata with tools to select and compose implementations based on type. We can also view type classes as shifting implementation from the definition site to the call site. Finally, can see type classes as a mechanism for ad-hoc polymorphism, allowing us to define common functionality for otherwise unrelated types.

Type classes were first described in *Parametric Overloading in Polymorphic Programming Languages* [42] and *How to make ad-hoc polymorphism less ad hoc* [88]. *Type Classes as Objects and Implicits* [65] details the encoding of type classes in Scala 2, and compares Scala's and Haskell's approach to type classes. Note that type classes are not restricted to Haskell and Scala. For examples, Rust's traits are essentially type classes.

As we have seen, Scala's support for type classes is based on implicit parameters (known as using clauses in Scala 3). Implicit parameters [51] were motivated by a desire to decompose type classes into smaller orthogonal language features, but they have been shown to be useful for other tasks. *Scala Implicits Are Everywhere: A Large-Scale Study of the Use of Scala Implicits in the Wild* [47] surveys different uses of implicits in Scala. There is a

particularly mind-bending example in *Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming* [64]. We'll see some of these different uses in later chapters.

Scala 3 has a few language features related to contextual abstraction that we haven't mentioned in this chapter. Context functions [62] allow functions to have using clauses. They are something the community is still exploring, and well defined use cases only beginning to emerge. Type class derivation[26] allows us to write code that generates type classes instances. Although this is extremely useful I think it's conceptually quite simple and doesn't warrant space in this book.

[26] https://docs.scala-lang.org/scala3/reference/contextual/derivation.html

6. Reified Interpreters

The interpreter strategy is perhaps the most important in all of functional programming. The central idea is to **separate description from action**. When we use the interpreter strategy our program consists of two parts: the description, instructions, or program that describes what we want to do, and the interpreter that carries the actions in the description. In this chapter we'll start exploring the design and implementation of interpreters, focusing on implementations using algebraic data types.

Interpreters arise whenever there is this distinction between description and action. You may think an interpreter is a complex piece requiring a lot of development effort, but I hope to show you this is not the case. You probably already use lots of interpreters in your daily coding without realizing it. For example, consider the code below which is taken from a web framework called Krop[27]

```
val route =
  Route(
    Request.get(Path.root / "user" / Param.int),
    Response.ok(Entity.text)
  ).handle(userId => s"You asked for the user
${userId.toString}")
```

This defines a route, which matches `GET` requests for the path `"/user/<int>"`, and responds with an `Ok` containing text. This kind of routing library is ubiquitous in web frameworks, is simple to write, and yet contains everything we need for the interpreter strategy.

Interpreters are so important because they are the key to enabling compositionality and reasoning, particularly while allowing effects. For example, imagine implementing a graphics library using the interpreter strategy. A program simply describes what

[27] https://github.com/creativescala/krop

we want to draw on the screen, but critically it does not draw anything. The interpreter takes this description and creates the drawing described by it. We can freely compose descriptions only because they do not carry out any effects. For example, if we have a description that describes a circle, and one for a square, we can compose them by saying we should draw the circle next to the square thereby creating a new description. If we immediately drew pictures there would be nothing to compose with. Similarly, it's easier to reason about pictures in this system because a program describes exactly what will appear on the screen, and there is no state from prior drawing that we need to worry about.

Throughout this chapter we will explore the interpreter strategy by building a series of interpreters for regular expressions. We've chosen to use regular expressions because they are already familiar to many and they are simple to work with. This means we can focus on the details of the interpreter strategy without getting caught up in problem specific details, but we still end up with a realistic and useful result.

We'll start with a basic implementation strategy that uses algebraic data types and structural recursion. We'll then look at transformations to turn our interpreter into a version that avoids using the stack and hence avoids the possibility of stack overflow.

6.1. Regular Expressions

We'll start this case study by briefly describing the usual task for regular expressions—matching text—and then take a more theoretical view. We'll then move on to implementation.

We most commonly use regular expressions to determine if a string matches a particular pattern. The simplest regular expression is one that matches only one string. In Scala we can create a regular expression by calling the `r` method on `String`.

Here's a regular expression that matches exactly the string `"Scala"`.

```
val regexp = "Scala".r
```

We can see that it matches only `"Scala"` and fails if we give it a shorter or longer input.

```
regexp.matches("Scala")
// res0: Boolean = true
regexp.matches("Sca")
// res1: Boolean = false
regexp.matches("Scalaland")
// res2: Boolean = false
```

Notice we already have a separation between description and action. The description is the regular expression itself, created by calling the `r` method, and the action is calling the `matches` method on the regular expression.

There are some characters that have a special meaning within the `String` describing a regular expression. For example, the character `*` matches the preceding character zero or more times.

```
val regexp = "Scala*".r
```

```
regexp.matches("Scal")
// res4: Boolean = true
regexp.matches("Scala")
// res5: Boolean = true
regexp.matches("Scalaaaa")
// res6: Boolean = true
```

We can also use parentheses to group sequences of characters. For example, if we wanted to match all the strings like `"Scala"`, `"Scalala"`, `"Scalalala"` and so on, we could use the following regular expression.

```
val regexp = "Scala(la)*".r
```

Let's check it matches what we're looking for.

```
regexp.matches("Scala")
// res8: Boolean = true
regexp.matches("Scalalalala")
// res9: Boolean = true
```

We should also check it fails to match as expected.

```
regexp.matches("Sca")
// res10: Boolean = false
regexp.matches("Scalal")
// res11: Boolean = false
regexp.matches("Scalaland")
// res12: Boolean = false
```

That's all I'm going to say about Scala's built-in regular expressions. If you'd like to learn more there are many resources online. The JDK documentation[28] is one example, which describes all the features available in the JVM implementation of regular expressions.

Let's turn to the theoretical description, such as we might find in a textbook. A regular expression is:

1. the empty regular expression that matches nothing;
2. a string, which matches exactly that string (including the empty string);
3. the concatenation of two regular expressions, which matches the first regular expression and then the second;
4. the union of two regular expressions, which matches if either expression matches; and
5. the repetition of a regular expression (often known as the Kleene star), which matches zero or more repetitions of the underlying expression.

[28] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/regex/Pattern.html

This kind of description may seem very abstract if you're not used to it. It is very useful for our purposes because it defines a minimal API that we can easily implement. Let's walk through the description and see how each part relates to code.

The empty regular expression is defining a constructor with type `() => Regexp`, which we can simplify to a value of type `Regexp`. In Scala we put constructors on the companion object, so this tells us we need

```
object Regexp {
  val empty: Regexp =
    ???
}
```

The second part tells us we need another constructor, this one with type `String => Regexp`.

```
object Regexp {
  val empty: Regexp =
    ???

  def apply(string: String): Regexp =
    ???
}
```

The other three components all take a regular expression and produce a regular expression. In Scala these will become methods on the `Regexp` type. Let's model this as a `trait` for now, and define these methods.

The first method, the concatenation of two regular expressions, is conventionally called ++ in Scala.

```
trait Regexp {
  def ++(that: Regexp): Regexp
}
```

Union is conventionally called `orElse`.

```
trait Regexp {
  def ++(that: Regexp): Regexp
  def orElse(that: Regexp): Regexp
}
```

Repetition we'll call `repeat`, and define an alias `*` that matches how this operation is written in conventional regular expression notation.

```
trait Regexp {
  def ++(that: Regexp): Regexp
  def orElse(that: Regexp): Regexp
  def repeat: Regexp
  def `*`: Regexp = this.repeat
}
```

We're missing one thing: a method to actually match our regular expression against some input. Let's call this method `matches`.

```
trait Regexp {
  def ++(that: Regexp): Regexp
  def orElse(that: Regexp): Regexp
  def repeat: Regexp
  def `*`: Regexp = this.repeat

  def matches(input: String): Boolean
}
```

This completes our API. Now we can turn to implementation. We're going to represent `Regexp` as an algebraic data type, and each method that returns a `Regexp` will return an instance of this algebraic data type. What should be the elements that make up the algebraic data type? There will be one element for each method, and the constructor arguments will be exactly the parameters passed to the method *including the hidden `this` parameter for methods on the trait.*

Here's the resulting code.

```
enum Regexp {
  def ++(that: Regexp): Regexp =
    Append(this, that)

  def orElse(that: Regexp): Regexp =
    OrElse(this, that)

  def repeat: Regexp =
    Repeat(this)

  def `*`: Regexp = this.repeat

  def matches(input: String): Boolean =
    ???

  case Append(left: Regexp, right: Regexp)
  case OrElse(first: Regexp, second: Regexp)
  case Repeat(source: Regexp)
  case Apply(string: String)
  case Empty
}
object Regexp {
  val empty: Regexp = Empty

  def apply(string: String): Regexp =
    Apply(string)
}
```

A quick note about `this`. We can think of every method on an object as accepting a hidden parameter that is the object itself. This is `this`. (If you have used Python, it makes this explicit as the `self` parameter.) As we consider `this` to be a parameter to a method call, and our implementation strategy is to capture all the method parameters in a data structure, we must make sure we capture `this` when it is available. The only case where we don't capture `this` is when we are defining a constructor on a companion object.

Notice that we haven't implemented `matches`. It doesn't return a `Regexp` so we cannot return an element of our algebraic data type. What should we do here? `Regexp` is an algebraic data type and `matches` transforms an algebraic data type into a `Boolean`.

Therefore we can use structural recursion! Let's write out the skeleton, including the recursion rule.

```
def matches(input: String): Boolean =
  this match {
    case Append(left, right) =>
      left.matches(???) ??? right.matches(???)
    case OrElse(first, second) =>
      first.matches(???) ??? second.matches(???)
    case Repeat(source) =>
      source.matches(???) ???
    case Apply(string) => ???
    case Empty => ???
  }
```

Now we can apply the usual strategies to complete the implementation. Let's reason independently by case, starting with the case for `Empty`. This case is trivial as it always fails to match, so we just return `false`.

```
def matches(input: String): Boolean =
  this match {
    case Append(left, right)   => left.matches(???) ???
right.matches(???)
    case OrElse(first, second) => first.matches(???) ???
second.matches(???)
    case Repeat(source)        => source.matches(???) ???
    case Apply(string)         => ???
    case Empty                 => false
  }
```

Let's move on to the `Append` case. This should match if the `left` regular expression matches the start of the `input`, and the `right` regular expression matches starting where the `left` regular expression stopped. This has uncovered a hidden requirement: we need to keep an index into the `input` that tells us where we should start matching from. Using a nested method is the easiest way to keep around additional information that we need. Here I've created a nested method that returns an `Option[Int]`. The `Int` is the new index to use, and we return an `Option` to indicate if the regular expression matched or not.

```
def matches(input: String): Boolean = {
  def loop(regexp: Regexp, idx: Int): Option[Int] =
    regexp match {
      case Append(left, right) =>
        loop(left, idx).flatMap(idx => loop(right, idx))
      case OrElse(first, second) =>
        loop(first, idx) ??? loop(second, ???)
      case Repeat(source) =>
        loop(source, idx) ???
      case Apply(string) =>
        ???
      case Empty =>
        None
    }

  // Check we matched the entire input
  loop(this, 0).map(idx => idx == input.size).getOrElse(false)
}
```

Now we can go ahead and complete the implementation.

```
def matches(input: String): Boolean = {
  def loop(regexp: Regexp, idx: Int): Option[Int] =
    regexp match {
      case Append(left, right) =>
        loop(left, idx).flatMap(i => loop(right, i))
      case OrElse(first, second) =>
        loop(first, idx).orElse(loop(second, idx))
      case Repeat(source) =>
        loop(source, idx)
          .flatMap(i => loop(regexp, i))
          .orElse(Some(idx))
      case Apply(string) =>
        Option.when(input.startsWith(string, idx))(idx +
string.size)
    }

  // Check we matched the entire input
  loop(this, 0).map(idx => idx == input.size).getOrElse(false)
}
```

The implementation for `Repeat` is a little tricky, so I'll walk through the code.

```
case Repeat(source) =>
  loop(source, idx)
    .flatMap(i => loop(regexp, i))
    .orElse(Some(idx))
```

The first line (`loop(source, index)`) is seeing if the `source` regular expression matches. If it does we loop again, but on `regexp` (which is `Repeat(source)`), not `source`. This is because we want to repeat an indefinite number of times. If we looped on `source` we would only try twice. Remember that failing to match is still a success; repeat matches zero or more times. This condition is handled by the `orElse` clause.

We should test that our implementation works.

Here's the example regular expression we started the chapter with.

```
val regexp = Regexp("Sca") ++ Regexp("la") ++ Regexp("la").repeat
```

Here are cases that should succeed.

```
regexp.matches("Scala")
// res14: Boolean = true
regexp.matches("Scalalalala")
// res15: Boolean = true
```

Here are cases that should fail.

```
regexp.matches("Sca")
// res16: Boolean = false
regexp.matches("Scalal")
// res17: Boolean = false
regexp.matches("Scalaland")
// res18: Boolean = false
```

Success! At this point we could add many extensions to our library. For example, regular expressions usually have a method (by convention denoted +) that matches one or more times, and one that matches zero or once (usually denoted ?). These are both

conveniences we can build on our existing API. However, our goal at the moment is to fully understand interpreters and the implementation technique we've used here. So in the next section we'll discuss these in detail.

Regular Expression Semantics

Our regular expression implementation handles union differently to Scala's built-in regular expressions. Look at the following example comparing the two.

```
val r1 = "(z|zxy)ab".r
val r2 = Regexp("z").orElse(Regexp("zxy")) ++ Regexp("ab")
```

```
r1.matches("zxyab")
// res19: Boolean = true
r2.matches("zxyab")
// res20: Boolean = false
```

The reason for this difference is that our implementation commits to the first branch in a union that successfully matches some of the input, regardless of how that affects later matching. We should instead try both branches, but doing so makes the implementation more complex. The semantics of regular expressions are not essential to what we're trying to do here; we're just using them as an example to motivate the programming strategies we're learning. I decided the extra complexity of implementing union in the usual way outweighed the benefits, and so kept the simpler implementation. Don't worry, we'll see how to do it properly in Chapter 16!

6.2. Interpreters and Reification

There are two different programming strategies at play in the regular expression code we've just written:

1. the interpreter strategy; and
2. the interpreter's implementation strategy of reification.

Remember the essence of the **interpreter strategy** is to separate description and action. Therefore, whenever we use the interpreter strategy we need at least two things: a description and an interpreter. Descriptions are programs; things that we want to happen. The interpreter runs the programs, carrying out the actions described within them.

In the regular expression example, a `Regexp` value is a program. It is a description of a pattern we are looking for within a `String`. The `matches` method is an interpreter. It carries out the instructions in the description, checking the pattern matches the entire input. We could have other interpreters, such as one that matches if at least some part of the input matches the pattern.

6.2.1. The Structure of Interpreters

All uses of the interpreter strategy have a particular structure to their methods. There are three different kinds of methods:

1. **constructors**, or **introduction forms**, with type `A => Program`. Here `A` is any type that isn't a program, and `Program` is the type of programs. Constructors conventionally live on the `Program` companion object in Scala. We see that `apply` is a constructor of `Regexp`. It has type `String => Regexp`, which matches the pattern `A => Program` for a constructor. The other constructor, `empty`, is just a value of type `Regexp`. This is

equivalent to a method with type `() => Regexp` and so it also matches the pattern for a constructor.

2. **combinators** have at least one program input and a program output. The type is similar to `Program => Program` but there are often additional parameters. All of `++`, `orElse`, and `repeat` are combinators in our regular expression example. They all have a `Regexp` input (the `this` parameter) and produce a `Regexp`. Some of them have additional parameters, such as `++` or `orElse`. For both these methods the single additional parameter is a `Regexp`, but it is not the case that additional parameters to a combinator must be of the program type. Conventionally these methods live on the `Program` type.
3. **destructors**, **interpreters**, or **elimination forms**, have type `Program => A`. In our regular expression example we have a single interpreter, `matches`, but we could easily add more. For example, we often want to extract elements from the input or find a match at any location in the input.

This structure is often called an **algebra** or **combinator library** in the functional programming world. When we talk about constructors and destructors in an algebra we're talking at a more abstract level then when we talk about constructors and destructors on algebraic data types. A constructor of an algebra is an abstract concept, at the theory level in my taxonomy, that we can choose to concretely implement at the craft level with the constructor of an algebraic data type. There are other possible implementations. We'll see one later.

6.2.2. Implementing Interpreters with Reification

Now that we understand the components of an interpreter we can talk more clearly about the implementation strategy we used. We

used a strategy called **reification**, **defunctionalization**, **deep embedding**, or an **initial algebra**.

Reification, in an abstract sense, means to make concrete what is abstract. Concretely, reification in the programming sense means to turn methods or functions into data. When using reification in the interpreter strategy we reify all the components that produce the `Program` type. This means reifying constructors and combinators.

Here are the rules for reification:

1. We define some type, which we'll call `Program`, to represent programs.
2. We implement `Program` as an algebraic data type.
3. All constructors and combinators become product types within the `Program` algebraic data type.
4. Each product type holds exactly the parameters to the constructor or combinator, including the `this` parameter for combinators.

Once we've defined the `Program` algebraic data type, the interpreter becomes a structural recursion on `Program`.

Exercise: Arithmetic

Now it's your turn to practice using reification. Your task is to implement an interpreter for arithmetic expressions. An expression is:

- a literal number, which takes a `Double` and produces an `Expression`;
- an addition of two expressions;
- a substraction of two expressions;
- a multiplication of two expressions; or
- a division of two expressions;

Reify this description as a type `Expression`.

Now implement an interpreter `eval` that produces a `Double`. This interpreter should interpret the expression using the usual rules of arithmetic.

Add methods `+`, `-` and so on that make your system a bit nicer to use. Then write some expressions and show that it works as expected.

6.3. Tail Recursive Interpreters

Structural recursion, as we have written it, uses the stack. This is not often a problem, but particularly deep recursions can lead to the stack running out of space. A solution is to write a **tail recursive** program. A tail recursive program does not need to use any stack space, and so is sometimes known as **stack safe**. Any program can be turned into a tail recursive version, which does not use the stack and therefore cannot run out of stack space.

The Call Stack

Method and function calls are usually implemented using an area of memory known as the call stack, or just the stack for short. Every method or function call uses a small amount of memory on the stack, called a stack frame. When the method or function returns, this memory is freed and becomes available for future calls to use.

A large number of method calls, without corresponding returns, can require more stack frames than the stack can accommodate. When there is no more memory available on the stack we say we have overflowed the stack. In Scala a `StackOverflowError` is raised when this happens.

In this section we will discuss tail recursion, converting programs to tail recursive form, and limitations and workarounds for the Scala's runtimes.

6.3.1. The Problem of Stack Safety

Let's start by seeing the problem. In Scala we can create a repeated `String` using the * method.

```
"a" * 4
// res0: String = "aaaa"
```

We can match such a `String` with a regular expression and `repeat`.

```
Regexp("a").repeat.matches("a" * 4)
// res1: Boolean = true
```

However, if we make the input very long the interpreter will fail with a stack overflow exception.

```
Regexp("a").repeat.matches("a" * 20000)
// java.lang.StackOverflowError
```

This is because the interpreter calls `loop` for each instance of a repeat, without returning. However, all is not lost. We can rewrite the interpreter in a way that consumes a fixed amount of stack space, and therefore match input that is as large as we like.

6.3.2. Tail Calls and Tail Position

Our starting point is **tail calls**. A tail call is a method call that does not take any additional stack space. Only method calls that are in **tail position** are candidates to be turned into tail calls. Even then,

runtime limitations mean that not all calls in tail position will be converted to tail calls.

A method call in tail position is a call that immediately returns the value returned by the call. Let's see an example. Below are two versions of a method to calculate the sum of the integers from 0 to `count`.

```
def isntTailRecursive(count: Int): Int =
  count match {
    case 0 => 0
    case n => n + isntTailRecursive(n - 1)
  }

def isTailRecursive(count: Int): Int = {
  def loop(count: Int, accum: Int): Int =
    count match {
      case 0 => accum
      case n => loop(n - 1, accum + n)
    }

  loop(count, 0)
}
```

The method call to `isntTailRecursive` in

```
case n => n + isntTailRecursive(n - 1)
```

is not in tail position, because the value returned by the call is then used in the addition. However, the call to `loop` in

```
case n => loop(n - 1, accum + n)
```

is in tail position because the value returned by the call to `loop` is itself immediately returned. Similarly, the call to `loop` in

```
loop(count, 0)
```

is also in tail position.

A method call in tail position is a candidate to be turned into a tail call. Limitations of Scala's runtimes mean that not all calls in tail position can be made tail calls. Currently, only calls from a method to itself that are also in tail position will be converted to tail calls. This means

```
case n => loop(n - 1, accum + n)
```

is converted to a tail call, because `loop` is calling itself. However, the call

```
loop(count, 0)
```

is not converted to a tail call, because the call is from `isTailRecursive` to `loop`. This will not cause issues with stack consumption, however, because this call only happens once.

Runtimes and Tail Calls

Scala supports three different platforms: the JVM, Javascript via Scala.js, and native code via Scala Native. Each platform provides what is known as a runtime, which is code that supports our Scala code when it is running. The garbage collector, for example, is part of the runtime.

At the time of writing none of Scala's runtimes support full tail calls. However, there is reason to think this may change in the future. Project Loom[29] should eventually add support for tail calls to the JVM. Scala Native is likely to support tail calls soon, as part of other work to implement continuations. Tail calls have been part of the Javascript specification for a long time, but remain unimplemented by

[29] https://wiki.openjdk.org/display/loom/Main

> the majority of Javascript runtimes. However, WebAssembly does support tail calls and will probably replace compiling Scala to Javascript in the medium term.

We can ask the Scala compiler to check that all self calls are in tail position by adding the `@tailrec` annotation to a method. The code will fail to compile if any calls from the method to itself are not in tail position.

```
import scala.annotation.tailrec

@tailrec
def isntTailRecursive(count: Int): Int =
  count match {
    case 0 => 0
    case n => n + isntTailRecursive(n - 1)
  }
// error:
// Cannot rewrite recursive call: it is not in tail position
//     case n => n + isntTailRecursive(n - 1)
//                   ^^^^^^^^^^^^^^^^^^^^^^^^
```

We can check the tail recursive version is truly tail recursive by passing it a very large input. The non-tail recursive version crashes.

```
isntTailRecursive(100000)
// java.lang.StackOverflowError
```

The tail recursive version runs just fine.

```
isTailRecursive(100000)
// res4: Int = 705082704
```

6.3.3. Continuation-Passing Style

Now that we know about tail calls, how do we convert the regular expression interpreter to use them? Any program can be converted to an equivalent program with all calls in tail position. This conversion is known as **continuation-passing style** or CPS for short. Our first step to understanding CPS is to understand **continuations**.

A continuation is an encapsulation of "what happens next". Let's return to our `Regexp` example. Here's the full code for reference.

```
enum Regexp {
  def ++(that: Regexp): Regexp =
    Append(this, that)

  def orElse(that: Regexp): Regexp =
    OrElse(this, that)

  def repeat: Regexp =
    Repeat(this)

  def `*` : Regexp = this.repeat

  def matches(input: String): Boolean = {
    def loop(regexp: Regexp, idx: Int): Option[Int] =
      regexp match {
        case Append(left, right) =>
          loop(left, idx).flatMap(i => loop(right, i))
        case OrElse(first, second) =>
          loop(first, idx).orElse(loop(second, idx))
        case Repeat(source) =>
          loop(source, idx)
            .flatMap(i => loop(regexp, i))
            .orElse(Some(idx))
        case Apply(string) =>
          Option.when(input.startsWith(string, idx))(idx +
string.size)
        case Empty =>
          None
      }

    // Check we matched the entire input
```

```
    loop(this, 0).map(idx => idx == input.size).getOrElse(false)
  }

  case Append(left: Regexp, right: Regexp)
  case OrElse(first: Regexp, second: Regexp)
  case Repeat(source: Regexp)
  case Apply(string: String)
  case Empty
}
object Regexp {
  val empty: Regexp = Empty

  def apply(string: String): Regexp =
    Apply(string)
}
```

Let's consider the case for `Append` in `matches`.

```
case Append(left, right) =>
  loop(left, idx).flatMap(i => loop(right, i))
```

What happens next when we call `loop(left, idx)`? Let's give the name `result` to the value returned by the call to `loop`. The answer is we run `result.flatMap(i => loop(right, i))`. We can represent this as a function, to which we pass `result`:

```
(result: Option[Int]) => result.flatMap(i => loop(right, i))
```

This is exactly the continuation, reified as a value.

As is often the case, there is a distinction between the concept and the representation. The concept of continuations always exists in code. A continuation means "what happens next". In other words, it is the program's control flow. There is always some concept of control flow, even if it is just "the program halts". We can represent continuations as functions in code. This transforms the abstract concept of continuations into concrete values in our program, and hence reifies them.

Now that we know about continuations, and their reification as functions, we can move on to continuation-passing style. In CPS we, as the name suggests, pass around continuations. Specifically, each function or method takes an extra parameter that is a continuation. Instead of returning a value it calls that continuation with the value. This is another example of duality, in this case between returning a value and calling a continuation.

Let's see how this works. We'll start with a simple example written in the normal style, also known as **direct style**.

```
(1 + 2) * 3
// res5: Int = 9
```

To rewrite this in CPS style we need to create replacements for + and * with the extra continuation parameter.

```
type Continuation = Int => Int

def add(x: Int, y: Int, k: Continuation) = k(x + y)
def mul(x: Int, y: Int, k: Continuation) = k(x * y)
```

Now we can rewrite our example in CPS. `(1 + 2)` becomes `add(1, 2, k)`, but what is `k`, the continuation? What we do next is multiply the result by `3`. Thus the continuation is `a => mul(a, 3, k2)`. What is the next continuation, `k2`? Here the program finishes, so we just return the value with the identity continuation `b => b`. Put it all together and we get

```
add(1, 2, a => mul(a, 3, b => b))
// res6: Int = 9
```

Notice that every continuation call is in tail position in the CPS code. This means that code written in CPS can potentially consume no stack space.

Now we can return to the interpreter loop for `Regexp`. We are going to CPS it, so we need to add an extra parameter for the

continuation. In this case the contination accepts and returns the result type of `loop`: `Option[Int]`.

```
def matches(input: String): Boolean = {
  // Define a type alias so we can easily write continuations
  type Continuation = Option[Int] => Option[Int]

  def loop(regexp: Regexp, idx: Int, cont: Continuation):
Option[Int] =
  // etc...
}
```

Now we go through each case and convert it to CPS. Each continuation we construct must call `cont` as its final step. This is tedious and a bit error-prone, so good tests are helpful.

```
def matches(input: String): Boolean = {
  // Define a type alias so we can easily write continuations
  type Continuation = Option[Int] => Option[Int]

  def loop(
      regexp: Regexp,
      idx: Int,
      cont: Continuation
  ): Option[Int] =
    regexp match {
      case Append(left, right) =>
        val k: Continuation = _ match {
          case None    => cont(None)
          case Some(i) => loop(right, i, cont)
        }
        loop(left, idx, k)

      case OrElse(first, second) =>
        val k: Continuation = _ match {
          case None => loop(second, idx, cont)
          case some => cont(some)
        }
        loop(first, idx, k)

      case Repeat(source) =>
        val k: Continuation =
          _ match {
            case None    => cont(Some(idx))
            case Some(i) => loop(regexp, i, cont)
```

```
          }
        loop(source, idx, k)

      case Apply(string) =>
        cont(Option.when(input.startsWith(string, idx))(idx +
string.size))

      case Empty =>
        cont(None)
    }

  // Check we matched the entire input
  loop(this, 0, identity).map(idx => idx ==
input.size).getOrElse(false)
}
```

Every call in this interpreter loop is in tail position. However Scala cannot convert these to tail calls because the calls go from `loop` to a continuation and vice versa. To make the interpreter fully stack safe we need to add **trampolining**.

Exercise: CPS Arithmetic

In a previous exercise we wrote an interpreter for arithmetic expressions. Your task now is to CPS this interpreter. For reference, the definition of an arithmetic expression is:

- a literal number, which takes a `Double` and produces an `Expression`;
- an addition of two expressions;
- a substraction of two expressions;
- a multiplication of two expressions; or
- a division of two expressions;

6.3.4. Trampolining

Earlier we said that CPS utilizes the duality between function calls and returns: instead of returning a value we call a function with a

value. This allows us to transform our code so it only has calls in tail positions. However, we still have a problem with stack safety. Scala's runtimes don't support full tail calls, so calls from a continuation to `loop` or from `loop` to a continuation will use a stack frame. We can use this same duality to avoid using the stack by, instead of making a call, returning a value that reifies the call we want to make. This idea is the core of trampolining. Let's see it in action, which will help clear up what exactly this all means.

Our first step is to reify all the method calls made by the interpreter loop and the continuations. There are three cases: calls to `loop`, calls to a continuation, and, to avoid an infinite loop, the case when we're done.

```
type Continuation = Option[Int] => Call

enum Call {
  case Loop(regexp: Regexp, index: Int, continuation:
Continuation)
  case Continue(index: Option[Int], continuation: Continuation)
  case Done(index: Option[Int])
}
```

Now we update `loop` to return instances of `Call` instead of making the calls directly.

```
def loop(regexp: Regexp, idx: Int, cont: Continuation): Call =
  regexp match {
    case Append(left, right) =>
      val k: Continuation = _ match {
        case None    => Call.Continue(None, cont)
        case Some(i) => Call.Loop(right, i, cont)
      }
      Call.Loop(left, idx, k)

    case OrElse(first, second) =>
      val k: Continuation = _ match {
        case None => Call.Loop(second, idx, cont)
        case some => Call.Continue(some, cont)
      }
      Call.Loop(first, idx, k)
```

```
    case Repeat(source) =>
      val k: Continuation =
        _ match {
          case None    => Call.Continue(Some(idx), cont)
          case Some(i) => Call.Loop(regexp, i, cont)
        }
      Call.Loop(source, idx, k)

    case Apply(string) =>
      Call.Continue(
        Option.when(input.startsWith(string, idx))(idx +
string.size),
        cont
      )

    case Empty =>
      Call.Continue(None, cont)
  }
```

This gives us an interpreter loop that returns values instead of making calls, and so does not consume stack space. However, we need to actually make these calls at some point, and doing this is the job of the trampoline. The trampoline is simply a tail recursive loop that makes calls until it reaches `Done`.

```
def trampoline(next: Call): Option[Int] =
  next match {
    case Call.Loop(regexp, index, continuation) =>
      trampoline(loop(regexp, index, continuation))
    case Call.Continue(index, continuation) =>
      trampoline(continuation(index))
    case Call.Done(index) => index
  }
```

Now every call has a corresponding return, so the stack usage is limited. Our interpreter can handle input of any size, up to the limits of available memory.

Here's the complete code for reference.

```
// Define a type alias so we can easily write continuations
type Continuation = Option[Int] => Call

enum Call {
  case Loop(regexp: Regexp, index: Int, continuation:
Continuation)
  case Continue(index: Option[Int], continuation: Continuation)
  case Done(index: Option[Int])
}

enum Regexp {
  def ++(that: Regexp): Regexp =
    Append(this, that)

  def orElse(that: Regexp): Regexp =
    OrElse(this, that)

  def repeat: Regexp =
    Repeat(this)

  def `*` : Regexp = this.repeat

  def matches(input: String): Boolean = {
    def loop(regexp: Regexp, idx: Int, cont: Continuation): Call
=
      regexp match {
        case Append(left, right) =>
          val k: Continuation = _ match {
            case None    => Call.Continue(None, cont)
            case Some(i) => Call.Loop(right, i, cont)
          }
          Call.Loop(left, idx, k)

        case OrElse(first, second) =>
          val k: Continuation = _ match {
            case None => Call.Loop(second, idx, cont)
            case some => Call.Continue(some, cont)
          }
          Call.Loop(first, idx, k)

        case Repeat(source) =>
          val k: Continuation =
            _ match {
              case None    => Call.Continue(Some(idx), cont)
              case Some(i) => Call.Loop(regexp, i, cont)
            }
          Call.Loop(source, idx, k)
```

```
        case Apply(string) =>
          Call.Continue(
            Option.when(input.startsWith(string, idx))(idx +
string.size),
            cont
          )

        case Empty =>
          Call.Continue(None, cont)
      }

    def trampoline(next: Call): Option[Int] =
      next match {
        case Call.Loop(regexp, index, continuation) =>
          trampoline(loop(regexp, index, continuation))
        case Call.Continue(index, continuation) =>
          trampoline(continuation(index))
        case Call.Done(index) => index
      }

    // Check we matched the entire input
    trampoline(loop(this, 0, opt => Call.Done(opt)))
      .map(idx => idx == input.size)
      .getOrElse(false)
  }

  case Append(left: Regexp, right: Regexp)
  case OrElse(first: Regexp, second: Regexp)
  case Repeat(source: Regexp)
  case Apply(string: String)
  case Empty
}
object Regexp {
  val empty: Regexp = Empty

  def apply(string: String): Regexp =
    Apply(string)
}
```

Exercise: Trampolined Arithmetic

Convert the CPSed arithmetic interpreter we wrote earlier to a trampolined version.

6.3.5. When Tail Recursion is Easy

Doing a full CPS conversion and trampoline can be quite involved. Some methods can be made tail recursive without so large a change. Remember these examples we looked at earlier?

```
def isntTailRecursive(count: Int): Int =
  count match {
    case 0 => 0
    case n => n + isntTailRecursive(n - 1)
  }

def isTailRecursive(count: Int): Int = {
  def loop(count: Int, accum: Int): Int =
    count match {
      case 0 => accum
      case n => loop(n - 1, accum + n)
    }

  loop(count, 0)
}
```

The tail recursive version doesn't seem to involve the complexity of CPS. How can we relate this to what we've just learned, and when can we avoid the work of CPS and trampolining?

Let's use substitution to show how the stack is used by each method, for a small value of `count`.

```
isntTailRecursive(2)
// expands to
(2 match {
  case 0 => 0
  case n => n + isntTailRecursive(n - 1)
})
// expands to
(2 + isntTailRecursive(1))
// expands to
(2 + (1 match {
        case 0 => 0
        case n => n + isntTailRecursive(n - 1)
      }))
// expands to
```

```
(2 + (1 + isntTailRecursive(n - 1)))
// expands to
(2 + (1 + (0 match {
             case 0 => 0
             case n => n + isntTailRecursive(n - 1)
           })))
// expands to
(2 + (1 + (0)))
// expands to
3
```

Here each set of brackets indicates a new method call and hence a stack frame allocation.

Now let's do the same for `isTailRecursive`.

```
isTailRecursive(2)
// expands to
(loop(2, 0))
// expands to
(2 match {
   case 0 => 0
   case n => loop(n - 1, 0 + n)
 })
// expands to
(loop(1, 2))
// call to loop is a tail call, so no stack frame is allocated
// expands to
(1 match {
   case 0 => 2
   case n => loop(n - 1, 2 + n)
 })
// expands to
(loop(0, 3))
// call to loop is a tail call, so no stack frame is allocated
// expands to
(0 match {
   case 0 => 3
   case n => loop(n - 1, 3 + n)
 })
// expands to
(3)
// expands to
3
```

The non-tail recursive function computes the result `(2 + (1 + (0)))` If we look closely, we'll see that the tail recursive version computes `(((2) + 1) + 0)`, which simply accumulates the result in the reverse order. This works because addition is associative, meaning `(a + b) + c == a + (b + c)`. This is our first criteria for using the "easy" method for converting to a tail recursive form: the operation that accumulates results must be associative.

This doesn't explain, though, how we come to realize that addition is the correct operation to use. The second criteria is that we don't need any memory beyond the partial result calculated from the data we've already seen. Some implications of this are that we can stop at any time and have a usable result, and that we are only applying a single operation to the data. This is not the case in the regular expression example. For example, we have the following code in the `Append` case:

```
case Append(left, right) =>
  loop(left, idx).flatMap(i => loop(right, i))
```

To compute the result for the `Append` we need to compute and combine results from both `left` and `right`. So when we have computed the result for `right` we need to remember both the result from `left` and that we're combining the two results using the rule for `Append` rather than, say, `OrElse`. It's remembering this that is exactly what the continuation does, and what stops us from using the easy method we saw when summing the elements of a list.

So, in summary, if we are applying only a single associative operation to data we can use the simple method for writing a tail recursive method:

1. define an structurally recursive loop with an additional parameter that is the partial result or accumulator;
2. in the base cases return the accumulator; and

3. in the recursive cases update the accumulator and call the loop in tail position.

You might be wondering how we handle tree-shaped data with this technique. One consequence of an associative operation is that we can transform any sequence of operations into a list-shaped sequence. If, for example, we have an expression tree that suggests we should call operations in the order `(1 + 2) + (3 + 4)` (where I'm using `+` to indicate the operation) we can rewrite that to `(((1 + 2) + 3) + 4)` via associativity. So we can transform our tree into a list and then apply the recipe above.

6.4. Conclusions

In this chapter we've discussed why we might want to build interpreters, and seen techniques for building them. To recap, the core of the interpreter strategy is a separation between description and action. The description is the program, and the interpreter is the action that carries out the program. This separation is allows for composition of programs, and managing effects by delaying them till the time the program is run. We sometimes call this structure an algebra, with constructs and combinators defining programs and destructors defining interpreters. Although the name of the strategy focuses on the interpreter, the design of the program is just as important as it is the user interface through which the programmer interacts with the system.

Our starting implementation strategy is reification of the algebra's constructors and compositional methods as an algebraic data type. The interpreter is then a structural recursion over this ADT. We saw that the straightforward implementation is not stack-safe, and which caused us to introduction the idea of tail recursion and continuations. We reified continuations as functions, and saw that we can convert any program into continuation-passing style

which has every method call in tail position. Due to Scala runtime limitations not all calls in tail position can be converted to tail calls, so we reified calls and returns into data structures used by a recursive loop called a trampoline. Underlying all these strategies in the concept of duality. We have seen a duality between functions and data, which we utilize in reification, and a duality between calling functions and returning data, which we use in continuations and trampolines.

Stack-safe interpreters are important in many situations, but the code is harder to read than the basic structural recursion. In some contexts a basic interpreter may be just fine. It's unlikely to run out of stack space when evaluating a straightforward expression tree, as in the arithmetic example. The depth of such a tree grows logarithmically with the number of elements, so only extremely large trees will have sufficient depth that stack safety becomes relevant. However, in the regular expression example the stack consumption is determined not by the depth of the regular expression tree, but by the length of the input being matched. In this situation stack safety is more important. There may still be other constraints that allow a simpler implementation. For example, if we know the library will only used in situations where inputs were guaranteed to be small. As always, only use coding techniques where they make sense.

These ideas are classics in programming language theory. *Definitional Interpreters for Higher-Order Programming Languages* [73] details defunctionalization, a limited form of reification and continuation-passing style. (If you want to read this paper, I suggest the re-typeset version from 1998[30], which is much more readable than the original typewriter version.) These ideas are expanded on in *Defunctionalization at Work* [17]. *Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity*

[30] https://homepages.inf.ed.ac.uk/wadler/papers/papers-we-love/reynolds-definitional-interpreters-1998.pdf

[36] is a very readable and elegant paper that highlights the importance of associativity in these transformations.

Part II: Type Classes

In this part of the book we move on to type classes. We looked at the implementation of type classes in Chapter 5. Our focus here is on a handful of specific type classes, that are both very useful for day-to-day programming tasks and as conceptual models that can drive program design. In this part we'll be looking more at their use for day-to-day programming, while the case studies will focus on their role in design.

In Chapter 7 we introduce the Cats[31] library. Cats provides implementation of the type classes we're interested in, and so it saves a lot of time and typing to use it.

[31] https://typelevel.org/cats

7. Using Cats

In this Chapter we'll learn how to use the Cats[32] library. Cats provides two main things: type classes and their instances, and some useful data structures. Our focus will mostly be on the type classes, though we will touch on the data structures where appropriate.

7.1. Quick Start

The easiest, and recommended, way to use Cats is to add the following imports:

```
import cats.*
import cats.syntax.all.*
```

The first import adds all the type classes (and makes their instances available, as they are found in the companion objects.) The second import adds the syntax helpers, which makes the type classes easier to work with. Note we don't need to `import cats.{*, given}` as, at the time of writing, Cats is written in Scala 2 style (using `implicits`) and these are imported by the wildcard import.

If we want use some of Cats' datastructures, we also need to add

```
import cats.data.*
```

[32]https://typelevel.org/cats

7.2. Using Cats

Let's now see how we work with Cats, using `cats.Show`[33] as an example.

`Show` is Cats' equivalent of the `Display` type class we defined in Section 5.5. It provides a mechanism for producing developer-friendly console output without using `toString`. Here's an abbreviated definition:

```
package cats

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

The easiest way to use `Show` is with the wildcard import above. However, we can also import `Show` directly from the `cats` package if we want:

```
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]
```

Once we have an instance we can call methods on it.

```
showInt.show(42)
// res0: String = "42"
```

More common, however, is to use the syntax or extension methods, which we imported with `import cats.syntax.all.*`. In the case of `Show`, an extension method `show` is defined.

[33] http://typelevel.org/cats/api/cats/Show.html

```
42.show
// res1: String = "42"
```

If, for some reason, we wanted just the syntax for `show`, we could import `cats.syntax.show`.

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

7.2.1. Defining Custom Instances

As we learned in Chapter 5, we can define an instance of `Show` by implementing a given instance of the trait for a specific type. Let's implement `Show` for `java.util.Date`.

```
import java.util.Date

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

It works as expected.

```
new Date().show
// res2: String = "1769000510800ms since the epoch."
```

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

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

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

```
    ???
}
```

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

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

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

Exercise: Cat Show

In this exercise we'll re-implement the `Cat` application from Section 5.5.1 using `Show` instead of `Display`.

Using this data type to represent a well-known type of furry animal:

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

create an implementation of `Display` for `Cat` that returns content in the following format:

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

Then 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!
```

7.3. Example: Eq

We will finish off this chapter by looking at another useful type class: `cats.Eq`[34]. `Eq` is designed to support *type-safe equality* and address annoyances using Scala's built-in == operator.

Almost every Scala developer has written code like this before:

```
List(1, 2, 3).map(Option(_)).filter(item => item == 1)
// error:
// Values of types Option[Int] and Int cannot be compared with ==
or !=
// List(1, 2, 3).map(Option(_)).filter(item => item == 1)
//                                             ^^^^^^^^^
```

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

This is programmer error—we should have compared `item` to `Some(1)` instead of `1`. However, it's not technically a type error because == works for any pair of objects, no matter what types we compare. `Eq` is designed to add some type safety to equality checks and work around this problem.[35]

[34] http://typelevel.org/cats/api/cats/kernel/Eq.html

[35] Scala 3 has it's own solution to this problem, called multiversal equality[36]. It also uses a type class, in this case called `CanEqual`. With the correct imports or compiler flags we can get the compiler to complain if we try to perform an equality check that doesn't make sense. So in practice we don't need `Eq` any more. However it's a simple type class to work with and makes a good introduction to using Cats.

[36] https://docs.scala-lang.org/scala3/reference/contextual/multiversal-equality.html

7.3.1. 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.eq`[37], provides two methods for performing equality checks provided there is an instance `Eq[A]` in scope:

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

7.3.2. Comparing Ints

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

```
import cats.*
```

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

```
val eqInt = Eq[Int]
```

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

```
eqInt.eqv(123, 123)
// res2: Boolean = true
eqInt.eqv(123, 234)
// res3: Boolean = false
```

[37] https://www.javadoc.io/doc/org.typelevel/cats-docs_3/latest/cats/syntax/EqSyntax.html

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

```
eqInt.eqv(123, "234")
// error:
// Found:    ("234" : String)
// Required: Int
// eqInt.eqv(123, "234")
//                ^^^^^
```

We can also import the interface syntax to use the === and =!= methods:

```
import cats.syntax.all.* // for === and =!=
```

Now the syntax methods are available.

```
123 === 123
// res5: Boolean = true
123 =!= 234
// res6: Boolean = true
```

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

```
123 === "123"
// error:
// Found:    ("123" : String)
// Required: Int
// 123 === "123"
//         ^^^^^
```

7.3.3. Comparing Options

Now for a more interesting example—`Option[Int]`.

```
Some(1) === None
// error:
// value === is not a member of Some[Int] - did you mean
```

```
Some[Int].==?
// Some(1) === None
// ^^^^^^^^^^^
```

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

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

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

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

or using special syntax from `cats.syntax.option`[38]:

```
1.some === none[Int]
// res11: Boolean = false
1.some =!= none[Int]
// res12: Boolean = true
```

7.3.4. 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

given dateEq: Eq[Date] =
```

[38] https://www.javadoc.io/doc/org.typelevel/cats-docs_3/latest/cats/syntax/OptionSyntax.html

```
  Eq.instance[Date] { (date1, date2) =>
    date1.getTime === date2.getTime
  }
```

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

```
x === x
// res13: Boolean = true
x === y
// res14: Boolean = true
```

Exercise: Equality, Liberty, and Felinity

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

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

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

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

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

7.4. Conclusions

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

- The type classes themselves are generic traits in the `cats` package.
- Each type class has a companion object with, an `apply` method for materializing instances, one or more construction methods

for creating instances, and a collection of other relevant helper methods.

- Many type classes have extension methods provided via the `cats.syntax` package.

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

8. Monoids and Semigroups

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

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 where we place brackets when we add elements, as 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
```

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

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

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

Multiplication, like addition, is associative:

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

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

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

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

and the empty string as the identity:

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

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

Once again, concatenation is associative:

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

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

Note that we used ++ above instead of the more usual + to suggest a parallel with sequences. We can do the same with other types of

sequence, using concatenation as the binary operator and the empty sequence as our identity.

8.1. Definition of a Monoid

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

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

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

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

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

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

def identityLaw[A](x: A)
      (using 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
// res14: Int = -4

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

In practice we only need to think about laws when we are writing our own `Monoid` instances. Unlawful instances are dangerous, not because using them can cause us to end up in jail, but because they can yield unpredictable results when used with the rest of Cats' machinery. Most of the time we can rely on the instances provided by Cats and assume the library authors knew what they were doing.

8.2. Definition of a Semigroup

A semigroup is just the `combine` part of a monoid, without the `empty` part. While many semigroups are also monoids, there are some data types for which we cannot define an `empty` element. For example, we have just seen that sequence concatenation and integer addition are monoids. However, if we restrict ourselves to non-empty sequences and positive integers, we are no longer able to define a sensible `empty` element. As a concrete example, Cats has a `NonEmptyList`[39] data type that has an implementation of `Semigroup` but no implementation of `Monoid`.

A more accurate (though still simplified) definition of Cats' `Monoid`[40] is:

[39] http://typelevel.org/cats/api/cats/data/NonEmptyList.html
[40] http://typelevel.org/cats/api/cats/Monoid.html

```
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.

8.2.0.1. 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
}
```

8.2.0.2. Exercise: All Set for Monoids

What monoids and semigroups are there for sets?

8.3. Monoids in Cats

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

8.3.1. The Monoid Type Class

The monoid type class is `cats.kernel.Monoid`, which is aliased as `cats.Monoid`[41]. `Monoid` extends `cats.kernel.Semigroup`, which is aliased as `cats.Semigroup`[42]. When using Cats we normally import type classes from the `cats`[43] package:

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

or just

```
import cats.*
```

> **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`[44] package, they are all aliased to

[41] http://typelevel.org/cats/api/cats/kernel/Monoid.html
[42] http://typelevel.org/cats/api/cats/kernel/Semigroup.html
[43] http://typelevel.org/cats/api/cats/
[44] http://typelevel.org/cats/api/cats/kernel/

the `cats`[45] package so we rarely need to be aware of the distinction.

The Cats Kernel type classes covered in this book are `Eq`[46], `Semigroup`[47], and `Monoid`[48]. All the other type classes we cover are part of the main Cats project and are defined directly in the `cats`[49] package.

8.3.2. Monoid Instances

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

```
import cats.Monoid
Monoid[String].combine("Hi ", "there")
// res1: String = "Hi there"
Monoid[String].empty
// res2: String = ""
```

which is equivalent to

```
Monoid.apply[String].combine("Hi ", "there")
// res3: String = "Hi there"
Monoid.apply[String].empty
// res4: String = ""
```

[45] http://typelevel.org/cats/api/cats/
[46] http://typelevel.org/cats/api/cats/kernel/Eq.html
[47] http://typelevel.org/cats/api/cats/kernel/Semigroup.html
[48] http://typelevel.org/cats/api/cats/kernel/Monoid.html
[49] http://typelevel.org/cats/api/cats/

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

```
import cats.Semigroup
```

and then summon instances of `Semigroup` in the usual way:

```
Semigroup[String].combine("Hi ", "there")
// res5: String = "Hi there"
```

The standard type class instances for `Monoid` are all found on the appropriate companion objects, and so are automatically in the given scope with no further imports required.

8.3.3. Monoid Syntax

Cats provides syntax for the `combine` method in the form of the `|+|` operator. Because `combine` technically comes from `Semigroup`, we access the syntax by importing from `cats.syntax.semigroup`[50]:

```
import cats.syntax.semigroup.* // for |+|
```

Now we can use `|+|` in place of calling `combine`.

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

val intResult = 1 |+| 2
// intResult: Int = 3
```

As always, unless there is compelling reason not, we recommend importing all the syntax with

[50] http://typelevel.org/cats/api/cats/syntax/package$$semigroup$

```
import cats.syntax.all.*
```

8.3.3.1. 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!

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!

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!

8.4. Applications of Monoids

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

8.4.1. Big Data

In big data applications like Spark and Flink 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 in Chapter 20.

8.4.2. 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 conflict-free 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.

8.4.3. 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 remainder of this book.

8.5. 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 two things: the type classes themselves, and the semigroup syntax to give us the `|+|` operator:

```
import cats.Monoid
import cats.syntax.semigroup.* // for |+|
```

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

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

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

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

```
map1 |+| map2
// res2: Map[String, Int] = Map("b" -> 5, "d" -> 4, "a" -> 1)
```

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

```
tuple1 |+| tuple2
// res3: Tuple2[String, Int] = ("helloworld", 444)
```

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

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

```
addAll(List(1, 2, 3))
// res4: Int = 6
addAll(List(None, Some(1), Some(2)))
// res5: Option[Int] = Some(value = 3)
```

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

9. Functors

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

9.1. Examples of Functors

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

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

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

Similarly, when we `map` over an `Option`, we transform the contents but leave the `Some` or `None` context unchanged. The same principle applies to `Either` with its `Left` and `Right` contexts. This general notion of transformation, along with the common pattern of type signatures shown in Figure 1, is what connects the behaviour of `map` across different data types.

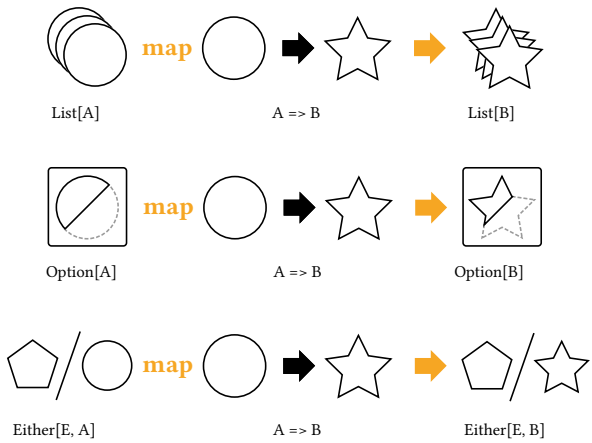

Figure 1: Type chart: mapping over List, Option, and Either

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

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

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

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

9.2. More Examples of Functors

The `map` methods of `List`, `Option`, and `Either` apply functions eagerly. However, the idea of sequencing computations is more

general than this. Let's investigate the behaviour of some other functors that apply the pattern in different ways.

9.2.1. Futures

`Future` is a functor that sequences asynchronous computations by queueing them and applying them as their predecessors complete. The type signature of its `map` method, shown in Figure 2, has the same shape as the signatures above. However, the behaviour is very different.

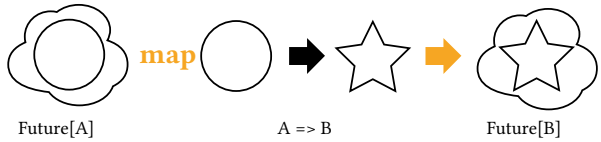

Figure 2: Type chart: mapping over a Future

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

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

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

```
Await.result(future, 1.second)
// res2: String = "248!"
```

Futures and Referential Transparency

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

```
import scala.util.Random

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

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

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

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

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

```
val result1 = Await.result(future1, 1.second)
// result1: Tuple2[Int, Int] = (-1155484576, -1155484576)
```

```
val result2 = Await.result(future2, 1.second)
// result2: Tuple2[Int, Int] = (-1155484576, -723955400)
```

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

This kind of discrepancy makes it hard to reason about programs involving `Futures` and side-effects. There also are other problematic aspects of `Future's` behaviour, such as the way it always starts computations immediately rather than allowing the user to dictate when the program should run. For more information see this excellent Reddit answer[51] by Rob Norris.

When we look at Cats Effect we'll see that the `IO` type solves these problems.

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

9.2.2. Functions (?!)

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

- start with `X => A`;

[51] https://www.reddit.com/r/scala/comments/3zofjl/why_is_future_totally_unusable/

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

If we alias `X => A` as `MyFunc[A]`, we see the same pattern of types we saw with the other examples in this chapter. We also see this in Figure 3:

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

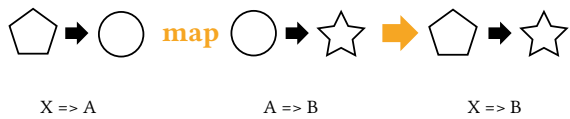

Figure 3: Type chart: mapping over a Function1

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

```
import cats.syntax.all.*     // for map

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

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

```
(func1.map(func2))(1)     // composition using map
(func1.andThen(func2))(1) // composition using andThen
// res3: Double = 2.0
func2(func1(1))           // composition written out by hand
// res4: Double = 2.0
```

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

of the operations are run in sequence. We can think of this as lazily queueing up operations similar to `Future`:

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

```
func(123)
// res5: String = "248.0!"
```

Partial Unification

For the above examples to work, in versions of Scala before 2.13, we need to add the following compiler option to `build.sbt`:

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

otherwise we'll get a compiler error:

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

We'll look at why this happens in detail in Section 9.8.

9.3. Definition of a Functor

Every example we've looked at so far is a functor: a class that encapsulates sequencing computations. Formally, a functor is a

type `F[A]` with an operation `map` with type `(A => B) => F[B]`. The general type chart is shown in Figure 4.

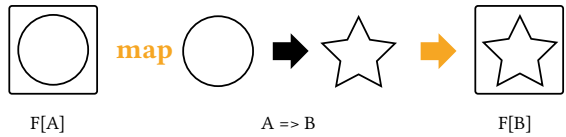

Figure 4: Type chart: generalised functor map

Cats encodes `Functor` as a type class, `cats.Functor`[52], so the method looks a little different. It accepts the initial `F[A]` as a parameter alongside the transformation function. Here's a simplified version of the definition:

```
package cats

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**.

Functor Laws

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

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

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

[52] http://typelevel.org/cats/api/cats/Functor.html

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)
```

9.4. Higher Kinds and Type Constructors

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

For example, `List` is a type constructor with one hole. We fill that hole with a type 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 by applying 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 by applying a value parameter
```

In Scala we declare type constructors using underscores. This specifies how many "holes" the type constructor has. However, to use them we refer to just the name.

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

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

  // ...
}
```

This is analogous to specifying function parameter types. When we declare a parameter we also give its type. However, to use them we refer to just the name.

```
// Declare f specifying parameter types
def f(x: Int): Int =
  // Reference x without type
  x * 2
```

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

Language Feature Imports

In versions of Scala before 2.13 we need to "enable" the higher kinded type language feature, to suppress warnings from the compiler, whenever we declare a type constructor with `A[_]` syntax. We can either do this with a "language import" as above:

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

or by adding the following to `scalacOptions` in `build.sbt`:

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

> In practice we find the `scalacOptions` flag to be the simpler of the two options.

9.5. Functors in Cats

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

9.5.1. The Functor Type Class and Instances

The functor type class is `cats.Functor`[53]. We obtain instances using the standard `Functor.apply` method on the companion object. As usual, default instances are found on companion objects and do not have to be explicity imported:

```
import cats.*
import cats.syntax.all.*
```

Once we have the imports we use the `map` method defined by `Functor`. In the examples below we are explicitly summoning the type class instances to avoid using the built-ins that are defined on `List` and `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)
```

[53] http://typelevel.org/cats/api/cats/Functor.html

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

`Functor` provides a method called `lift`, which converts a function of type `A => B` to one that operates over a functor and has type `F[A] => F[B]`: Let's lift a function into the `Option` functor.

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

val liftedFunc = Functor[Option].lift(func)
```

Now we can directly apply it to an `Option`.

```
liftedFunc(Option(1))
// res1: Option[Int] = Some(value = 2)
```

The `as` method is the other method you are likely to use. It replaces the value inside the `Functor` with the given value.

```
Functor[List].as(list1, "As")
// res2: List[String] = List("As", "As", "As")
```

9.5.2. Functor Syntax

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

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

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

Once we've constructed a function using `map` we can apply it.

```
func4(123)
// res3: String = "248!"
```

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

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

We can write this more compactly with a context bound.

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

It works as expected, using whatever `Functor` instance we pass it.

```
doMath(Option(20))
// res4: Option[Int] = Some(value = 41)
doMath(List(1, 2, 3))
// res5: List[Int] = List(3, 5, 7)
```

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

```
extension [F[_], A](src: F[A]) {
  def map[B](func: A => B)
      (using functor: Functor[F]): F[B] =
```

```
    functor.map(src)(func)
}
```

The compiler can use this extension method to insert a `map` method wherever no built-in `map` is available. If we have the code

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

and assume `foo` has no built-in `map` method, the compiler detects the potential error and uses the extension method to fix it. The `map` extension method requires a given `Functor` as a parameter. This means this code will only compile if we have a `Functor` for `F` in scope. If we don't, we get a compiler error.

Here's an example of the error. First we define a new type that has no `Functor` instance.

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

val box = Box[Int](123)
```

Now attempting to call `map` fails. Notice the error message gives us a hint as to what went wrong.

```
box.map(value => value + 1)
// error:
// No given instance of type cats.Functor[Box] was found for
parameter functor of method map
// box.map(value => value + 1)
//                  ^
```

The `as` method is also available as syntax, and works in the same way.

```
List(1, 2, 3).as("As")
// res8: List[String] = List("As", "As", "As")
```

9.5.3. 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`[54]. The implementation is trivial—we simply call `Option`'s `map` method:

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

Sometimes we need to inject dependencies into our instances. For example, if we had to define a custom `Functor` for `Future` (another hypothetical example—Cats provides one in `cats.instances.future`) we would need to account for the given `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}

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

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

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

[54] http://typelevel.org/cats/api/cats/instances/

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

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

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`:

```
enum Tree[+A] {
  case Branch[A](left: Tree[A], right: Tree[A])
    extends Tree[A]

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

9.6. Contravariant and Invariant Functors

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

This Section is Optional!

You don't need to know about contravariant and invariant functors to understand monads, which are the most important type class in this book and the focus of the next chapter. However, contravariant and invariant do come in handy in our discussion of `Semigroupal` and `Applicative` in Chapter 12.

If you want to move on to monads now, feel free to skip straight to Chapter 10. Come back here before you read Chapter 12.

9.6.1. Contravariant Functors and the `contramap` Method

The first of our type classes, the *contravariant functor*, provides an operation called `contramap` that represents "prepending" an operation to a chain. The general type signature is shown in Figure 5.

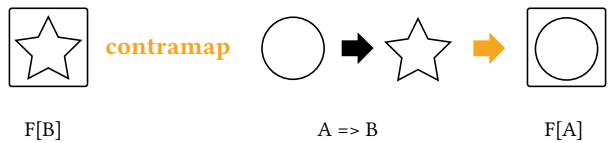

Figure 5: Type chart: the contramap method

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

can define `contramap` for the `Display` type class we discussed in Section 5.5:

```
trait Display[A] {
  def display(value: A): String
}
```

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

```
trait Display[A] {
  def display(value: A): String

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

def display[A](value: A)(using p: Display[A]): String =
  p.display(value)
```

Exercise: Showing Off with Contramap

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

```
trait Display[A] {
  def display(value: A): String

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

If you get stuck, think about the types. You need to turn `value`, which is of type `B`, into a `String`. What functions and methods do

you have available and in what order do they need to be combined?

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

```
given stringDisplay: Display[String] with {
  def display(value: String): String =
    s"'${value}'"
}

given booleanDisplay: Display[Boolean] with {
  def display(value: Boolean): String =
    if value then "yes" else "no"
}
```

```
display("hello")
// res2: String = "'hello'"
display(true)
// res3: String = "yes"
```

Now define an instance of `Display` for the following `Box` case class. This is an example of type class composotion as described in Section 5.3:

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

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

Your instance should work as follows:

```
display(Box("hello world"))
// res4: String = "'hello world'"
display(Box(true))
// res5: String = "yes"
```

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

```
display(Box(123))
// error:
// No given instance of type
repl.MdocSession.MdocApp1.Display[repl.MdocSession.MdocApp1.Box[Int]]
was found for parameter p of method display in object MdocApp1.
// I found:
//
//     repl.MdocSession.MdocApp1.boxDisplay[Int](
//       /* missing */
summon[repl.MdocSession.MdocApp1.Display[Int]])
//
// But no implicit values were found that match type
repl.MdocSession.MdocApp1.Display[Int].
// display(Box(123))
//                 ^
```

9.6.2. Invariant functors and the imap method

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

The most intuitive examples of this are a type class that represents encoding and decoding as some data type, such as Circe's `Codec`[55] and Play JSON's `Format`[56]. We can build our own `Codec` by enhancing `Display` to support encoding and decoding to and from a `String`:

```
trait Codec[A] {
  def encode(value: A): String
  def decode(value: String): A
```

[55] https://github.com/circe/circe/blob/series/0.14.x/modules/core/shared/src/main/scala/io/circe/Codec.scala

[56] https://www.playframework.com/documentation/2.6.x/ScalaJsonCombinators#Format

```
  def imap[B](dec: A => B, enc: B => A): Codec[B] = ???
}
```

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

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

The type chart for `imap` is shown in Figure 6. If we have a `Codec[A]` and a pair of functions `A => B` and `B => A`, the `imap` method creates a `Codec[B]`:

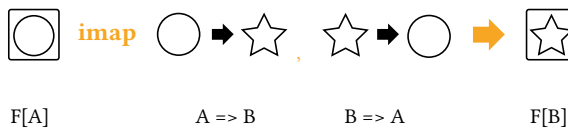

Figure 6: Type chart: the imap method

As an example use case, imagine we have a basic `Codec[String]`, whose `encode` and `decode` methods both simply return the value they are passed:

```
given stringCodec: Codec[String] with {
  def encode(value: String): String = value
  def decode(value: String): String = value
}
```

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

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

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

Coping with Failure

Note that the `decode` method of our `Codec` type class doesn't account for failures. If we want to model more sophisticated relationships we can move beyond functors to look at *lenses* and *optics*.

Optics are beyond the scope of this book. However, the Monocle[57] provides a great starting point for further investigation.

9.6.2.1. Transformative Thinking with imap

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

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

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

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

Your instances should work as follows:

```
encode(123.4)
// res11: String = "123.4"
decode[Double]("123.4")
// res12: Double = 123.4

encode(Box(123.4))
// res13: String = "123.4"
decode[Box[Double]]("123.4")
// res14: Box[Double] = Box(value = 123.4)
```

[57] https://github.com/optics-dev/Monocle

What's With the Names?

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

As we discussed in Section 5.6.1, variance affects subtyping, which is essentially our ability to use a value of one type in place of a value of another type without breaking the code.

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

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

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

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

9.7. Contravariant and Invariant in Cats

Let's look at the implementation of contravariant and invariant

functors in Cats, provided by the `cats.Contravariant`[58] and `cats.Invariant`[59] type classes respectively. Here's a simplified version of the code:

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

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

9.7.1. Contravariant in Cats

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

```
import cats.*

val showString = Show[String]

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

```
showSymbol.show(Symbol("dave"))
// res1: String = "'dave"
```

More conveniently, we can use `cats.syntax.contravariant`[60], which provides a `contramap` extension method:

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

[58] http://typelevel.org/cats/api/cats/Contravariant.html
[59] http://typelevel.org/cats/api/cats/Invariant.html
[60] http://typelevel.org/cats/api/cats/syntax/package$$contravariant$

```
showString
  .contramap[Symbol](sym => s"'${sym.name}")
  .show(Symbol("dave"))
// res2: String = "'dave"
```

9.7.2. Invariant in Cats

Among other types, Cats provides an instance of `Invariant` for `Monoid`. This is a little different from the `Codec` example we introduced in Section 9.6.2. If you recall, this is what `Monoid` looks like:

```
package cats

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

Imagine we want to produce a `Monoid` for Scala's `Symbol`[61] type. Cats doesn't provide a `Monoid` for `Symbol` but it does provide a `Monoid` for a similar type: `String`. We can write our new semigroup with an `empty` method that relies on the empty `String`, and a `combine` method that works as follows:

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

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

[61] https://www.scala-lang.org/api/3.3.3/scala/Symbol.html#

```
import cats.*
import cats.syntax.invariant.* // for imap
import cats.syntax.semigroup.* // for |+|

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

```
Monoid[Symbol].empty
// res3: Symbol = '

Symbol("a") |+| Symbol("few") |+| Symbol("words")
// res4: Symbol = 'afewwords
```

9.8. Aside: Partial Unification

In Section 9.2 we saw a functor instance for `Function1`.

```
import cats.*
import cats.syntax.functor.*     // for map

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

```
val func3 = func1.map(func2)
// func3: Function1[Int, Double] =
cats.instances.Function1Instances0$$anon$11$
$Lambda$13184/0x0000000803958040@2ecffd5e
```

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

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

However, `Functor` accepts a type constructor with one parameter:

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

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

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

We know that the first one is the correct choice. However the compiler doesn't understand what the code means. Instead it relies on a simple rule, implementing what is called "partial unification".

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

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

val functor = Functor[F]
```

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

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

either.map(_ + 1)
// res0: Either[String, Int] = Right(value = 124)
```

> Partial unification is the default behaviour in Scala 2.13. In earlier versions of Scala we need to add the `-Ypartial-`

`unification` compiler flag. In sbt we would add the compiler flag in `build.sbt`:

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

The rationale behind this change is discussed in SI-2712[62].

9.8.1. Limitations of Partial Unification

There are situations where left-to-right elimination is not the correct choice. One example is the `Or` type in Scalactic[63], which is a conventionally left-biased equivalent of `Either`:

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

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

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

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

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

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

[62] https://issues.scala-lang.org/browse/SI-2712
[63] http://scalactic.org

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

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

```
val func3c = func2.contramap(func1)
// error:
// value contramap is not a member of Double => Double.
// An extension method was tried, but could not be fully
constructed:
//
//     cats.syntax.contravariant.toContravariantOps[[R] =>>
Double => R, A](
//       repl.MdocSession.MdocApp.func2)(
//       cats.Invariant.catsContravariantForFunction1[R])
// val func3c = func2.contramap(func1)
//              ^^^^^^^^^^^^^^^
```

The problem here is that the `Contravariant` for `Function1` fixes the return type and leaves the parameter type varying, requiring the compiler to eliminate type parameters from right to left, as shown below and in Figure 7:

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

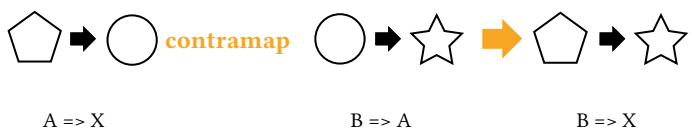

Figure 7: Type chart: contramapping over a Function1

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

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

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

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

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

```
val func3c = func2b.contramap(func1)
// func3c: Function1[Int, Double] = scala.Function1$
$Lambda$13249/0x0000000803965040@5b3e48b8
```

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

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

9.9. Conclusions

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

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

- `Invariant` functors, with their `imap` method, represent bidirectional transformations.

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

Functors for collections are extremely important, as they transform each element independently of the rest. This allows us to parallelise or distribute transformations on large collections, a technique leveraged heavily in "map-reduce" frameworks popularized by Hadoop[64]. We will investigate this approach in more detail in the map-reduce case study later in Chapter 20.

The `Contravariant` and `Invariant` type classes are less widely applicable but are still useful for building data types that represent transformations. We will revisit them to discuss the `Semigroupal` type class later in Chapter 12.

[64]http://hadoop.apache.org

10. Monads

Monads are one of the best known abstractions in functional programming, but also the one that perhaps leads to the most confusion. Many programmers have used and are intuitively familiar with monads, even if we don't know them by name.

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

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 see how we can create a concrete type as a type class. We'll then look at their implementation in Cats. Finally, we'll tour some interesting monads that you may not have seen, providing introductions and examples of their use.

10.1. What is a Monad?

This is the question that has been posed in a thousand blog posts, with explanations and analogies involving concepts as diverse as cats, Mexican food, space suits full of toxic waste, and monoids in the category of endofunctors (whatever that means). We're going to solve the problem of explaining monads once and for all by stating very simply: a monad is a mechanism for *sequencing computations*.

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

In Section 9.1 we said that functors allow us to sequence computations ignoring some complication. However, functors are limited in that they only allow this complication to occur once, at the beginning of the sequence. They don't account for further complications at each step in the sequence.

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

10.1.1. Options as Monads

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

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

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

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

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

```
      divide(aNum, bNum)
    }
  }
```

The semantics are:

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

At each step, `flatMap` chooses whether to call our function, and our function generates the next computation in the sequence. This is shown in Figure 8.

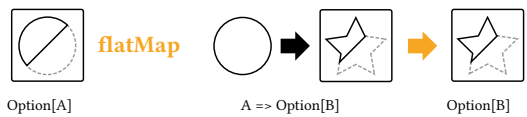

Figure 8: Type chart: flatMap for Option

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

```
stringDivideBy("6", "2")
// res0: Option[Int] = Some(value = 3)
stringDivideBy("6", "0")
// res1: Option[Int] = None
stringDivideBy("6", "foo")
// res2: Option[Int] = None
stringDivideBy("bar", "2")
// res3: Option[Int] = None
```

Every monad is also a functor (see below for proof), so we can rely on both `flatMap` and `map` to sequence computations that do and

don't introduce a new monad respectively. 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
```

10.1.2. Lists as Monads

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

```
for {
  x <- (1 to 3).toList
  y <- (4 to 5).toList
} yield (x, y)
// res5: List[Tuple2[Int, Int]] = List(
//   (1, 4),
//   (1, 5),
//   (2, 4),
//   (2, 5),
//   (3, 4),
//   (3, 5)
// )
```

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

For example, in the for comprehension above there are three possible values of `x` and two possible values of `y`. This means there are six possible values of `(x, y)`. `flatMap` is generating these

combinations from our code, which states the sequence of operations:

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

The type chart in Figure 9 illustrates this behaviour[65].

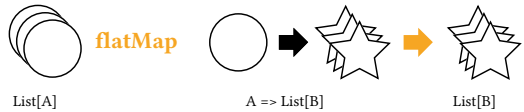

Figure 9: Type chart: flatMap for List

10.1.3. Futures as Monads

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

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

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

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

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

[65] 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 `As` and `Bs`.

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

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

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

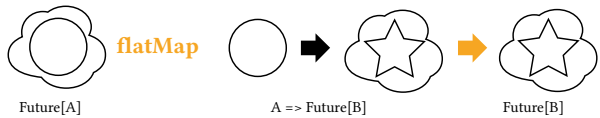

Figure 10: Type chart: flatMap for Future

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

10.1.4. Definition of a Monad

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

- `pure`, of type `A => F[A]`;
- `flatMap`[66], of type `(F[A], A => F[B]) => F[B]`.

[66]In the programming literature and Haskell, `pure` is often referred to as `point` or `return` and `flatMap` is often referred to as `bind` or `>>=`. This is purely a difference in terminology. We'll use the term `flatMap` for compatibility with Cats and the Scala standard library.

`pure` abstracts over constructors, providing a way to create a new monadic context from a plain value. `flatMap` provides the sequencing step we have already discussed, extracting the value from a context and generating the next context in the sequence.

We can represent this as a type class. Here is a simplified version of the `Monad` type class in Cats:

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

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

Monad Laws

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

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

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

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

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

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

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

10.1.5. Exercise: Getting Func-y

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

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

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

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

Try defining `map` yourself now.

10.2. 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.

10.2.1. The Monad Type Class

The monad type class is `cats.Monad`[67]. `Monad` extends two other type classes: `FlatMap`, which provides the `flatMap` method, and `Applicative`, which provides `pure`. `Applicative` also extends

[67] http://typelevel.org/cats/api/cats/Monad.html

`Functor`, which gives every `Monad` a `map` method as we saw in the exercise above. We'll discuss `Applicatives` in Chapter 12.

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

```
import cats.Monad
```

```
val opt1 = Monad[Option].pure(3)
// opt1: Option[Int] = Some(value = 3)
val opt2 = Monad[Option].flatMap(opt1)(a => Some(a + 2))
// opt2: Option[Int] = Some(value = 5)
val opt3 = Monad[Option].map(opt2)(a => 100 * a)
// opt3: Option[Int] = Some(value = 500)

val list1 = Monad[List].pure(3)
// list1: List[Int] = List(3)
val list2 = Monad[List].
  flatMap(List(1, 2, 3))(a => List(a, a*10))
// list2: List[Int] = List(1, 10, 2, 20, 3, 30)
val list3 = Monad[List].map(list2)(a => a + 123)
// list3: List[Int] = List(124, 133, 125, 143, 126, 153)
```

`Monad` provides many other methods, including all of the methods from `Functor`. See the documentation[68] for more information.

10.2.2. Default Instances

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

Let's import `Future` (and some other imports we will use later.)

[68] http://typelevel.org/cats/api/cats/Monad.html

```
import scala.concurrent.*
import scala.concurrent.duration.*
```

We see that compilation fails without an `ExecutionContext` available.

```
val fm = Monad[Future]
// error:
// No given instance of type cats.Monad[scala.concurrent.Future]
was found for parameter instance of method apply in object Monad.
// I found:
//
//     cats.Invariant.catsInstancesForFuture(
//       /* missing */summon[scala.concurrent.ExecutionContext])
//
// But no implicit values were found that match type
scala.concurrent.ExecutionContext.
// val fm = Monad[Future]
//                       ^
```

Now we bring the `ExecutionContext` into scope.

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

This provides the given instance required to summon the `Monad[Future]` instance:

```
val fm = Monad[Future]
// fm: Monad[[T >: Nothing <: Any] =>> Future[T]] =
cats.instances.FutureInstances$$anon$1@7f39cfbb
```

The `Monad` instance uses the captured `ExecutionContext` for subsequent calls to `pure` and `flatMap`. We can construct a `Future` using calls to the monad instance we summoned above.

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

If we await the result of the `Future` we get the expected result.

```
Await.result(future, 1.second)
// res1: Int = 3
```

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

10.2.3. Monad Syntax

The syntax for monads comes from three places:

- `cats.syntax.flatMap`[69] provides syntax for `flatMap`;
- `cats.syntax.functor`[70] provides syntax for `map`;
- `cats.syntax.applicative`[71] provides syntax for `pure`.

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

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

```
import cats.syntax.applicative.* // for pure
```

```
1.pure[Option]
// res2: Option[Int] = Some(value = 1)
1.pure[List]
// res3: 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

[69] http://typelevel.org/cats/api/cats/syntax/package$$flatMap$
[70] http://typelevel.org/cats/api/cats/syntax/package$$functor$
[71] http://typelevel.org/cats/api/cats/syntax/package$$applicative$

generic function that performs a calculation on parameters that come wrapped in a monad of the user's choice:

```
import cats.Monad
import cats.syntax.functor.* // for map
import cats.syntax.flatMap.* // for flatMap

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

```
sumSquare(Option(3), Option(4))
// res4: Option[Int] = Some(value = 25)
sumSquare(List(1, 2, 3), List(4, 5))
// res5: List[Int] = List(17, 26, 20, 29, 25, 34)
```

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

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

```
sumSquare(Option(3), Option(4))
// res7: Option[Int] = Some(value = 25)
sumSquare(List(1, 2, 3), List(4, 5))
// res8: List[Int] = List(17, 26, 20, 29, 25, 34)
```

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

10.3. The Identity Monad

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

```
import cats.Monad
import cats.syntax.all.*

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

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

```
sumSquare(3, 4)
// error:
// Found:    (3 : Int)
// Required: ([_] =>> Any)[Int]
// Note that implicit conversions were not tried because the
result of an implicit conversion
// must be more specific than ([_] =>> Any)[Int]
// sumSquare(3, 4)
//           ^
// error:
// Found:    (4 : Int)
// Required: ([_] =>> Any)[Int]
// Note that implicit conversions were not tried because the
result of an implicit conversion
// must be more specific than ([_] =>> Any)[Int]
// sumSquare(3, 4)
//              ^
```

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

```
import cats.Id
```

```
sumSquare(3 : Id[Int], 4 : Id[Int])
// res1: Int = 25
```

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

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

```
package cats

type Id[A] = A
```

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

```
"Dave" : Id[String]
// res2: String = "Dave"
123 : Id[Int]
// res3: Int = 123
List(1, 2, 3) : Id[List[Int]]
// res4: List[Int] = List(1, 2, 3)
```

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

```
val a = Monad[Id].pure(3)
// a: Int = 3
val b = Monad[Id].flatMap(a)(_ + 1)
// b: Int = 4
```

```
import cats.syntax.functor.* // for map
import cats.syntax.flatMap.* // for flatMap
```

```
for {
  x <- a
```

```
  y <- b
} yield x + y
// res5: Int = 7
```

The ability to abstract over monadic and non-monadic code is extremely powerful. For example, we can run code asynchronously in production using `Future` and synchronously in test using `Id`. We'll see this in our first case study in Chapter 18.

10.3.1. Exercise: Monadic Secret Identities

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

The `Id` monad does find occasional use in highly generic code, but I think it is more useful as a tool for understanding monads in general. Remember we said a monad is a tool for sequencing computations. When we write

```
a.flatMap(b)
```

we are saying that `b` occurs after `a`, subject to whatever complications the concrete monad and `a` might introduce. In other words, monads express control flow. Our programming languages already have built-in ways of expressing control flow. In Scala, like most languages, control flow goes top-to-bottom and left-to-right. We can think of this as an "ambient" monad, a monad that conceptually exists but we don't work with directly. When we write

```
1 + 2
// res10: Int = 3
```

we can instead express it in monadic terms as

```
Id(1).flatMap(_ + 2)
// res11: Int = 3
```

This shows us that monads are reifying control flow, making it explicit. This in turn puts the control flow under the control of the monad, which allows, for example, the error handling behaviour we saw with `Option`.

10.4. Either

Let's look at another useful monad: the `Either` type from the Scala standard library. `Either` has two cases, `Left` and `Right`. By convention `Right` represents a success case, and `Left` a failure. When we call `flatMap` on `Either`, computation continues if we have a `Right` case.

```
Right(10).flatMap(a => Right(a + 32))
// res0: Either[Nothing, Int] = Right(value = 42)
```

A `Left`, however, stops the computation.

```
Right(10).flatMap(a => Left("Oh no!"))
// res1: Either[String, Nothing] = Left(value = "Oh no!")
```

AS these examples suggest, `Either` is typically used to implement fail-fast error handling. We sequence computations using `flatMap` as usual. If one computation fails, the remaining computations are not run. Here's an example where we fail if we attempt to divide by zero.

```
for {
  a <- Right(1)
  b <- Right(0)
  c <- if(b == 0) Left("DIV0")
       else Right(a / b)
```

```
} yield c * 100
// res2: Either[String, Int] = Left(value = "DIV0")
```

We can see `Either` as similar to `Option`, but it allows us to record some information in the case of failure, whereas `Option` represents failure by `None`. In the examples above we used strings to hold information about the cause of failure, but we can use any type we like. For example, we could use `Throwable` instead:

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

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

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

```
enum LoginError {
  case UserNotFound(username: String)
  case PasswordIncorrect(username: String)
  case UnexpectedError
}
```

We could use the `LoginError` type along with `Either` as shown below.

```
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:

```
import LoginError.*
```

```
// 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")
  }
```

Here's an example of use.

```
val result1: LoginResult = Right(User("dave", "passw0rd"))
// result1: Either[LoginError, User] = Right(
//   value = User(username = "dave", password = "passw0rd")
// )
val result2: LoginResult = Left(UserNotFound("dave"))
// result2: Either[LoginError, User] = Left(
//   value = UserNotFound(username = "dave")
// )

result1.fold(handleError, println)
// User(dave,passw0rd)
result2.fold(handleError, println)
// User not found: dave
```

We have much more to say about error handling in Chapter 19.

10.4.1. Cats Utilities

Cats provides several utilities for working with `Either`. Here we go over the most useful of them.

10.4.1.1. Creating Instances

In addition to creating instances of `Left` and `Right` directly, we can also use the `asLeft` and `asRight` extension methods from Cats. For these methods we need to import the Cats syntax:

```
import cats.syntax.all.*
```

Now we can construct instances using the extensions.

```
val a = 3.asRight[String]
// a: Either[String, Int] = Right(value = 3)
val b = 4.asRight[String]
// b: Either[String, Int] = Right(value = 4)

for {
  x <- a
  y <- b
} yield x*x + y*y
// res5: Either[String, Int] = Right(value = 25)
```

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

```
def countPositive(nums: List[Int]) =
  nums.foldLeft(Right(0)) { (accumulator, num) =>
    if(num > 0) {
      accumulator.map(_ + 1)
    } else {
      Left("Negative. Stopping!")
    }
  }
// error:
// Found:    Either[Nothing, Int]
// Required: Right[Nothing, Int]
//       accumulator.map(_ + 1)
//       ^^^^^^^^^^^^^^^^^^^^^^
// error:
// Found:    Left[String, Any]
// Required: Right[Nothing, Int]
```

```
//       Left("Negative. Stopping!")
//       ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This code fails to compile for two reasons:

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

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

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

```
countPositive(List(1, 2, 3))
// res7: Either[String, Int] = Right(value = 3)
countPositive(List(1, -2, 3))
// res8: Either[String, Int] = Left(value = "Negative.
Stopping!")
```

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

```
Either.catchOnly[NumberFormatException]("foo".toInt)
// res9: Either[NumberFormatException, Int] = Left(
//   value = java.lang.NumberFormatException: For input string:
"foo"
// )
Either.catchNonFatal(sys.error("Badness"))
// res10: Either[Throwable, Nothing] = Left(
```

```
//   value = java.lang.RuntimeException: Badness
// )
```

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

```
Either.fromTry(scala.util.Try("foo".toInt))
// res11: Either[Throwable, Int] = Left(
//   value = java.lang.NumberFormatException: For input string:
"foo"
// )
Either.fromOption[String, Int](None, "Badness")
// res12: Either[String, Int] = Left(value = "Badness")
```

10.4.1.2. Transforming Eithers

Cats syntax also adds some useful methods for instances of `Either`.

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

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

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

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

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

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

```
"foo".asLeft[Int].leftMap(_.reverse)
// res16: Either[String, Int] = Left(value = "oof")
6.asRight[String].bimap(_.reverse, _ * 7)
// res17: Either[String, Int] = Right(value = 42)
"bar".asLeft[Int].bimap(_.reverse, _ * 7)
// res18: Either[String, Int] = Left(value = "rab")
```

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

```
123.asRight[String]
// res19: Either[String, Int] = Right(value = 123)
123.asRight[String].swap
// res20: Either[Int, String] = Left(value = 123)
```

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

Exercise: What is Best?

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

10.5. Aside: Error Handling and MonadError

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

this Section is Optional!

You won't need to use `MonadError` unless you need to abstract over error handling monads. For example, you can use `MonadError` to abstract over `Future` and `Try`, or over `Either` and `EitherT` (which we will meet in Chapter 11).

If you don't need this kind of abstraction right now, feel free to skip onwards to Section 10.6.

10.5.1. The MonadError Type Class

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

```
package cats

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

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

  // Handle all errors, recovering from them:
  def handleError[A](fa: F[A])(f: E => A): F[A]

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

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

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

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

```
import cats.MonadError

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

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

> **ApplicativeError**
>
> In reality, `MonadError` extends another type class called `ApplicativeError`. However, we won't encounter `Applicatives` until Chapter 12. The semantics are the same for each type class so we can ignore this detail for now.

10.5.2. Raising and Handling Errors

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

```
val success = monadError.pure(42)
// success: Either[String, Int] = Right(value = 42)
val failure = monadError.raiseError("Badness")
// failure: Either[String, Nothing] = Left(value = "Badness")
```

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

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

  case _ =>
    monadError.raiseError("It's not ok")
}
// res0: Either[String, String] = Right(value = "It's ok")
```

If we know we can handle all possible errors we can use `handleWith`.

```
monadError.handleError(failure) {
  case "Badness" => 42

  case _ => -1
}
// res1: Either[String, Int] = Right(value = 42)
```

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

```
monadError.ensure(success)("Number too low!")(_ > 1000)
// res2: Either[String, Int] = Left(value = "Number too low!")
```

Cats provides syntax for `raiseError` and `handleErrorWith` via `cats.syntax.applicativeError`[72] and `ensure` via `cats.syntax.monadError`[73]:

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

```
val success = 42.pure[ErrorOr]
// success: Either[String, Int] = Right(value = 42)
val failure = "Badness".raiseError[ErrorOr, Int]
// failure: Either[String, Int] = Left(value = "Badness")
failure.handleErrorWith{
  case "Badness" =>
    256.pure

  case _ =>
    ("It's not ok").raiseError
}
// res4: Either[String, Int] = Right(value = 256)
```

[72] http://typelevel.org/cats/api/cats/syntax/package$$applicativeError$

[73] http://typelevel.org/cats/api/cats/syntax/package$$monadError$

```
success.ensure("Number to low!")(_ > 1000)
// res5: Either[String, Int] = Left(value = "Number to low!")
```

There are other useful variants of these methods. See the source of `cats.MonadError`[74] and `cats.ApplicativeError`[75] for more information.

10.5.3. Instances of MonadError

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

```
import scala.util.Try

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

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

10.5.4. Exercise: Abstracting

Implement a method `validateAdult` with the following signature

```
def validateAdult[F[_]](age: Int)(implicit me: MonadError[F,
Throwable]): F[Int] =
  ???
```

[74] http://typelevel.org/cats/api/cats/MonadError.html
[75] http://typelevel.org/cats/api/cats/ApplicativeError.html

When passed an `age` greater than or equal to 18 it should return that value as a success. Otherwise it should return a error represented as an `IllegalArgumentException`.

Here are some examples of use.

```
validateAdult[Try](18)
// res7: Try[Int] = Success(value = 18)
validateAdult[Try](8)
// res8: Try[Int] = Failure(
//   exception = java.lang.IllegalArgumentException: Age must be
greater than or equal to 18
// )
type ExceptionOr[A] = Either[Throwable, A]
validateAdult[ExceptionOr](-1)
// res9: Either[Throwable, Int] = Left(
//   value = java.lang.IllegalArgumentException: Age must be
greater than or equal to 18
// )
```

10.6. The Eval Monad

`cats.Eval` is a monad that allows us to abstract over different **models of evaluation**. We first met this concept, also known as evaluation strategies, in Section 4.3. We typically talk of two such models: **eager** and **lazy**, also called **call-by-value** and **call-by-name** respectively. `Eval` also allows for a result to be **memoized**, which gives us **call-by-need** evaluation.

`Eval` is also **stack-safe**, which means we can use it in very deep recursions without blowing up the stack.

10.6.1. Eager, Lazy, Memoized, Oh My!

What do these terms for models of evaluation mean? Let's see some examples.

Let's first look at Scala `vals`. We can see the evaluation model using a computation with a visible side-effect. In the following example, the code to compute the value of `x` happens at the place where it is defined rather than on access. Accessing `x` recalls the stored value without re-running the code.

```
val x = {
  println("Computing X")
  math.random()
}
// Computing X
// x: Double = 0.12372446328627063

x // first access
// res0: Double = 0.12372446328627063
x // second access
// res1: Double = 0.12372446328627063
```

This is an example of call-by-value evaluation:

- the computation is evaluated at the point where it is defined (eager); and
- the computation is evaluated once (memoized).

Let's look at an example using a `def`. The code to compute `y` below is not run until we use it, and is re-run on every access:

```
def y = {
  println("Computing Y")
  math.random()
}

y // first access
// Computing Y
// res2: Double = 0.01513707311373147
y // second access
// Computing Y
// res3: Double = 0.08155577503973321
```

These are the properties of call-by-name evaluation:

- the computation is evaluated at the point of use (lazy); and

- the computation is evaluated each time it is used (not memoized).

Last but not least, `lazy vals` are an example of call-by-need evaluation. The code to compute `z` below is not run until we use it for the first time (lazy). The result is then cached and re-used on subsequent accesses (memoized):

```
lazy val z = {
  println("Computing Z")
  math.random()
}

z // first access
// Computing Z
// res4: Double = 0.7756931106737277
z // second access
// res5: Double = 0.7756931106737277
```

Let's summarize. There are two properties of interest:

- evaluation at the point of definition (eager) versus at the point of use (lazy); and
- values are saved once evaluated (memoized) or not (not memoized).

There are three possible combinations of these properties:

- call-by-value which is eager and memoized;
- call-by-name which is lazy and not memoized; and
- call-by-need which is lazy and memoized.

The final combination, eager and not memoized, is not possible.

10.6.2. Eval's Models of Evaluation

fs `Eval` has three subtypes: `Now`, `Always`, and `Later`. They correspond to call-by-value, call-by-name, and call-by-need respectively. We construct these with three constructor methods,

which create instances of the three classes and return them typed as `Eval`:

```
import cats.Eval
```

```
val now = Eval.now(math.random() + 1000)
// now: Eval[Double] = Now(value = 1000.648061968798)
val always = Eval.always(math.random() + 3000)
// always: Eval[Double] = cats.Always@692f4eaa
val later = Eval.later(math.random() + 2000)
// later: Eval[Double] = cats.Later@ebb01bf
```

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

```
now.value
// res6: Double = 1000.648061968798
always.value
// res7: Double = 3000.906485399325
later.value
// res8: Double = 2000.7074758395897
```

Each type of `Eval` calculates its result using one of the evaluation models defined above. `Eval.now` captures a value *right now*. Its semantics are similar to a `val`—eager and memoized:

```
val x = Eval.now{
  println("Computing X")
  math.random()
}
// Computing X
// x: Eval[Double] = Now(value = 0.18404443716000762)

x.value // first access
// res10: Double = 0.18404443716000762
x.value // second access
// res11: Double = 0.18404443716000762
```

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

```
val y = Eval.always{
  println("Computing Y")
```

```
  math.random()
}
// y: Eval[Double] = cats.Always@25db3188

y.value // first access
// Computing Y
// res12: Double = 0.7506360379891952
y.value // second access
// Computing Y
// res13: Double = 0.2441797695545187
```

Finally, `Eval.later` captures a lazy, memoized computation, similar to a `lazy val`:

```
val z = Eval.later{
  println("Computing Z")
  math.random()
}
// z: Eval[Double] = cats.Later@3b7a3435

z.value // first access
// Computing Z
// res14: Double = 0.18423088162834222
z.value // second access
// res15: Double = 0.18423088162834222
```

The three behaviours are summarized below.

Scala	Cats	Properties
`val`	`Now`	eager, memoized
`def`	`Always`	lazy, not memoized
`lazy val`	`Later`	lazy, memoized

10.6.3. Eval as a Monad

Like all monads, `Eval's` `map` and `flatMap` methods add computations to a chain. In this case, however, the chain is stored

explicitly as a list of functions. The functions aren't run until we call `Eval's value` method to request a result:

```
val greeting = Eval
  .always{ println("Step 1"); "Hello" }
  .map{ str => println("Step 2"); s"$str world" }
// greeting: Eval[String] = cats.Eval$$anon$4@676e8e38

greeting.value
// Step 1
// Step 2
// res16: 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: Eval[Int] = cats.Eval$$anon$4@68ec0e7

ans.value // first access
// Calculating B
// Adding A and B
// res17: Int = 42
ans.value // second access
// Calculating B
// Adding A and B
// res18: Int = 42
```

`Eval` has a `memoize` method that allows us to memoize a chain of computations. The result of the chain up to the call to `memoize` is cached, whereas calculations after the call retain their original semantics:

```
val saying = Eval
  .always{ println("Step 1"); "The cat" }
  .map{ str => println("Step 2"); s"$str sat on" }
  .memoize
  .map{ str => println("Step 3"); s"$str the mat" }
// saying: Eval[String] = cats.Eval$$anon$4@731102e

saying.value // first access
// Step 1
// Step 2
// Step 3
// res19: String = "The cat sat on the mat"
saying.value // second access
// Step 3
// res20: String = "The cat sat on the mat"
```

10.6.4. Trampolining and Eval.defer

One useful property of `Eval` is that its `map` and `flatMap` methods are *trampolined*. This means we can nest calls to `map` and `flatMap` arbitrarily without consuming stack frames. We call this property *"stack safety"*.

For example, consider this function for calculating factorials:

```
def factorial(n: BigInt): BigInt =
  if(n == 1) n else n * factorial(n - 1)
```

It is relatively easy to make this method stack overflow:

```
factorial(50000)
// java.lang.StackOverflowError
//    ...
```

We can rewrite the method using `Eval` to make it stack safe:

```
def factorial(n: BigInt): Eval[BigInt] =
  if(n == 1) {
    Eval.now(n)
```

```
  } else {
    factorial(n - 1).map(_ * n)
  }
```

```
factorial(50000).value
// java.lang.StackOverflowError
//   ...
```

Oops! That didn't work—our stack still blew up! This is because we're still making all the recursive calls to `factorial` before we start working with `Eval`'s `map` method. We can work around this using `Eval.defer`, which takes an existing instance of `Eval` and defers its evaluation. The `defer` method is trampolined like `map` and `flatMap`, so we can use it as a quick way to make an existing operation stack safe:

```
def factorial(n: BigInt): Eval[BigInt] =
  if(n == 1) {
    Eval.now(n)
  } else {
    Eval.defer(factorial(n - 1).map(_ * n))
  }
```

```
factorial(50000).value
// res: A very big value
```

`Eval` is a useful tool to enforce stack safety when working on very large computations and data structures. However, we must bear in mind that trampolining is not free. It avoids consuming stack by creating a chain of function objects on the heap. There are still limits on how deeply we can nest computations, but they are bounded by the size of the heap rather than the stack.

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
  }
```

10.7. The Writer Monad

`cats.data.Writer` is a monad that lets us carry a log along with a computation. We can use it to record messages, errors, or additional data about a computation, and extract the log alongside the final result.

A common use for `Writer` is recording sequences of steps in multi-threaded computations, where standard imperative logging techniques can result in interleaved messages from different contexts. With `Writer` the log for the computation is tied to the result, so we can run concurrent computations without mixing logs.

> **Cats Data Types**
>
> `Writer` is the first data type we've seen from the `cats.data` package. This package provides instances of various type classes that produce useful semantics. Other examples from `cats.data` include the monad transformers that we will see in Chapter 11, and the `Validated` type we will encounter in Chapter 12.

10.7.1. Creating and Unpacking Writers

A `Writer[W, A]` carries two values: a *log* of type `W` and a *result* of type `A`. We can create a `Writer` from values of each type as follows:

```
import cats.data.Writer

Writer(Vector(
  "It was the best of times",
  "it was the worst of times"
), 1859)
// res0: WriterT[Id, Vector[String], Int] = WriterT(
//   run = (Vector("It was the best of times", "it was the worst
of times"), 1859)
// )
```

Notice that the type reported on the console is actually `WriterT[Id, Vector[String], Int]` instead of `Writer[Vector[String], Int]` as we might expect. In the spirit of code reuse, Cats implements `Writer` in terms of another type, `WriterT`. `WriterT` is an example of a new concept called a **monad transformer**, which we will cover in Chapter 11.

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. In the example below we use the `Monoid` instance for `Vector`, which Scala will find on the `Monoid` companion object.

```
import cats.syntax.all.*
```

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

123.pure[Logged]
// res1: WriterT[Id, Vector[String], Int] = WriterT(run =
(Vector(), 123))
```

If we have a log and no result we can create a `Writer[Unit]` using the `tell` syntax.

```
Vector("msg1", "msg2", "msg3").tell
// res2: WriterT[Id, Vector[String], Unit] = WriterT(
//   run = (Vector("msg1", "msg2", "msg3"), ())
// )
```

If we have both a result and a log, we can either use `Writer.apply` or we can use the `writer` syntax.

```
val a = Writer(Vector("msg1", "msg2", "msg3"), 123)
// a: WriterT[Id, Vector[String], Int] = WriterT(
//   run = (Vector("msg1", "msg2", "msg3"), 123)
// )
val b = 123.writer(Vector("msg1", "msg2", "msg3"))
// b: WriterT[Id, Vector[String], Int] = WriterT(
//   run = (Vector("msg1", "msg2", "msg3"), 123)
// )
```

We can extract the result and log from a `Writer` using the `value` and `written` methods respectively:

```
val aResult: Int =
  a.value
// aResult: Int = 123
val aLog: Vector[String] =
  a.written
// aLog: Vector[String] = Vector("msg1", "msg2", "msg3")
```

We can extract both values at the same time using the `run` method:

```
val (log, result) = b.run
// log: Vector[String] = Vector("msg1", "msg2", "msg3")
// result: Int = 123
```

10.7.2. Composing and Transforming Writers

The log in a `Writer` is preserved when we `map` or `flatMap` over it. `flatMap` appends the logs from the source `Writer` and the result of the user's sequencing function. For this reason it's good practice to use a log type that has an efficient append and concatenate operations, such as a `Vector`.

```
val writer1 = for {
  a <- 10.pure[Logged]
  _ <- Vector("a", "b", "c").tell
  b <- 32.writer(Vector("x", "y", "z"))
} yield a + b
// writer1: WriterT[Id, Vector[String], Int] = WriterT(
//   run = (Vector("a", "b", "c", "x", "y", "z"), 42)
// )

writer1.run
// res3: Tuple2[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: WriterT[Id, Vector[String], Int] = WriterT(
//   run = (Vector("A", "B", "C", "X", "Y", "Z"), 42)
// )

writer2.run
// res4: Tuple2[Vector[String], Int] = (
//   Vector("A", "B", "C", "X", "Y", "Z"),
//   42
// )
```

We can transform both log and result simultaneously using `bimap` or `mapBoth`. `bimap` takes two function parameters, one for the log and one for the result. `mapBoth` takes a single function that accepts two parameters.

```
val writer3 = writer1.bimap(
  log => log.map(_.toUpperCase),
  res => res * 100
)
// writer3: WriterT[Id, Vector[String], Int] = WriterT(
//   run = (Vector("A", "B", "C", "X", "Y", "Z"), 4200)
// )

writer3.run
// res5: Tuple2[Vector[String], Int] = (
//   Vector("A", "B", "C", "X", "Y", "Z"),
//   4200
// )

val writer4 = writer1.mapBoth { (log, res) =>
  val log2 = log.map(_ + "!")
  val res2 = res * 1000
  (log2, res2)
}
// writer4: WriterT[Id, Vector[String], Int] = WriterT(
//   run = (Vector("a!", "b!", "c!", "x!", "y!", "z!"), 42000)
// )

writer4.run
// res6: Tuple2[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: WriterT[Id, Vector[String], Int] = WriterT(run =
(Vector(), 42))

writer5.run
// res7: Tuple2[Vector[String], Int] = (Vector(), 42)

val writer6 = writer1.swap
// writer6: WriterT[Id, Int, Vector[String]] = WriterT(
//   run = (42, Vector("a", "b", "c", "x", "y", "z"))
// )

writer6.run
```

```
// res8: Tuple2[Int, Vector[String]] = (
//   42,
//   Vector("a", "b", "c", "x", "y", "z")
// )
```

Exercise: Show Your Working

`Writers` are useful for logging operations in multi-threaded environments. Let's confirm this by computing (and logging) some factorials.

The `factorial` function below computes a factorial and prints out the intermediate steps as it runs. The `slowly` helper function ensures this takes a while to run, even on the very small examples below:

```
def slowly[A](body: => A) =
  try body finally Thread.sleep(100)

def factorial(n: Int): Int = {
  val ans = slowly(if(n == 0) 1 else n * factorial(n - 1))
  println(s"fact $n $ans")
  ans
}
```

Here's the output—a sequence of monotonically increasing values:

```
factorial(5)
// fact 0 1
// fact 1 1
// fact 2 2
// fact 3 6
// fact 4 24
// fact 5 120
// res9: 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.*
import scala.concurrent.duration.*

Await.result(Future.sequence(Vector(
  Future(factorial(5)),
  Future(factorial(5))
)), 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
// fact 4 24
// fact 4 24
// fact 5 120
// fact 5 120
// res: 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.

10.8. The Reader Monad

`cats.data.Reader` is a monad that allows us to sequence operations that depend on some input. Instances of `Reader` wrap up functions of one argument, providing us with useful methods for composing them.

One common use for `Readers` is dependency injection. If we have a number of operations that all depend on some external configuration, we can chain them together using a `Reader` to produce one large operation that accepts the configuration as a parameter and runs our program in the order specified.

10.8.1. Creating and Unpacking Readers

We can create a `Reader[A, B]` from a function `A => B` using the `Reader.apply` constructor:

```
import cats.data.Reader

final case class Cat(name: String, favoriteFood: String)

val catName: Reader[Cat, String] =
  Reader(cat => cat.name)
// catName: Kleisli[Id, Cat, String] = Kleisli(
//   run = repl.MdocSession$MdocApp$
$Lambda$13474/0x0000000803b08040@22fb5ef5
// )
```

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: String = "Garfield"
```

So far so simple, but what advantage do `Readers` give us over the raw functions?

10.8.2. Composing Readers

The power of `Readers` comes from their `map` and `flatMap` methods, which represent different kinds of function composition. We typically create a set of `Readers` that accept the same type of configuration, combine them with `map` and `flatMap`, and then call `run` to inject the config at the end.

The `map` method simply extends the computation in the `Reader` by passing its result through a function.

```
val greetKitty: Reader[Cat, String] =
  catName.map(name => s"Hello ${name}")
```

```
greetKitty.run(Cat("Heathcliff", "junk food"))
// res1: String = "Hello Heathcliff"
```

The `flatMap` method is more interesting. It allows us to combine readers that depend on the same input type. To illustrate this, let's extend our greeting example to also feed the cat.

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

val greetAndFeed: Reader[Cat, String] =
  for {
    greet <- greetKitty
    feed  <- feedKitty
  } yield s"$greet. $feed."
```

```
greetAndFeed(Cat("Garfield", "lasagne"))
// res2: String = "Hello Garfield. Have a nice bowl of lasagne."
greetAndFeed(Cat("Heathcliff", "junk food"))
// res3: String = "Hello Heathcliff. Have a nice bowl of junk
food."
```

Exercise: Hacking on Readers

The classic use of `Readers` is to build programs that accept a configuration as a parameter. Let's ground this with a complete example of a simple login system. Our configuration will consist of two databases: a list of valid users and a list of their passwords:

```
final 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.

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] =
  ???
```

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] =
  ???
```

You should be able to use `checkLogin` as follows:

```
val users = Map(
  1 -> "dade",
  2 -> "kate",
  3 -> "margo"
)

val passwords = Map(
  "dade"  -> "zerocool",
  "kate"  -> "acidburn",
  "margo" -> "secret"
)

val db = Db(users, passwords)
```

```
checkLogin(1, "zerocool").run(db)
// res6: Boolean = true
checkLogin(4, "davinci").run(db)
// res7: Boolean = false
```

10.8.3. When to Use Readers?

`Readers` provide a tool for doing dependency injection. We write steps of our program as instances of `Reader`, chain them together with `map` and `flatMap`, and build a function that accepts the dependency as input.

There are many ways of implementing dependency injection in Scala, from simple techniques like methods with multiple parameter lists, through implicit parameters and type classes, to complex techniques like the cake pattern and DI frameworks.

`Readers` are most useful in situations where:

- we are constructing a program that can easily be represented by a function;
- we need to defer injection of a known parameter or set of parameters;
- we want to be able to test parts of the program in isolation.

By representing the steps of our program as `Readers` we can test them as easily as pure functions, plus we gain access to the `map` and `flatMap` combinators.

For more complicated problems where we have lots of dependencies, or where a program isn't easily represented as a pure function, other dependency injection techniques tend to be more appropriate.

> **Kleisli Arrows**
>
> You may have noticed from console output that `Reader` is implemented in terms of another type called `Kleisli`. **Kleisli arrows** provide a more general form of `Reader` that

generalise over the type constructor of the result type. We will encounter Kleislis again in Chapter 11.

10.9. The State Monad

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

10.9.1. Creating and Unpacking State

Boiled down to their simplest form, instances of `State[S, A]` represent functions of type `S => (S, A)`. `S` is the type of the state and `A` is the type of the result. In the example below, the state has type `Int`, and we return a `String` result computed from the state.

```
import cats.data.State

val a = State[Int, String]{ state =>
  (state, s"The state is $state")
}
// a: IndexedStateT[[A >: Nothing <: Any] =>> Eval[A], Int, Int,
String] = cats.data.IndexedStateT@24cf3bad
```

State and IndexedStateT

You may have noticed the console output reports a type of `IndexedStateT` when we create an instance of `State`. Just

like with `Writer` and `Reader`, `State` is defined as a type alias of a more complicated type `IndexedStateT`:

```
type State[S, A] = IndexedStateT[Eval, S, S, A]
```

We can ignore this more complicated type.

In other words, an instance of `State` is a function that does two things:

- transforms an input state to an output state;
- computes a result.

We can "run" our monad by supplying an initial state. `State` provides three methods—`run`, `runS`, and `runA`—that return different combinations of state and result. Each method returns an instance of `Eval`, which `State` uses to maintain stack safety. We call the `value` method as usual to extract the actual result:

```
// Get the state and the result:
val (state, result) = a.run(10).value
// state: Int = 10
// result: String = "The state is 10"

// Get the state, ignore the result:
val justTheState = a.runS(10).value
// justTheState: Int = 10

// Get the result, ignore the state:
val justTheResult = a.runA(10).value
// justTheResult: String = "The state is 10"
```

10.9.2. Composing and Transforming State

As we've seen with `Reader` and `Writer`, the power of the `State` monad comes from combining instances. The `map` and `flatMap` methods thread the state from one instance to another. Each

individual instance represents an atomic state transformation, and their combination represents a complete sequence of changes:

```
val step1 = State[Int, String]{ num =>
  val ans = num + 1
  (ans, s"Result of step1: $ans")
}

val step2 = State[Int, String]{ num =>
  val ans = num * 2
  (ans, s"Result of step2: $ans")
}

val both = for {
  a <- step1
  b <- step2
} yield (a, b)
```

When we run this program we get the result of applying each step in sequence. State is threaded from step to step even though we don't interact with it in the for comprehension.

```
val (state, result) = both.run(20).value
// state: Int = 42
// result: Tuple2[String, String] = (
//   "Result of step1: 21",
//   "Result of step2: 42"
// )
```

The general model for using the `State` monad is to represent each step of a computation as an instance and compose the steps using the standard monad operators. Cats provides several convenience constructors for creating primitive steps:

- `get` extracts the state as the result;
- `set` updates the state and returns unit as the result;
- `pure` ignores the state and returns a supplied result;
- `inspect` extracts the state via a transformation function;
- `modify` updates the state using an update function.

```
State.get[Int].run(10).value
// res0: Tuple2[Int, Int] = (10, 10)

State.set[Int](30).run(10).value
// res1: Tuple2[Int, Unit] = (30, ())

State.pure[Int, String]("Result").run(10).value
// res2: Tuple2[Int, String] = (10, "Result")

State.inspect[Int, String](x => s"${x}!").run(10).value
// res3: Tuple2[Int, String] = (10, "10!")

State.modify[Int](_ + 1).run(10).value
// res4: Tuple2[Int, Unit] = (11, ())
```

We can assemble these building blocks using a for comprehension. We typically ignore the result of intermediate stages that only represent transformations on the state.

```
import cats.data.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)
```

As we expect, the result is the composition of the individual stages.

```
val (state, result) = program.run(1).value
// state: Int = 3
// result: Tuple3[Int, Int, Int] = (1, 2, 3000)
```

Exercise: Post-Order Calculator

The `State` monad allows us to implement simple interpreters for complex expressions, passing the values of mutable registers along

with the result. We can see a simple example of this by implementing a calculator for post-order integer arithmetic expressions.

In case you haven't heard of post-order expressions before (don't worry if you haven't), they are a mathematical notation where we write the operator *after* its operands. So, for example, instead of writing `1 + 2` we would write:

```
1 2 +
```

Although post-order expressions are difficult for humans to read, they are easy to evaluate in code. All we need to do is traverse the symbols from left to right, carrying a *stack* of operands with us as we go:

- when we see a number, we push it onto the stack;
- when we see an operator, we pop two operands off the stack, operate on them, and push the result in their place.

This allows us to evaluate complex expressions without using parentheses. For example, we can evaluate `(1 + 2) * 3)` as follows:

```
1 2 + 3 * // see 1, push onto stack
2 + 3 *   // see 2, push onto stack
+ 3 *     // see +, pop 1 and 2 off of stack,
          //        push (1 + 2) = 3 in their place
3 3 *     // see 3, push onto stack
3 *       // see 3, push onto stack
*         // see *, pop 3 and 3 off of stack,
          //        push (3 * 3) = 9 in their place
```

Let's write an interpreter for these expressions. We can parse each symbol into a `State` instance representing a transformation on the stack and an intermediate result. The `State` instances can be threaded together using `flatMap` to produce an interpreter for any sequence of symbols.

Start by writing a function `evalOne` that parses a single symbol into an instance of `State`. Use the code below as a template. Don't worry about error handling for now—if the stack is in the wrong configuration, it's OK to throw an exception.

```
import cats.data.State

type CalcState[A] = State[List[Int], A]

def evalOne(sym: String): CalcState[Int] = ???
```

If this seems difficult, think about the basic form of the `State` instances you're returning. Each instance represents a functional transformation from a stack to a pair of a stack and a result. You can ignore any wider context and focus on just that one step:

```
State[List[Int], Int] { oldStack =>
  val newStack = someTransformation(oldStack)
  val result   = someCalculation
  (newStack, result)
}
```

Feel free to write your `Stack` instances in this form or as sequences of the convenience constructors we saw above.

`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
// res8: 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: IndexedStateT[[A >: Nothing <: Any] =>> Eval[A],
List[Int], List[Int], Int] = cats.data.IndexedStateT@4ad4991f

program.runA(Nil).value
// res9: 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] =
  ???
```

We can use `evalAll` to conveniently evaluate multi-stage expressions:

```
val multistageProgram = evalAll(List("1", "2", "+", "3", "*"))
// multistageProgram: IndexedStateT[[A >: Nothing <: Any] =>>
Eval[A], List[Int], List[Int], Int] =
cats.data.IndexedStateT@1edc9ba3

multistageProgram.runA(Nil).value
// res11: Int = 9
```

Because `evalOne` and `evalAll` both return instances of `State`, we can thread these results together using `flatMap`. `evalOne` produces a simple stack transformation and `evalAll` produces a complex one, but they're both pure functions and we can use them in any order as many times as we like:

```
val biggerProgram = for {
  _   <- evalAll(List("1", "2", "+"))
  _   <- evalAll(List("3", "4", "+"))
  ans <- evalOne("*")
} yield ans
// biggerProgram: IndexedStateT[[A >: Nothing <: Any] =>>
```

```
Eval[A], List[Int], List[Int], Int] =
cats.data.IndexedStateT@3c478f3a

biggerProgram.runA(Nil).value
// res12: Int = 21
```

Complete the exercise by implementing an `evalInput` function that splits an input `String` into symbols, calls `evalAll`, and runs the result with an initial stack.

10.10. Defining Custom Monads

We can define a `Monad` for a custom type by providing implementations of three methods: `flatMap`, `pure`, and a method we haven't seen yet called `tailRecM`. Here is an implementation of `Monad` for `Option` as an example:

```
import cats.Monad
import scala.annotation.tailrec

val optionMonad = new Monad[Option] {
  def flatMap[A, B](opt: Option[A])
      (fn: A => Option[B]): Option[B] =
    opt.flatMap(fn)

  def pure[A](opt: A): Option[A] =
    Some(opt)

  @tailrec
  def tailRecM[A, B](a: A)(fn: A => Option[Either[A, B]]):
Option[B] = {
    fn(a) match {
      case None           => None
      case Some(Left(a1)) => tailRecM(a1)(fn)
      case Some(Right(b)) => Some(b)
    }
  }
}
```

The `tailRecM` method is an optimisation used in Cats to limit the amount of stack space consumed by nested calls to `flatMap`. The technique comes from a 2015 paper by PureScript creator Phil Freeman [29]. The method should recursively call itself until the result of `fn` returns a `Right`.

To motivate its use let's use the following example: suppose we want to write a method that calls a function until the function indicates it should stop. The function will return a monad instance because, as we know, monads represent sequencing and many monads have some notion of stopping.

We can write this method in terms of `flatMap`.

```
import cats.syntax.all.*

def retry[F[_]: Monad, A](start: A)(f: A => F[A]): F[A] =
  f(start).flatMap{ a =>
    retry(a)(f)
  }
```

Unfortunately it is not stack-safe. It works for small input.

```
retry(100)(a => if(a == 0) None else Some(a - 1))
// res1: Option[Int] = None
```

but if we try large input we get a `StackOverflowError`.

```
retry(100000)(a => if(a == 0) None else Some(a - 1))
// KABLOOIE!!!!
```

We can instead rewrite this method using `tailRecM`.

```
def retryTailRecM[F[_]: Monad, A](start: A)(f: A => F[A]): F[A] =
  Monad[F].tailRecM(start){ a =>
    f(a).map(a2 => Left(a2))
  }
```

Now it runs successfully no matter how many time we recurse.

```
retryTailRecM(100000)(a => if(a == 0) None else Some(a - 1))
// res2: Option[Int] = None
```

It's important to note that we have to explicitly call `tailRecM`. There isn't a code transformation that will convert non-tail recursive code into tail recursive code that uses `tailRecM`. However there are several utilities provided by the `Monad` type class that makes these kinds of methods easier to write. For example, we can rewrite `retry` in terms of `iterateWhileM` and we don't have to explicitly call `tailRecM`.

```
def retryM[F[_]: Monad, A](start: A)(f: A => F[A]): F[A] =
  start.iterateWhileM(f)(a => true)
```

This implementation is stack-safe.

```
retryM(100000)(a => if(a == 0) None else Some(a - 1))
// res3: Option[Int] = None
```

We'll see more methods that use `tailRecM` in Section 13.1.

All of the built-in monads in Cats have tail-recursive implementations of `tailRecM`, although writing one for custom monads can be a challenge… as we shall see.

Exercise: Branching out Further with Monads

Let's write a `Monad` for our `Tree` data type from last chapter. Here's the type again:

```
enum Tree[+A] {
  case Branch(left: Tree[A], right: Tree[A])
  case Leaf(value: A) extends Tree[A]
}
object Tree {
  def branch[A](left: Tree[A], right: Tree[A]): Tree[A] =
    Branch(left, right)
```

```
  def leaf[A](value: A): Tree[A] =
    Leaf(value)
}
```

Verify that the code works on instances of `Branch` and `Leaf`, and that the `Monad` provides `Functor`-like behaviour for free.

Also verify that having a `Monad` in scope allows us to use for comprehensions, despite the fact that we haven't directly implemented `flatMap` or `map` on `Tree`.

Don't feel you have to make `tailRecM` tail-recursive. Doing so is quite difficult. We've included both tail-recursive and non-tail-recursive implementations in the solutions so you can check your work.

10.11. Conclusions

In this chapter we've seen monads up-close. We saw that `flatMap` can be viewed as an operator for sequencing computations, dictating the order in which operations must happen. From this viewpoint, `Option` represents a computation that can fail without an error message, `Either` represents computations that can fail with a message, `List` represents multiple possible results, and `Future` represents a computation that may produce a value at some point in the future.

We've also seen some of the custom types and data structures that Cats provides, including `Id`, `Reader`, `Writer`, and `State`. These cover a wide range of use cases.

Finally, in the unlikely event that we have to implement a custom monad, we've learned about defining our own instance using `tailRecM`. `tailRecM` is an odd wrinkle that is a concession to building a functional programming library that is stack-safe by default. We don't need to understand `tailRecM` to understand

monads, but having it around gives us benefits of which we can be grateful when writing monadic code.

The Essence of Functional Programming [92] introduced monads to functional programming. It describes monads in terms of `bind` and `unit`, which in Scala we call `flatMap` and `pure` respectively. It has several examples of interpreters built using monads, and relates monads to continuation-passing style, which we first met in Section 6.3.3.

11. Monad Transformers

Monads are like burritos[76], which means that once you acquire a taste, you'll find yourself returning to them again and again. This is not without issues. As burritos can bloat the waist, monads can bloat the code base through nested for-comprehensions.

Imagine we are interacting with a database. We want to look up a user record. The user may or may not be present, so we return an `Option[User]`. Our communication with the database could fail for many reasons (network issues, authentication problems, and so on), so this result is wrapped up in an `Either`, giving us a final result of `Either[Error, Option[User]]`.

To use this value we must nest `flatMap` calls (or equivalently, for-comprehensions):

```
def lookupUserName(id: Long): Either[Error, Option[String]] =
  for {
    optUser <- lookupUser(id)
  } yield {
    for { user <- optUser } yield user.name
  }
```

This quickly becomes very tedious.

11.1. Composing Monads

A question arises. Given two arbitrary monads, can we combine them in some way to make a single monad? That is, do monads *compose*? We can try to write the code but we soon hit problems:

[76] http://blog.plover.com/prog/burritos.html

```
// Hypothetical example. This won't actually compile:
def compose[M1[_]: Monad, M2[_]: Monad] = {
  type Composed[A] = M1[M2[A]]

  new Monad[Composed] {
    def pure[A](a: A): Composed[A] =
      a.pure[M2].pure[M1]

    def flatMap[A, B](fa: Composed[A])
        (f: A => Composed[B]): Composed[B] =
      // Problem! How do we write flatMap?
      ???
  }
}
```

It is impossible to write a general definition of `flatMap` without knowing something about `M1` or `M2`. However, if we *do* know something about one or other monad, we can typically complete this code. For example, if we fix `M2` above to be `Option`, a definition of `flatMap` comes to light:

```
def flatMap[A, B](fa: Composed[A])
    (f: A => Composed[B]): Composed[B] =
  fa.flatMap(_.fold[Composed[B]](None.pure[M1])(f))
```

Notice that the definition above makes use of `None`, an `Option`-specific concept that doesn't appear in the general `Monad` interface. We need this extra detail to combine `Option` with other monads. Similarly, there are things about other monads that help us write composed `flatMap` methods for them. This is the idea behind monad transformers: Cats defines transformers for a variety of monads, each providing the extra knowledge we need to compose that monad with others. Let's look at some examples.

11.2. A Transformative Example

Cats provides transformers for many monads, each named with a `T` suffix: `EitherT` composes `Either` with other monads, `OptionT` composes `Option`, and so on.

Here's an example that uses `OptionT` to compose `List` and `Option`. We can use `OptionT[List, A]`, aliased to `ListOption[A]` for convenience, to transform a `List[Option[A]]` into a single monad:

```
import cats.data.OptionT

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

Note how we build `ListOption` from the inside out: we pass `List`, the type of the outer monad, as a parameter to `OptionT`, the transformer for the inner monad.

We can create instances of `ListOption` using the `OptionT` constructor, or more conveniently using `pure`:

```
import cats.syntax.all.*

val result1: ListOption[Int] = OptionT(List(Option(10)))
// result1: OptionT[List, Int] = OptionT(value = List(Some(value
= 10)))

val result2: ListOption[Int] = 32.pure[ListOption]
// result2: OptionT[List, Int] = OptionT(value = List(Some(value
= 32)))
```

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

```
result1.flatMap { (x: Int) =>
  result2.map { (y: Int) =>
    x + y
  }
}
```

```
// res1: OptionT[List, Int] = OptionT(value = List(Some(value =
42)))
```

This is the basis of all monad transformers. The combined `map` and `flatMap` methods allow us to use both component monads without having to recursively unpack and repack values at each stage in the computation. Now let's look at the API in more depth.

11.3. Monad Transformers in Cats

Each monad transformer is a data type, defined in `cats.data`[77], that allows us to *wrap* stacks of monads to produce new monads. We use the monads we've built via the `Monad` type class. The main concepts we have to cover to understand monad transformers are:

- the available transformer classes;
- how to build stacks of monads using transformers;
- how to construct instances of a monad stack; and
- how to pull apart a stack to access the wrapped monads.

11.3.1. The Monad Transformer Classes

By convention, in Cats a monad `Foo` will have a transformer class called `FooT`. In fact, many monads in Cats are defined by combining a monad transformer with the `Id` monad. Concretely, some of the available instances are:

- `cats.data.OptionT` for `Option`;
- `cats.data.EitherT` for `Either`;
- `cats.data.ReaderT` for `Reader`;
- `cats.data.WriterT` for `Writer`;
- `cats.data.StateT` for `State`;

[77] http://typelevel.org/cats/api/cats/data/

- `cats.data.IdT` for the `Id` monad.

> **Kleisli Arrows**
>
> In Section 10.8 we mentioned that the `Reader` monad was a specialisation of a more general concept called a "kleisli arrow", represented in Cats as `cats.data.Kleisli`.
>
> We can now reveal that `Kleisli` and `ReaderT` are, in fact, the same thing! `ReaderT` is actually a type alias for `Kleisli`. Hence, we were creating `Readers` last chapter and seeing `Kleislis` on the console.

11.3.2. Building Monad Stacks

All of these monad transformers follow the same convention. The transformer itself represents the *inner* monad in a stack, while the first type parameter specifies the outer monad. The remaining type parameters are the types we've used to form the corresponding monads.

For example, our `ListOption` type above is an alias for `OptionT[List, A]` but the result is effectively a `List[Option[A]]`. In other words, we build monad stacks from the inside out:

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

Many monads and all transformers have at least two type parameters, so we often end up defining 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 can use a type alias to convert the type constructor to the correct shape.

```
// Alias Either to a type constructor with one parameter:
type ErrorOr[A] = Either[String, A]

// Build our final monad stack using OptionT:
type ErrorOrOption[A] = OptionT[ErrorOr, A]
```

`ErrorOrOption` is a monad, just like `ListOption`. We can use `pure`, `map`, and `flatMap` as usual to create and transform instances.

```
val a = 10.pure[ErrorOrOption]
// a: OptionT[ErrorOr, Int] = OptionT(value = Right(value =
Some(value = 10)))
val b = 32.pure[ErrorOrOption]
// b: OptionT[ErrorOr, Int] = OptionT(value = Right(value =
Some(value = 32)))

val c = a.flatMap(x => b.map(y => x + y))
// c: OptionT[ErrorOr, Int] = OptionT(value = Right(value =
Some(value = 42)))
```

Things become even more confusing when we want to stack three or more monads.

For example, let's create a `Future` of an `Either` of `Option`. Once again we build this from the inside out with an `OptionT` of an `EitherT` of `Future`. However, defining this in one line is harder because `EitherT` has three type parameters:

```
final case class EitherT[F[_], E, A](stack: F[Either[E, A]]) {
  // etc...
}
```

The three type parameters are as follows:

- `F[_]` is the outer monad in the stack (`Either` is the inner);
- `E` is the error type for the `Either`;
- `A` is the result type for the `Either`.

The simplest approach is to create an alias for `EitherT` that fixes `Future` and `Error` but allows `A` to vary.

```
import scala.concurrent.Future

type FutureEither[A] = EitherT[Future, String, A]

type FutureEitherOption[A] = OptionT[FutureEither, A]
```

Our mammoth stack now composes three monads and our `map` and `flatMap` methods cut through three layers of abstraction.

```
import scala.concurrent.Await
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration.*

val futureEitherOr: FutureEitherOption[Int] =
  for {
    a <- 10.pure[FutureEitherOption]
    b <- 32.pure[FutureEitherOption]
  } yield a + b
```

Type Lambdas

If you frequently find yourself defining multiple type aliases when building monad stacks, you may want to try Scala 3's type lambdas. In Scala 2.13 you can use the Kind Projector[78] compiler plugin to get the same functionality with slightly different syntax.

Type lambdas make it more compact to define partially applied type constructors. For example we can write

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

[78] https://github.com/typelevel/kind-projector

instead of the longer (but perhaps clearer!)

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

type FutureEitherOption[A] = OptionT[FutureEither, A]
```

11.3.3. Constructing and Unpacking Instances

As we saw above, we can create transformed monad stacks using the relevant monad transformer's `apply` method or the usual `pure` syntax[79].

```
// Create using apply:
val errorStack1 = OptionT[ErrorOr, Int](Right(Some(10)))
// errorStack1: OptionT[ErrorOr, Int] = OptionT(
//   value = Right(value = Some(value = 10))
// )

// Create using pure:
val errorStack2 = 32.pure[ErrorOrOption]
// errorStack2: OptionT[ErrorOr, Int] = OptionT(
//   value = Right(value = Some(value = 32))
// )
```

Once we've finished with a monad transformer stack, we can unpack it using its `value` method. This returns the untransformed stack. We can then manipulate the individual monads in the usual way.

```
// Extracting the untransformed monad stack:
errorStack1.value
// res3: Either[String, Option[Int]] = Right(value = Some(value =
10))
```

[79] Cats provides an instance of `MonadError` for `EitherT`, allowing us to create instances using `raiseError` as well as `pure`.

```
// Mapping over the Either in the stack:
errorStack2.value.map(_.getOrElse(-1))
// res4: Either[String, Int] = Right(value = 32)
```

Each call to `value` unpacks a single monad transformer. We may need more than one call to completely unpack a large stack. For example, to `Await` the `FutureEitherOption` stack above, we need to call `value` twice.

```
futureEitherOr
// res5: OptionT[FutureEither, Int] = OptionT(
//   value = EitherT(value = Future(Success(Right(Some(42)))))
// )

val intermediate = futureEitherOr.value
// intermediate: EitherT[[T >: Nothing <: Any] =>> Future[T],
String, Option[Int]] = EitherT(
//   value = Future(Success(Right(Some(42))))
// )

val stack = intermediate.value
// stack: Future[Either[String, Option[Int]]] =
Future(Success(Right(Some(42))))

Await.result(stack, 1.second)
// res6: Either[String, Option[Int]] = Right(value = Some(value =
42))
```

11.3.4. Default Instances

Many monads in Cats are defined using the corresponding transformer and the `Id` monad. This is reassuring as it confirms that the APIs for monads and transformers are identical. `Reader`, `Writer`, and `State` are all defined in this way:

```
type Reader[E, A] = ReaderT[Id, E, A] // = Kleisli[Id, E, A]
type Writer[W, A] = WriterT[Id, W, A]
type State[S, A]  = StateT[Id, S, A]
```

In other cases monad transformers are defined separately to their corresponding monads. In these cases, the methods of the transformer tend to mirror the methods on the monad. For example, `OptionT` defines `getOrElse`, and `EitherT` defines `fold`, `bimap`, `swap`, and other useful methods.

11.3.5. Usage Patterns

Widespread use of monad transformers is sometimes difficult because they fuse monads together in predefined ways. Without careful thought, we can end up having to unpack and repack monads in different configurations to operate on them in different contexts.

The most practical solution is to forego monad transformers entirely, and use a single "super-monad" that combines several useful monads into one. This is the approach taken by so-called IO monads, such as Cats Effect[80]. These monad types usually provide asynchronicity, error-handling, and more in one type.

A similar approach is to create a single "super stack" and sticking to it throughout our code base. This works if the code is simple and largely uniform in nature. For example, in a web application, we could decide that all request handlers are asynchronous and all can fail with the same set of HTTP error codes. We could design a custom ADT representing the errors and use a fusion `Future` and `Either` everywhere in our code:

```
sealed abstract class HttpError
final case class NotFound(item: String) extends HttpError
final case class BadRequest(msg: String) extends HttpError
// etc...

type FutureEither[A] = EitherT[Future, HttpError, A]
```

[80] https://typelevel.org/cats-effect/

The “super stack” approach starts to fail in larger, more heterogeneous code bases where different stacks make sense in different contexts. Another design pattern that makes more sense in these contexts uses monad transformers as local “glue code”. We expose untransformed stacks at module boundaries, transform them to operate on them locally, and untransform them before passing them on. This allows each module of code to make its own decisions about which transformers to use:

```
import cats.data.Writer

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

// Methods generally return untransformed stacks:
def parseNumber(str: String): Logged[Option[Int]] =
  util.Try(str.toInt).toOption match {
    case Some(num) => Writer(List(s"Read $str"), Some(num))
    case None      => Writer(List(s"Failed on $str"), None)
  }

// Consumers use monad transformers locally to simplify
composition:
def addAll(a: String, b: String, c: String): Logged[Option[Int]]
= {
  import cats.data.OptionT

  val result = for {
    a <- OptionT(parseNumber(a))
    b <- OptionT(parseNumber(b))
    c <- OptionT(parseNumber(c))
  } yield a + b + c

  result.value
}
```

```
// This approach doesn't force OptionT on other users' code:
val result1 = addAll("1", "2", "3")
// result1: WriterT[Id, List[String], Option[Int]] = WriterT(
//   run = (List("Read 1", "Read 2", "Read 3"), Some(value = 6))
// )
val result2 = addAll("1", "a", "3")
// result2: WriterT[Id, List[String], Option[Int]] = WriterT(
```

```
//   run = (List("Read 1", "Failed on a"), None)
// )
```

Unfortunately, there aren't one-size-fits-all approaches to working with monad transformers. The best approach for you may depend on a lot of factors: the size and experience of your team, the complexity of your code base, and so on. You may need to experiment and gather feedback from colleagues to determine whether monad transformers are a good fit.

Exercise: Monads: Transform and Roll Out

The Autobots, well-known robots in disguise, frequently send messages during battle requesting the power levels of their team mates. This helps them coordinate strategies and launch devastating attacks. The message sending method looks like this:

```
def getPowerLevel(autobot: String): Response[Int] =
  ???
```

Transmissions take time in Earth's viscous atmosphere, and messages are occasionally lost due to satellite malfunction or sabotage by pesky Decepticons[81]. `Responses` are therefore represented as a stack of monads:

```
type Response[A] = Future[Either[String, A]]
```

Optimus Prime is getting tired of the nested for comprehensions in his neural matrix. Help him by rewriting `Response` using a monad transformer.

Now test the code by implementing `getPowerLevel` to retrieve data from a set of imaginary allies. Here's the data we'll use:

[81]It is a well known fact that Autobot neural nets are implemented in Scala. Decepticon brains are, of course, dynamically typed.

```
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.

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] =
  ???
```

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 =
  ???
```

You should be able to use `report` as follows:

```
tacticalReport("Jazz", "Bumblebee")
// res12: String = "Jazz and Bumblebee need a recharge."
tacticalReport("Bumblebee", "Hot Rod")
// res13: String = "Bumblebee and Hot Rod are ready to roll out!"
tacticalReport("Jazz", "Ironhide")
// res14: String = "Comms error: Ironhide unreachable"
```

11.4. Conclusions

In this chapter we introduced monad transformers, which eliminate the need for nested for comprehensions and pattern matching when working with "stacks" of nested monads.

Each monad transformer, such as `FutureT`, `OptionT` or `EitherT`, provides the code needed to merge its related monad with other monads. The transformer is a data structure that wraps a monad stack, equipping it with `map` and `flatMap` methods that unpack and repack the whole stack.

The type signatures of monad transformers are written from the inside out, so an `EitherT[Option, String, A]` is a wrapper for an `Option[Either[String, A]]`. It is often useful to use type aliases when writing transformer types for deeply nested monads.

With this look at monad transformers, we have now covered everything we need to know about monads and the sequencing of computations using `flatMap`. In the next chapter we will switch tack and discuss two new type classes, `Semigroupal` and `Applicative`, that support new kinds of operation such as `zipping` independent values within a context.

12. Semigroupal and Applicative

In previous chapters we saw how functors and monads let us sequence operations using `map` and `flatMap`. While functors and monads are both immensely useful abstractions, there are certain types of program flow that they cannot represent.

One such example is form validation. When we validate a form we want to return *all* the errors to the user, not stop on the first error we encounter. If we model this with a monad like `Either`, we fail fast and lose errors. For example, consider `parseInt` below that represents errors with `Either`.

```
import cats.syntax.all.*

def parseInt(str: String): Either[String, Int] =
  Either.catchOnly[NumberFormatException](str.toInt).
    leftMap(_ => s"Couldn't read $str")
```

Uses of `parseInt` fail on the first call and don't go any further.

```
for {
  a <- parseInt("a")
  b <- parseInt("b")
  c <- parseInt("c")
} yield (a + b + c)
// res0: Either[String, Int] = Left(value = "Couldn't read a")
```

Another example is the concurrent evaluation of `Futures`. If we have several long-running independent tasks, it makes sense to execute them concurrently. However, monadic comprehension only allows us to run them in sequence. `map` and `flatMap` aren't quite capable of capturing what we want because they make the assumption that each computation is *dependent* on the previous one:

```
// context2 is dependent on value1:
context1.flatMap(value1 => context2)
```

The calls to `parseInt` and `Future.apply` above are *independent* of one another, but `map` and `flatMap` can't exploit this. We need a weaker construct—one that doesn't guarantee sequencing—to achieve the result we want. In this chapter we will look at three type classes that support this pattern:

- `Semigroupal` encompasses the notion of composing pairs of contexts. Cats provides syntax that makes use of `Semigroupal` and `Functor` to allow users to sequence functions with multiple arguments.
- `Parallel` converts types with a `Monad` instance to a related type with a `Semigroupal` instance.
- `Applicative` extends `Semigroupal` and `Functor`. It provides a way of applying functions to parameters within a context. `Applicative` is the source of the `pure` method we introduced in Chapter 10.

Applicatives are often formulated in terms of function application, instead of the semigroupal formulation that is emphasised in Cats. This alternative formulation provides a link to much of the published research, and to other languages such as Haskell. We'll take a look at different formulations of Applicative, as well as the relationships between `Semigroupal`, `Functor`, `Applicative`, and `Monad`, towards the end of the chapter.

12.1. Semigroupal

`cats.Semigroupal` is a type class that allows us to combine contexts. If we have two objects of type `F[A]` and `F[B]`, a

`Semigroupal[F]` allows us to combine them to form an `F[(A, B)]`. Its definition in Cats is

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

The parameters `fa` and `fb` are independent of one another: we can compute them in either order before passing them to `product`. This is in contrast to `flatMap`, which imposes a strict order on its parameters. This gives us more freedom when defining instances of `Semigroupal` than we get when defining `Monads`.

12.1.1. Joining Two Contexts

While `Semigroup` allows us to join values, `Semigroupal` allows us to join contexts. Let's join some `Options` as an example:

```
import cats.Semigroupal

Semigroupal[Option].product(Some(123), Some("abc"))
// res1: Option[Tuple2[Int, String]] = Some(value = (123, "abc"))
```

If both parameters are instances of `Some`, we end up with a tuple of the values within. If either parameter evaluates to `None`, the entire result is `None`:

```
Semigroupal[Option].product(None, Some("abc"))
// res2: Option[Tuple2[Nothing, String]] = None
Semigroupal[Option].product(Some(123), None)
// res3: Option[Tuple2[Int, Nothing]] = None
```

12.1.2. Joining Three or More Contexts

The companion object for `Semigroupal` defines a set of methods on top of `product`. For example, the methods `tuple2` through `tuple22` generalise `product` to different arities:

```
Semigroupal.tuple3(Option(1), Option(2), Option(3))
// res4: Option[Tuple3[Int, Int, Int]] = Some(value = (1, 2, 3))
Semigroupal.tuple3(Option(1), Option(2), Option.empty[Int])
// res5: Option[Tuple3[Int, Int, Int]] = None
```

The methods `map2` through `map22` apply a user-specified function to the values inside 2 to 22 contexts:

```
Semigroupal.map3(Option(1), Option(2), Option(3))(_ + _ + _)
// res6: Option[Int] = Some(value = 6)

Semigroupal.map2(Option(1), Option.empty[Int])(_ + _)
// res7: Option[Int] = None
```

There are also methods `contramap2` through `contramap22` and `imap2` through `imap22`, that require instances of `Contravariant` and `Invariant` respectively.

12.1.3. Semigroupal Laws

There is only one law for `Semigroupal`: the `product` method must be associative.

```
product(a, product(b, c)) == product(product(a, b), c)
```

12.2. Semigroupal Syntax

Cats' syntax provides shorthands for the methods described above.[82]

Below is an example of the `tupled` syntax method applied to a tuple of `Options`. It uses the `Semigroupal` for `Option` to zip the values inside the `Options`, creating a single `Option` of a tuple.

```
import cats.syntax.all.*

(Option(123), Option("abc")).tupled
// res8: Option[Tuple2[Int, String]] = Some(value = (123, "abc"))
```

We can use the same trick on tuples of up to 22 values. Cats defines a separate `tupled` method for each arity.

```
(Option(123), Option("abc"), Option(true)).tupled
// res9: Option[Tuple3[Int, String, Boolean]] = Some(
//   value = (123, "abc", true)
// )
```

In addition to `tupled`, Cats' provides a method called `mapN` that accepts an implicit `Functor` and a function of the correct arity to combine the values. Let's start with the following case class.

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

We can use `mapN` to convert optional values into an instance of the case class as shown below.

```
(
  Option("Garfield"),
  Option(1978),
```

[82]Some of this syntax is defined for instances of the `cats.Apply` typeclass. Almost all instances of `Semigroupal` are also instances of `Apply`, so the distinction is not particularly important in practice.

```
  Option("Orange & black")
).mapN(Cat.apply)
// res10: Option[Cat] = Some(
//   value = Cat(name = "Garfield", born = 1978, color = "Orange
& black")
// )
```

Of all the methods mentioned here, it is most common to use `mapN`.

Internally `mapN` uses the `Semigroupal` to extract the values from the `Option` and the `Functor` to apply the values to the function.

12.2.1. Fancy Functors and Apply Syntax

Cats' syntax also has `contramapN` and `imapN` methods that accept Contravariant and Invariant functors (see Section 9.6). For example, we can combine `Monoids` using `Invariant`. Here's an example:

```
import cats.Monoid
import cats.syntax.all.*

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

val tupleToCat: (String, Int, List[String]) => Cat =
  Cat.apply

val catToTuple: Cat => (String, Int, List[String]) =
  cat => (cat.name, cat.yearOfBirth, cat.favoriteFoods)

given catMonoid: Monoid[Cat] = (
  Monoid[String],
  Monoid[Int],
  Monoid[List[String]]
).imapN(tupleToCat)(catToTuple)
```

Let's define some Cats.

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

Now our `Monoid` allows us to create "empty" `Cats`, and add `Cats` together using the syntax we first saw in Chapter 8.

```
garfield |+| heathcliff
// res12: Cat = Cat(
//   name = "GarfieldHeathcliff",
//   yearOfBirth = 3966,
//   favoriteFoods = List("Lasagne", "Junk Food")
// )
```

12.3. Semigroupal Applied to Different Types

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

12.3.1. Semigroupal Applied to List

Combining `Lists` with `Semigroupal` produces some potentially unexpected results. We might expect code like the following to *zip* the lists, but we actually get the cartesian product of their elements:

```
import cats.Semigroupal
import cats.syntax.all.*

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

This is perhaps surprising. Zipping lists tends to be a more common operation. We'll see why we get this behaviour in a moment, but let's first look at `Either`.

12.3.2. Semigroupal Applied to Either

We opened this chapter with a discussion of fail-fast versus accumulating error-handling. We might expect `product` applied to `Either` to accumulate errors instead of fail fast. Again, perhaps surprisingly, we find that `product` implements the same fail-fast behaviour as `flatMap`.

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

Semigroupal[ErrorOr].product(
  Left(Vector("Error 1")),
  Left(Vector("Error 2"))
)
// res1: Either[Vector[String], Tuple2[Nothing, Nothing]] = Left(
//   value = Vector("Error 1")
// )
```

In this example `product` sees the first failure and stops, even though it is possible to examine the second parameter and see that it is also a failure.

12.3.3. Semigroupal Applied to Monads

The reason for the surprising results for `List` and `Either` is that they are both monads. If we have a monad we can implement `product` as follows.

```
import cats.Monad

def product[F[_]: Monad, A, B](fa: F[A], fb: F[B]): F[(A,B)] =
  fa.flatMap(a =>
```

```
    fb.map(b =>
      (a, b)
    )
  )
```

It would be very strange if we had different semantics for `product` depending on how we implemented it. To ensure consistent semantics, Cats' `Monad` (which extends `Semigroupal`) provides a standard definition of `product` in terms of `map` and `flatMap` as we showed above.

So why bother with `Semigroupal` at all? The answer is that we can create useful data types that have instances of `Semigroupal` (and `Applicative`) but not `Monad`. This frees us to implement `product` in different ways. We'll examine this further in a moment when we look at an alternative data type for error handling.

Exercise: The Product of Lists

Why does `product` for `List` produce the Cartesian product? We saw an example above. Here it is again.

```
Semigroupal[List].product(List(1, 2), List(3, 4))
// res2: List[Tuple2[Int, Int]] = List((1, 3), (1, 4), (2, 3),
(2, 4))
```

We can also write this in terms of `tupled`.

```
(List(1, 2), List(3, 4)).tupled
// res3: List[Tuple2[Int, Int]] = List((1, 3), (1, 4), (2, 3),
(2, 4))
```

12.4. Parallel

In the previous section we saw that when call `product` on a type that has a `Monad` instance we get sequential semantics. This makes sense from the point-of-view of keeping consistency with implementations of `product` in terms of `flatMap` and `map`. However it's not always what we want. The `Parallel` type class, and its associated syntax, allows us to access alternate semantics for certain monads.

We've seen how the `product` method on `Either` stops at the first error. If we define some `Left` instances of `Either`

```
import cats.Semigroupal

type ErrorOr[A] = Either[Vector[String], A]
val error1: ErrorOr[Int] = Left(Vector("Error 1"))
val error2: ErrorOr[Int] = Left(Vector("Error 2"))
```

and then call `product` on them

```
Semigroupal[ErrorOr].product(error1, error2)
// res0: Either[Vector[String], Tuple2[Int, Int]] = Left(
//   value = Vector("Error 1")
// )
```

we see we only get the one `Left`.

We can also write this using `tupled` as a short-cut.

```
import cats.syntax.all.*

(error1, error2).tupled
```

To collect all the errors we simply replace `tupled` with its "parallel" version called `parTupled`.

```
(error1, error2).parTupled
// res2: Either[Vector[String], Tuple2[Int, Int]] = Left(
//   value = Vector("Error 1", "Error 2")
// )
```

Notice that both errors are returned! This behaviour is not special to using `Vector` as the error type. Any type that has a `Semigroup` instance will work. For example, we can use a use `List` instead

```
type ErrorOrList[A] = Either[List[String], A]
val errStr1: ErrorOrList[Int] = Left(List("error 1"))
val errStr2: ErrorOrList[Int] = Left(List("error 2"))
```

and `parTupled` will collect all the errors as before.

```
(errStr1, errStr2).parTupled
// res3: Either[List[String], Tuple2[Int, Int]] = Left(
//   value = List("error 1", "error 2")
// )
```

There are many syntax methods provided by `Parallel` for methods on `Semigroupal` and related types, but the most commonly used is `parMapN`.

Let's define some successes

```
val success1: ErrorOr[Int] = Right(1)
val success2: ErrorOr[Int] = Right(2)
val addTwo = (x: Int, y: Int) => x + y
```

and see how we can use `parMapN` to apply a function in an error handling situation.

```
(error1, error2).parMapN(addTwo)
(success1, success2).parMapN(addTwo)
// res4: Either[Vector[String], Int] = Right(value = 3)
```

It's time we looked into how `Parallel` works. The definition below is the core of `Parallel`.

```
trait Parallel[M[_]] {
  type F[_]

  def applicative: Applicative[F]
  def monad: Monad[M]
  def parallel: ~>[M, F]
}
```

This tells us if there is a `Parallel` instance for some type constructor `M` then:

- there must be a `Monad` instance for `M`;
- there is a related type constructor `F` that has an `Applicative` instance; and
- we can convert `M` to `F`.

We haven't seen `~>` before. It's a type alias for `cats.arrow.FunctionK` and is what performs the conversion from `M` to `F`. A normal function `A => B` converts values of type `A` to values of type `B`. Remember that `M` and `F` are not types; they are type constructors. A `FunctionK M ~> F` is a function from a value with type `M[A]` to a value with type `F[A]`. Let's see a quick example by defining a `FunctionK` that converts an `Option` to a `List`.

```
import cats.arrow.FunctionK

object optionToList extends FunctionK[Option, List] {
  def apply[A](fa: Option[A]): List[A] =
    fa match {
      case None    => List.empty[A]
      case Some(a) => List(a)
    }
}
```

We can use it like a function to perform the expected transformation.

```
optionToList(Some(1))
// res5: List[Int] = List(1)
```

```
optionToList(None)
// res6: List[Nothing] = List()
```

As the type parameter `A` is generic a `FunctionK` cannot inspect any values contained with the type constructor `M`. The conversion must be performed purely in terms of the structure of the type constructors `M` and `F`. We can see in `optionToList` above this is indeed the case.

So in summary, `Parallel` allows us to take a type that has a monad instance and convert it to some related type that instead has an applicative (which is equivalent to `Semigroupal`) instance. This related type will have some useful alternate semantics. We've seen the case above where the related applicative for `Either` allows for accumulation of errors instead of fail-fast semantics.

Now we've seen `Parallel` it's time to finally learn about `Applicative`.

Exercise: Parallel List

Does `List` have a `Parallel` instance? If so, what does the `Parallel` instance do?

12.5. Apply and Applicative

Semigroupals aren't mentioned frequently in the wider functional programming literature. They provide a subset of the functionality of a related type class called an **applicative functor** ("applicative" for short).

Cats models applicatives using two type classes. The first, `cats.Apply`, extends `Semigroupal` and `Functor` and adds an `ap` method that applies a parameter to a function within a context. The second, `cats.Applicative`, extends `Apply` and adds the `pure`

method introduced in Chapter 10. Here's a simplified definition in code:

```
trait Apply[F[_]] extends Semigroupal[F] with Functor[F] {
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]

  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)] =
    ap(map(fa)(a => (b: B) => (a, b)))(fb)
}

trait Applicative[F[_]] extends Apply[F] {
  def pure[A](a: A): F[A]
}
```

Breaking this down, the `ap` method applies a parameter `fa` to a function `ff` within a context `F[_]`. The `product` method from `Semigroupal` is defined in terms of `ap` and `map`.

Don't worry too much about the implementation of `product`—it's difficult to read and the details aren't particularly important. The main point is that there is a tight relationship between `product`, `ap`, and `map` that allows any one of them to be defined in terms of the other two.

`Applicative` also introduces the `pure` method. This is the same `pure` we saw in `Monad`. It constructs a new applicative instance from an unwrapped value. In this sense, `Applicative` is related to `Apply` as `Monoid` is related to `Semigroup`.

12.5.1. The Hierarchy of Sequencing Type Classes

With the introduction of `Apply` and `Applicative`, we can zoom out and see a whole family of type classes that concern themselves with sequencing computations in different ways. Figure 11 shows the relationship between the type classes covered in this book.

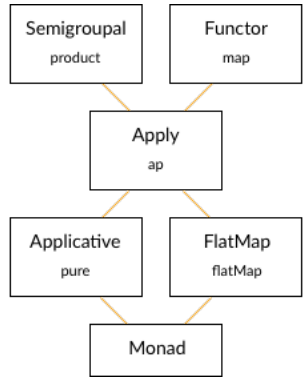

Figure 11: Monad type class hierarchy

Each type class in the hierarchy represents a particular set of sequencing semantics, introduces a set of characteristic methods, and defines the functionality of its supertypes in terms of them:

- every monad is an applicative;
- every applicative a semigroupal;
- and so on.

Because of the lawful nature of the relationships between the type classes, the inheritance relationships are constant across all instances of a type class. `Apply` defines `product` in terms of `ap` and `map`; `Monad` defines `product`, `ap`, and `map`, in terms of `pure` and `flatMap`.

To illustrate this let's consider two hypothetical data types:

- `Foo` is a monad. It has an instance of the `Monad` type class that implements `pure` and `flatMap` and inherits standard definitions of `product`, `map`, and `ap`;
- `Bar` is an applicative functor. It has an instance of `Applicative` that implements `pure` and `ap` and inherits standard definitions of `product` and `map`.

What can we say about these two data types without knowing more about their implementation?

We know strictly more about `Foo` than `Bar`: `Monad` is a subtype of `Applicative`, so we can guarantee properties of `Foo` (namely `flatMap`) that we cannot guarantee with `Bar`. Conversely, we know that `Bar` may have a wider range of behaviours than `Foo`. It has fewer laws to obey (no `flatMap`), so it can implement behaviours that `Foo` cannot.

This demonstrates the classic trade-off of power (in the mathematical sense) versus constraint. The more constraints we place on a data type, the more guarantees we have about its behaviour, but the fewer behaviours we can model.

Monads happen to be a sweet spot in this trade-off. They are flexible enough to model a wide range of behaviours and restrictive enough to give strong guarantees about those behaviours. However, there are situations where monads aren't the right tool for the job. Sometimes we want Thai food, and burritos just won't satisfy.

Whereas monads impose a strict *sequencing* on the computations they model, applicatives and semigroupals impose no such restriction. This puts them in a different sweet spot in the hierarchy. We can use them to represent classes of 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!

12.6. Summary

While monads and functors are the most widely used sequencing data types we've covered in this book, semigroupals and applicatives are the most general. These type classes provide a

generic mechanism to combine values and apply functions within a context, from which we can fashion monads and a variety of other combinators.

`Semigroupal` and `Applicative` are most commonly used as a means of combining independent values such as the results of validation rules. Cats provides the `Parallel` type class to allow to easily switch between a monad and an alternative applicative (or semigroupal) semantics.

Applicative and semigroupal are both introduced in *Applicative Programming with Effects* [54][83].

[83]Semigroupal is referred to as "monoidal" in the paper.

13. Foldable and Traverse

In this chapter we'll look at two type classes that capture iteration over collections:

- `Foldable` abstracts the familiar `foldLeft` and `foldRight` operations;
- `Traverse` is a higher-level abstraction that uses `Applicatives` to iterate with less pain than folding.

We'll start by looking at `Foldable`, and then examine cases where folding becomes complex and `Traverse` becomes convenient.

13.1. Foldable

The `Foldable` type class captures the `foldLeft` and `foldRight` methods we're used to using with sequences like `List` and `Vector`. 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.

13.1.1. Folds and Folding

Let's start with a quick recap of the general concept of folding a sequence. Here's an example of a fold.

```
def show[A](list: List[A]): String =
  list.foldLeft("nil")((accum, item) => s"$item then $accum")
```

This produces output like the following.

```
show(Nil)
// res0: String = "nil"

show(List(1, 2, 3))
// res1: String = "3 then 2 then 1 then nil"
```

There are two parameters we pass to `foldLeft`: an **accumulator** value and a **binary function** to combine it with each item in the sequence.

The `foldLeft` method works recursively down the sequence. Our binary function is called repeatedly for each item, the result of each call becoming the accumulator for the next. When we reach the end of the sequence, the final accumulator becomes our final result.

Depending on the operation we're performing, the order in which we fold may be important. Because of this there are two standard variants of fold:

- `foldLeft` traverses from "left" to "right" (start to finish);
- `foldRight` traverses from "right" to "left" (finish to start).

Figure 12 illustrates each direction.

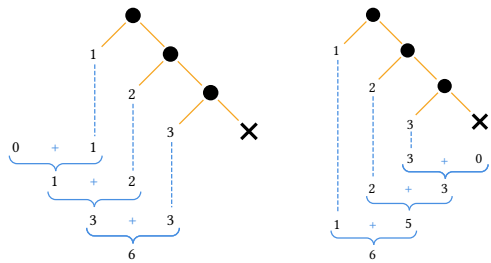

Figure 12: Illustration of foldLeft and foldRight

`foldLeft` and `foldRight` are equivalent if our binary operation is associative. For example, we can sum a `List[Int]` by folding in either direction, using `0` as our accumulator and addition as our operation:

```
List(1, 2, 3).foldLeft(0)(_ + _)
// res2: Int = 6
List(1, 2, 3).foldRight(0)(_ + _)
// res3: Int = 6
```

If we provide a non-associative operator the order of evaluation makes a difference. For example, if we fold using subtraction, we get different results in each direction:

```
List(1, 2, 3).foldLeft(0)(_ - _)
// res4: Int = -6
List(1, 2, 3).foldRight(0)(_ - _)
// res5: Int = 2
```

> **Folds on Sequences**
>
> In Section 3.3 we learned that folds are an abstraction of structural recursion. Here we are looking only at folds on sequences, which are ordered collections of data. Every sequence, regardless of implementation, can be viewed as a list. This means it is either an empty sequence or contains an elements and a sequence. Using this view we can define `foldLeft` and `foldRight` for any sequence.

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?

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`.

13.1.2. Foldable in Cats

Cats' `Foldable` abstracts `foldLeft` and `foldRight` into a type class. Instances of `Foldable` define these two methods and inherit a host of derived methods. Cats provides out-of-the-box instances of `Foldable` for a handful of Scala data types: `List`, `Vector`, `LazyList`, and `Option`.

We can summon instances as usual using `Foldable.apply` and call their implementations of `foldLeft` directly. Let's define an instance of `List`.

```
val ints = List(1, 2, 3)
```

Here is an example using the `Foldable` instance on `List`.

```
import cats.Foldable

Foldable[List].foldLeft(ints, 0)(_ + _)
// res0: Int = 6
```

Other sequences like `Vector` and `LazyList` work in the same way. Here is an example using `Option`, which is treated like a sequence of zero or one elements.

```
val maybeInt = Option(123)

Foldable[Option].foldLeft(maybeInt, 10)(_ * _)
```

13.1.2.1. 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 `LazyList` is not stack safe. The longer the lazy list, the larger the stack requirements for the fold. A sufficiently large lazy list will trigger a `StackOverflowError`:

```
val bigData = (1 to 100000).to(LazyList)

bigData.foldRight(0L)(_ + _)
// java.lang.StackOverflowError ...
```

Using `Foldable` forces us to use stack safe operations.

```
import cats.Eval

val eval: Eval[Long] =
  Foldable[LazyList].
    foldRight(bigData, Eval.now(0L)) { (num, eval) =>
      eval.map(_ + num)
    }
```

This fixes the overflow exception.

```
eval.value
// res3: Long = 5000050000L
```

Stack Safety in the Standard Library

Stack safety isn't typically an issue when using the standard library. The most commonly used collection types, such as `List` and `Vector`, provide stack safe implementations of `foldRight`.

```
(1 to 100000).toList.foldRight(0L)(_ + _)
(1 to 100000).toVector.foldRight(0L)(_ + _)
// res4: Long = 5000050000L
```

We've called out `LazyList` because it is an exception to this rule. Whatever data type we're using, though, it's useful to know that `Eval` has our back.

13.1.2.2. Folding with Monoids

`Foldable` provides us with a host of useful methods defined on top of `foldLeft`. Many of these are copies of familiar methods from the standard library: `find`, `exists`, `forall`, `toList`, `isEmpty`, `nonEmpty`, and so on:

```
Foldable[Option].nonEmpty(Option(42))
// res5: Boolean = true

Foldable[List].find(List(1, 2, 3))(_ % 2 == 0)
// res6: Option[Int] = Some(value = 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 a `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]`.

```
Foldable[List].combineAll(List(1, 2, 3))
// res7: Int = 6
```

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

```
Foldable[List].foldMap(List(1, 2, 3))(_.toString)
// res8: String = "123"
```

Finally, we can compose `Foldables` to work on nested sequences. Take `ints` below.

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

Composing `Foldables` for `List` and `Vector` supports deep traversal on `ints`.

```
(Foldable[List].compose(using Foldable[Vector])).combineAll(ints)
// res9: Int = 21
```

13.1.2.3. Syntax for Foldable

Every method in `Foldable` is available in syntax form. In each case, the first argument to the method on `Foldable` becomes the receiver of the method call:

```
import cats.syntax.all.*

List(1, 2, 3).combineAll
// res10: Int = 6

List(1, 2, 3).foldMap(_.toString)
// res11: 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)(_ + _)
// res12: Int = 6
```

whereas the following generic code will use `Foldable`.

```
def sum[F[_]: Foldable](values: F[Int]): Int =
  values.foldLeft(0)(_ + _)
```

We typically don't need to worry about this distinction. It's a feature! We call the method we want and the compiler uses a `Foldable` when needed to ensure our code works as expected. If we need a stack-safe implementation of `foldRight`, using `Eval` as the accumulator is enough to force the compiler to select the method from Cats.

13.2. 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.

13.2.1. 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
  }
```

Intuitively, we iterate over `hostnames`, call `func` for each item, and combine the results into a list. This produces a correct result.

```
Await.result(allUptimes, 1.second)
// res0: List[Int] = List(1020, 960, 840)
```

However, 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)
```

This produces the same result as before, but the code is much clearer and more concise.

```
Await.result(allUptimes, 1.second)
// res2: List[Int] = List(1020, 960, 840)
```

Let's see how it works. If we ignore distractions like `CanBuildFrom` and `ExecutionContext`, the implementation of `Future.traverse` in the standard library looks like this:

```
def traverse[A, B](values: List[A])
    (func: A => Future[B]): Future[List[B]] =
  values.foldLeft(Future(List.empty[B])) { (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]`; and
- 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 doesn't require us to provide a transformation 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]]`; and
- end up with a `Future[List[A]]`.

`Future.traverse` and `Future.sequence` solve a very specific problem: they allow us to iterate over a sequence of `Futures` and accumulate a result. The simplified examples above only work with `Lists`, but the real `Future.traverse` and `Future.sequence` work with any standard Scala collection.

Cats' `Traverse` type class generalises these patterns to work with any type of `Applicative`: `Future`, `Option`, `List`, and so on. We'll approach `Traverse` in the next sections in two steps: first we'll generalise over the `Applicative`, then we'll generalise over the sequence type. We'll end up with an extremely valuable tool that trivialises many operations involving sequences and other data types.

13.2.2. 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])
```

can be generalized to a use of `Applicative.pure`.

```
import cats.syntax.all.*

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
}
```

can be rewritten to use `mapN`, another `Applicative` operation.

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

By substituting these snippets back into the definition of `traverse` we can generalise it to to work with any `Applicative`.

```
import cats.Applicative

def listTraverse[F[_]: Applicative, A, B]
      (list: List[A])(func: A => F[B]): F[List[B]] =
  list.foldLeft(List.empty[B].pure[F]) { (accum, item) =>
    (accum, func(item)).mapN(_ :+ _)
  }

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

We can use `listTraverse` to re-implement our uptime example.

```
val totalUptime = listTraverse(hostnames)(getUptime)
```

Once again, we get the same result.

```
Await.result(totalUptime, 1.second)
// res5: List[Int] = List(1020, 960, 840)
```

or we can use it with other `Applicative` data types as shown in the following exercises.

Exercise: Traversing with Vectors

What is the result of the following?

```
listSequence(List(Vector(1, 2), Vector(3, 4)))
```

What about a list of three parameters?

```
listSequence(List(Vector(1, 2), Vector(3, 4), Vector(5, 6)))
```

Exercise: Traversing with Options

Here's an example that uses `Options`:

```
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))
```

Exercise: Traversing with Validated

Finally, here is an example that uses `Validated`:

```
import cats.data.Validated

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))
```

13.2.3. Traverse in Cats

Our `listTraverse` and `listSequence` methods work with any type of `Applicative`, but they only work with one type of sequence: `List`. We can generalise over different sequence types using a type class, which brings us to Cats' `Traverse`. Here's the abbreviated definition:

```
package cats

trait Traverse[F[_]] {
  def traverse[G[_]: Applicative, A, B]
      (inputs: F[A])(func: A => G[B]): G[F[B]]

  def sequence[G[_]: Applicative, B]
      (inputs: F[G[B]]): G[F[B]] =
```

```
    traverse(inputs)(identity)
}
```

Cats provides instances of `Traverse` for `List`, `Vector`, `Stream`, `Option`, `Either`, and a variety of other types. We can summon instances as usual using `Traverse.apply` and use the `traverse` and `sequence` methods as described in the previous section.

```
import cats.Traverse

val totalUptime: Future[List[Int]] =
  Traverse[List].traverse(hostnames)(getUptime)
```

We get the same result as before.

```
Await.result(totalUptime, 1.second)
// res0: List[Int] = List(1020, 960, 840)
```

There are also syntax versions of the methods.

```
import cats.syntax.all.*

val numbers  = hostnames.traverse(getUptime)
val numbers2 =
  List(Future(1), Future(2), Future(3)).sequence
```

As you can see, this is much more compact and readable than the `foldLeft` code we started with earlier this chapter!

13.3. Conclusions

In this chapter we were introduced to `Foldable` and `Traverse`, two type classes for iterating over sequences.

`Foldable` abstracts the `foldLeft` and `foldRight` methods we know from collections in the standard library. It adds stack-safe implementations of these methods to a handful of extra data types,

and defines a host of situationally useful additions. That said, `Foldable` doesn't introduce much that we didn't already know.

The real power comes from `Traverse`, which abstracts and generalises the `traverse` and `sequence` methods we know from `Future`. Using these methods we can turn an `F[G[A]]` into a `G[F[A]]` for any `F` with an instance of `Traverse` and any `G` with an instance of `Applicative`. In terms of the reduction we get in lines of code, `Traverse` is one of the most powerful patterns in this book. We can reduce `folds` of many lines down to a single `traverse`.

As far as I know, the `Foldable` type class was cooked up by the Haskell community as a simple abstraction over the well known left- and right-folds. The `traverse` method, however, comes from *The Essence of the Iterator Pattern* [33].

Part III: Interpreters

14. Indexed Types

In this chapter we look at **indexed types**. An indexed type is a type constructor, so a type like `F[_]`, along with a set of types that can fill in the constructor's type parameters. Let's say this set of types is `Int`, `String`, and `Option[Double]`. Then, for a type constructor `F` we can construct an indexed type from the set `F[Int]`, `F[String]`, and `F[Option[Double]]`. The types `Int`, `String`, and `Option[Double]` act as indices into the set `F[Int]`, `F[String]`, and `F[Option[Double]]`, hence the name. The type constructor `F` can be either data and codata.

The description above is very abstract, and doesn't help us understand how indexed types are useful. We'll see a lot of details and examples in this chapter, but let's start with a more useful high-level overview. We can think of indexed types as working with proofs that a type parameter is equal to a particular element from the set of indices. Indexed *data* provides this evidence when we destructure it, while indexed *codata* requires this evidence when we call methods. Remember the definition of algebras we gave in Section 6.2, where we said an algebra consists of three different kinds of methods: constructors, combinators, and interpreters. Indexed types allows us to do two things:

- We can restrict where constructors and combinators can be used. We can think of representing some state using a type parameter of `F`, and we can only call particular methods when we are in the correct state. In this case we are working with **indexed codata**.
- We restrict the types produced by interpreters, enabling us to create type-safe interpreters that guarantee they only encounter particular states when they run. Again these constraints are represented using type parameters. In this case we are working with **indexed data**.

Indexed data are more usually known as **generalized algebraic data types**. Indexed codata are sometimes known as **typestate**. Both can make use of what is known as **phantom types**. Indeed, an early name for indexed data was **first-class phantom types**. As you might expect, indexed data and indexed codata are dual to one another.

14.1. Phantom Types

Phantom types are a basic building block of indexed types, so we'll start with an example of them. A phantom type is simply a type parameter that doesn't correspond to any value. In the example below, the type parameter `A` is a phantom type, because there is no value of type `A`, while `B` is not because there is a value of that type.

```
final case class PhantomExample[A, B](value: B)
```

Phantom types are used to shift constraints to compile time. A simple example involves units of measurement. Most of the world has standardized on SI units, such as metres and litres. However, other measuring systems, such as Imperial units, remain in use some countries or in some niches within countries that otherwise use metric. Differences between different measurement systems can cause problems. A dramatic example is the [loss of the Mars orbiter][mars], caused by two software components using incompatible measurements (one using metric, and one using US customary measurements.)

With phantom types we can annotate measurements with their units, which in turn can prevent us ever using incompatible units. Let's work with just length, which is sufficient to show the idea. We'll start by defining a length type with a phantom type recording the unit, and a method that allows us to add together lengths.

```
final case class Length[Unit](value: Double) {
  def +(that: Length[Unit]): Length[Unit] =
    Length[Unit](this.value + that.value)
}
```

We'll need to define a few unit types to use this, and some `Lengths` using these units.

```
trait Metres
trait Feet

val threeMetres = Length[Metres](3)
val threeFeetAndRising = Length[Feet](3)
```

Now we can add `Lengths` together if they have the same unit.

```
threeMetres + threeMetres
// res0: Length[Metres] = Length(value = 6.0)
```

However if we try to add `Lengths` with different units the code will not compile.

```
threeMetres + threeFeetAndRising
// error:
// Found:    (repl.MdocSession.MdocApp.threeFeetAndRising :
//
repl.MdocSession.MdocApp.Length[repl.MdocSession.MdocApp.Feet])
// Required:
repl.MdocSession.MdocApp.Length[repl.MdocSession.MdocApp.Metres]
// threeMetres + threeFeetAndRising
//               ^^^^^^^^^^^^^^^^^^
```

There is one big problem with phantom types on their own: there is no way to use the information stored in the phantom type in further processing. For example, force times length gives torque (with the SI unit of newton metres). However we cannot define a `*` method on `Length` that can only be called if the `Unit` is `Metre` using just the tool of phantom types. Similarly, we cannot define, say, a `toString` method that uses the `Unit` type to appropriately

print the result. Solving these problems leads us to indexed codata, so let's now look at that.

[mars]: https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#Cause_of_failure

14.2. Indexed Codata

The basic idea of indexed codata is to prevent methods being called unless certain conditions, encoded in types, are met. More precisely, methods are guarded by type equalities that callers must prove they satisfy to call a method. The contextual abstraction features, `given` instances and `using` clauses, are used to implement this in Scala.

We'll start our exploration of indexed codata with a very simple example. We are going to define a switch that can only be turned on when it is off, and off when it is on. Since this is codata, we start with an interface.

```
trait Switch {
  def on: Switch
  def off: Switch
}
```

There are no constraints on this interface as defined; we can turn any switch on, even if it is already on, and vice versa. The first step to implement such a constraint is to add a type parameter, which will hold the state of the `Switch`. This type parameter doesn't correspond to any data we store in `Switch`, so it is a phantom type.

```
trait Switch[A] {
  def on: Switch[A]
  def off: Switch[A]
}
```

We are now going to add constraints that say we can only call certain methods when this type parameter corresponds to particular concrete types. It is in this way that indexed codata goes beyond what phantom types alone can do: we can inspect, at compile-time, the type of a type parameter and make decisions based on this type.

Implementing these constraints has two parts. The first is defining types to represent on and off.

```
trait On
trait Off
```

The second step is to add the constraints to the relevant methods on `Switch`. Here is how we do it.

```
trait Switch[A] {
  def on(using ev: A =:= Off): Switch[On]
  def off(using ev: A =:= On): Switch[Off]
}
```

We can create an implementation to show it really works.

```
final case class SimpleSwitch[A]() extends Switch[A] {
  def on(using ev: A =:= Off): Switch[On] =
    SimpleSwitch()
  def off(using ev: A =:= On): Switch[Off] =
    SimpleSwitch()
}
object SimpleSwitch {
  val on: Switch[On] = SimpleSwitch()
  val off: Switch[Off] = SimpleSwitch()
}
```

Here are some examples of using it correctly

```
SimpleSwitch.on.off
// res2: Switch[Off] = SimpleSwitch()
SimpleSwitch.off.on
// res3: Switch[On] = SimpleSwitch()
```

Incorrect uses fail to compile.

```
SimpleSwitch.on.on
// error:
// Cannot prove that MdocApp1.this.On =:= MdocApp1.this.Off.
```

The constraint is made of two parts: using clauses, which we learned about in Chapter 5, and the [`A =:= B`][scala.=:=] construction, which is new. `=:=` represents a type equality. If a given instance `A =:= B` exists, then the type `A` is equal to the type `B`. (Note we can write this with the more familiar prefix notation `=:=[A, B]` if we prefer.) We never create these instances ourselves; instead the compiler creates them for us. In the method `on` we are asking the compiler to construct an instance `A =:= Off`, which can only be done if `A` *is* `Off`. This in turn means we can only call the method when the `Switch` is `Off`. This is the core idea of indexed codata: we raise states into types, and restrict method calls to a subset of states.

This is a different use of contextual abstraction to type classes. Type classes associate operations with types. What we're doing here is proving some property of a type with respect to another type. More precisely we're proving that a type parameter is equal to a particular type. The given instance only exists when the compiler can prove this is the case. Hence these given instances are sometimes called **evidence** or **witnesses**. This different view subsumes type classes, as we can think of type classes as evidence that a type implements a certain interface.

14.2.0.1. Exercise: Torque {-}

In Section 14.1 we saw we could use phantom types to represent units. We also ran into a limitation: we had no way to inspect the phantom types and hence make decisions based on them. Now, with indexed codata, we can do solve this problem.

Below if the definition of `Length` we previously used. Your mission is to:

1. implement a type `Force`, parameterized by a phantom type that represents the units of force;
2. implement a type `Torque`, parameterized by a phantom type that represents the units of torque;
3. define types `Newtons` and `NewtonMetres` to represent force in SI units;
4. implement a method `*` on `Force` that accepts a `Length` and returns a `Torque`. It can only be called if the `Force` is in `Newtons` and the `Length` is in `Metres`. In this case the `Torque` is in `NewtonMetres`. (Torque is force times length.)

```
final case class Length[Unit](value: Double) {
  def +(that: Length[Unit]): Length[Unit] =
    Length[Unit](this.value + that.value)
}
```

14.2.1. API Protocols

An API protocol defines the order in which methods must be called. The protocol in the case of `Switch` is that we can only call `off` after calling `on` and vice versa. This protocol is a simple finite state machine, and illustrated in Figure 13. Many common types have similar protocols. For example, files can only be read once they are opened and cannot be read once they have been closed.

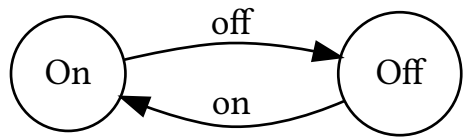

Figure 13: The switch API protocol

Indexed codata allows us to enforce API protocols at compile-time. Often these protocols are finite-state machines. We can represent

these protocols with a single type parameter that represents the state, as we did with `Switch`. We can also use multiple type parameters if that makes for a more convenient representation.

Let's see an example using multiple type parameters. We're going to build an API that represents a very limited subset of [HTML][html], the language the defines web pages. An example of HTML is below.

```
<!DOCTYPE html>
<html>
  <head><title>Our Amazing Web Page</title></head>
  <body>
    <h1>This Is Our Amazing Web Page</h1>
    <p>Please be in awe of its <strong>amazingness</strong></p>
  </body>
</html>
```

In HTML the content of the page is marked up with tags, like `<h1>`, that give it meaning. For example, `<h1>` means a heading at level one, and `<p>` means a paragraph. An opening tag is closed by a corresponding closing tag, such as `</h1>` for `<h1>` and `</p>` for `<p>`.

There are several rules for valid HTML[^valid-html]. We're going to focus on the following:

1. Within the `html` tag there can only be a `head` and a `body` tag, in that order.
2. Within the `head` tag there must be exactly one `title`, and there can be any other number of allowed tags (of which we're only going to model `link`).
3. Within the `body` there can be any number of allowed tags (of which we are only going to model `h1` and `p`).

We're going to use a Church-encoded representation for HTML, so tags are created by method calls. Figure 14 shows the finite state machine representation of the API protocol. I find it easier to read as a regular expression, which we can write down as

head link title link body (h1 | p)*

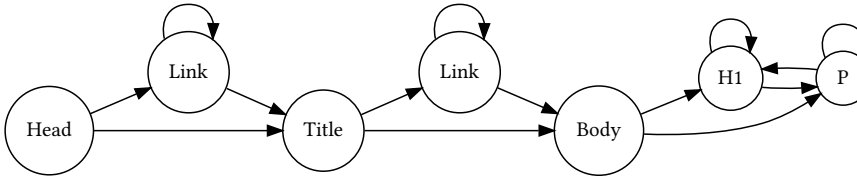

Figure 14: The HTML API protocol

As the code is fairly repetitive I will just present all the code and then discuss the important parts. Here's the implementation.

```
sealed trait StructureState
trait Empty extends StructureState
trait InHead extends StructureState
trait InBody extends StructureState

sealed trait TitleState
trait WithoutTitle extends TitleState
trait WithTitle extends TitleState

// Not a case class so external users cannot copy it
// and break invariants
final class Html[S <: StructureState, T <: TitleState](
    head: Vector[String],
    body: Vector[String]
) {
  // Head tags -----------------------------------------------

  def head(using S =:= Empty): Html[InHead, WithoutTitle] =
    Html(head, body)

  def title(
      text: String
  )(using S =:= InHead, T =:= WithoutTitle): Html[InHead,
WithTitle] =
    Html(head :+ s"<title>$text</title>", this.body)

  def link(rel: String, href: String)(using S =:= InHead):
Html[InHead, T] =
    Html(head :+ s"<link rel=\"$rel\" href=\"$href\"/>", body)

  // Body tags -----------------------------------------------
```

```
  def body(using S =:= InHead, T =:= WithTitle): Html[InBody,
WithTitle] =
    Html(head, body)

  def h1(text: String)(using S =:= InBody): Html[InBody, T] =
    Html(head, body :+ s"<h1>$text</h1>")

  def p(text: String)(using S =:= InBody): Html[InBody, T] =
    Html(head, body :+ s"<p>$text</p>")

  // Interpreter ----------------------------------------

  override def toString(): String = {
    val h = head.mkString("  <head>\n    ", "\n    ", "\n  </
head>")
    val b = body.mkString("  <body>\n    ", "\n    ", "\n  </
body>")

    s"\n<html>\n$h\n$b\n</html>"
  }
}
object Html {
  val empty: Html[Empty, WithoutTitle] = Html(Vector.empty,
Vector.empty)
}
```

The key point is that we factor the state into two components. `StructureState` represents where in the overall structure we are (inside the `head`, inside the `body`, or inside neither). `TitleState` represents the state when defining the elements inside the `head`, specifically whether we have a `title` element or not. We could certainly represent this with one state type variable, but I find the factored representation both easier to work with and easier for other developers to understand.

Here's an example in use.

```
Html.empty.head
  .link("stylesheet", "styles.css")
  .title("Our Amazing Webpage")
  .body
  .h1("Where Amazing Exists")
```

```
  .p("Right here")
  .toString
// res6: String = """
// <html>
//   <head>
//     <link rel="stylesheet" href="styles.css"/>
//     <title>Our Amazing Webpage</title>
//   </head>
//   <body>
//     <h1>Where Amazing Exists</h1>
//     <p>Right here</p>
//   </body>
// </html>"""
```

Here's an example of the type system preventing an invalid construction, in this case the lack of a title.

```
Html.empty.head
  .link("stylesheet", "styles.css")
  .body
  .h1("This Shouldn't Work")
// error:
// Cannot prove that MdocApp2.this.WithoutTitle =:=
MdocApp2.this.WithTitle.
```

These error messages are not great. We'll address this in Chapter 17.

We can implement more complex protocols, such as those that can be represented by context-free or even context-sensitive grammars, using the same technique.

14.2.1.1. Exercise: HTML API Design {-}

I don't particularly like the HTML API we developed above, as the flat method call structure doesn't match the nesting in the HTML structure we're creating. I would prefer to write the following.

```
Html.empty
  .head(_.title("Our Amazing Webpage"))
```

```
  .body(_.h1("Where Amazing Happens").p("Right here"))
  .toString
```

We still require the `head` is specified before the `body`, but now the nesting of the method calls matches the nesting of the structure. Notice we're still using a Church-encoded representation.

Can you think of how to implement this? You'll need to use indexed codata, and perhaps a bit of inspiration. This is a very open ended question, so don't worry if you struggle with it!

[html]: https://html.spec.whatwg.org/multipage/

[^valid-html]: The HTML specification allows for very lenient parsing of HTML. For example, if we don't define the `head` tag it will usually be inferred. However we aren't going to allow that kind of leniency in our API.

14.2.2. Beyond Equality Constraints

Indexed data is all about equality constraints: proofs that some type parameter is equal to some type. However we can go beyond equality constraints with contextual abstraction. We can use [`<:<`][scala.<:<] for evidence of a subtyping relationship, and [`NotGiven`][scala.NotGiven] for evidence that no given instance exists (with which we can test that types are not equal, for example). Beyond that, we can view any given instance as evidence.

Let's return to our example of length, force, and torque to see how this is useful. In the exercise where we defined torque as force times length, we fixed the computation to have SI units. The example code is below.

```
final case class Force[Unit](value: Double) {
  def *[L](length: Length[L])(using Unit =:= Newtons, L =:=
```

```
Metres): Torque[NewtonMetres] =
    Torque(this.value * length.value)
}
```

This is a reasonable thing to do, as other units are insane, but there are a lot of insane people out there. To accommodate other unit types we can create given instances that represent the result types of operations of interest. In this case we want to represent the result of multiplying a length unit by the force unit. In code we can write the following.

```
// Weird units
trait Feet
trait Pounds
trait PoundsFeet

// An instance exists if A * B = C
trait Multiply[A, B, C]
object Multiply {
  given Multiply[Metres, Newtons, NewtonMetres] with {}
  given Multiply[Feet, Pounds, PoundsFeet] with {}
}
```

Now we can define * methods on `Length` and `Force` in terms of `Multiply`.

```
final case class Length[L](value: Double) {
  def *[F, T](that: Force[F])(using Multiply[L, F, T]): Torque[T]
=
    Torque(this.value * that.value)
}

final case class Force[F](value: Double) {
  def *[L, T](that: Length[L])(using Multiply[F, L, T]):
Torque[T] =
    Torque(this.value * that.value)
}
```

Here's an example showing it works.

```
Length[Metres](3) * Force[Newtons](4)
// res11: Torque[NewtonMetres] = Torque(value = 12.0)

Length[Feet](3) * Force[Pounds](4)
// res12: Torque[PoundsFeet] = Torque(value = 12.0)
```

Note that's it hard to think of `Multiply` as a type class, as it does not provide *any* methods. Viewing it as evidence, however, does make sense.

14.2.2.1. Exercise: Commutivitiy {-}

In the example above we defined a `Multiply` type class to represent that metres times newtons gives newton metres. Multiplication is commutative. If $A \times B = C$, then $B \times A = C$. However we have not represented this, and if we try newtons times metres, as in the example below, the code will fail.

```
Force[Newtons](3) * Length[Metres](4)
// error:
// No given instance of type
MdocApp4.this.Multiply[MdocApp4.this.Newtons,
MdocApp4.this.Metres, Any] was found for parameter x$2 of method
* in class Force
// Force[Newtons](3) * Length[Metres](4)
//                                    ^
```

Add evidence to `Multiply` that if `Multiply[A, B, C]` exists, then so does `Multiply[B, A, C]`, and show that it solves this problem.

Now that we have learned about indexed codata, we'll turn to its dual, indexed data.

14.3. Indexed Data

The key idea of indexed data is to encode type equalities in data. When we come to inspect the data (usually, via structural

recursion) we discover these equalities, which in turn limit what values we can produce. Notice, again, the duality with codata. Indexed codata limits methods we can call. Indexed data limits values we can produce. Also, remember that indexed data is often known as generalized algebraic data types. We are using the simpler term indexed data to emphasise the relationship to indexed codata, and also because it's much easier to type!

Concretely, indexed data in Scala occurs when:

1. we define a sum type with at least one type parameter; and
2. cases within the sum instantiate that type parameter with a concrete type.

Let's see an example. Imagine we are implementing a programming language. We need some representation of values within the language. Suppose our language supports strings, integers, and doubles, which we will represent with the corresponding Scala types. The code below shows how we can implement this as a standard algebraic data type.

```
enum Value {
  case VString(value: String)
  case VInt(value: Int)
  case VDouble(value: Double)
}
```

Using indexed data we can use the alternate implementation below.

```
enum Value[A] {
  case VString(value: String) extends Value[String]
  case VInt(value: Int) extends Value[Int]
  case VDouble(value: Double) extends Value[Double]
}
```

This is indexed data, as it meets the criteria above: we have a type parameter `A` that is instantiated with a concrete type in the cases `VString`, `VInt`, and `VDouble`. It's quite easy to use indexed data in

Scala, and people often do so not knowing that it is anything special. The natural next question is why is this useful? It will take a more involved example to show why, so let us now dive into one that makes good use of indexed data.

14.3.1. The Probability Monad

For our case study of indexed data we will create a probability monad. This is a composable abstraction for defining probability distributions. The probability monad has a lot of uses. The most relevant to most developers is generating data for property-based tests, so we'll focus on this use case. However, it can also be used, for example, for statistical inference or for creating generative art. See the conclusions (Section 14.4) for some pointers to these uses.

Let's start with an example of generating random data. [Doodle] [doodle] is a Scala library for graphics and visualization. A core part of the library is representing colors. Doodle has two different representations of colors, RGB and OkLCH, with conversions between the two. These conversions involve some somewhat tricky mathematics. Testing these conversions is an excellent use of property-based testing. If we can generate many, say, random RGB colors, we can test the conversion by checking the roundrip from RGB to OkLCH and back results in the original color[^numerics].

To create an RGB color we need three unsigned bytes, so our first task is to define how we generate a random byte. Doodle happens to have an implementation of the probability monad that we will use. Here is how we can do it.

```
import cats.syntax.all.*
import doodle.core.Color
import doodle.core.UnsignedByte
import doodle.random.{*, given}
```

```
val randomByte: Random[UnsignedByte] =
  Random.int(0, 255).map(UnsignedByte.clip)
```

Note that once again we see the interpreter strategy. A `Random[A]` is a value representing a program that will generate a random value of type `A` when it runs.

With three random unsigned bytes we can create a random RGB color.

```
val randomRGB: Random[Color] =
  (randomByte, randomByte, randomByte)
    .mapN((r, g, b) => Color.rgb(r, g, b))
```

We might want to check our code by generating a few random values.

```
randomRGB.replicateA(2).run
// res1: List[Color] = List(
//   Rgb(
//     r = UnsignedByte(value = 84),
//     g = UnsignedByte(value = 23),
//     b = UnsignedByte(value = -112),
//     a = Normalized(get = 1.0)
//   ),
//   Rgb(
//     r = UnsignedByte(value = -46),
//     g = UnsignedByte(value = -36),
//     b = UnsignedByte(value = 69),
//     a = Normalized(get = 1.0)
//   )
// )
```

It seems to be working.

Once we have a source of random data we can write tests using it. We can easily generate more data than is feasible for a programmer to write by hand, and therefore have a higher degree of certainty that our code is correct than we would get with manual testing. The details of writing the tests are not important to us here, so let's move on.

We have seen is an illustration of using the probability monad to generate random data. The probability monad works the same way as every other algebra: we have constructors (`Random.int`), combinators (`map`, and `mapN`), and interpreters (`run`). Being a monad means the algebra has some specific structure. For example, it tells us that we have `pure` and `flatMap` available, from which we can derive `mapN`.

Let's sketch an plausible interface for our probability monad.

```
trait Random[A] {
  def flatMap[B](f: A => Random[B]): Random[B]
}
object Random {
  def pure[A](value: A): Random[A] = ???

  // Generate a uniformly distributed random  Double greater
  // than or equal to zero and less than one.
  val double: Random[Double] = ???

  // Generate a uniformly distributed random  Int
  val int: Random[Int] = ???
}
```

The interface has the minimum requirements to be a monad, and a few other constructors. We can make progress on the implementation by applying the reification strategy, introduced in Section 6.2.

```
enum Random[A] {
  def flatMap[B](f: A => Random[B]): Random[B] =
    RFlatMap(this, f)

  case RFlatMap[A, B](source: Random[A], f: A => Random[B])
      extends Random[B]
  case RPure(value: A)
  case RDouble extends Random[Double]
  case RInt extends Random[Int]
}
object Random {
  import Random.{RPure, RDouble, RInt}
```

```
  def pure[A](value: A): Random[A] = RPure(value)

  // Generate a uniformly distributed random  Double greater
  // than or equal to zero and less than one.
  val double: Random[Double] = RDouble

  // Generate a uniformly distributed random  Int
  val int: Random[Int] = RInt
}
```

The next step is to implement an interpreter, which is a standard structural recursion. The interpreter has a parameter that provides a source of random numbers.

```
def run(rng: scala.util.Random = scala.util.Random): A =
  this match {
    case RFlatMap(source, f) => f(source.run(rng)).run(rng)
    case RPure(value)        => value
    case RDouble             => rng.nextDouble()
    case RInt                => rng.nextInt()
  }
```

This is an example of indexed data, as the cases `RDouble` and `RInt` provide a concrete type for the type parameter `A`. This means that these cases in the interpreter can produce values of that concrete type. If we did not use indexed data we could only generate values of type `A`, which the programmer would have to supply to use like in the `RPure` case.

To finish this implementation we should implement the `Monad` type class, which would give us `mapN` and other methods for free. However, this is outside the scope of this case study, which is focused on indexed data. I encourage you to do this yourself if you feel you would benefit from the practice.

Note that indexed data can mix concrete and generic types. Let's say we add a `product` method to `Random`.

```
enum Random[A] {
  // ...
```

```
  def product[B](that: Random[B]): Random[(A, B)] =
    RProduct(this, that)

  case RProduct[A, B](left: Random[A], right: Random[B]) extends
Random[(A, B)]
  // .. other cases here
}
```

The right-hand side of the `RProduct` case instantiates the type parameter to `(A, B)`, which mixes the concrete tuple type with the generic types `A` and `B`

There are a few tricks to using indexed data that are essential in Scala 2, and can sometimes be useful in Scala 3. Take the following translation of the probability monad into Scala 2. (I've placed a `using` directive in this code, so if you paste it into a file and run it with the Scala CLI it will use the latest version of Scala 2.13.)

```
//> using scala 2.13

sealed trait Random[A] {
  import Random._

  def flatMap[B](f: A => Random[B]): Random[B] =
    RFlatMap(this, f)

  def product[B](that: Random[B]): Random[(A, B)] =
    RProduct(this, that)

  def run(rng: scala.util.Random = scala.util.Random): A =
    this match {
      case RFlatMap(source, f) => f(source.run(rng)).run(rng)
      case RProduct(l, r)      => (l.run(rng), r.run(rng))
      case RPure(value)        => value
      case RDouble             => rng.nextDouble()
      case RInt                => rng.nextInt()
    }

}
object Random {
  final case class RFlatMap[A, B](source: Random[A], f: A =>
Random[B])
```

```
      extends Random[B]
  final case class RProduct[A, B](left: Random[A], right:
Random[B])
      extends Random[(A, B)]
  final case class RPure[A](value: A) extends Random[A]
  case object RDouble extends Random[Double]
  case object RInt extends Random[Int]

  def pure[A](value: A): Random[A] = RPure(value)

  // Generate a uniformly distributed random  Double greater
  // than or equal to zero and less than one.
  val double: Random[Double] = RDouble

  // Generate a uniformly distributed random  Int
  val int: Random[Int] = RInt
}
```

In Scala 2 this generates a lot of type errors like

```
[error] constructor cannot be instantiated to expected type;
[error]  found   : Random.RProduct[A(in class RProduct),B]
[error]  required: Random[A(in trait Random)]
[error]       case RProduct(l, r)      => (l.run(rng),
r.run(rng))
[error]            ^^^^^^^^
```

To solve this we need to create a nested method with a fresh type parameter in the interpreter, as shown below. With this change Scala 2's type inference works and it can successfully compile the code.

```
def run(rng: scala.util.Random = scala.util.Random): A = {
  def loop[A](random: Random[A]): A =
    random match {
      case RFlatMap(source, f)   => loop(f(loop(source)))
      case RProduct(left, right) => (loop(left), loop(right))
      case RPure(value)          => value
      case RDouble               => rng.nextDouble()
      case RInt                  => rng.nextInt()
    }
```

```
  loop(this)
}
```

The other trick is for when we want to use pattern matches that match type tags. This means the form like

```
case r: RPure[A] => ???
```

rather than

```
case RPure(value) => ???
```

For cases like `RProduct` it is not clear how to write these pattern matches, as the type parameters `A` and `B` for `RProduct` don't correspond to the type parameter `A` on `Random`. The solution is use lower case names from the type parameters. Concretely, this means we can write

```
case r: RProduct[a, b] => ???
```

The type parameters `a` and `b` are existential types; we know they exist but we don't know what concrete type they correspond to. I've found this is occasionally necessary in Scala 2, but very rare in Scala 3.

[^numerics]: Due to numeric issues there may be small differences between the colors that we should ignore.

[doodle]: https://www.creativescala.org/doodle/

14.4. Conclusions

In this chapter we looked at indexed data and indexed codata. The key idea of indexed types is to encode equality constraints that a type parameter equals some type. With indexed data these

constraints are encoded in the data and we discover them when we destructure the data. In this way indexed data is a producer of equalities. With indexed codata these constraints must be shown to hold when methods are called. Hence indexed codata is a consumer of equalities. We also saw that we can go beyond equalities constraints with contextual abstraction, by encoding other types of constraints in given instances.

Indexed types build on phantom types. The earliest reference I've found to phantom types is Daan Leijen and Erik Meijer. [48]. Type equalities were added soon afterwards, creating what we now know as generalized algebraic data types or indexed data [13,78,93]. Most work on generalized algebraic data types is concerned with type inference algorithms (e.g. [68]), which is not so relevant to the working programmer. Chuan-kai Lin and Tim Sheard. [52] is not different in this respect, but it does have a particularly clear breakdown of how GADTs are used in the most common case.

Interest in indexed codata is much more recent [83], reflecting the general lack of attention that codata has received in programming language research (or at least the parts that I read.) Scala has excellent support for indexed codata but, even so, we can see in Scala a lack of symmetry in the support for indexed data and codata. While indexed data is built into the language, indexed codata is something we must built ourselves from contextual abstractions. This is not necessarily a bad thing, as contextual abstraction allows us to go beyond the simple type equalities of indexed data and codata. Recent research has looked to address this asymmetry. For example, Klaus Ostermann and Julian Jabs. [66] considers indexed data and indexed codata as related by transposition of a matrix defining the API and Weixin Zhang, Cristina David, and Meng Wang. [94] develops a system, implemented in Scala, that translates between data and codata.

In a case study we used indexed codata to implement an API protocol: a restriction on the order in which methods can be called. We can view this as an elaboration on the basic algebra or combinator library strategy we have seen in some many different case studies. We can also relate it to work in the object-oriented programming (OOP) community. It is worth doing so to show that these problems bridge programming communities and sometimes disparate communities discover very similar solutions.

In the OOP world a combinator library is called a fluent interface. The same article that introduces the term fluent interface also mentions the need for API protocols: "choose your return type based on what you need to continue fluent action" [28]. Many case studies have explored fluent interfaces (e.g. [30]; [41]; [18]; [80]) and this style of code is increasing in popularity [61]. Encoding an API protocol can be quite involved, so another research direction is the creation of tools to generate code from a protocol definition [50]; [60]; [37]; [87]. [74] translates API protocols back to the functional world, showing a variety of encodings in Standard ML.

The probability monad we developed, which is specialized to sampling data, is only one of many possibilities. Sampling gives us an approximate representation of a distribution. Small discrete distributions can be represented exactly. [26] show how this can be done, in addition to the sampling approach we used. [43] shows how the exact and sampling approaches can be factored into monad transformer stacks. [76] uses probability monad as the underlying abstraction on which a variety of different statistical inference algorithms are defined. This is application of the idea of multiple interpretations that we have stressed throughout this book. [77] expands on this idea, breaking down inference algorithms into reusable components.

We introduced the probability monad in the context of property based testing [14]. Randomly generating test data is not the only approach. [75] describes an elegant way of enumerating data. Also

see [23] for an approach specialized to enumerating algebraic data types. More recently machine learning techniques are being explored. See, for example, [72] and [49]. [38] studies how property based testing is used in practice.

I mentioned that the probability monad can be used in generative art. Generative art is, broadly, art that is generated by some algorithmic process. This can include an element of randomness. While there are papers on generative art (e.g. [7]; [20]), and many other resources that discuss it, it's much more fun to create some yourself. Figure 15 shows an example of generative art. The code is below, and it has many knobs that you can play with to create your own example. Just add the `@main` annotation to the `cycloid` method and you can run the code from the Scala CLI. Have fun!

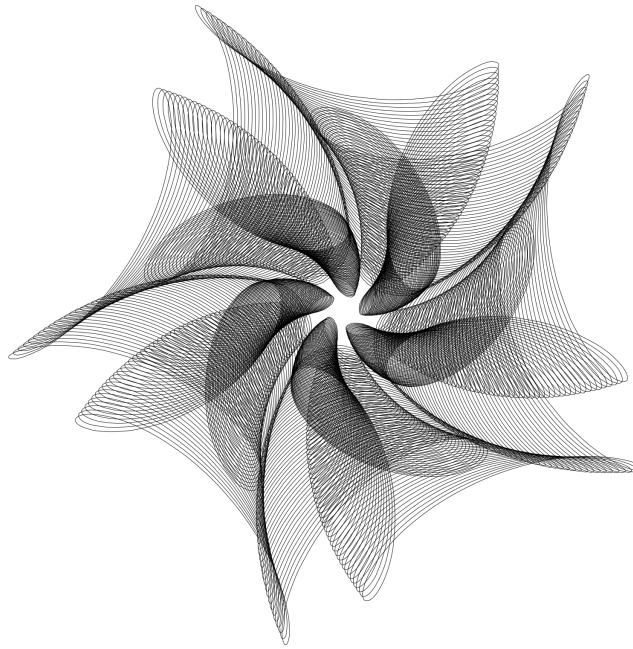

Figure 15: A set of cycloids showing five-fold symmetry

```
//> using dep org.creativescala::doodle:0.30.0

import cats.Monoid
import cats.syntax.all.*
import cats.effect.unsafe.implicits.global
import doodle.core.*
import doodle.core.format.{Pdf, Png}
import doodle.interact.syntax.interpolation.*
import doodle.random.{*, given}
import doodle.syntax.all.*
import doodle.java2d.*

def cycloid(): Unit = {
  given Monoid[Angle => Vec] with {
    def combine(a: Angle => Vec, b: Angle => Vec): Angle => Vec =
```

```
      angle => a(angle) + b(angle)

    val empty: Angle => Vec = angle => Vec.zero
  }

  /** Reverse the rolling direction of the cycloid. */
  val reverse: Angle => Angle = angle => -angle

  /** Multiply the angle by the given speed, which determines how
rapidly the
    * cycloid rotates.
    */
  def speed(speed: Double): Angle => Angle =
    angle => angle * speed

  /** Increment the angle by the given amount. In other words
move it out of
    * phase.
    */
  def phase(p: Angle): Angle => Angle =
    angle => p - angle

  /** Set the radius of the cycloid */
  def radius(r: Double): Angle => Vec =
    angle => Vec(r, angle)

  /** Cycloid is speed of rotation and radius (+ve or -ve) */
  def cycloid(v: Double, r: Double): Angle => Vec =
    speed(v).andThen(radius(r))

  /** Inspired by "Creating Symmetry" by Frank Farris. */
  def c1(amplitude: Double) =
    cycloid(1.0, amplitude) |+| cycloid(6.0, 0.5 * amplitude) |+|
(speed(14.0)
      .andThen(phase(90.degrees))
      .andThen(radius(0.33 * amplitude)))

  val randomCycloid: Random[Double => Angle => Vec] =
    for {
      d <- Random.int(3, 25) // Degree of symmetry
      n <- Random.natural(d) // Offset from d
      m1 <- Random.int(1, 5)
      m2 <- Random.int(m1, m1 + 5)
    } yield amplitude =>
      cycloid(n, amplitude) |+| cycloid(
        m1 * d + n,
        0.5 * amplitude
```

```
      ) |+| phase(90.degrees).andThen(cycloid(m2 * d + n, 0.33 *
amplitude))

  def drawCycloid(
      cycloid: Angle => Vec,
      start: Angle = 0.degrees,
      stop: Angle = 720.degrees,
      steps: Int = 1000
  ): Picture[Unit] =
    interpolatingSpline[Algebra](
      (start)
        .upTo(stop)
        .forSteps(steps)
        .map(angle => cycloid(angle).toPoint)
        .toList
    )

  /** Repeatedly draw a cycloid with increasing size and a slow
turn */
  def drawCycloids(cycloid: Double => Angle => Vec):
Picture[Unit] =
    (0.0)
      .upTo(1.0)
      .forSteps(30)
      .map { m =>
        drawCycloid(cycloid(350 * m + 100)).rotate(30.degrees *
m)
      }
      .toList
      .allOn

  // val picture = drawCycloids(c1)
  val picture = randomCycloid.map(drawCycloids).run

  val frame = Frame.default

  picture.drawWithFrame(frame)
  picture.write[Png]("cycloid.png", frame)
  picture.write[Pdf]("cycloid.pdf", frame)
}
```

15. Tagless Final Interpreters

In this chapter we'll explore the codata approach to interpreters, building up to a strategy known as **tagless final**. Along the way we will build two interpreters: one for terminal interaction and one for user interfaces.

We've seen the duality between data and codata in many places, starting with Chapter 4. This chapter will begin by applying that duality to build an interpreter using codata, which contrasts with the data approach we saw in Section 6.2. This will illustrate the technique and give us a concrete example to discuss its shortcoming. In particular we'll see that extensibility is limited, a problem we first encountered in Section 4.5.

Solving the problem of extensibility, otherwise known as the **expression problem**, will lead us to tagless final. In the context of interpreters, solving the expression problem means allowing extensibility of both the programs we write and the interpreters that run them. We'll start with the standard encoding of tagless final in Scala, and see that it is a bit painful to use in practice. We'll then develop an alternative encoding that is easier to use. Solving the expression problem allows for very expressive code but it adds complexity, so we'll finish by talking about when tagless final is appropriate and when it's best to use a different strategy.

15.1. Codata Interpreters

In this section we'll explore codata interpreters, using a DSL for terminal interaction as a case study. The terminal is familiar to

most programmers, and terminal applications are common for developer focused tools. Most terminal features are controlled by writing so-called escape codes to the terminal. However, applications benefit from higher-level abstractions, motivating textual user interface (TUI) libraries that present a more ergonomic interface[^tuis]. Our library will showcase codata interpreters, monads, and the central role of designing for composition and reasoning.

15.1.1. The Terminal

The modern terminal is an accretion of features that started with the VT-100 in 1978 and continues [to this day][kitty-kp]. Most terminal features are accessed by reading and writing ANSI escape codes, which are sequence of characters starting with the escape character. We will work only with escape codes that change the text style. This allows us to produce interesting output, and raises all the design issues we want to address, but keeps the system simple. The ideas here are extended to a more complete system in the Terminus library.

The code below is written so that with a single change it can pasted into a file and run with any recent version of Scala with just `scala <filename>`. The required change is to add the `@main` annotation before the method `go`. That is, change

```
def go(): Unit =
```

to

```
@main def go(): Unit =
```

(This is due to a limitation of the software that compiles the code in the book.)

The examples should work with any terminal from the last 40 odd years. If you're on Windows you can use Windows Terminal, WSL, or another terminal that runs on Windows such as WezTerm.

15.1.2. Color Codes

We will start by writing color codes straight to the terminal. This will introduce us to controlling the terminal, and show the problems of using ANSI escape codes directly. Here's our starting point:

```
val csiString = "\u001b["

def printRed(): Unit =
  print(csiString)
  print("31")
  print("m")

def printReset(): Unit =
  print(csiString)
  print("0")
  print("m")

def go(): Unit =
  print("Normal text, ")
  printRed()
  print("now red text, ")
  printReset()
  println("and now back to normal.")
```

Try running the above code (e.g. add the `@main` annotation to `go`, save it to a file `ColorCodes.scala` and run `scala ColorCodes.scala`.) You should see text in the normal style for your terminal, followed by text colored red, and then some more text in the normal style. The change in color is controlled by writing escape codes. These are strings starting with `ESC` (which is the character `'\u001b'`) followed by `'['`. This is the value of `csiString` (where CSI stands for Control Sequence Introducer). The CSI is followed by a string indicating the text style to use, and

ended with a `"m"` The string `"\u001b[31m"` tells the terminal to set the text foreground color to red, and the string `"\u001b[0m"` tells the terminal to reset all text styling to the default.

15.1.3. The Trouble with Escape Codes

Escape codes are simple for the terminal to process but lack useful structure for the programmer generating them. The code above shows one potential problem: we must remember to reset the color when we finish a run of styled text. This problem is no different to that of remembering to free manually allocated memory, and the long history of memory safety problems in C programs show us that we cannot expect to do this reliably. Luckily, we're unlikely to crash our program if we forget an escape code!

To solve this problem we might decide to write functions like `printRed` below, which prints a colored string and resets the styling afterwards.

```
val csiString = "\u001b["
val redCode = s"${csiString}31m"
val resetCode = s"${csiString}0m"

def printRed(output: String): Unit =
  print(redCode)
  print(output)
  print(resetCode)

def go(): Unit =
  print("Normal text, ")
  printRed("now red text, ")
  println("and now back to normal.")
```

Changing color is the not the only way that we can style terminal output. We can also, for example, turn text bold. Continuing the above design gives us the following.

```
val csiString = "\u001b["
val redCode = s"${csiString}31m"
val resetCode = s"${csiString}0m"
val boldOnCode = s"${csiString}1m"
val boldOffCode = s"${csiString}22m"

def printRed(output: String): Unit =
  print(redCode)
  print(output)
  print(resetCode)

def printBold(output: String): Unit =
  print(boldOnCode)
  print(output)
  print(boldOffCode)

def go(): Unit =
  print("Normal text, ")
  printRed("now red text, ")
  printBold("and now bold.\n")
```

This works, but what if we want text that is *both* red and bold? We cannot express this with our current design, without creating methods for every possible combination of styles. Concretely this means methods like

```
def printRedAndBold(output: String): Unit =
  print(redCode)
  print(boldOnCode)
  print(output)
  print(resetCode)
```

This is not feasible to implement for all possible combinations of styles. The root problem is that our design is not compositional: there is no way to build a combination of styles from smaller pieces.

15.1.4. Programs and Interpreters

To solve the problem above we need `printRed` and `printBold` to accept not a `String` to print but a program to run. We don't need to know what these programs do; we just need a way to run them. Then the combinators `printRed`, `printBold`, and so on, can also return programs. These programs will set the style appropriately before running their program parameter, and reset it after the parameter program has finished running. By accepting and returning programs the combinators have the property of closure, meaning that type of the input (a program) is the same as the type of the output. Closure in turn makes composition possible.

How should we represent a program? We will choose codata and in particular functions, the simplest form of codata. In the code below we define the type `Program[A]`, which is a function `() => A`. The interpreter, which is the thing that runs programs, is just function application. To make it clearer when we are running programs I have a created method `run` that does just that.

```
type Program[A] = () => A

val csiString = "\u001b["
val redCode = s"${csiString}31m"
val resetCode = s"${csiString}0m"
val boldOnCode = s"${csiString}1m"
val boldOffCode = s"${csiString}22m"

def run[A](program: Program[A]): A = program()

def print(output: String): Program[Unit] =
  () => Console.print(output)

def printRed[A](output: Program[A]): Program[A] =
  () => {
    run(print(redCode))
    val result = run(output)
    run(print(resetCode))

    result
```

```
  }


def printBold[A](output: Program[A]): Program[A] =
  () => {
    run(print(boldOnCode))
    val result = run(output)
    run(print(boldOffCode))

    result
  }


def go(): Unit =
  run(() => {
    run(print("Normal text, "))
    run(printRed(print("now red text, ")))
    run(printBold(print("and now bold ")))
    run(printBold(printRed(print("and now bold and red.\n"))))
  })
```

Notice that we have the usual structure for an algebra, which we first met in Section 6.2.1:

1. we have a constructor in `print`;
2. we have two combinators in `printRed` and `printBold`; and
3. we have an interpreter in `run`.

This code works, for the example we have chosen, but there are two issues: composition and ergonomics. That we have a problem with composition is perhaps surprising, as that's the problem we set out to solve. We have made the system compositional in some aspects, but there are still ways in which it does not work correctly. For example, take the following code:

```
run(printBold(() => {
  run(print("This should be bold, "))
  run(printBold(print("as should this ")))
  run(print("and this.\n"))
}))
```

We would expect output like

This should be bold, as should this and this

but we get

This should be bold, as should this and this.

The inner call to `printBold` resets the bold styling when it finishes, which means the surrounding call to `printBold` does not have effect on later statements.

The issue with ergonomics is that this code is tedious and error-prone to write. We have to pepper calls to `run` in just the right places, and even in these small examples I found myself making mistakes. This is actually another failing of composition, because we don't have methods to combine together programs. For example, we don't have methods to say that the program above is the sequential composition of three sub-programs.

We can solve the first problem by keeping track of the state of the terminal. If `printBold` is called within a state that is already printing bold it should do nothing, otherwise it should update the state to indicate bold styling has been turned on. This means the type of programs changes from `() => A` to `Terminal => (Terminal, A)`, where `Terminal` holds the current state of the terminal.

To solve the second problem we're looking for a way to sequentially compose programs. Remember programs have type `Terminal => (Terminal, A)` and pass around the state in `Terminal`. When you hear the phrase "sequentially compose", or see that type, your monad sense might start tingling. You are correct: this is an instance of the state monad, which we first met in Section 10.9.

Using Cats we can define

```
import cats.data.State
type Program[A] = State[Terminal, A]
```

assuming some suitable definition of `Terminal`. Let's accept this definition for now, and focus on defining `Terminal`.

`Terminal` has two pieces of state: the current bold setting and the current color. (The real terminal has much more state, but these are representative and modelling additional state does not introduce any new concepts.) The bold setting could simply be a toggle that is either on or off, but when we come to the implementation it will be easier to work with a counter that records the depth of the nesting. The current color must be a stack. We can nest color changes, and the color should change back to the surrounding color when a nested level exits. Concretely, we should be able to write code like

```
printBlue(.... printRed(...) ...)
```

and have output in blue or red as we would expect.

Given this we can define `Terminal` as

```
final case class Terminal(bold: Int, color: List[String]) {
  def boldOn: Terminal = this.copy(bold = bold + 1)
  def boldOff: Terminal = this.copy(bold = bold - 1)
  def pushColor(c: String): Terminal = this.copy(color = c ::
color)
  // Only call this when we know there is at least one color on
the
  // stack
  def popColor: Terminal = this.copy(color = color.tail)
  def peekColor: Option[String] = this.color.headOption
}
```

where we use `List` to represent the stack of color codes. (We could also use a mutable stack, as working with the state monad ensures the state will be threaded through our program.) I've also defined some convenience methods to simplify working with the state.

With this in place we can write the rest of the code, which is shown below. Compared to the previous code I've shortened a few method names and abstracted the escape codes. Remember this

code can be directly executed by `scala`. Just copy it into a file (e.g. `Terminal.scala`), add the `@main` annotation to `go`, and run `scala Terminal.scala`.

```
//> using dep org.typelevel::cats-core:2.13.0

import cats.data.State
import cats.syntax.all.*

object AnsiCodes {
  val csiString: String = "\u001b["

  def csi(arg: String, terminator: String): String =
    s"${csiString}${arg}${terminator}"

  // SGR stands for Select Graphic Rendition.
  // All the codes that change formatting are SGR codes.
  def sgr(arg: String): String =
    csi(arg, "m")

  val reset: String = sgr("0")
  val boldOn: String = sgr("1")
  val boldOff: String = sgr("22")
  val red: String = sgr("31")
  val blue: String = sgr("34")
}

final case class Terminal(bold: Int, color: List[String]) {
  def boldOn: Terminal = this.copy(bold = bold + 1)
  def boldOff: Terminal = this.copy(bold = bold - 1)
  def pushColor(c: String): Terminal = this.copy(color = c ::
color)
  // Only call this when we know there is at least one color on
the
  // stack
  def popColor: Terminal = this.copy(color = color.tail)
  def peekColor: Option[String] = this.color.headOption
}
object Terminal {
  val empty: Terminal = Terminal(0, List.empty)
}

type Program[A] = State[Terminal, A]
object Program {
  def print(output: String): Program[Unit] =
    State[Terminal, Unit](
```

```
      terminal => (terminal, Console.print(output))
    )

  def bold[A](program: Program[A]): Program[A] =
    for {
      _ <- State.modify[Terminal] { terminal =>
        if terminal.bold == 0 then
Console.print(AnsiCodes.boldOn)
        terminal.boldOn
      }
      a <- program
      _ <- State.modify[Terminal] { terminal =>
        val newTerminal = terminal.boldOff
        if terminal.bold == 0 then
Console.print(AnsiCodes.boldOff)
        newTerminal
      }
    } yield a

  // Helper to construct methods that deal with color
  def withColor[A](code: String)(program: Program[A]): Program[A]
=
    for {
      _ <- State.modify[Terminal] { terminal =>
        Console.print(code)
        terminal.pushColor(code)
      }
      a <- program
      _ <- State.modify[Terminal] { terminal =>
        val newTerminal = terminal.popColor
        newTerminal.peekColor match {
          case None    => Console.print(AnsiCodes.reset)
          case Some(c) => Console.print(c)
        }
        newTerminal
      }
    } yield a

  def red[A](program: Program[A]): Program[A] =
    withColor(AnsiCodes.red)(program)

  def blue[A](program: Program[A]): Program[A] =
    withColor(AnsiCodes.blue)(program)

  def run[A](program: Program[A]): A =
    program.runA(Terminal.empty).value
}
```

```
def go(): Unit = {
  val program =
    Program.blue(
      Program.print("This is blue ") >>
        Program.red(Program.print("and this is red ")) >>
        Program.bold(Program.print("and this is blue and bold "))
    ) >>
      Program.print("and this is back to normal.\n")

  Program.run(program)
}
```

Having defined the structure of `Terminal`, the majority of the remaining code manipulates the `Terminal` state. Most of the methods on `Program` have a common structure that specifies a state change before and after the main program runs.

Notice we don't need to implement combinators like `flatMap` or `>>` because we get them from the `State` monad. This is one of the big benefits of reusing abstractions like monads: we get a full library of methods without doing additional work.

15.1.5. Composition and Reasoning

In Section 1.2.1 I argued that the core of functional programming is reasoning and composition. Both of these are central to this case study. We've explicitly designed the DSL for ease of reasoning. Indeed that's the whole point of creating a DSL instead of just spitting control codes at the terminal. An example is how we paid attention to making sure nested calls work as we'd expect. Composition comes in at two levels: both our design and our implementation are compositional. Within the case study we discussed compositionality in the design. Implementationally, a `Program` is a composition of the state monad and the functions inside the state monad. The state monad provides the sequential

flow of the `Terminal` state, and the functions provide the domain specific actions.

15.1.6. Codata and Extensibility

We made a seemingly arbitrary choice to use a codata interpreter. Let's now explore this choice and its implications.

We described codata as programming to an interface. The interface for functions is essentially one method: the ability to apply them. This corresponds to the single interpretation we have for `Program`: run it and carry out the effects therein. If we wanted to have multiple interpretations (such as logging the `Terminal` state or saving the output to a buffer) we would need to have a richer interface. In Scala this would be a `trait` or `class` exposing more than one method.

Keen readers will recall that data makes it easy to add new interpreters but hard to add new operations, while codata makes it easy to add new operations but hard to add new interpreters. We see that in action here. For example, it's trivial to add a new color combinator by defining a method like the below.

```
def green[A](program: Program[A]): Program[A] =
  withColor(AnsiCodes.sgr("32"))(program)
```

However, changing `Program` to something that allows more interpretations requires changing all of the existing code.

Another advantage of codata is that we can mix in arbitrary other Scala code. For example, we can use `map` like shown below.

```
Program.print("Hello").map(_ => 42)
```

Using the native representation of programs (i.e. functions) gives us the entire Scala language for free. In a data representation we

have to reify every kind of expression we wish to support. There is a downside to this as well: we get Scala semantics whether we like them or not. A codata representation would not be appropriate if we wanted to make an exotic language that worked in a different way.

We could factor the interpreter in different ways, and it would still be a codata interpreter. For example, we could put a method to write to the terminal on the `Terminal` type. This would give us a bit more flexibility as changing the implementation of `Terminal` could, say, write to a network socket or a terminal embedded in a browser. We still have the limitation that we cannot create truly different interpretations, such as serializing programs to disk, with the codata approach. We'll address this limitation in the next section where we look at tagless final.

[^tuis]: If you're interested in TUI libraries you might like to look at the brilliantly named ratatui for Rust, brick for Haskell, or Textual for Python.

15.2. Tagless Final Interpreters

We'll now explore tagless final, an extension to the basic codata interpreter. In the terminal DSL case study we used an ad-hoc process to produce the DSL, fixing problems as we uncovered them. In this section we will be more systematic, illustrating how we can apply strategies to derive code. This will in turn make it clearer how we can derive tagless final for the basic codata interpreter.

We'll start by being explicit about the role of the different types in the codata interpreter. Following Section 6.2.1, remember there are three different kinds of methods in an algebra:

- constructors, with type `A => Program`,

- combinators, with type `Program => Program`, and
- interpreters, with type `Program => A`.

In the terminal DSL we defined the `Program` type as

```
type Program[A] = State[Terminal, A]
```

There is a single constructor, `print`, with type `String => Program[Unit]`. All of the methods that change the output style, such as `bold`, `red`, and `blue`, are combinators with the type `Program[A] => Program[A]`. Finally, there is a single interpreter, function application, with type `Program[A] => A`.

In a codata interpreter the available interpretations are limited to the methods available on the `Program` type. The terminal DSL represents programs as functions, and therefore only has a single interpretation available. The key idea in tagless final, to get around this restriction, is to parameterize the `Program` type by the program operations. It's not entirely clear what this means, so let's see a simple example of tagless final to illustrate it.

Our example will be arithmetic expressions. This is not a particularly compelling example, but it is familiar. This means we can focus on the details of tagless final without any confusion about the domain. We'll see a more compelling example soon.

We'll start with a data interpreter, convert it to a codata interpreter, and then apply tagless final. Here's our program type, defined using an algebraic data type. We don't need to explicitly define constructors as they come as part of the ADT.

```
enum Expr {
  case Add(l: Expr, r: Expr)
  case Sub(l: Expr, r: Expr)
  case Mul(l: Expr, r: Expr)
  case Div(l: Expr, r: Expr)

  case Literal(value: Double)
}
```

We will now define two interpreters, one that evaluates `Expr` to a `Double` and one that prints them to `String`. They are implemented using structural recursion.

```
object EvalInterpreter {
  import Expr.*

  def eval(expr: Expr): Double =
    expr match {
      case Add(l, r) => eval(l) + eval(r)
      case Sub(l, r) => eval(l) - eval(r)
      case Mul(l, r) => eval(l) * eval(r)
      case Div(l, r) => eval(l) / eval(r)
      case Literal(value) => value
    }
}
object PrintInterpreter {
  import Expr.*

  def print(expr: Expr): String =
    expr match {
      case Add(l, r) => s"(${print(l)} + ${print(r)})"
      case Sub(l, r) => s"(${print(l)} - ${print(r)})"
      case Mul(l, r) => s"(${print(l)} * ${print(r)})"
      case Div(l, r) => s"(${print(l)} / ${print(r)})"
      case Literal(value) => value.toString
    }
}
```

Finally, let's see a quick example. We start by defining an expression, in this case representing `1 + 2`.

```
val onePlusTwo = Expr.Add(Expr.Literal(1), Expr.Literal(2))
```

Now we can interpret this expression in two different ways.

```
EvalInterpreter.eval(onePlusTwo)
// res0: Double = 3.0
PrintInterpreter.print(onePlusTwo)
// res1: String = "(1.0 + 2.0)"
```

We have the usual trade-off for data: we can easily add more interpreters, but we cannot extend the program type with new operations.

Let's now convert this to codata. The interpreters become methods on the `Expr` type.

```
trait Expr {
  def eval: Double
  def print: String
}
```

The constructors and combinators create instances of `Expr`. We could define explicit subtypes of `Expr` but here I've used anonymous subtypes to keep the code more compact. The implementation uses structural corecursion.

```
trait Expr {
  def eval: Double
  def print: String

  def +(that: Expr): Expr = {
    val self = this
    new Expr {
      def eval: Double = self.eval + that.eval
      def print: String = s"(${self.print} + ${that.print})"
    }
  }

  def -(that: Expr): Expr = {
    val self = this
    new Expr {
      def eval: Double = self.eval - that.eval
      def print: String = s"(${self.print} - ${that.print})"
    }
  }

  def *(that: Expr): Expr = {
    val self = this
    new Expr {
      def eval: Double = self.eval * that.eval
      def print: String = s"(${self.print} * ${that.print})"
    }
```

```
  }

  def /(that: Expr): Expr = {
    val self = this
    new Expr {
      def eval: Double = self.eval / that.eval
      def print: String = s"(${self.print} / ${that.print})"
    }
  }
}
object Expr {
  def literal(value: Double): Expr =
    new Expr {
      def eval: Double = value
      def print: String = value.toString
    }
}
```

Now we can create the same example as before

```
val onePlusTwo = Expr.literal(1) + Expr.literal(2)
```

and interpret it as before

```
onePlusTwo.eval
// res4: Double = 3.0
onePlusTwo.print
// res5: String = "(1.0 + 2.0)"
```

As expected we have the opposite extensibility. We can add new program operations such as `sin`.

```
def sin(expr: Expr): Expr = {
  new Expr {
    def eval: Double = Math.sin(expr.eval)
    def print: String = s"sin(${expr.print})"
  }
}
```

However we are restricted to the two interpretations we have defined on `Expr`, `eval` and `print`.

We now need to introduce a bit of terminology, so we can talk more precisely. We will use the term **program algebras** to refer to constructors and combinators, as they are the portion of the algebra used to create programs. We must also distinguish between programs and the **program type**. In the example above, `Expr` is the program type. A program is an expression that produces a value of the program type.

The core of tagless final is to:

1. define program algebras parameterized by their program type, and
2. parameterize programs by the program algebras they depend on.

For the example we have just seen we could define a program algebra as follows:

```
trait Arithmetic[Expr] {
  def +(l: Expr, r: Expr): Expr
  def -(l: Expr, r: Expr): Expr
  def *(l: Expr, r: Expr): Expr
  def /(l: Expr, r: Expr): Expr

  def literal(value: Double): Expr
}
```

Notice how it is parameterized by a type `Expr`. This is the program type.

Now we can create a program. Here's the same example we saw above, but written in tagless final style.

```
def onePlusTwo[Expr](arithmetic: Arithmetic[Expr]): Expr =
  arithmetic.+(arithmetic.literal(1.0), arithmetic.literal(2.0))
```

Notice the distinction between a program and the program type: a program creates a value of the program type, but a program is not itself of the program type. In tagless final a program is a function from program algebras to the program type.

We can finish our example by creating an instance of `Arithmetic`.

```
object DoubleArithmetic extends Arithmetic[Double] {
  def +(l: Double, r: Double): Double =
    l + r
  def -(l: Double, r: Double): Double =
    l - r
  def *(l: Double, r: Double): Double =
    l * r
  def /(l: Double, r: Double): Double =
    l / r

  def literal(value: Double): Double =
    value
}
```

Now we can run our example.

```
onePlusTwo(DoubleArithmetic)
// res7: Double = 3.0
```

Tagless final gives us both forms of extensibility. We can add a new interpreter.

```
object PrintArithmetic extends Arithmetic[String] {
  def +(l: String, r: String): String =
    s"($l + $r)"
  def -(l: String, r: String): String =
    s"($l - $r)"
  def *(l: String, r: String): String =
    s"($l * $r)"
  def /(l: String, r: String): String =
    s"($l / $r)"

  def literal(value: Double): String =
    value.toString
}
```

This works in the same way.

```
onePlusTwo(PrintArithmetic)
// res8: String = "(1.0 + 2.0)"
```

We can also define new program algebras

```
trait Trigonometry[Expr] {
  def sin(expr: Expr): Expr
}
```

and use them in a program.

```
def sinOnePlusTwo[Expr](
    arithmetic: Arithmetic[Expr],
    trigonometry: Trigonometry[Expr]
  ): Expr =
  trigonometry.sin(onePlusTwo(arithmetic))
```

Notice that we are using composition here; the program `sinOnePlusTwo` reuses `onePlusTwo`.

A few notes before we move on.

In this example the program type is the same as the type we interpret to. We can use `Double` as the program type when we want to interpret to `Double`, and likewise with `String`. This is usually *not* the case. It's just a coincidence of using arithmetic that we don't need any additional information to calculate the final result, and hence the program type and interpreter result type are the same.

There is quite a high notational overhead of tagless final, compared to the data and codata interpreters. We'll address this later, and end up with an encoding of tagless final in Scala that looks like ordinary code. First, however, we'll introduce a more compelling example: cross-platform user interfaces.

15.3. Algebraic User Interfaces

Changing the interpretation of our terminal programs is more a theoretical than a practical problem. While it is true that different

interpretations, such as saving to a text buffer, or tracing the state changes, will have niche uses, the vast majority of the time we'll use the default interpretation. A much more motivating example is a cross-platform user interface library. Frameworks such as Flutter, React Native, and Capacitor derive a lot of their value by allowing programmers to define a single interface that works across web and mobile. We will build such a library here, but our ambitions are a bit reduced: we will create a terminal backend but leave other backends up to your inspiration and perspiration.

Broadly speaking, there are two kinds of user interfaces. When operating, say, a digital musical instrument, we require a continuous stream of values from the user interface. In contrast, when working with a form we only require the values once, when the form is submitted. Modelling a continuous stream of values is certainly doable (see functional reactive programming) but it adds inessential complexity. Therefore we will stick with the simpler kind of interface where the user submits values once.

We'll consider each of constructors, combinators, and interpreters in turn.

Constructors will define the atomic units of user interface for our library. The granularity we use here trades off expressivity and convenience. At the very lowest level we could work with vertex buffers and the like, which would make our library a general purpose graphics library. This gives us the ultimate flexibility but is far too low level for this case study. At a higher level we might think of atomic units as user interface elements like labels, buttons, text inputs, and so on. This is the level at which HTML operates. At this level we still usually require multiple elements to construct a complete control. For example, in HTML what is conceptually a single form field will often consist of separate DOM elements for the label, the input control, and the control to show validation errors, plus some Javascript to add interactivity. We will go even higher level. Our atomic elements will specify the kind of

user input we wants, such as a choice between a number of elements, and leave it up to the interpreter to decide how to render this using the platform's available controls. For example, we could render a one-of-many control using either radio buttons or a dropdown, or choose between the two depending on the number of choices. We'll also add labels, and optional validation rules, to each element. Let's model two such elements, to illustrate the idea.

```
type Validation[A] = A => Either[String, A]

// The validation rule that always succeeds
def succeed[A](value: A): Either[String, A] = Right(value)

trait Controls[Ui[_]] {
  def textInput(
      label: String,
      placeholder: String,
      validation: Validation[String] = succeed
  ): Ui[String]

  def choice[A](label: String, options: Seq[(String, A)]): Ui[A]
}
```

Here we defined two controls:

- `textInput`, which creates a text input where the user can enter any text that passes the validation rule; and
- `choice`, which gives the user a choice of one of the given items.

Notice how our modelling decisions restrict our expressivity. For example, `textInput` has a placeholder, which is displayed before the user enters input, but does not have a default value. By reducing expressivity we gain convenience. If the user's requirements fit our model it is very easy to create controls. Also notice that we don't have any way to control the appearance of controls. This is deliberate; we are pushing that concern into the interpreters.

These controls generate an element of the program type `Ui`. Each particular interpreter, corresponding to a backend, will choose a

concrete type for `Ui` corresponding to the needs of the user interface toolkit it is working with.

These two constructors are enough to illustrate the idea, so we will move on to combinators. In the context of user interfaces the most common combinators will specify the layout of elements. As with the constructors there are a number of possible designs: we could allow a lot of precision in layout, as CSS does for HTML, or we could provide a few pre-defined layouts, or we could even push layout into the interpreter. In keeping with our design for the constructors, and with the need to keep things simple, we will go with a very high-level design. Our single combinator, `and`, only specifies that two elements should occur together. It leaves it up to the interpreter how this should be rendered on the screen.

```
trait Layout[Ui[_]] {
  def and[A, B](first: Ui[A], second: Ui[B]): Ui[(A, B)]
}
```

You might have noticed that `and` is another name for `product` from `Semigroupal`, which we encountered in Section 12.1. It has exactly the same signature, apart from the name, and it represents the same concept as applied to user interfaces.

At this point we have defined two program algebras, `Controls` and `Layout`, and shown examples of both constructors and combinators. The next step is to create an interpreter. Here we are going to create an extremely simple interpreter to illustrate the idea and to allow us to show how to write programs using our algebras. More full featured interpreters are certainly possible, but they don't introduce any new concepts and take considerably more code.

Our interpreter will use the Console IO features of the standard library to interact with the user.

```scala
import cats.syntax.all.*
import scala.io.StdIn
import scala.util.Try

type Program[A] = () => A

object Simple extends Controls[Program], Layout[Program] {
  def and[A, B](first: Program[A], second: Program[B]):
Program[(A, B)] =
    // Use Cats Semigroupal for Function0
    (first, second).tupled

  def textInput(
      label: String,
      placeholder: String,
      validation: Validation[String] = succeed
  ): Program[String] =
    () => {
      def loop(): String = {
        println(s"$label (e.g. $placeholder):")
        val input = StdIn.readLine

        validation(input).fold(
          msg => {
            println(msg)
            loop()
          },
          value => value
        )
      }

      loop()
    }

  def choice[A](label: String, options: Seq[(String, A)]):
Program[A] =
    () => {
      def loop(): A = {
        println(label)
        options.zipWithIndex.foreach { case ((desc, _), idx) =>
          println(s"$idx: $desc")
        }

        Try(StdIn.readInt).fold(
          _ => {
            println("Please enter a valid number.")
            loop()
```

```
          },
          idx => {
            if idx >= 0 && idx < options.size then options(idx)
(1)
            else {
              println("Please enter a valid number.")
              loop()
            }
          }
        )
      }

      loop()
    }
}
```

Now we can implement a simple example.

```
def quiz[Ui[_]](
    controls: Controls[Ui],
    layout: Layout[Ui]
): Ui[(String, Int)] =
  layout.and(
    controls.textInput("What is your name?", "John Doe"),
    controls.choice(
      "Tagless final is the greatest thing ever",
      Seq(
        "Strongly disagree" -> 1,
        "Disagree" -> 2,
        "Neutral" -> 3,
        "Agree" -> 4,
        "Strongly agree" -> 5
      )
    )
  )
```

We can run this example with code like the following.

```
val (name, rating) = quiz(Simple, Simple)()
println(s"Hello $name!")
println(s"You gave tagless final a rating of $rating.")
```

Here is an example of interaction.

```
What is your name? (e.g. John Doe):
Noel Welsh
Tagless final is the greatest thing ever
0: Strongly disagree
1: Disagree
2: Neutral
3: Agree
4: Strongly agree
4
Hello Noel Welsh!
You gave tagless final a rating of 5.
```

We have a basic example working, but it is not very nice to work with. The way in which we write code in tagless final style is very convoluted compared to normal code. In the next section we'll see a different encoding of tagless final that gives the user a much better experience.

15.4. A Better Encoding

The basic implementation of tagless final has quite a poor developer experience. Consider the refactoring of our example below.

```
def name[Ui[_]](controls: Controls[Ui]): Ui[String] =
  controls.textInput("What is your name?", "John Doe")

def rating[Ui[_]](controls: Controls[Ui]): Ui[Int] =
  controls.choice(
    "Tagless final is the greatest thing ever",
    Seq(
      "Strongly disagree" -> 1,
      "Disagree" -> 2,
      "Neutral" -> 3,
      "Agree" -> 4,
      "Strongly agree" -> 5
    )
  )
```

```
def quiz[Ui[_]](
    controls: Controls[Ui],
    layout: Layout[Ui]
): Ui[(String, Int)] =
  layout.and(name(controls), rating(controls))
```

This style of code quickly becomes tedious to write. The method signatures are quite involved, and passing the program algebras from method to method is annoying busy work.

An improvement is to make the program algebras `given` instances. If we define accessors

```
object Controls {
  def apply[Ui[_]](using controls: Controls[Ui]): Controls[Ui] =
    controls
}

object Layout {
  def apply[Ui[_]](using layout: Layout[Ui]): Layout[Ui] =
    layout
}
```

we can then write

```
def name[Ui[_]: Controls]: Ui[String] =
  Controls[Ui].textInput("What is your name?", "John Doe")

def rating[Ui[_]: Controls]: Ui[Int] =
  Controls[Ui].choice(
    "Tagless final is the greatest thing ever",
    Seq(
      "Strongly disagree" -> 1,
      "Disagree" -> 2,
      "Neutral" -> 3,
      "Agree" -> 4,
      "Strongly agree" -> 5
    )
  )

def quiz[Ui[_]: Controls: Layout]: Ui[(String, Int)] =
  Layout[Ui].and(name, rating)
```

This is the encoding of tagless final that is common in the Scala community, but there is still a lot of notational overhead for the developer who has to write this code. We can use Scala language features to reduce the overhead of writing code using a tagless final style to the point where is a simple as standard code.

We'll use a combination of five techniques:

1. creating a base type for program algebras;
2. making the program type a type member;
3. defining a type for programs;
4. defining constructors on companion objects; and
5. using extension methods for combinators.

This is quite involved, but each step is relatively simple. Let's see how it works.

Our first step is to create a base type for algebras. This is just a trait like

```
trait Algebra[Ui[_]]
```

Our program algebras extend this trait.

```
trait Controls[Ui[_]] extends Algebra[Ui]{
  def textInput(
      label: String,
      placeholder: String,
      validation: Validation[String] = succeed
  ): Ui[String]

  def choice[A](label: String, options: Seq[(String, A)]): Ui[A]
}

trait Layout[Ui[_]] extends Algebra[Ui]{
  def and[A, B](first: Ui[A], second: Ui[B]): Ui[(A, B)]
}
```

Now we make the program type a type member.

```
trait Algebra {
  type Ui[_]
}

trait Controls extends Algebra {
  def textInput(
      label: String,
      placeholder: String,
      validation: Validation[String] = succeed
  ): Ui[String]

  def choice[A](label: String, options: Seq[(String, A)]): Ui[A]
}

trait Layout extends Algebra {
  def and[A, B](first: Ui[A], second: Ui[B]): Ui[(A, B)]
}
```

At this point we've made sufficient changes that our example program is meaningfully changed. Our starting point was

```
def quiz[Ui[_]: Controls: Layout](
    controls: Controls[Ui],
    layout: Layout[Ui]
): Ui[(String, Int)] =
  Layout[Ui].and(
    Controls[Ui].textInput("What is your name?", "John Doe"),
    Controls[Ui].choice(
      "Tagless final is the greatest thing ever",
      Seq(
        "Strongly disagree" -> 1,
        "Disagree" -> 2,
        "Neutral" -> 3,
        "Agree" -> 4,
        "Strongly agree" -> 5
      )
    )
  )
```

With the changes above we can instead write

```
def quiz(using alg: Controls & Layout): alg.Ui[(String, Int)] =
  alg.and(
    alg.textInput("What is your name?", "John Doe"),
```

```
    alg.choice(
      "Tagless final is the greatest thing ever",
      Seq(
        "Strongly disagree" -> 1,
        "Disagree" -> 2,
        "Neutral" -> 3,
        "Agree" -> 4,
        "Strongly agree" -> 5
      )
    )
  )
```

The key changes are:

1. the program algebras are a single parameter to the method, which is possible because they extend a common base type;
2. the `Ui` type parameter is no longer needed, as it has become a type member; and
3. we must now use a dependent method to specify the result type.

Our next step is to define a type for programs. Programs are conceptually functions from an algebra to a program type, so we can define such a type.

```
trait Program[-Alg <: Algebra, A] {
  def apply(alg: Alg): alg.Ui[A]
}
```

Pay particular attention to the result type, `alg.Ui[A]`. As `Program` requires a dependent method type it cannot be a standard function.

The example now becomes

```
val quiz =
  new Program[Controls & Layout, (String, Int)] {
    def apply(alg: Controls & Layout) =
      alg.and(
        alg.textInput("What is your name?", "John Doe"),
        alg.choice(
```

```
          "Tagless final is the greatest thing ever",
          Seq(
            "Strongly disagree" -> 1,
            "Disagree" -> 2,
            "Neutral" -> 3,
            "Agree" -> 4,
            "Strongly agree" -> 5
          )
        )
      )
  }
```

Programs are now values instead of methods. Notice that first type parameter of `Program` declares all the program algebras the program requires. It's still quite involved to write this code, though we can simplify it a bit by using the *single abstract method* technique, which means a `trait` with a single abstract method (like `Program`) can be implemented with a function.

```
val quiz: Program[Controls & Layout, (String, Int)] =
  (alg: Controls & Layout) =>
    alg.and(
      alg.textInput("What is your name?", "John Doe"),
      alg.choice(
        "Tagless final is the greatest thing ever",
        Seq(
          "Strongly disagree" -> 1,
          "Disagree" -> 2,
          "Neutral" -> 3,
          "Agree" -> 4,
          "Strongly agree" -> 5
        )
      )
    )
```

Programs-as-values is the key that unlocks the next two improvements. The first is to define constructors as methods on companion objects.

```
object Controls {
  def textInput(
      label: String,
```

```
      placeholder: String,
      validation: Validation[String] = succeed
  ): Program[Controls, String] =
    alg => alg.textInput(label, placeholder, validation)

  def choice[A](
    label: String,
    options: Seq[(String, A)]
  ): Program[Controls, A] =
    alg => alg.choice(label, options)
}
```

This works because methods can now return programs.

The second and final improvement is to define extension methods for combinators. Since we only have one combinator, `and`, that means a single extension method.

```
extension [Alg <: Algebra, A](p: Program[Alg, A]) {
  def and[Alg2 <: Algebra, B](
    second: Program[Alg2, B]
  ): Program[Alg & Alg2 & Layout, (A, B)] =
    alg => alg.and(p(alg), second(alg))
}
```

Pay particular attention to how the types are defined for this extension method. We define the extension on a `Program` requiring algebras `Alg`. The parameter to the `and` method is a `Program` requiring algebras `Alg2`. The result requires algebras `Alg & Alg2 & Layout`, which is the union of the algebras required by the two programs and the `Layout` algebra. In this way the combinators build up the algebras required for the program.

The net result is that users can write

```
val quiz  =
  Controls
    .textInput("What is your name?", "John Doe")
    .and(
      Controls.choice(
        "Tagless final is the greatest thing ever",
```

```
        Seq(
          "Strongly disagree" -> 1,
          "Disagree" -> 2,
          "Neutral" -> 3,
          "Agree" -> 4,
          "Strongly agree" -> 5
        )
      )
    )
```

which looks just like normal code. The type of `quiz` shows that type inference has correctly inferred all the needed program algebras.

```
quiz
// res0: Program[Controls & Layout, Tuple2[String, Int]] =
repl.MdocSession$MdocApp$
$Lambda$13618/0x0000000803a75840@669e8eb8
```

This encoding requires more work from the library developer. However this is a one off cost, and result is that library users write much simpler code. For most applications of tagless final I think this is an appropriate trade off.

15.5. Conclusions

In this chapter we looked at codata interpreters, and their extension to tagless final. Tagless final is particularly interesting because it solves the expression problem, allowing us to extend both the operations a program can perform and the interpretations of that program.

Our exploration of tagless final nicely illustrates the distinction between theory and craft introduced in Section 1.1. We saw two different encoding of tagless final in Scala (three, if we count using context bounds as a different encoding). They are both tagless final at the theory level, but are very different to implement or use as a

programmer. The “standard” encoding is relatively easy to implement for the library author, but tedious and potentially confusing for the user. The improved encoding places more work on the library author, but the user writes code in a natural style.

Tagless final is very powerful and it can be tempting to use it everywhere. I want to caution against this urge. Tagless final can cause problems, both for the author and the user. From the user’s point of view everything works fine until they make a mistake. Then the errors can be confusing. Consider this code, where we have missed a parameter to `and`.

```
Controls.textInput("Name", "John Doe").and()
// error:
// missing argument for parameter second of method and in object
MdocApp: (second: repl.MdocSession.MdocApp.Program[Alg2, B]):
//   repl.MdocSession.MdocApp.Program[
//     Alg & Alg2 & repl.MdocSession.MdocApp.Layout, (String, B)]
// Controls.textInput("Name", "John Doe").and()
//                                        ^^^
```

The error message *does* tell us the problem, but it exposes a lot of the internal machinery that the user is not normally exposed to, and hence they’ll probably have difficult understanding. A straightforward data or codata interpreter does not have this problem.

From the library author’s point of view, it is a lot more work to create tagless final code. It can also be difficult to onboard new developers to this code, as the techniques are not familiar to most.

As always, the applicability of tagless final comes down to the context in which it is used. In cases where the extensibility is truly justified it is a powerful tool. In other cases it just introduces unwarranted complexity.

The term “expression problem” was first introduced in an email by Phil Wadler [89], but there are much earlier sources that discuss the same issue. (One example is William Cook. [16].) Tagless final

was first introduced in Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. [10] and expanded on in Oleg Kiselyov. [45]. It is just one of many solutions that have been proposed to the expression problem. I'm no expert on the wider field of solutions to the expression problem, but of the papers I've read the ones I'd like to highlight are object algebras [63] and data types à la carte [82]. Object algebras are, in all essentials, the same as tagless final. They were developed in object-oriented languages rather than functional programming languages, making an interesting case of convergent evolution in two distinct, but connected, fields of research. The object algebras paper is also a good read for a more formal, if brief, discussion of the theory behind the concepts we've been dealing with. Data types à la carte is a data, rather than codata, approach to the expression problem, and so makes an interesting contrast to tagless final. I find tagless final much simpler, so we have not explored data types à la carte in this book. Another noteworthy paper is Jeremy Gibbons and Nicolas Wu. [34], which discuss the duality between data and codata and its implication for embedded domain specific languages.

Tagless final was introduced using Haskell as the implementation language. The standard encoding in Scala is a direct translation of the Haskell implementation. The improved Scala encoding is my own creation. The use of the single abstract method shortcut was suggested by Jakub Kozłowski.

16. Optimizing Interpreters and Compilers

In a previous chapter we introduced interpreters as a key strategy in functional programming. In many cases simple structurally recursive interpreters are sufficient. However, in a few cases we need more performance than they can offer so in this chapter we'll turn to optimization. This is a huge subject, which we cannot hope to cover in just one book chapter. Instead we'll focus on two techniques that I believe use key ideas found in more complex techniques: algebraic manipulation and compilation to a virtual machine.

We'll start looking at algebraic manipulation, returning to the regular expression example we used earlier. We'll then move to virtual machine, this time using a simple arithmetic interpreter example. We'll see how we can compile code to a stack machine, and then look at some of the optimizations that are available when we use a virtual machine.

16.1. Algebraic Manipulation

Reifying a program represents it as a data structure. We can **rewrite** this data structure to several ends: as a way to simplify and therefore optimize the program being interpreted, but also as a general form of computation implementing the interpreter. In this section we're going to return to our regular expression example, and show how rewriting can be used perform both of these tasks.

We will use a technique known as regular expression derivatives. Regular expression derivatives provide a simple way to match a regular expression against input (with the correct semantics for union, which you may recall we didn't deal with in the previous chapter). The derivative of a regular expression, with respect to a character, is the regular expression that remains after matching that character. Say we have the regular expression that matches the string `"osprey"`. In our library this would be `Regexp("osprey")`. The derivative with respect to the character `o` is `Regexp("sprey")`. In other words it's the regular expression that is looking for the string `"sprey"`. The derivative with respect to the character `a` is the regular expression that matches nothing, which is written `Regexp.empty` in our library. To take a more complicated example, the derivative with respect to `c` of `Regexp("cats").repeat` is `Regexp("ats") ++ Regexp("cats").repeat`. This indicates we're looking for the string `"ats"` followed by zero or more repeats of `"cats"`

All we need to do to determine if a regular expression matches some input is to calculate successive derivatives with respect to the characters in the input in the order in which they occur. If the resulting regular expression matches the empty string then we have a successful match. Otherwise it has failed to match.

To implement this algorithm we need three things:

1. an explicit representation of the regular expression that matches the empty string;
2. a method that tests if a regular expression matches the empty string; and
3. a method that computes the derivative of a regular expression with respect to a given character.

Our starting point is the basic reified interpreter we developed in the previous chapter. This is the simplest code and therefore the easiest to work with.

```
enum Regexp {
  def ++(that: Regexp): Regexp =
    Append(this, that)

  def orElse(that: Regexp): Regexp =
    OrElse(this, that)

  def repeat: Regexp =
    Repeat(this)

  def `*` : Regexp = this.repeat

  def matches(input: String): Boolean = {
    def loop(regexp: Regexp, idx: Int): Option[Int] =
      regexp match {
        case Append(left, right) =>
          loop(left, idx).flatMap(i => loop(right, i))
        case OrElse(first, second) =>
          loop(first, idx).orElse(loop(second, idx))
        case Repeat(source) =>
          loop(source, idx)
            .flatMap(i => loop(regexp, i))
            .orElse(Some(idx))
        case Apply(string) =>
          Option.when(input.startsWith(string, idx))(idx +
string.size)
        case Empty =>
          None
      }

    // Check we matched the entire input
    loop(this, 0).map(idx => idx == input.size).getOrElse(false)
  }

  case Append(left: Regexp, right: Regexp)
  case OrElse(first: Regexp, second: Regexp)
  case Repeat(source: Regexp)
  case Apply(string: String)
  case Empty
}
object Regexp {
  val empty: Regexp = Empty

  def apply(string: String): Regexp =
    Apply(string)
}
```

We want to explicitly represent the regular expression that matches the empty string, as it plays an important part in the algorithms that follow. This is simple to do: we just reify it and adjust the constructors as necessary. I've called this case "epsilon", which matches the terminology used in the literature.

```
enum Regexp {
  // ...
  case Epsilon
}
object Regexp {
  val epsilon: Regexp = Epsilon

  def apply(string: String): Regexp =
    if string.isEmpty() then Epsilon
    else Apply(string)
}
```

Next up we will create a predicate that tells us if a regular expression matches the empty string. Such a regular expression is called "nullable". The code is so simple it's easier to read it than try to explain it in English.

```
def nullable: Boolean =
  this match {
    case Append(left, right) => left.nullable && right.nullable
    case OrElse(first, second) => first.nullable ||
second.nullable
    case Repeat(source) => true
    case Apply(string) => false
    case Epsilon => true
    case Empty => false
  }
```

Now we can implement the actual regular expression derivative. It consists of two parts: the method to calculate the derivative which in turn depends on a method that handles a nullable regular expression. Both parts are quite simple so I'll give the code first and then explain the more complicated parts.

```
def delta: Regexp =
  if nullable then Epsilon else Empty

def derivative(ch: Char): Regexp =
  this match {
    case Append(left, right) =>
      (left.derivative(ch) ++ right).orElse(left.delta ++
right.derivative(ch))
    case OrElse(first, second) =>
      first.derivative(ch).orElse(second.derivative(ch))
    case Repeat(source) =>
      source.derivative(ch) ++ this
    case Apply(string) =>
      if string.size == 1 then
        if string.charAt(0) == ch then Epsilon
        else Empty
      else if string.charAt(0) == ch then Apply(string.tail)
      else Empty
    case Epsilon => Empty
    case Empty => Empty
  }
```

I think this code is reasonably straightforward, except perhaps for the cases for `OrElse` and `Append`. The case for `OrElse` is trying to match both regular expressions simultaneously, which gets around the problem in our earlier implementation. The definition of `nullable` ensures we match if either side matches. The case for `Append` is attempting to match the `left` side if it is still looking for characters; otherwise it is attempting to match the `right` side.

With this we redefine `matches` as follows.

```
def matches(input: String): Boolean = {
  val r = input.foldLeft(this){ (regexp, ch) =>
regexp.derivative(ch) }
  r.nullable
}
```

We can show the code works as expected.

```
val regexp = Regexp("Sca") ++ Regexp("la") ++ Regexp("la").repeat
```

```
regexp.matches("Scala")
// res1: Boolean = true
regexp.matches("Scalalalala")
// res2: Boolean = true
regexp.matches("Sca")
// res3: Boolean = false
regexp.matches("Scalal")
// res4: Boolean = false
```

It also solves the problem with the earlier implementation.

```
Regexp("cat").orElse(Regexp("cats")).matches("cats")
// res5: Boolean = true
```

This is a nice result for a very simple algorithm. However there is a problem. You might notice that regular expression matching can become very slow. In fact we can run out of heap space trying a simple match like

```
Regexp("cats").repeat.matches("catscatscatscats")
// java.lang.OutOfMemoryError: Java heap space
```

This happens because the derivative of the regular expression can grow very large. Look at this example, after only a few derivatives.

```
Regexp("cats").repeat.derivative('c').derivative('a').derivative('t')
// res6: Regexp =
OrElse(OrElse(Append(Apply(s),Repeat(Apply(cats))),Append(Empty,Append(Empty,
```

The root cause is that the derivative rules for `Append`, `OrElse`, and `Repeat` can produce a regular expression that is larger than the input. However this output often contains redundant information. In the example above there are multiple occurrences of `Append(Empty, ...)`, which is equivalent to just `Empty`. This is similar to adding zero or multiplying by one in arithmetic, and we can use similar algebraic simplification rules to get rid of these unnecessary elements.

We can implement this simplification in one of two ways: we can make simplification a separate method that we apply to an existing `Regexp`, or we can do the simplification as we construct the `Regexp`. I've chosen to do the latter, modifying `++`, `orElse`, and `repeat` as follows:

```
def ++(that: Regexp): Regexp = {
  (this, that) match {
    case (Epsilon, re2) => re2
    case (re1, Epsilon) => re1
    case (Empty, _) => Empty
    case (_, Empty) => Empty
    case _ => Append(this, that)
  }
}

def orElse(that: Regexp): Regexp = {
  (this, that) match {
    case (Empty, re) => re
    case (re, Empty) => re
    case _ => OrElse(this, that)
  }
}

def repeat: Regexp = {
  this match {
    case Repeat(source) => this
    case Epsilon => Epsilon
    case Empty => Empty
    case _ => Repeat(this)
  }
}
```

With this small change in-place, our regular expressions stay at a reasonable size for any input.

```
Regexp("cats").repeat.derivative('c').derivative('a').derivative('t')
// res8: Regexp = Append(Apply(s),Repeat(Apply(cats)))
```

Here's the final code.

```
enum Regexp {
  def ++(that: Regexp): Regexp = {
```

```
    (this, that) match {
      case (Epsilon, re2) => re2
      case (re1, Epsilon) => re1
      case (Empty, _) => Empty
      case (_, Empty) => Empty
      case _ => Append(this, that)
    }
  }

  def orElse(that: Regexp): Regexp = {
    (this, that) match {
      case (Empty, re) => re
      case (re, Empty) => re
      case _ => OrElse(this, that)
    }
  }

  def repeat: Regexp = {
    this match {
      case Repeat(source) => this
      case Epsilon => Epsilon
      case Empty => Empty
      case _ => Repeat(this)
    }
  }

  def `*` : Regexp = this.repeat

  /** True if this regular expression accepts the empty string */
  def nullable: Boolean =
    this match {
      case Append(left, right) => left.nullable && right.nullable
      case OrElse(first, second) => first.nullable ||
second.nullable
      case Repeat(source) => true
      case Apply(string) => false
      case Epsilon => true
      case Empty => false
    }

  def delta: Regexp =
    if nullable then Epsilon else Empty

  def derivative(ch: Char): Regexp =
    this match {
      case Append(left, right) =>
        (left.derivative(ch) ++ right).orElse(left.delta ++
```

```
right.derivative(ch))
      case OrElse(first, second) =>
        first.derivative(ch).orElse(second.derivative(ch))
      case Repeat(source) =>
        source.derivative(ch) ++ this
      case Apply(string) =>
        if string.size == 1 then
          if string.charAt(0) == ch then Epsilon
          else Empty
        else if string.charAt(0) == ch then Apply(string.tail)
        else Empty
      case Epsilon => Empty
      case Empty => Empty
    }

  def matches(input: String): Boolean = {
    val r = input.foldLeft(this){ (regexp, ch) =>
regexp.derivative(ch) }
    r.nullable
  }

  case Append(left: Regexp, right: Regexp)
  case OrElse(first: Regexp, second: Regexp)
  case Repeat(source: Regexp)
  case Apply(string: String)
  case Epsilon
  case Empty
}
object Regexp {
  val empty: Regexp = Empty

  val epsilon: Regexp = Epsilon

  def apply(string: String): Regexp =
    if string.isEmpty() then Epsilon
    else Apply(string)
}
```

Notice that our implementation is tail recursive. The only "looping" is the call to the tail recursive `foldLeft` in `matches`. No continuation-passing style transform is necessary here! (Calculating the derivatives is not tail recursive but it very unlikely this would overflow the stack.) This may not be surprising if you've studied theory of computation. A key result from that field is the equivalence between regular expressions and finite state

machines. If you know this you may have found it a bit surprising we had to use a stack at all in our prior implementations. But hold on a minute. If we think carefully about regular expression derivatives we'll see that they actually are continuations! A continuation means "what comes next", which is exactly what a regular expression derviative defines for a regular expression and a particular character. So our interpreter does use CPS, but reified as a regular expression not a function, and derived through a different route.

Continuations reify control-flow. That is, they give us an explicit representation of how control moves through our program. This means we can change the control flow by applying continuations in a different order. Let's make this concrete. A regular expression derivative represents a continuation. So imagine we're running a regular expression on data that arrives asynchronously; we want to match as much data as we have available, and then suspend the regular expression and continue matching when more data arrives. This is trival. When we run out of data we just store the current derivative. When more data arrives we continue processing using the derivative we stored. Here's an example.

Start by defining the regular expression.

```
val cats = Regexp("cats").repeat
```

Process the first piece of data and store the continuation.

```
val next = "catsca".foldLeft(cats){ (regexp, ch) =>
regexp.derivative(ch) }
```

Continue processing when more data arrives.

```
"tscats".foldLeft(next){ (regexp, ch) => regexp.derivative(ch) }
```

Notice that we could just as easily go back to a previous regular expression if we wanted to. This would give us backtracking. We

don't need backtracking for regular expressions, but for more general parsers we do. In fact with continuations we can define any control flow we like, including backtracking search, exceptions, cooperative threading, and much much more.

In this section we've also seen the power of rewrites. Regular expression matching using derivatives works solely by rewriting the regular expression. We also used rewriting to simplify the regular expressions, avoiding the explosion in size that derivatives can cause. The abstract type of these methods is `Program => Program` so we might think they are combinators. However the implementation uses structural recursion and they serve the role of interpreters. Rewrites are the one place where the types alone can lead us astray.

I hope you find regular expression derivatives interesting and a bit surprising. I certainly did when I first read about them. There is a deeper point here, which runs throughout the book: most problems have already been solved and we can save a lot of time if we can just find those solutions. I elevate this idea of the status of a strategy, which I call **read the literature** for reasons that will soon be clear. Most developers read the occasional blog post and might attend a conference from time to time. Many fewer, I think, read academic papers. This is unfortunate. Part of the fault is with the academics: they write in a style that is hard to read without some practice. However I think many developers think the academic literature is irrelevant. One of the goals of this book is to show the relevance of academic work, which is why each chapter conclusion sketches the development of its main ideas with links to relevant papers.

16.2. From Continuations to Stacks

In the previous section we explored regular expression derivatives. We saw that they are continuations, but reified as data structures rather than the functions we used when we first worked with continuation-passing style. In this section we'll reify continuations-as-functions as data. In doing so we'll find continuations implicitly encode a stack structure. Explicitly reifying this structure is a step towards implementing a stack machine.

We'll start with the CPSed regular expression interpreter (not using derivatives), shown below.

```
enum Regexp {
  def ++(that: Regexp): Regexp =
    Append(this, that)

  def orElse(that: Regexp): Regexp =
    OrElse(this, that)

  def repeat: Regexp =
    Repeat(this)

  def `*` : Regexp = this.repeat

  def matches(input: String): Boolean = {
    // Define a type alias so we can easily write continuations
    type Continuation = Option[Int] => Option[Int]

    def loop(regexp: Regexp, idx: Int, cont: Continuation):
Option[Int] =
      regexp match {
        case Append(left, right) =>
          val k: Continuation = _ match {
            case None    => cont(None)
            case Some(i) => loop(right, i, cont)
          }
          loop(left, idx, k)
```

```
        case OrElse(first, second) =>
          val k: Continuation = _ match {
            case None => loop(second, idx, cont)
            case some => cont(some)
          }
          loop(first, idx, k)

        case Repeat(source) =>
          val k: Continuation =
            _ match {
              case None    => cont(Some(idx))
              case Some(i) => loop(regexp, i, cont)
            }
          loop(source, idx, k)

        case Apply(string) =>
          cont(Option.when(input.startsWith(string, idx))(idx +
string.size))

        case Empty =>
          cont(None)
      }

    // Check we matched the entire input
    loop(this, 0, identity).map(idx => idx ==
input.size).getOrElse(false)
  }

  case Append(left: Regexp, right: Regexp)
  case OrElse(first: Regexp, second: Regexp)
  case Repeat(source: Regexp)
  case Apply(string: String)
  case Empty
}
object Regexp {
  val empty: Regexp = Empty

  def apply(string: String): Regexp =
    Apply(string)
}
```

To reify the continuations we can apply the same recipe as before: we create a case for each place in which we construct a continuation. In our interpreter loop this is for `Append`, `OrElse`, and `Repeat`. We also construct a continuation using the identity

function when we first call `loop`, which represents the continuation to call when the loop has finished. This gives us four cases.

```
enum Continuation {
  case AppendK
  case OrElseK
  case RepeatK
  case DoneK
}
```

What data does each case next to hold? Let's let look at the structure of the cases within the CPS interpreter. The case for `Append` is typical.

```
case Append(left, right) =>
  val k: Cont = _ match {
    case None    => cont(None)
    case Some(i) => loop(right, i, cont)
  }
  loop(left, idx, k)
```

The continuation `k` refers to the `Regexp` `right`, the method `loop`, and the continuation `cont`. Our reification should reflect this by holding the same data. If we consider all the cases we end up with the following definition. Notice that I implemented an `apply` method so we can still call these continuations like a function.

```
type Loop = (Regexp, Int, Continuation) => Option[Int]
enum Continuation {
  case AppendK(right: Regexp, loop: Loop, next: Continuation)
  case OrElseK(second: Regexp, index: Int, loop: Loop, next:
Continuation)
  case RepeatK(regexp: Regexp, index: Int, loop: Loop, next:
Continuation)
  case DoneK

  def apply(idx: Option[Int]): Option[Int] =
    this match {
      case AppendK(right, loop, next) =>
        idx match {
```

```
          case None    => next(None)
          case Some(i) => loop(right, i, next)
        }

      case OrElseK(second, index, loop, next) =>
        idx match {
          case None => loop(second, index, next)
          case some => next(some)
        }

      case RepeatK(regexp, index, loop, next) =>
        idx match {
          case None    => next(Some(index))
          case Some(i) => loop(regexp, i, next)
        }

      case DoneK =>
        idx
    }
}
```

Now we can rewrite the interpreter loop using the `Continuation` type.

```
def matches(input: String): Boolean = {
  def loop(
      regexp: Regexp,
      idx: Int,
      cont: Continuation
  ): Option[Int] =
    regexp match {
      case Append(left, right) =>
        val k: Continuation = AppendK(right, loop, cont)
        loop(left, idx, k)

      case OrElse(first, second) =>
        val k: Continuation = OrElseK(second, idx, loop, cont)
        loop(first, idx, k)

      case Repeat(source) =>
        val k: Continuation = RepeatK(regexp, idx, loop, cont)
        loop(source, idx, k)

      case Apply(string) =>
        cont(Option.when(input.startsWith(string, idx))(idx +
```

```
string.size))

      case Empty =>
        cont(None)
    }

  // Check we matched the entire input
  loop(this, 0, DoneK)
    .map(idx => idx == input.size)
    .getOrElse(false)
}
```

The point of this construction is that we've reified the stack: it's now explicitly represented as the `next` field in each `Continuation`. The stack is a last-in first-out (LIFO) data structure: the last element we add to the stack is the first element we use. (This is exactly the same as efficient use of a `List`.) We construct continuations by adding elements to the front of the existing continuation, which is exactly how we construct lists or stacks. We use continuations from front-to-back; in other words in last-in first-out (LIFO) order. This is the correct access pattern to use a list efficiently, and also the access pattern that defines a stack. Reifying the continuations as data has reified the stack. In the next section we'll use this fact to build a compiler that targets a stack machine.

16.3. Compilers and Virtual Machines

We've reified continuations and seen they contain a stack structure: each continuation contains a references to the next continuation, and continuations are constructed in a last-in first-out order. We'll now, once again, reify this structure. This time we'll create an explicit stack, giving rise to a stack-based **virtual machine** to run our code. We'll also introduce a compiler, transforming our code into a sequence of operations that run on

this virtual machine. We'll then look at optimizing our virtual machine. As this code involves benchmarking, there is an [accompanying repository][stack-machine] that contains benchmarks you can run on your own computer.

16.3.1. Virtual and Abstract Machines

A virtual machine is a computational machine implemented in software rather than hardware. A virtual machine runs programs written in some **instruction set**. The Java Virtual Machine (JVM), for example, runs programs written in Java bytecode. Closely related are **abstract machines**. The two terms are sometimes used interchangeably but I'll make the distinction that a virtual machine has an implementation in software, while an abstract machine is a theoretical model without an implementation. Thus we can think of an abstract machine as a concept, and a virtual machine as a realization of a concept. This is a distinction we've made in many other parts of the book.

As an abstract machine, stack machines are represented by models such as push down automata and the SECD machine. From abstract stack machines we firstly get the concept itself of a stack machine. The two core operations for a stack are pushing a value on to the top of the stack, and popping the top value off the stack. Function arguments and results are both passed via the stack. So, for example, a binary operation like addition will pop the top two values off the stack, add them, and push the result onto the stack. Abstract stack machines also tell us that stack machines with a single stack are not universal computers. In other words, they are not as powerful as Turing machines. If we add a second stack, or some other form of additional memory, we have a universal computer. This informs the design of virtual machines based on a stack machine.

Stack machines are also very common virtual machines. The Java Virtual Machine is a stack machine, as are the .Net and WASM virtual machines. They are easy to implement, and to write compilers for. We've already seen how easy it is to implement an interpreter so why should we care about stack machines, or virtual machines in general? The usual answer is performance. Implementing a virtual machine opens up opportunities for optimizations that are difficult to implement in interpreters. Virtual machines also give us a lot of flexibility. It's simple to trace or otherwise inspect the execution of a virtual machine, which makes debugging easier. They are easy to port to different platforms and languages. Virtual machines are often very compact, as is the code they run. This makes them suitable for embedded devices. Our focus will be on performance. Although we won't go down the rabbit-hole of compiler and virtual machine optimizations, which would easily take up an entire book, we'll at least tip-toe to the edge and peek down.

16.3.2. Compilation

Let's now briefly talk about compilation. A compiler transforms a program from one representation to another. In our case we will transform our programs represented as an algebraic data type of reified constructors and combinators into the instruction set for our virtual machine. The virtual machine itself is an interpreter for its instruction set. Computation always bottoms out in interpretation: a hardware CPU is nothing but an interpreter for it's machine code.

Notice there are two notions of program here, and two corresponding instruction sets: there is the program the structurally recursive interpreter executes, with an instruction set consisting of reified constructors and combinators, and there is the program we compile this into for the stack machine using the

stack machine's instruction set. We will call these the interpreter program and instruction set, and stack machine program and instruction set respectively.

The structurally recursive interpreter is an example of a **tree-walking interpreter** or **abstract syntax tree (AST) interpreter**. The stack machine is an example of a **byte-code interpreter**.

16.4. From Interpreter to Stack Machine

There are three parts to transforming an interpreter to a stack machine:

1. creating the instruction set the stack machine will run;
2. creating the compiler from interpreter programs to stack machine programs; and
3. implementing the stack machine to execute stack machine instructions.

Let's make this concrete by returning to our arithmetic interpreter.

```
enum Expression {
  def +(that: Expression): Expression = Addition(this, that)
  def *(that: Expression): Expression = Multiplication(this,
that)
  def -(that: Expression): Expression = Subtraction(this, that)
  def /(that: Expression): Expression = Division(this, that)

  def eval: Double =
    this match {
      case Literal(value)              => value
      case Addition(left, right)       => left.eval + right.eval
      case Subtraction(left, right)    => left.eval - right.eval
      case Multiplication(left, right) => left.eval * right.eval
      case Division(left, right)       => left.eval / right.eval
    }
```

```
  case Literal(value: Double)
  case Addition(left: Expression, right: Expression)
  case Subtraction(left: Expression, right: Expression)
  case Multiplication(left: Expression, right: Expression)
  case Division(left: Expression, right: Expression)
}
object Expression {
  def literal(value: Double): Expression = Literal(value)
}
```

Interpreter programs are defined by the interpreter instruction set

```
enum Expression {
  case Literal(value: Double)
  case Addition(left: Expression, right: Expression)
  case Subtraction(left: Expression, right: Expression)
  case Multiplication(left: Expression, right: Expression)
  case Division(left: Expression, right: Expression)
}
```

Transforming the interpreter instruction set to the stack machine instruction set works as follows:

- each constructor interpreter instruction corresponds to stack machine instruction carrying exactly the same data; and
- each combinator interpreter instruction has a corresponding stack machine instruction that carries only non-recursive data. Recursive data, which is executed by recursive calls to the interpreter, will be represented by data on the stack machine's stack.

Turning to the arithmetic interpreter's instruction set, we see that `Literal` is our sole constructor and thus has a mirror in our stack machine's instruction set. Here I've named the interpreter instruction set `Op` (short for "operation"), and shortened the name from `Literal` to `Lit` to make it clearer which instruction set we are using.

```
enum Op {
  case Lit(value: Double)
}
```

The other instructions are all combinators. They also all only contain values of type `Expression`, and hence in the stack machine the corresponding values will be found on the stack. This gives us the complete stack machine instruction set.

```
enum Op {
  case Lit(value: Double)
  case Add
  case Sub
  case Mul
  case Div
}
```

This completes the first step of the process. The second step is to implement the compiler. The secret to compiling for a stack machine is to transfrom instructions into **reverse polish notation (RPN)**. In RPN operations follow their operands. So, instead of writing `1 + 2` we write `1 2 +`. This is exactly the order in which a stack machine works. To evaluate `1 + 2` we should first push `1` onto the stack, then push `2`, and finally pop both these values, perform the addition, and push the result back to the stack. RPN also does not need nesting. To represent `1 + (2 + 3)` in RPN we simply use `2 3 + 1 +`. Doing away with brackets means that stack machine programs can be represented as a linear sequence of instructions, not a tree. Concretely, we can use `List[Op]`.

How we should we implement the conversion to RPN. We are performing a transformation on an algebraic data type, our interpreter instruction set and therefore we can use structural recursion. The following code shows one way to implement this. It's not very efficient (appending lists is a slow operation) but this doesn't matter for our purposes.

```
def compile: List[Op] =
  this match {
    case Literal(value) => List(Op.Lit(value))
    case Addition(left, right) =>
      left.compile ++ right.compile ++ List(Op.Add)
    case Subtraction(left, right) =>
      left.compile ++ right.compile ++ List(Op.Sub)
    case Multiplication(left, right) =>
      left.compile ++ right.compile ++ List(Op.Mul)
    case Division(left, right) =>
      left.compile ++ right.compile ++ List(Op.Div)
  }
```

We now are left to implement the stack machine. We'll start by sketching out the interface for the stack machine.

```
final case class StackMachine(program: List[Op]) {
  def eval: Double = ???
}
```

In this design the program is fixed for a given `StackMachine` instance, but we can run the program multiple times.

Now we'll implement `eval`. It is a structural recursion over an algebraic data type, in this case the `program` of type `List[Op]`. It's a little bit more complicated than some of the structural recursions we have seen, because we need to implement the stack as well. We'll represent the stack as a `List[Double]`, and define methods to push and pop the stack.

```
final case class StackMachine(program: List[Op]) {
  def eval: Double = {
    def pop(stack: List[Double]): (Double, List[Double]) =
      stack match {
        case head :: next => (head, next)
        case Nil =>
          throw new IllegalStateException(
            s"The data stack does not have any elements."
          )
      }

    def push(value: Double, stack: List[Double]): List[Double] =
```

```
      value :: stack

    ???
  }
}
```

Now we can define the main stack machine loop. It takes as parameters the program and the stack, and is a structural recursion over the program.

```
def eval: Double = {
  // pop and push defined here ...

  def loop(stack: List[Double], program: List[Op]): Double =
    program match {
      case head :: next =>
        head match {
          case Op.Lit(value) => loop(push(value, stack), next)
          case Op.Add =>
            val (a, s1) = pop(stack)
            val (b, s2) = pop(s1)
            val s = push(a + b, s2)
            loop(s, next)
          case Op.Sub =>
            val (a, s1) = pop(stack)
            val (b, s2) = pop(s1)
            val s = push(a + b, s2)
            loop(s, next)
          case Op.Mul =>
            val (a, s1) = pop(stack)
            val (b, s2) = pop(s1)
            val s = push(a + b, s2)
            loop(s, next)
          case Op.Div =>
            val (a, s1) = pop(stack)
            val (b, s2) = pop(s1)
            val s = push(a + b, s2)
            loop(s, next)
        }

      case Nil => stack.head
    }

  loop(List.empty, program)
}
```

I've implemented a simple benchmark for this code (see [the repository][stack-machine]) and it's roughly five times slower than the interpreter we started with. Clearly some optimization is needed.

[stack-machine]: https://github.com/scalawithcats/stack-machine

16.4.1. Effectful Interpreters

One of the reasons for using the interpreter strategy is to isolate effects, such as state or input and output. An interpreter can be effectful without impacting the ability to reason about or compose the programs the interpreter runs. Sometimes the effects are the entire point of the interpreter as the program may describe effectful actions, such as parsing network data or drawing on a screen, which the interpreter then carries out. Sometimes effects may just be optimizations, which is how we are going to use them in our arithmetic stack machine.

There are many inefficiencies in the stack machine we have just created. A `List` is a poor choice of data structure for both the stack and program. We can avoid a lot of pointer chasing and memory allocation by using a fixed size `Array`. The program never changes in size, and we can simply allocate a large enough stack that resizing it becomes very unlikely. We can also avoid the indirection of pushing and popping and operate directly on the stack array.

The code below shows a simple implementation, which in my benchmarking is about thirty percent faster than the tree-walking interpreter.

```
final case class StackMachine(program: Array[Op]) {
  // The data stack
  private val stack: Array[Double] = Array.ofDim[Double](256)
```

```
  def eval: Double = {
    // sp points to first free element on the stack
    // stack(sp - 1) is the first element with data
    //
    // pc points to the current instruction in program
    def loop(sp: Int, pc: Int): Double =
      if (pc == program.size) stack(sp - 1)
      else
        program(pc) match {
          case Op.Lit(value) =>
            stack(sp) = value
            loop(sp + 1, pc + 1)
          case Op.Add =>
            val a = stack(sp - 1)
            val b = stack(sp - 2)
            stack(sp - 2) = (a + b)
            loop(sp - 1, pc + 1)
          case Op.Sub =>
            val a = stack(sp - 1)
            val b = stack(sp - 2)
            stack(sp - 2) = (a - b)
            loop(sp - 1, pc + 1)
          case Op.Mul =>
            val a = stack(sp - 1)
            val b = stack(sp - 2)
            stack(sp - 2) = (a * b)
            loop(sp - 1, pc + 1)
          case Op.Div =>
            val a = stack(sp - 1)
            val b = stack(sp - 2)
            stack(sp - 2) = (a / b)
            loop(sp - 1, pc + 1)
        }

    loop(0, 0)
  }
}
```

16.4.2. Further Optimization

The above optimization is, to me, the most obvious and straightforward to implement. In this section we'll attempt to go further, by looking at some of the optimizations described in the

literature. We'll see that there is not always a straight path to faster code.

The benchmark I used is the simple recursive Fibonacci. Calculating the n^{th} Fibonacci number produces a large expression for a modest choice of n. I used a value of 25, and the expression has over one million elements. Notably the expressions only involve addition, and the only literals in use are zero and one. This limits the applicability of the optimizations to a wider range of inputs, but the intention is not to produce an optimized interpreter for this specific case but rather to discuss possible optimizations and issues that arise when attempting to optimize an interpreter in general.

We'll look at four different optimizations, which all use the optimized stack machine above as their base:

- **Algebraic simplification** performs simplifications at compile-time to produce smaller expressions. A small expression should require fewer interpreter steps and hence be faster. The only simplification I used was replacing $x + 0$ or $0 + x$ with x. This occurs frequently in the Fibonacci series. Since the expressions we are working with have no variables or control flow we could simplify the entire expression to a single literal at compile-time. This would be an extremely good optimization but rather defeats the purpose of trying to generalize to other applications.
- **Byte code** replaces the `Op` algebraic data type with a single byte. The hope here is that the smaller representation will lead to better cache utilization, and possibly a faster `match` expression, and therefore a faster overall interpreter. In this representation literals are also stored in a separate array of `Doubles`. More on this later.
- **Stack caching** stores the top of the stack in a variable, which we hope will be allocated to a register and therefore be extremely fast to access. The remainder of the stack is stored in an array as above. Stack caching involves more work when

pushing values on to the stack, as we must copy the value from the top into the array, but less work when popping values off the stack. The hope is that the savings will outweigh the costs.

- **Superinstructions** replace common sequences of instructions with a single instruction. We already do this to an extent; a typical stack machine would have separate instructions for pushing and popping, but our instruction set merges these into the arithmetic operations. I used two superinstructions: one for incrementing a value, which frequently occurs in the Fibonacci, and one for adding two values from the stack and a literal.

Below are the benchmarks results obtained on an AMD Ryzen 5 3600 and an Apple M1, both running JDK 21. Results are shown in operations per second. The Baseline interpreter is the one using structural recursion. The Stack interpreter uses a `List` to represent the stack and program. The Optimized Stack represents the stack and program as arrays. The other interpreters build on the Optimized Stack interpreter and add the optimizations described above. The All interpreter has all the optimizations.

Interpreter	Ryzen 5	Speedup	M1	Speedup
Baseline	2754.43	1.00	3932.93	1.00
Stack	676.43	0.25	1004.16	0.26
Optimized Stack	3631.19	1.32	2953.21	0.75
Algebraic Simplification	1630.93	0.59	4818.45	1.23
Byte Code	4057.11	1.47	3355.75	0.85
Stack Caching	3698.10	1.34	3237.17	0.82
Superinstructions	3706.10	1.35	4689.02	1.19

Interpreter	Ryzen 5	Speedup	M1	Speedup
All	7612.45	2.76	7098.06	1.80

There are a few lessons to take from this. The most important, in my opinion, is that *performance is not compositional*. The results of applying two optimizations is not simply the sum of applying the optimizations individually. You can see that most of the optimizations on their own make little or no change to performance relative to the Optimized Stack interpreter. Taken together, however, they make a significant improvement.

Basic structural recursion, the Baseline interpreter, is surprisingly fast; a bit slower than the Optimized Stack interpreter on the Ryzen 5 but faster on the M1. A stack machine emulates the processor's built-in call stack. The native call stack is extremely fast, so we need a good reason to avoid using it.

Details really matter in optimization. We see the choice of data structure makes a massive difference between the Stack and Optimized Stack interpreters. An earlier version of the Byte Code interpreter had worse performance than the Optimized Stack. As best I could tell this was because I was storing literals alongside byte code, and loading a `Double` from an `Array[Byte]` (using a `ByteBuffer`) was slow. Superinstructions are very dependent on the chosen superinstructions. The superinstruction to add two values from the stack plus a literal had little effect on it's own; in fact the interpreter with this single superinstruction was much slower on the Ryzen 5.

Compilers, and JIT compilers in particular, are difficult to understand. I cannot explain why, for example, the Algebraic Simplification interpreter is so slow on the Ryzen 5. This interpreter does strictly less work than the Optimized Stack interpreter. Just like the interpreter optimizations I implemented, compiler optimizations apply in restricted cases that the algorithms recognize. If code does not match the patterns the

algorithms look for, the optimizations will not apply, which can lead to strange performance cliffs. My best guess is that something about my implementation caused me to run afoul of such an issue.

Finally, differences between platforms are also significant. It's hard to know how much this due to differences in the computer's architecture, and how much is down to differences in the JVM. Either way, be aware of which platform or platforms you expect the majority of users to run on, and don't naively assume performance on one platform will directly translate to another.

16.5. Conclusions

In this chapter we explored two main techniques for optimizing interpeters: algebraic simplification of programs, and interpretation in a virtual machine.

Our regular expression derivative algorithm is taken from *Regular-Expression Derivatives Re-Examined* [67]. Regular expression derivatives are very easy to implement and nicely illustrate algebraic simplification. However we have to recompute the derivative on each input character. If we instead compile the regular expression to a finite state machine ahead of time, we save time when parsing input. The details of this algorithm are in the paper.

Regular-Expression Derivatives Re-Examined [67] is in turn based on *Derivatives of Regular Expressions* [8], published in 1964. Although the style of the paper will be immediately recognizable to anyone familiar with the more theoretical end of computer science, anachronisms like "State Diagram Construction" are a reminder that this comes from the very beginnings of the discipline.

Regular expression derivatives can be extended to context-free grammars and therefore used to implement parsers [57]. Other work has added additional operators to regular expression derivatives, such as anchors and restricted lookaround, and created best-in-class regular expression engines [59,86]. The ease of algebraically manipulating regular expression derivatives a key to this advance.

A lot of work has looked at systematically transforming an interpreter into a compiler and virtual machine. See, for example, *From Interpreter to Compiler and Virtual Machine: A Functional Derivation* [2] for some earlier work, and *Calculating correct compilers* [3] for more recent work. These are only a few examples; there is too much work in this field for me to adequately summarise.

Interpreters and their optimization has a similarly enormous body of work. However, we spent a bit more time on this, and it's also a personal interest, so I've been a bit more through in collecting references for this section.

We looked at four techniques for optimization: algebraic simplification, byte code, stack caching, and superinstructions. Algebraic simplification is as old as algebra, and something familiar to any secondary school student. In the world of compilers, different aspects of algebraic simplification are known as constant folding, constant propagation, and common subexpression elimination. Byte code is probably as old as interpreters, and dates back to at least the 1960s in the form of P-code[84]. *Stack Caching for Interpreters* [25] introduces the idea of stack caching, and shows some rather more complex realizations than the simple system I used. Superinstructions were introduced in *Optimizing an ANSI C Interpreter with Superoperators* [71]. *Towards Superinstructions for Java Interpreters* [11] is a nice example of applying superinstructions to an interpreted JVM.

[84]https://en.wikipedia.org/wiki/P-code_machine

Let's now talk about instruction dispatch, which is area we did not consider for optimization. Instruction dispatch is the process by which the interpreter chooses the code to run for a given interpreter instruction. *The Structure and Performance of Efficient Interpreters* [24] argues that instruction dispatch makes up a major portion of an interpreter's execution time. The approach we used is known as switch dispatch in the literature. There are several alternative approaches. Direct threading [5] represents an instruction by the function that implements it. This requires first-class functions and full tail calls. It is generally considered the fastest form of dispatch. Notice that it leverages the duality between data and functions. Subroutine threading is like direct threading, but uses normal calls and returns instead of tail calls. Indirect threaded code [19] represents each bytecode as an index into a lookup table that points to the implementing function.

Stack machines are not the only virtual machine used for implementing interpreters. Register machines are the most common alternative. The Lua virtual machine, for example, is a register machine. *Virtual Machine Showdown: Stack versus Registers* [79] compares the two and concludes that register machines are faster. However, register machines are more complex to implement.

If you're interested in the design considerations in a general purpose stack based instruction set, *Bringing the Web up to Speed with WebAssembly* [39] is the paper for you. It covers the design of WebAssembly, and the rationale behind the design choices. An interpreter for WebAssembly is described in *A Fast In-Place Interpreter for WebAssembly* [84]. Notice how often tail calls arise in the discussion!

Part IV: Case Studies

17. Creating Usable Code

APIs are interfaces and should be designed as such.

`scala.annotation.implicitNotFound` and `scala.annotation.implicitAmbiguous`

18. Case Study: Testing Asynchronous Code

We'll start with a straightforward case study: how to simplify unit tests for asynchronous code by making them synchronous.

Let's return to the example from Chapter 13 where we're measuring the uptime on a set of servers. We'll flesh out the code into a more complete structure. There will be two components. The first is an `UptimeClient` that polls remote servers for their uptime:

```
import scala.concurrent.Future

trait UptimeClient {
  def getUptime(hostname: String): Future[Int]
}
```

We'll also have an `UptimeService` that maintains a list of servers and allows the user to poll them for their total uptime:

```
import cats.instances.future._ // for Applicative
import cats.instances.list._   // for Traverse
import cats.syntax.traverse._  // for traverse
import scala.concurrent.ExecutionContext.Implicits.global

class UptimeService(client: UptimeClient) {
  def getTotalUptime(hostnames: List[String]): Future[Int] =
    hostnames.traverse(client.getUptime).map(_.sum)
}
```

We've modelled `UptimeClient` as a trait because we're going to want to stub it out in unit tests. For example, we can write a test client that allows us to provide dummy data rather than calling out to actual servers:

```
class TestUptimeClient(hosts: Map[String, Int]) extends
UptimeClient {
  def getUptime(hostname: String): Future[Int] =
    Future.successful(hosts.getOrElse(hostname, 0))
}
```

Now, suppose we're writing unit tests for `UptimeService`. We want to test its ability to sum values, regardless of where it is getting them from. Here's an example:

```
def testTotalUptime() = {
  val hosts    = Map("host1" -> 10, "host2" -> 6)
  val client   = new TestUptimeClient(hosts)
  val service  = new UptimeService(client)
  val actual   = service.getTotalUptime(hosts.keys.toList)
  val expected = hosts.values.sum
  assert(actual == expected)
}
// error:
// Values of types scala.concurrent.Future[Int] and Int cannot be
compared with == or !=
//   assert(actual == expected)
//          ^^^^^^^^^^^^^^^^^^
```

The code doesn't compile because we've made a classic error[^warnings]. We forgot that our application code is asynchronous. Our `actual` result is of type `Future[Int]` and our `expected` result is of type `Int`. We can't compare them directly!

[^warnings]: Technically this is a *warning* not an error. It has been promoted to an error in our case because we're using the `-Xfatal-warnings` flag on `scalac`.

There are a couple of ways to solve this problem. We could alter our test code to accommodate the asynchronousness. However, there is another alternative. Let's make our service code synchronous so our test works without modification!

18.1. Abstracting over Type Constructors

We need to implement two versions of `UptimeClient`: an asynchronous one for use in production and a synchronous one for use in our unit tests:

```
trait RealUptimeClient extends UptimeClient {
  def getUptime(hostname: String): Future[Int]
}

trait TestUptimeClient extends UptimeClient {
  def getUptime(hostname: String): Int
}
```

The question is: what result type should we give to the abstract method in `UptimeClient`? We need to abstract over `Future[Int]` and `Int`:

```
trait UptimeClient {
  def getUptime(hostname: String): ???
}
```

At first this may seem difficult. We want to retain the `Int` part from each type but "throw away" the `Future` part in the test code. Fortunately, Cats provides a solution in terms of the *identity type*, `Id`, that we discussed way back in Section 10.3. `Id` allows us to "wrap" types in a type constructor without changing their meaning:

```
package cats

type Id[A] = A
```

`Id` allows us to abstract over the return types in `UptimeClient`. Implement this now:

- write a trait definition for `UptimeClient` that accepts a type constructor `F[_]` as a parameter;
- extend it with two traits, `RealUptimeClient` and `TestUptimeClient`, that bind `F` to `Future` and `Id` respectively;
- write out the method signature for `getUptime` in each case to verify that it compiles.

You should now be able to flesh your definition of `TestUptimeClient` out into a full class based on a `Map[String, Int]` as before.

18.2. Abstracting over Monads

Let's turn our attention to `UptimeService`. We need to rewrite it to abstract over the two types of `UptimeClient`. We'll do this in two stages: first we'll rewrite the class and method signatures, then the method bodies. Starting with the method signatures:

- comment out the body of `getTotalUptime` (replace it with `???` to make everything compile);
- add a type parameter `F[_]` to `UptimeService` and pass it on to `UptimeClient`.

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`.

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()
```

18.3. Summary

This case study provides an example of how Cats can help us abstract over different computational scenarios. We used the `Applicative` type class to abstract over asynchronous and synchronous code. Leaning on a functional abstraction allows us to specify the sequence of computations we want to perform without worrying about the details of the implementation.

Back in Figure 11, 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.

19. Error Handling

20. Case Study: Map-Reduce

<!– TODO:

- DONE - talk about map-reduce - it's just foldMap
- DONE - introduce/reimplement foldMap
- DONE - implement parallelFoldMap to mimic map-reduce
  - DONE - mention that we're specifically imitating multi-machine map-reduce where we need to split data between machines in large blocks
  - DONE - implement in terms of our foldMap first
  - DONE - then implement in terms of Cats' foldMap
  - DONE - talk about traverse
- summary
  - DONE - real-world map-reduce has communication costs
  - DONE - multi-cpu map-reduce doesn't have communication costs
  - DONE - parallelFoldMap mimics multi-machine
  - DONE - our final version of parallelFoldMap (based on traverse) is far simpler
  - talk about substitution and the things it doesn't model:
    - performance
    - parallelism
    - side-effects (future starts immediately)
    - etc…

TODO:

- DONE - drop the current foldMapM stuff
- DONE - maybe move it elsewhere

–>

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[85], which is a programming model for doing parallel data processing across clusters of machines (aka "nodes"). As the name suggests, the model is built around a *map* phase, which is the same `map` function we know from Scala and the `Functor` type class, and a *reduce* phase, which we usually call `fold`[^hadoop-shuffle] in Scala.

[^hadoop-shuffle]: In Hadoop there is also a shuffle phase that we will ignore here.

20.1. 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]`:

!generic-map.svg, caption: [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`.

!generic-foldleft.svg, caption: [Type chart: fold]

[85] http://research.google.com/archive/map-reduce.html

By distributing the reduce step we lose control over the order of traversal. Our overall reduction may not be entirely left-to-right—we may reduce left-to-right across several subsequences and then combine the results. To ensure correctness we need a reduction operation that is *associative*:

```
reduce(a1, reduce(a2, a3)) == reduce(reduce(a1, a2), a3)
```

If we have associativity, we can arbitrarily distribute work between our nodes provided the subsequences at every node stay in the same order as the initial dataset.

Our fold operation requires us to seed the computation with an element of type `B`. Since fold may be split into an arbitrary number of parallel steps, the seed should not affect the result of the computation. This naturally requires the seed to be an *identity* element:

```
reduce(seed, a1) == reduce(a1, seed) == a1
```

In summary, our parallel fold will yield the correct results if:

- we require the reducer function to be associative;
- we seed the computation with the identity of this function.

What does this pattern sound like? That's right, we've come full circle back to `Monoid`, the first type class we discussed in this book. We are not the first to recognise the importance of monoids. The monoid design pattern for map-reduce jobs[86] is at the core of recent big data systems such as Twitter's Summingbird[87].

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.

[86] http://arxiv.org/abs/1304.7544
[87] https://github.com/twitter/summingbird

20.2. 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.

Now implement the body of `foldMap`. Use the flow chart in Figure 16 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`.

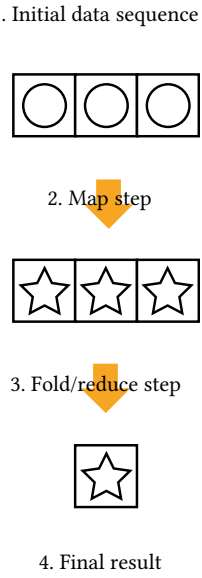

Figure 16: *foldMap* algorithm

Here's some sample output for reference:

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

```
foldMap(Vector(1, 2, 3))(identity)
// res1: Int = 6
```

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

```
// Mapping to a String uses the concatenation monoid:
foldMap(Vector(1, 2, 3))(_.toString + "! ")
// res2: String = "1! 2! 3! "

// Mapping over a String to produce a String:
foldMap("Hello world!".toVector)(_.toString.toUpperCase)
// res3: String = "HELLO WORLD!"
```

20.3. 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 17:

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.

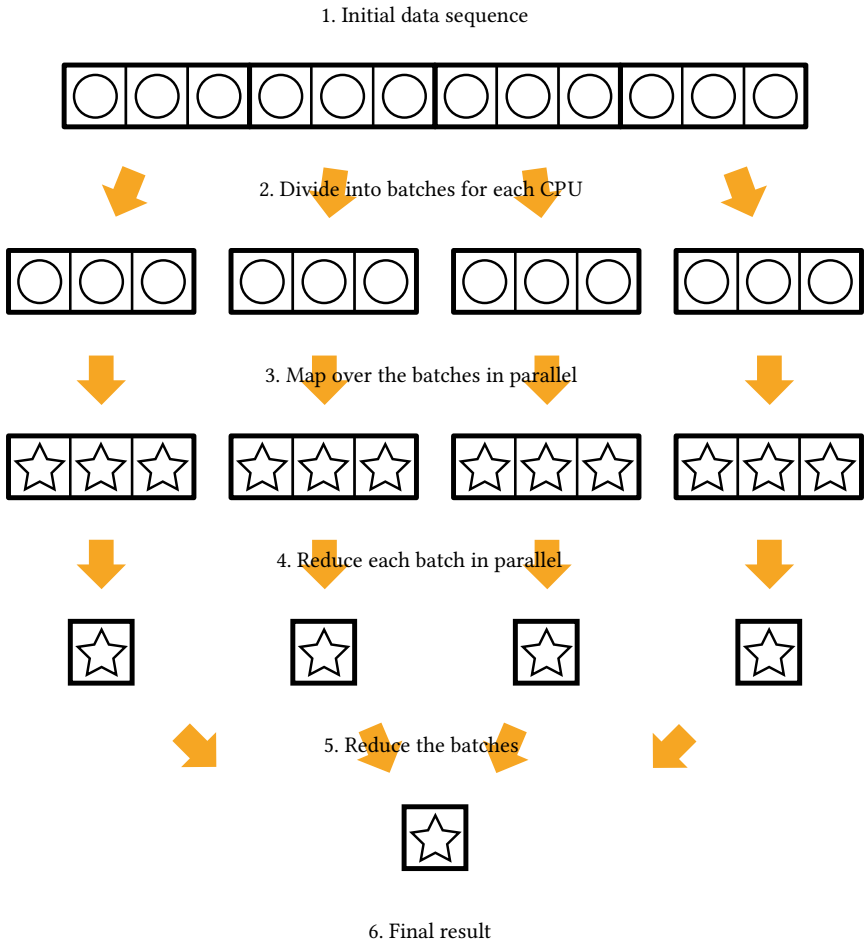

Figure 17: *parallelFoldMap* algorithm

Scala provides some simple tools to distribute work amongst threads. We could use the parallel collections library[88] to implement a solution, but let's challenge ourselves by diving a bit deeper and implementing the algorithm ourselves using `Futures`.

[88]http://docs.scala-lang.org/overviews/parallel-collections/overview.html

20.3.1. 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: Future[Int] = Future(Success(5050))

val future2 = Future {
  (100 to 200).toList.foldLeft(0)(_ + _)
}
// future2: Future[Int] = Future(Success(15150))
```

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: Future[String] = Future(Success(5050))

val future4 = for {
  a <- future1
  b <- future2
} yield a + b
// future4: Future[Int] = Future(Success(20200))
```

As we saw in Section 13.2, we can convert a `List[Future[A]]` to a `Future[List[A]]` using `Future.sequence`:

```
Future.sequence(List(Future(1), Future(2), Future(3)))
// res6: Future[List[Int]] = Future(Success(List(1, 2, 3)))
```

or an instance of `Traverse`:

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

```
List(Future(1), Future(2), Future(3)).sequence
// res7: 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 for the result
// res8: Int = 1
```

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

```
import cats.{Monad, Monoid}
import cats.instances.int._    // for Monoid
import cats.instances.future._ // for Monad and Monoid
```

```
Monad[Future].pure(42)

Monoid[Future[Int]].combine(Future(1), Future(2))
```

20.3.2. 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
// res11: Int = 4
```

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
// res12: List[List[Int]] = List(
//   List(1, 2, 3),
//   List(4, 5, 6),
//   List(7, 8, 9),
//   List(10)
// )
```

20.3.3. 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 17 if you need to review the overall algorithm.

For bonus points, process the batches for each CPU using your implementation of `foldMap` from above.

20.3.4. 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 13.1.

Reimplement `parallelFoldMap` using Cats' `Foldable` and `Traverseable` type classes.

20.4. Summary

In this case study we implemented a system that imitates map-reduce as performed on a cluster. Our algorithm followed three steps:

1. batch the data and send one batch to each "node";
2. perform a local map-reduce on each batch;
3. combine the results using monoid addition.

Our toy system emulates the batching behaviour of real-world map-reduce systems such as Hadoop. However, in reality we are running all of our work on a single machine where communcation between nodes is negligible. We don't actually need to batch data to gain efficient parallel processing of a list. We can simply map using a `Functor` and reduce using a `Monoid`.

Regardless of the batching strategy, mapping and reducing with `Monoids` is a powerful and general framework that isn't limited to

simple tasks like addition and string concatenation. Most of the tasks data scientists perform in their day-to-day analyses can be cast as monoids. There are monoids for all the following:

- approximate sets such as the Bloom filter;
- set cardinality estimators, such as the HyperLogLog algorithm;
- vectors and vector operations like stochastic gradient descent;
- quantile estimators such as the t-digest

to name but a few.

21. Case Study: Data Validation

In this case study we will build a library for validation. What do we mean by validation? Almost all programs must check their input meets certain criteria. Usernames must not be blank, email addresses must be valid, and so on. This type of validation often occurs in web forms, but it could be performed on configuration files, on web service responses, and any other case where we have to deal with data that we can't guarantee is correct. Authentication, for example, is just a specialised form of validation.

We want to build a library that performs these checks. What design goals should we have? For inspiration, let's look at some examples of the types of checks we want to perform:

- A user must be over 18 years old or must have parental consent.
- A `String` ID must be parsable as a `Int` and the `Int` must correspond to a valid record ID.
- A bid in an auction must apply to one or more items and have a positive value.
- A username must contain at least four characters and all characters must be alphanumeric.
- An email address must contain a single `@` sign. Split the string at the `@`. The string to the left must not be empty. The string to the right must be at least three characters long and contain a dot.

With these examples in mind we can state some goals:

- We should be able to associate meaningful messages with each validation failure, so the user knows why their data is not valid.

- We should be able to combine small checks into larger ones. Taking the username example above, we should be able to express this by combining a check of length and a check for alphanumeric values.
- We should be able to transform data while we are checking it. There is an example above requiring we parse data, changing its type from `String` to `Int`.
- Finally, we should be able to accumulate all the failures in one go, so the user can correct all the issues before resubmitting.

These goals assume we're checking a single piece of data. We will also need to combine checks across multiple pieces of data. For a login form, for example, we'll need to combine the check results for the username and the password. This will turn out to be quite a small component of the library, so the majority of our time will focus on checking a single data item.

21.1. 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 abstractly represent this as a value in a context, where the context is the possibility of an error message as shown in Figure 18.

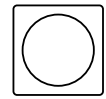

Figure 18: A validation result

A check itself is therefore a function that transforms a value into a value in a context as shown in Figure 19.

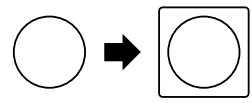

Figure 19: A validation check

Combine checks

How do we combine smaller checks into larger ones? Is this an applicative or semigroupal as shown in Figure 20?

Figure 20: Applicative combination of checks

Not really. With applicative combination, both checks are applied to the same value and result in a tuple with the value repeated. What we want feels more like a monoid as shown in Figure 21. We can define a sensible identity—a check that always passes—and two binary combination operators—*and* and *or*:

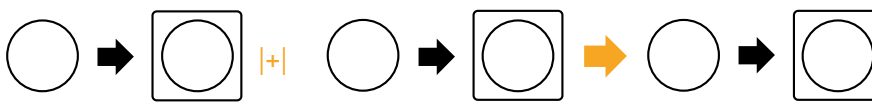

Figure 21: Monoid combination of checks

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 22.

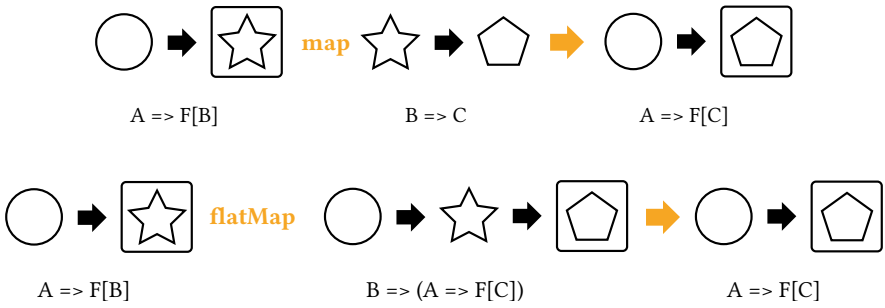

Figure 22: Monadic combination of checks

We've now broken down our library into familiar abstractions and are in a good position to begin development.

21.2. The Check Datatype

Our design revolves around a `Check`, which we said was a function from a value to a value in a context. As soon as you see this description you should think of something like

```
type Check[A] = A => Either[String, A]
```

Here we've represented the error message as a `String`. This is probably not the best representation. We may want to accumulate messages in a `List`, for example, or even use a different representation that allows for internationalization or standard error codes.

We could attempt to build some kind of `ErrorMessage` type that holds all the information we can think of. However, we can't predict the user's requirements. Instead let's let the user specify what they want. We can do this by adding a second type parameter to `Check`:

```
type Check[E, A] = A => Either[E, A]
```

We will probably want to add custom methods to `Check` so let's declare it as a `trait` instead of a type alias:

```
trait Check[E, A] {
  def apply(value: A): Either[E, A]

  // other methods...
}
```

As we said in [Essential Scala][link-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.

21.3. Basic Combinators

Let's add some combinator methods to `Check`, starting with `and`. This method combines two checks into one, succeeding only if both checks succeed. Think about implementing this method now. You should hit some problems. Read on when you do!

```
trait Check[E, A] {
  def and(that: Check[E, A]): Check[E, A] =
    ???

  // other methods...
}
```

The problem is: what do you do when *both* checks fail? The correct thing to do is to return both errors, but we don't currently have any way to combine `E`s. We need a *type class* that abstracts over the concept of "accumulating" errors as shown in Figure 23 What type class do we know that looks like this? What method or operator should we use to implement the `?` operation?

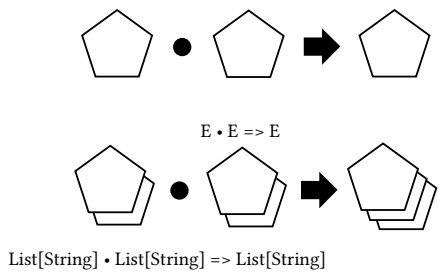

Figure 23: Combining error messages

There is another semantic issue that will come up quite quickly: should `and` short-circuit if the first check fails. What do you think the most useful behaviour is?

Use this knowledge to implement `and`. Make sure you end up with the behaviour you expect!

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.

Our implementation is looking pretty good now. Implement an `or` combinator to complement `and`.

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.

21.4. Transforming Data

One of our requirements is the ability to transform data. This allows us to support additional scenarios like parsing input. In this section we'll extend our check library with this additional functionality.

The obvious starting point is `map`. When we try to implement this, we immediately run into a wall. Our current definition of `Check` requires the input and output types to be the same:

```
type Check[E, A] = A => Either[E, A]
```

When we map over a check, what type do we assign to the result? It can't be `A` and it can't be `B`. We are at an impasse:

```
def map(check: Check[E, A])(func: A => B): Check[E, ???]
```

To implement `map` we need to change the definition of `Check`. Specifically, we need to a new type variable to separate the input type from the output:

```
type Check[E, A, B] = A => Either[E, B]
```

Checks can now represent operations like parsing a `String` as an `Int`:

```
val parseInt: Check[List[String], String, Int] =
  // etc...
```

However, splitting our input and output types raises another issue. Up until now we have operated under the assumption that a `Check` always returns its input when successful. We used this in `and` and `or` to ignore the output of the left and right rules and simply return the original input on success:

```
(this(a), that(a)) match {
  case And(left, right) =>
    (left(a), right(a))
      .mapN((result1, result2) => Right(a))

  // etc...
}
```

In our new formulation we can't return `Right(a)` because its type is `Either[E, A]` not `Either[E, B]`. We're forced to make an arbitrary choice between returning `Right(result1)` and `Right(result2)`. The same is true of the `or` method. From this we can derive two things:

- we should strive to make the laws we adhere to explicit; and
- the code is telling us we have the wrong abstraction in `Check`.

21.4.1. Predicates

We can make progress by pulling apart the concept of a *predicate*, which can be combined using logical operations such as *and* and *or*, and the concept of a *check*, which can transform data.

What we have called `Check` so far we will call `Predicate`. For `Predicate` we can state the following *identity law* encoding the notion that a predicate always returns its input if it succeeds:

> For a predicate `p` of type `Predicate[E, A]` > and elements `a1` and `a2` of type `A`, > if `p(a1) == Success(a2)` then `a1 == a2`.

Making this change gives us the following code:

```
import cats.Semigroup
import cats.data.Validated
import cats.syntax.semigroup._ // for |+|
import cats.syntax.apply._     // for mapN
import cats.data.Validated._   // for Valid and Invalid
```

```
sealed trait Predicate[E, A] {
  def and(that: Predicate[E, A]): Predicate[E, A] =
    And(this, that)

  def or(that: Predicate[E, A]): Predicate[E, A] =
    Or(this, that)

  def apply(a: A)(implicit s: Semigroup[E]): Validated[E, A] =
    this match {
      case Pure(func) =>
        func(a)

      case And(left, right) =>
        (left(a), right(a)).mapN((_, _) => a)

      case Or(left, right) =>
        left(a) match {
          case Valid(_)   => Valid(a)
          case Invalid(e1) =>
            right(a) match {
              case Valid(_)   => 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]
```

21.4.2. Checks

We'll use `Check` to represent a structure we build from a `Predicate` that also allows transformation of its input. Implement `Check` with the following interface:

```
sealed trait Check[E, A, B] {
  def apply(a: A): Validated[E, B] =
    ???

  def map[C](func: B => C): Check[E, A, C] =
    ???
}
```

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 24.

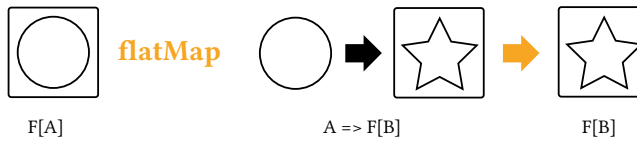

Figure 24: Type chart for flatMap

How do we relate `F` in the figure to `Check` in our code? `Check` has *three* type variables while `F` only has one.

To unify the types we need to fix two of the type parameters. The idiomatic choices are the error type `E` and the input type `A`. This

gives us the relationships shown in Figure 25. In other 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]`.

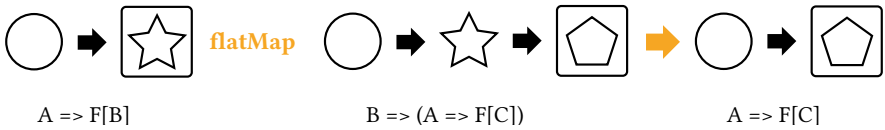

Figure 25: Type chart for flatMap applied to Check

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.

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!

21.4.3. Recap

We now have two algebraic data types, `Predicate` and `Check`, and a host of combinators with their associated case class implementations. Look at the following solution for a complete definition of each ADT.

We have a complete implementation of `Check` and `Predicate` that do most of what we originally set out to do. However, we are not finished yet. You have probably recognised structure in `Predicate` and `Check` that we can abstract over: `Predicate` has a monoid and `Check` has a monad. Furthermore, in implementing `Check` you might have felt the implementation doesn't do much—all we do is call through to underlying methods on `Predicate` and `Validated`.

There are a lot of ways this library could be cleaned up. However, let's implement some examples to prove to ourselves that our library really does work, and then we'll turn to improving it.

Implement checks for some of the examples given in the introduction:

- A username must contain at least four characters and consist entirely of alphanumeric characters
- An email address must contain an `@` sign. Split the string at the `@`. The string to the left must not be empty. The string to the right must be at least three characters long and contain a dot.

You might find the following predicates useful:

```
import cats.data.{NonEmptyList, Validated}
```

```
type Errors = NonEmptyList[String]

def error(s: String): NonEmptyList[String] =
  NonEmptyList(s, Nil)

def longerThan(n: Int): Predicate[Errors, String] =
```

```
  Predicate.lift(
    error(s"Must be longer than $n characters"),
    str => str.size > n)

val alphanumeric: Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must be all alphanumeric characters"),
    str => str.forall(_.isLetterOrDigit))

def contains(char: Char): Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must contain the character $char"),
    str => str.contains(char))

def containsOnce(char: Char): Predicate[Errors, String] =
  Predicate.lift(
    error(s"Must contain the character $char only once"),
    str => str.filter(c => c == char).size == 1)
```

21.5. 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 26. 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)
```

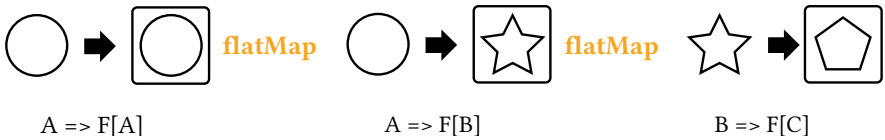

Figure 26: Sequencing monadic transforms

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`][cats.data.Kleisli] that wraps a function just as `Check` does. `Kleisli` has all the methods of `Check` plus some additional ones. If `Kleisli` seems familiar to you, then congratulations. You've seen through its disguise and recognised it as another concept from earlier in the book: `Kleisli` is just another name for `ReaderT`.

Here is a simple example using `Kleisli` to transform an integer into a list of integers through three steps:

```
import cats.data.Kleisli
import cats.instances.list._ // for Monad
```

These steps each transform an input `Int` into an output of type `List[Int]`:

```
val step1: Kleisli[List, Int, Int] =
  Kleisli(x => List(x + 1, x - 1))

val step2: Kleisli[List, Int, Int] =
  Kleisli(x => List(x, -x))

val step3: Kleisli[List, Int, Int] =
  Kleisli(x => List(x * 2, x / 2))
```

We can combine the steps into a single pipeline that combines the underlying `Lists` using `flatMap`:

```
val pipeline = step1 andThen step2 andThen step3
```

The result is a function that consumes a single `Int` and returns eight outputs, each produced by a different combination of transformations from `step1`, `step2`, and `step3`:

```
pipeline.run(20)
// res0: 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.

Now rewrite our username and email validation example in terms of `Kleisli` and `Predicate`. Here are few tips in case you get stuck:

First, remember that the `run` method on `Predicate` takes an implicit parameter. If you call `aPredicate.run(a)` it will try to pass the implicit parameter explicitly. If you want to create a function from a `Predicate` and immediately apply that function, use `aPredicate.run.apply(a)`

Second, type inference can be tricky in this exercise. We found that the following definitions helped us to write code with fewer type declarations.

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

type Check[A, B] = Kleisli[Result, A, B]

// Create a check from a function:
def check[A, B](func: A => Result[B]): Check[A, B] =
  Kleisli(func)

// Create a check from a Predicate:
def checkPred[A](pred: Predicate[Errors, A]): Check[A, A] =
  Kleisli[Result, A, A](pred.run)
```

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.

21.6. Summary

This case study has been an exercise in removing rather than building abstractions. We started with a fairly complex `Check` type. Once we realised we were conflating two concepts, we separated out `Predicate` leaving us with something that could be implemented with `Kleisli`.

<!– `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 `ands`) or a failing disjunction (two `ors`)? 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. ->

We made several design choices above that reasonable developers may disagree with. Should the method that converts a `Predicate` to a function really be called `run` instead of, say, `toFunction`? Should `Predicate` be a subtype of `Function` to begin with? Many functional programmers prefer to avoid subtyping because it plays poorly with implicit resolution and type inference, but there could be an argument to use it here. As always the best decisions depend on the context in which the library will be used.

22. Case Study: CRDTs

In this case study we will explore **Commutative Replicated Data Types (CRDTs)**, a family of data structures that can be used to reconcile eventually consistent data.

We'll start by describing the utility and difficulty of eventually consistent systems, then show how we can use monoids and their extensions to solve the issues that arise. Finally, we will model the solutions in Scala.

Our goal here is to focus on the implementation in Scala of a particular type of CRDT. We're not aiming at a comprehensive survey of all CRDTs. CRDTs are a fast-moving field and we advise you to read the literature to learn about more.

22.1. Eventual Consistency

As soon as a system scales beyond a single machine we have to make a fundamental choice about how we manage data.

One approach is to build a system that is *consistent*, meaning that all machines have the same view of data. For example, if a user changes their password then all machines that store a copy of that password must accept the change before we consider the operation to have completed successfully.

Consistent systems are easy to work with but they have their disadvantages. They tend to have high latency because a single change can result in many messages being sent between machines. They also tend to have relatively low uptime because outages can cut communications between machines creating a *network partition*. When there is a network partition, a consistent system

may refuse further updates to prevent inconsistencies across machines.

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

Eventually consistent systems require less communication between machines so latency can be lower. A partitioned machine can still accept updates and reconcile its changes when the network is fixed, so systems can also have better uptime.

The big question is: how do we do this reconciliation between machines? CRDTs provide one approach to the problem.

22.2. The GCounter

Let's look at one particular CRDT implementation. Then we'll attempt to generalise properties to see if we can find a general pattern.

The data structure we will look at is called a *GCounter*. It is a distributed *increment-only* counter that can be used, for example, to count the number of visitors to a web site where requests are served by many web servers.

22.2.1. Simple Counters

To see why a straightforward counter won't work, imagine we have two servers storing a simple count of visitors. Let's call the machines `A` and `B`. Each machine is storing an integer counter and the counters all start at zero as shown in Figure 27.

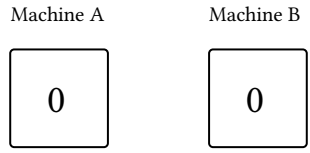

Figure 27: Simple counters: initial state

Now imagine we receive some web traffic. Our load balancer distributes five incoming requests to `A` and `B`, `A` serving three visitors and `B` two. The machines have inconsistent views of the system state that they need to *reconcile* to achieve consistency. One reconciliation strategy with simple counters is to exchange counts and add them as shown in Figure 28.

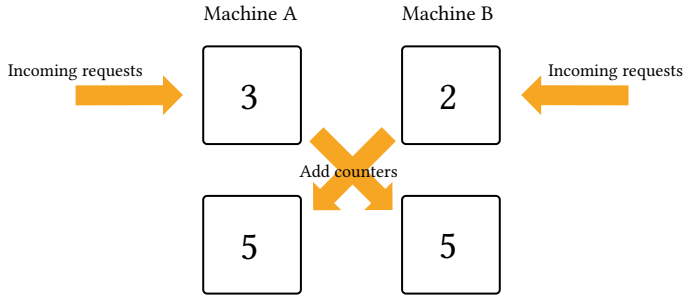

Figure 28: Simple counters: first round of requests and reconciliation

So far so good, but things will start to fall apart shortly. Suppose `A` serves a single visitor, which means we've seen six visitors in total. The machines attempt to reconcile state again using addition leading to the answer shown in Figure 29.

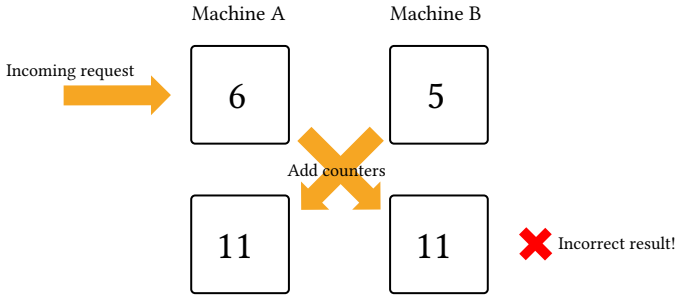

Figure 29: Simple counters: second round of requests and (incorrect) reconciliation

This is clearly wrong! The problem is that simple counters don't give us enough information about the history of interactions between the machines. Fortunately we don't need to store the *complete* history to get the correct answer—just a summary of it. Let's look at how the GCounter solves this problem.

22.2.2. GCounters

The first clever idea in the GCounter is to have each machine storing a *separate* counter for every machine it knows about (including itself). In the previous example we had two machines, `A` and `B`. In this situation both machines would store a counter for `A` and a counter for `B` as shown in Figure 30.

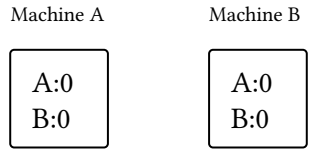

Figure 30: GCounter: initial state

The rule with GCounters is that a given machine is only allowed to increment its own counter. If `A` serves three visitors and `B` serves two visitors the counters look as shown in Figure 31.

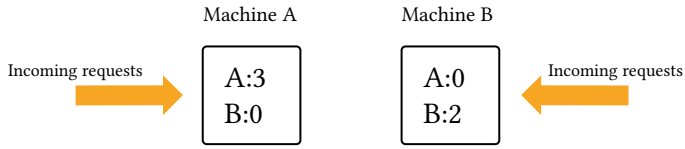

Figure 31: GCounter: first round of web requests

When two machines reconcile their counters the rule is to take the largest value stored for each machine. In our example, the result of the first merge will be as shown in Figure 32.

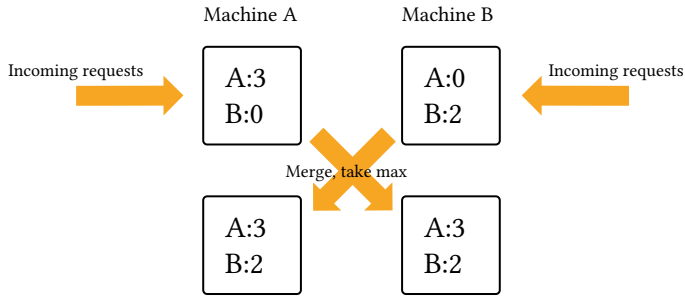

Figure 32: GCounter: first reconciliation

Subsequent incoming web requests are handled using the increment-own-counter rule and subsequent merges are handled using the take-maximum-value rule, producing the same correct values for each machine as shown in Figure 33.

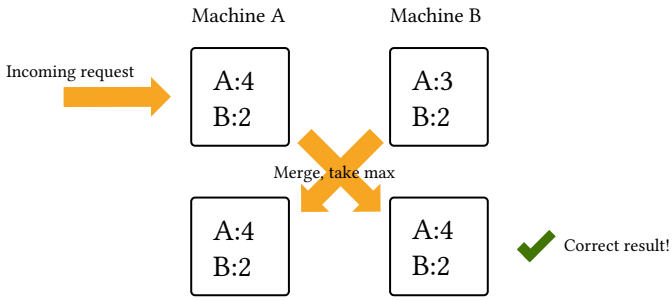

Figure 33: GCounter: second reconciliation

GCounters allow each machine to keep an accurate account of the state of the whole system without storing the complete history of interactions. If a machine wants to calculate the total traffic for the whole web site, it sums up all the per-machine counters. The result is accurate or near-accurate depending on how recently we performed a reconciliation. Eventually, regardless of network outages, the system will always converge on a consistent state.

22.2.3. Exercise: GCounter Implementation

We can implement a GCounter with the following interface, where we represent machine IDs as `Strings`.

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

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

  def total: Int =
    ???
}
```

Finish the implementation!

22.3. Generalisation

We've now created a distributed, eventually consistent, increment-only counter. This is a useful achievement but we don't want to stop here. In this section we will attempt to abstract the operations in the GCounter so it will work with more data types than just natural numbers.

The GCounter uses the following operations on natural numbers:

- addition (in `increment` and `total`);
- maximum (in `merge`);
- and the identity element 0 (in `increment` and `merge`).

You can probably guess that there's a monoid in here somewhere, but let's look in more detail at the properties we're relying on.

As a refresher, in Chapter 8 we saw that monoids must satisfy two laws. The binary operation + must be associative:

```
(a + b) + c == a + (b + c)
```

and the empty element must be an identity:

```
0 + a == a + 0 == a
```

We need an identity in `increment` to initialise the counter. We also rely on associativity to ensure the specific sequence of `merges` gives the correct value.

In `total` we implicitly rely on associativity and commutativity to ensure we get the correct value no matter what arbitrary order we choose to sum the per-machine counters. We also implicitly assume an identity, which allows us to skip machines for which we do not store a counter.

The properties of `merge` are a bit more interesting. We rely on commutativity to ensure that machine `A` merging with machine `B` yields the same result as machine `B` merging with machine `A`. We need associativity to ensure we obtain the correct result when three or more machines are merging data. We need an identity element to initialise empty counters. Finally, we need an additional property, called *idempotency*, to ensure that if two machines hold the same data in a per-machine counter, merging data will not lead to an incorrect result. Idempotent operations are ones that return the same result again and again if they are executed multiple times. Formally, a binary operation `max` is idempotent if the following relationship holds:

```
a max a = a
```

Written more compactly, we have:

——————————————————————— Method Identity Commutative Associative Idempotent ————— ———— ————- ————- ————- `increment` Y N Y N

`merge` Y Y Y Y

`total` Y Y Y N ———————————————————

From this we can see that

- `increment` requires a monoid;
- `total` requires a commutative monoid; and
- `merge` required an idempotent commutative monoid, also called a *bounded semilattice*.

Since `increment` and `get` both use the same binary operation (addition) it's usual to require the same commutative monoid for both.

This investigation demonstrates the powers of thinking about properties or laws of abstractions. Now we have identified these properties we can substitute the natural numbers used in our GCounter with any data type with operations satisfying these properties. A simple example is a set, with the binary operation being union and the identity element the empty set. With this simple substitution of `Int` for `Set[A]` we can create a GSet type.

22.3.1. Implementation

Let's implement this generalisation in code. Remember `increment` and `total` require a commutative monoid and `merge` requires a bounded semilattice (or idempotent commutative monoid).

Cats provides a type class for both `Monoid` and `CommutativeMonoid`, but doesn't provide one for bounded semilattice[^spire]. That's why we're going to implement our own `BoundedSemiLattice` type class.

```
import cats.kernel.CommutativeMonoid

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

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

22.3.2. Exercise: BoundedSemiLattice Instances

Implement `BoundedSemiLattice` type class instances for `Int`s and for `Set`s. The instance for `Int` will technically only hold for non-negative numbers, but you don't need to model non-negativity explicitly in the types.

22.3.3. Exercise: Generic GCounter

Using `CommutativeMonoid` and `BoundedSemiLattice`, generalise `GCounter`.

When you implement this, look for opportunities to use methods and syntax on `Monoid` to simplify your implementation. This is a good example of how type class abstractions work at multiple levels in our code. We're using monoids to design a large component—our CRDTs—but they are also useful in the small, simplifying our code and making it shorter and clearer.

[^spire]: A closely related library called Spire already provides that abstractions.

22.4. Abstracting GCounter to a Type Class

We've created a generic GCounter that works with any value that has instances of `BoundedSemiLattice` and `CommutativeMonoid`. However we're still tied to a particular representation of the map from machine IDs to values. There is no need to have this restriction, and indeed it can be useful to abstract away from it. There are many key-value stores that we want to work with, from a simple `Map` to a relational database.

If we define a `GCounter` type class we can abstract over different concrete implementations. This allows us to, for example, seamlessly substitute an in-memory store for a persistent store when we want to change performance and durability tradeoffs.

There are a number of ways we can implement this. One approach is to define a `GCounter` type class with dependencies on `CommutativeMonoid` and `BoundedSemiLattice`. We define this as a type class that takes a type constructor with *two* type parameters represent the key and value types of the map abstraction.

```
trait GCounter[F[_,_],K, V] {
  def increment(f: F[K, V])(k: K, v: V)
        (implicit m: CommutativeMonoid[V]): F[K, V]

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

  def total(f: F[K, V])
        (implicit m: CommutativeMonoid[V]): V
}
```

```
object GCounter {
  def apply[F[_,_], K, V]
        (implicit counter: GCounter[F, K, V]) =
    counter
}
```

Try defining an instance of this type class for `Map`. You should be able to reuse your code from the case class version of `GCounter` with some minor modifications.

You should be able to use your instance as follows:

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

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

val counter = GCounter[Map, String, Int]
```

```
val merged = counter.merge(g1, g2)
// merged: Map[String, Int] = Map("a" -> 7, "b" -> 5)
val total  = counter.total(merged)
// total: Int = 12
```

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

22.5. Abstracting a Key Value Store

One solution is to capture the idea of a key-value store within a type class, and then generate `GCounter` instances for any type that has a `KeyValueStore` instance. Here's the code for such a type class:

```
trait KeyValueStore[F[_,_]] {
  def put[K, V](f: F[K, V])(k: K, v: V): F[K, V]

  def get[K, V](f: F[K, V])(k: K): Option[V]

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

  def values[K, V](f: F[K, V]): List[V]
}
```

Implement your own instance for `Map`.

With our type class in place we can implement syntax to enhance data types for which we have instances:

```
implicit class KvsOps[F[_,_], K, V](f: F[K, V]) {
  def put(key: K, value: V)
        (implicit kvs: KeyValueStore[F]): F[K, V] =
    kvs.put(f)(key, value)

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

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

  def values(implicit kvs: KeyValueStore[F]): List[V] =
    kvs.values(f)
}
```

Now we can generate `GCounter` instances for any data type that has instances of `KeyValueStore` and `CommutativeMonoid` using an `implicit def`:

```
implicit def gcounterInstance[F[_,_], K, V]
    (implicit kvs: KeyValueStore[F], km: CommutativeMonoid[F[K,
V]]): GCounter[F, K, V] =
  new GCounter[F, K, V] {
    def increment(f: F[K, V])(key: K, value: V)
          (implicit m: CommutativeMonoid[V]): F[K, V] = {
      val total = f.getOrElse(key, m.empty) |+| value
```

```
      f.put(key, total)
    }

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

    def total(f: F[K, V])(implicit m: CommutativeMonoid[V]): V =
      f.values.combineAll
  }
```

The complete code for this case study is quite long, but most of it is boilerplate setting up syntax for operations on the type class. We can cut down on this using compiler plugins such as [Simulacrum][link-simulacrum] and [Kind Projector][link-kind-projector].

22.6. Summary

In this case study we've seen how we can use type classes to model a simple CRDT, the GCounter, in Scala. Our implementation gives us a lot of flexibility and code reuse: we aren't tied to the data type we "count", nor to the data type that maps machine IDs to counters.

The focus in this case study has been on using the tools that Scala provides, not on exploring CRDTs. There are many other CRDTs, some of which operate in a similar manner to the GCounter, and some of which have very different implementations. A [fairly recent survey][link-crdt-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{Appendices} appendix

23. Acknowledgements

No book is an island. This book wouldn't exist without it's predecessor, *Scala with Cats*, and everyone involved in creating that book implicitly played some part in this book's creation. See below for that book's acknowledgements, but in particular I want to highlight my coauthor, Dave "Lord of Types" Pereira-Gurnell, without whom that book would not exist and hence neither would this one. Thanks Dave!

Thanks also to Adam Rosien, who gave me low-key encouragement and put up with my bullshit. Also my wife and children, who put up with even more of my bullshit, and gave me the space to finish this project. The members of ScalaBridge London and attendees at various training courses acted as experimental subjects for a lot of the material here. Thank you for being willing test subjects; you greatly helped improve the content. Thanks for the members of the PLT research group who inspired me directly back in the day, and continue to provide inspiration from afar. Finally, thanks to the following who sponsored my work or contributed with corrections and suggestions:

Aleksandr Andreev, Charles Adetiloye, Johanna Odersky, Lunfu Zhong, Maciej Gorywoda , Mathieu Pichette, Murat Cetin , Olya Mazhara, Pavel Syvak, Philip Schwarz, Seth Tisue, Tim Eccleston (@combinatorist).

23.1. Acknowledgements from Scala with Cats

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

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

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

23.1.1. Backers

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

copy before we released it as open source. This book wouldn't exist without you:

A battle-hardened technologist, Aaron Pritzlaff, Abhishek Srivastava, Aleksey "Daron" Terekhin, Algolia, Allen George (@allenageorge), Andrew Johnson, Andrew Kerr, Andy Dwelly, Anler, anthony@dribble.ai, Aravindh Sridaran, Araxis Ltd, ArtemK, Arthur Kushka (@arhelmus), Artur Zhurat, Arturas Smorgun, Attila Mravik, Axel Gschaider, Bamboo Le, bamine, Barry Kern, Ben Darfler (@bdarfler), Ben Letton, Benjamin Neil, Benoit Hericher, Bernt Andreas Langøien, Bill Leck, Blaze K, Boniface Kabaso, Brian Wongchaowart, Bryan Dragon, @cannedprimates, Ceschiatti (@6qat), Chris Gojlo, Chris Phelps, @CliffRedmond, Cody Koeninger, Constantin Gonciulea, Dadepo Aderemi, Damir Vandic, Damon Rolfs, Dan Todor, Daniel Arndt, Daniela Sfregola, David Greco, David Poltorak, Dennis Hunziker, Dennis Vriend, Derek Morr, Dimitrios Liapis, Don McNamara, Doug Clinton, Doug Lindholm (dlindhol), Edgar Mueller, Edward J Renauer Jr, Emiliano Martinez, esthom, Etienne Peiniau, Fede Silva, Filipe Azevedo, Franck Rasolo, Gary Coady, George Ball, Gerald Loeffler, Integrational, Giles Taylor, Guilherme Dantas (@gamsd), Harish Hurchurn, Hisham Ismail, Iurii Susuk, Ivan (SkyWriter) Kasatenko, Ivano Pagano, Jacob Baumbach, James Morris, Jan Vincent Liwanag, Javier Gonzalez, Jeff Gentry, Joel Chovanec, Jon Bates, Jorge Aliss (@jaliss), Juan Macias (@1macias1), Juan Ortega, Juan Pablo Romero Méndez, Jungsun Kim, Kaushik Chakraborty (@kaychaks), Keith Mannock, Ken Hoffman, Kevin Esler, Kevin Kyyro, kgillies, Klaus Rehm, Kostas Skourtis, Lance Linder, Liang, Guang Hua, Loïc Girault, Luke Tebbs, Makis A, Malcolm Robbins, Mansur Ashraf (@mansur_ashraf), Marcel Lüthi, Marek Prochera @hicolour, Marianudo (Mariano Navas), Mark Eibes, Mark van Rensburg, Martijn Blankestijn, Martin Studer, Matthew Edwards, Matthew Pflueger, mauropalsgraaf, mbarak, Mehitabel,

Michael Pigg, Mikael Moghadam, Mike Gehard (@mikegehard), MonadicBind, arjun.mukherjee@gmail.com, Stephen Arbogast, Narayan Iyer, @natewave, Netanel Rabinowitz, Nick Peterson, Nicolas Sitbon, Oier Blasco Linares, Oliver Daff, Oliver Schrenk, Olly Shaw, P Villela, pandaforme, Patrick Garrity, Pawel Wlodarski from JUG Lodz, @peel, Peter Perhac, Phil Glover, Philipp Leser-Wolf, Rachel Bowyer, Radu Gancea (@radusw), Rajit Singh, Ramin Alidousti, Raymond Tay, Riccardo Sirigu, Richard (Yin-Wu) Chuo, Rob Vermazeren, Robert “Kemichal” Andersson, Robin Taylor (@badgermind), Rongcui Dong, Rui Morais, Rupert Bates, Rustem Suniev, Sanjiv Sahayam, Shane Delmore, Stefan Plantikow, Sundy Wiliam Yaputra, Tal Pressman, Tamas Neltz, theLXK, Tim Pigden, Tobias Lutz, Tom Duhourq, @tomzalt, Utz Westermann, Vadym Shalts, Val Akkapeddi, Vasanth Loka, Vladimir Bacvanski, Vladimir Bystrov aka udav_pit, William Benton, Wojciech Langiewicz, Yann Ollivier (@ya2o), Yoshiro Naito, zero323, and zeronone.

backmatter

Bibliography

1. Harold Abelson and Gerald Jay Sussman. 1996. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA.

2. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. From Interpreter to Compiler and Virtual Machine: A Functional Derivation. *BRICS Report Series* 10, 14.

3. Patrick Bahr and Graham Hutton. 2015. Calculating correct compilers. *Journal of Functional Programming* 25: e14. https://doi.org/10.1017/S0956796815000180

4. Kent Beck. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

5. James R. Bell. 1973. Threaded Code. *Communications of the ACM* 16, 6: 370–372. https://doi.org/10.1145/362248.362270

6. Joshua Bloch. 2017. *Effective Java*. Addison-Wesley Professional.

7. Margaret A. Boden and Ernest A. Edmonds. 2009. What is generative art?. *Digital Creativity*, 20: 21–46. https://doi.org/10.1080/14626260902867915

8. Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *Journal of the ACM (JACM)* 11, 4: 481–494. https://doi.org/10.1145/321239.321249

9. R. M. Burstall. 1969. Proving Properties of Programs by Structural Induction. *The Computer Journal* 12, 1: 41–48. https://doi.org/10.1093/comjnl/12.1.41

10. Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of*

Functional Programming 5: 509–543. Retrieved from https://okmij.org/ftp/tagless-final/JFP.pdf

11. Kevin Casey, David Gregg, M. Anton Ertl, and Andrew Nisbet. 2003. Towards Superinstructions for Java Interpreters. In *Software and Compilers for Embedded Systems, 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003, Proceedings* (Lecture Notes in Computer Science), 329–343. https://doi.org/10.1007/978-3-540-39920-9_23

12. Giuseppe Castagna, Guillaume Duboc, and José Valim. 2023. The Design Principles of the Elixir Type System. *The Art, Science, and Engineering of Programming* 8, 2. https://doi.org/10.22152/programming-journal.org/2024/8/4

13. James Cheney and Ralf Hinze. 2003. *First-class Phantom Types*. Cornell University. Retrieved from https://ecommons.cornell.edu/server/api/core/bitstreams/6cf38000-1bc3-4572-b71f-9ddb06f3565c/content

14. Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (ICFP '00), 268–279. https://doi.org/10.1145/351240.351266

15. David Conley. 2014. *Learning Strategies as Metacognitive Factors: A Critical Review*. Educational Policy Improvement Center, Eugene, OR.

16. William Cook. 1990. Object-Oriented Programming Versus Abstract Data Types. In *Proceedings of the REX Workshop/School on the Foundations of Object-Oriented Languages (FOOL)* (LNCS), 151–178. Retrieved from https://www.cs.utexas.edu/~wcook/papers/OOPvsADT/CookOOPvsADT90.pdf

17. Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *Proceedings of the 3rd ACM SIGPLAN International*

Conference on Principles and Practice of Declarative Programming (PPDP '01), 162–174. https://doi.org/10.1145/773184.773202

18. Nina Dethlefs and Ken Hawick. 2017. DEFIne: A Fluent Interface DSL for Deep Learning Applications. In *Proceedings of the 2nd International Workshop on Real World Domain Specific Languages* (RWDSL17). https://doi.org/10.1145/3039895.3039898

19. Robert B. K. Dewar. 1975. Indirect Threaded Code. *Communications of the ACM* 18, 6: 330–331. https://doi.org/10.1145/360825.360849

20. Alan Dorin, Jonathan McCabe, Jon McCormack, Gordon Monro, and Mitchell Whitelaw. 2012. A Framework for Understanding Generative Art. *Digital Creativity*: 239–259. https://doi.org/10.1080/14626268.2012.709940

21. Paul Downen and Zena M. Ariola. 2021. Classical (Co)Recursion: Programming. *CoRR*. Retrieved from https://arxiv.org/abs/2103.06913

22. Paul Downen, Zachary Sullivan, Zena M Ariola, and Simon Peyton Jones. 2019. Codata in Action. In *European Symposium on Programming*, 119–146. Retrieved from https://www.microsoft.com/en-us/research/uploads/prod/2020/01/CoDataInAction.pdf

23. Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. *ACM SIGPLAN Notices* 47, 12: 61–72. https://doi.org/10.1145/2430532.2364515

24. M. Anton Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *Journal of Instruction Level Parallelism* 5. Retrieved from http://www.jilp.org/vol5/v5paper12.pdf

25. M. Anton Ertl. 1995. Stack Caching for Interpreters. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, 315–327. https://doi.org/10.1145/207110.207165

26. Martin Erwig and Steve Kollmansberger. 2006. Functional Pearls: Probabilistic Functional Programming in Haskell. *Journal of Functional Programming* 16, 1: 21–34. https://doi.org/10.1017/S0956796805005721

27. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs*. The MIT Press, Cambridge, MA. Retrieved from https://htdp.org/

28. Martin Fowler. 2005. Fluent Interface. Retrieved May 12, 2025 from https://www.martinfowler.com/bliki/FluentInterface.html

29. Phil Freeman. 2015. Stack Safety for Free. Retrieved from https://functorial.com/stack-safety-for-free/index.pdf

30. Steve Freeman and Nat Pryce. 2006. Evolving an Embedded Domain-Specific Language in Java. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '06), 855–865. https://doi.org/10.1145/1176617.1176735

31. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

32. Jeremy Gibbons and Geraint Jones. 1998. The Under-appreciated Unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (ICFP '98), 273–279. https://doi.org/10.1145/289423.289455

33. Jeremy Gibbons and Bruno César dos Santos Oliveira. 2009. The Essence of the Iterator Pattern. *Journal of Functional*

Programming 19, 3&4: 377–402. https://doi.org/10.1017/S0956796809007291

34. Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). *SIGPLAN Not.* 49, 9: 339–347. https://doi.org/10.1145/2692915.2628138

35. Jeremy Gibbons. 2021. How to Design Co-Programs. *Journal of Functional Programming* 31: e15. https://doi.org/10.1017/S0956796821000113

36. Jeremy Gibbons. 2022. Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity. *The Art, Science, and Engineering of Programming* 6, 7. https://doi.org/https://doi.org/10.22152/programming-journal.org/2022/6/7

37. Yossi Gil and Ori Roth. 2019. Fling - A Fluent API Generator. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)* (Leibniz International Proceedings in Informatics (LIPIcs), 13:1–13:25. https://doi.org/10.4230/LIPIcs.ECOOP.2019.13

38. Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (ICSE '24). https://doi.org/10.1145/3597503.3639581

39. Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, 185–200. https://doi.org/10.1145/3062341.3062363

40. Tatsuya Hagino. 1989. Codatatypes in ML. *Journal of Symbolic Computation* 8, 6: 629–650. https://doi.org/https://doi.org/10.1016/S0747-7171(89)80065-3

41. K. A. Hawick. 2013. Fluent Interfaces and Domain-Specific Languages for Graph Generation and Network Analysis Calculations. In *Procceedings of the International Conference on Software Engineering (SE'13)*, 752–759.

42. Stefan Kaes. 1988. Parametric Overloading in Polymorphic Programming Languages. In *ESOP '88*, 131–144.

43. Eric Kidd. 2007. Build your own probability monads. Retrieved from https://www.randomhacks.net/files/build-your-own-probability-monads.pdf

44. Oleg Kiselyov. 2005. Beyond Church encoding: Boehm-Berarducci isomorphism of algebraic data types and polymorphic lambda-terms. Retrieved from https://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html

45. Oleg Kiselyov. 2012. Typed Tagless Final Interpreters. In *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, Jeremy Gibbons (ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 130–174. https://doi.org/10.1007/978-3-642-32202-0_3

46. Ansten Klev. 2019. A Comparison of Type Theory with Set Theory. In *Reflections on the Foundations of Mathematics: Univalent Foundations, Set Theory and General Thoughts*, Stefania Centrone, Deborah Kant and Deniz Sarikaya (eds.). Springer International Publishing, Cham, 271–292. https://doi.org/10.1007/978-3-030-15655-8_12

47. Filip Křikava, Heather Miller, and Jan Vitek. 2019. Scala Implicits Are Everywhere: A Large-Scale Study of the Use of Scala Implicits in the Wild. *Proceedings of the ACM on*

Programming Languages 3, OOPSLA. https://doi.org/10.1145/3360589

48. Daan Leijen and Erik Meijer. 2000. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages* (DSL '99), 109–122. https://doi.org/10.1145/331960.331977

49. Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

50. Tomer Levy. 2016. A Fluent API for Automatic Generation of Fluent APIs in Java. Technion.

51. Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '00), 108–118. https://doi.org/10.1145/325694.325708

52. Chuan-kai Lin and Tim Sheard. 2010. Pointwise generalized algebraic data types. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (TLDI '10), 51–62. https://doi.org/10.1145/1708016.1708024

53. Tadhg E. MacIntyre, Eric R. Igou, Mark J. Campbell, Aidan P. Moran, and James Matthews. 2014. Metacognition and Action: A New Pathway to Understanding Social and Cognitive Aspects of Expertise in Sport. *Frontiers in Psychology* 5, 1155. https://doi.org/10.3389/fpsyg.2014.01155

54. Conor Mcbride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1: 1–13. https://doi.org/10.1017/S0956796807006326

55. Conor McBride. 2001. The Derivative of a Regular Type is its Type of One-Hole Contexts. Retrieved from http://strictlypositive.org/diff.pdf

56. Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture*, 124–144. Retrieved from https://ris.utwente.nl/ws/portalfiles/portal/6142049/meijer91functional.pdf

57. Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with Derivatives: a Functional Pearl. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, 189–195. https://doi.org/10.1145/2034773.2034801

58. James H. Morris. 1973. Types Are Not Sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL '73), 120–124. https://doi.org/10.1145/512927.512938

59. Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *Proceedings of the ACM on Programming Languages* 7, PLDI. https://doi.org/10.1145/3591262

60. Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. 2017. Silverchain: A Fluent API Generator. *SIGPLAN Not.* 52, 12: 199–211. https://doi.org/10.1145/3170492.3136041

61. Tomoki Nakamaru, Tomomasa Matsunaga, Tetsuro Yamazaki, Soramichi Akiyama, and Shigeru Chiba. 2020. An Empirical Study of Method Chaining in Java. In *Proceedings of the 17th*

International Conference on Mining Software Repositories (MSR '20), 93–102. https://doi.org/10.1145/3379597.3387441

62. Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.* 2, POPL. https://doi.org/10.1145/3158130

63. Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the masses: practical extensibility with object algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (ECOOP'12), 2–27. https://doi.org/10.1007/978-3-642-31057-7_2

64. Bruno C. D. S. Oliveira and Jeremy Gibbons. 2010. Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming. *Journal of Functional Programming* 20, 3–4: 303–352. https://doi.org/10.1017/S0956796810000171

65. Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes as Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA '10), 341–360. https://doi.org/10.1145/1869459.1869489

66. Klaus Ostermann and Julian Jabs. 2018. Dualizing Generalized Algebraic Data Types by Matrix Transposition. In *Programming Languages and Systems*, 60–85.

67. Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-Expression Derivatives Re-Examined. *Journal of Functional Programming* 19, 2: 173–190. https://doi.org/10.1017/S0956796808007090

68. Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. *SIGPLAN Not.* 41, 9: 50–61. https://doi.org/10.1145/1160074.1159811

69. Benjamin C. Pierce. 2002. *Types and Programming Languages*. The MIT Press, Cambridge, MA.

70. Paul R. Pintrich. 2002. The Role of Metacognitive Knowledge in Learning, Teaching, and Assessing. *Theory into Practice* 41, 4.

71. Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, 322–332. https://doi.org/10.1145/199448.199526

72. Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 1410–1421. Retrieved from https://ieeexplore.ieee.org/document/9284117

73. John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (ACM '72), 717–740. https://doi.org/10.1145/800194.805852

74. Ori Roth and Yossi Gil. 2023. Fluent APIs in Functional Languages. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1: 876–901. https://doi.org/10.1145/3586057

75. Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Haskell '08), 37–48. https://doi.org/10.1145/1411286.1411292

76. Adam Scibior, Zoubin Ghahramani, and Andrew D. Gordon. 2015. Practical Probabilistic Programming with Monads. In

Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15), 165–176. https://doi.org/10.1145/2804302.2804317

77. Adam Scibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional Programming for Modular Bayesian Inference. *Proc. ACM Program. Lang.* 2, ICFP. https://doi.org/10.1145/3236778

78. Tim Sheard and Emir Pasalic. 2008. Meta-programming With Built-in Type Equality. *Electronic Notes in Theoretical Computer Science* 199: 49–65. https://doi.org/https://doi.org/10.1016/j.entcs.2007.11.012

79. Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. 2008. Virtual Machine Showdown: Stack versus Registers. *ACM Trans. Archit. Code Optim.* 4, 4: 2:1–2:36. https://doi.org/10.1145/1328195.1328197

80. Nischal Shrestha, Titus Barik, and Chris Parnin. 2021. Unravel: A Fluent Code Explorer for Data Wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (UIST '21), 198–207. https://doi.org/10.1145/3472749.3474744

81. Zachary Sullivan. 2019. *Exploring Codata: The Relation to Object-Orientation.* University of Oregon, Computer, Information Sciences Department. Retrieved from https://www.cs.uoregon.edu/Reports/DRP-201905-Sullivan.pdf

82. Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4: 423–436. https://doi.org/10.1017/S0956796808006758

83. David Thibodeau, Andrew Cave, and Brigitte Pientka. 2016. Indexed Codata Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (ICFP 2016), 351–363. https://doi.org/10.1145/2951913.2951929

84. Ben L. Titzer. 2022. A Fast In-Place Interpreter for WebAssembly. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2: 646–672. https://doi.org/10.1145/3563311

85. Phillip Merlin Uesbeck. 2019. On the Human Factors Impact of Polyglot Programming on Programmer Productivity. University of Nevada, Las Vegas. Retrieved from https://web.cs.unlv.edu/stefika/documents/MerlinDissertation.pdf

86. Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. *Proceedings of the ACM on Programming Languages* 9, POPL. https://doi.org/10.1145/3704837

87. Milica Vuković, Vladimir Vujović, Zorana Štaka, and Snježana Milinković. 2023. Domain-Specific Language for Modeling Fluent API. In *2023 15th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, 1–6. https://doi.org/10.1109/ECAI58194.2023.10194083

88. P. Wadler and S. Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '89), 60–76. https://doi.org/10.1145/75277.75283

89. Phil Wadler. 1998. The Expression Problem. Retrieved from https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

90. Philiip Wadler, Walid Taha, and David MacQueen. 1998. How to add laziness to a strict language without even being odd. In *SML'98, The SML workshop*. Retrieved from https://www.diva-portal.org/smash/get/diva2:413532/FULLTEXT01.pdf

91. Philip Wadler. 1989. Theorems for free!. In *Proceedings of the Fourth International Conference on Functional Programming*

Languages and Computer Architecture (FPCA '89), 347–359. https://doi.org/10.1145/99370.99404

92. Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '92), 1–14. https://doi.org/10.1145/143165.143169

93. Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '03), 224–235. https://doi.org/10.1145/604131.604150

94. Weixin Zhang, Cristina David, and Meng Wang. 2022. Decomposition Without Regret. Retrieved from https://arxiv.org/abs/2204.10411