Functional Programming Strategies

Functional Programming Strategies

Noel Welsh

Draft built on 2026-02-24

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:

```scala
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:

```scala
"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.

```scala
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`.

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

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

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

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

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

```scala
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

```scala
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

```scala
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.

```scala
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.

```scala
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.

```scala
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.

```scala
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.

```scala
// 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`.

```scala
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.

```scala
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's strategy is **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

```scala
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

```scala
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`.

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

```scala
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`.

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

```scala
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

```scala
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

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

In Scala 2 we write

```scala
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

```scala
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.

```scala
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

```scala
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

```scala
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`.

See the solution (Solution 0).

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

```scala
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.

```scala
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

```scala
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.

```scala
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:

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

Or, in the Scala 2 encoding:

```scala
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.

See the solution (Solution 1).

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

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

We know the structural recursion skeleton for `MyList` is

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

Implementing fold for `MyList` means defining a method

```scala
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).

```scala
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.

```scala
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

```scala
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.

See the solution (Solution 2).

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.

See the solution (Solution 3).

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

```scala
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`:

```scala
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.

```scala
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.

```scala
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.

```scala
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`

```scala
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`.

See the solution (Solution 4).

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.

See the solution (Solution 5).

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.

```scala
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

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

We can use tuples to avoid creating lots of names.

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

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

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

or just

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

Exercise: Identities

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

See the solution (Solution 6).

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

See the solution (Solution 7).

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.

```scala
final case class Person(name: String, permissions: Permissions)
enum Permissions {
  case User
  case Moderator
}
```

Written in mathematical notation, this is

$$\begin{aligned}\text{Person} &= \text{String} \times \text{Permissions} \\ \text{Permissions} &= \text{User} + \text{Moderator}\end{aligned}$$

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

```scala
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.

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

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.

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

```scala
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

```scala
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.

```scala
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.

```scala
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.

```scala
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.

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

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

```scala
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

```scala
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.

```scala
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`.

```scala
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.

```scala
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.

```scala
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.

```scala
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`.

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

We can check it works.

```scala
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.

```scala
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]
}
```

See the solution (Solution 8).

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

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

As usual, we should check it works.

```scala
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.

```scala
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.

```scala
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.

```scala
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.

```scala
randoms.take(5)
// res19: List[Double] = List(
//   0.5362908758546415,
//   0.18091326208566982,
//   0.8472997337174366,
//   0.6824013114786737,
//   0.6292251340151807
// )
randoms.take(5)
// res20: List[Double] = List(
//   0.786224366594783,
//   0.7760723959986657,
//   0.9631798514061348,
//   0.45347308835950506,
//   0.21424264934514292
// )
```

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(
//   0.3057894522265686,
//   0.3057894522265686,
//   0.3057894522265686,
//   0.3057894522265686,
//   0.3057894522265686
// )
randomsByNeed.take(5)
// res22: List[Double] = List(
//   0.3057894522265686,
//   0.3057894522265686,
//   0.3057894522265686,
//   0.3057894522265686,
//   0.3057894522265686
// )
```

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.

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

Then check it works.

```scala
randomsByNeed2.take(5)
// res23: List[Double] = List(
//   0.8770882725496386,
//   0.5098905632585569,
//   0.28132845799245376,
//   0.8668588360944173,
//   0.4568121280453893
// )
randomsByNeed2.take(5)
// res24: List[Double] = List(
//   0.8770882725496386,
//   0.5098905632585569,
//   0.28132845799245376,
//   0.8668588360944173,
//   0.4568121280453893
// )
```

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.)

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

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

```scala
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`.

```scala
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.

See the solution (Solution 9).

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.

```scala
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.

```scala
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.

```scala
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

```scala
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.

```scala
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.

```scala
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.

```scala
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`.

```scala
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.

See the solution (Solution 10).

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.)

See the solution (Solution 11).

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.

See the solution (Solution 12).

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.

```scala
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.

```scala
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.

```scala
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.

```scala
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.

```scala
{
  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:

```scala
// 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.

```scala
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.

```scala
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:

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

```scala
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.

```scala
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$stringWriter$@1bb3d2d8
```

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:

```scala
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:

```scala
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`.

```scala
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`.

See the solution (Solution 13).

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!
```

See the solution (Solution 14).

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.

See the solution (Solution 15).

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:

```scala
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`:

```scala
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`:

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

```scala
val circles: List[Circle] = List(Circle(5.0))
val shapes: List[Shape] = circles
```