The Science of Functional Programming

A Tutorial, with Examples in Scala

by Sergei Winitzki, Ph.D.

Draft version of November 5, 2019

Published by **lulu.com** in 2019

Published by **lulu.com**

ISBN: 978-0-359-76877-6

A *Transparent* copy of the source code for the book (LyX, LaTeX, graphics files, and build scripts), together with a full-color hyperlinked PDF file, is available at https://github.com/winitzki/sofp. The source code is also included as a "file attachment" `sofp-src.tar.bz2` within the PDF file `sofp.pdf`. To extract, run the command `‘pdftk sofp.pdf unpack_files output .‘` and then `‘tar jxvf sofp-src.tar.bz2‘`. See the file `README.md` for compilation instructions.

This pedagogical tutorial book presents the theoretical knowledge needed by practitioners of the functional programming paradigm. The main topics covered are: working with functional collections and recursion; the fundamental type and code constructions in functional programming; reasoning about types and code with the Curry-Howard correspondence; laws, structure theorems, and code derivation for functors, monads, and other important typeclasses; free type constructions; and recursive types. Detailed explanations are logically developed and accompanied by full derivations and worked examples tested in the Scala interpreter, as well as exercises. Readers should have a working knowledge of programming; e.g. be able to write code that reads a small text file and prints the number of words in it. Readers should also know school-level mathematics; for example, should be able to simplify the expressions $\frac{1}{x-2} - \frac{1}{x+2}$ and $\frac{d}{dx}\left((x+1)\,e^{-x}\right)$.

Contents

Preface

The goal of this book is to teach programmers how to reason mathematically about types and code, in a way that is directly relevant to software practice.

The material is presented at medium to advanced level. Readers will need to learn difficult concepts through prolonged mental concentration and effort. The book requires a certain amount of mathematical experience, at the level of familiarity with high-school algebra.

The first part is introductory and may be suitable for beginners in programming. Starting from Chapter 5, the material becomes unsuitable for beginners.

The presentation in this book is self-contained, defining and explaining all required notations, concepts, and Scala language features from scratch. The emphasis is on clarity and understandability of all examples, mathematical notions, derivations, and code. To achieve a clearer presentation of the material, the book uses some non-standard notations (Appendix A) and terminology (Appendix B).

The vision of this book is to explain the mathematical principles that guide the practice of functional programming – that is, help people to write code. Therefore, all mathematical developments in this book are motivated and justified by practical programming issues and are accompanied by code examples that illustrate their usage. For instance, the equational laws for standard typeclasses (functor, applicative, monad, etc.) are first motivated heuristically through code examples. Only then a set of mathematical equations is derived and the laws are formulated as equations.

Each new concept or technique is clafiried by solved examples and exercises. Answers to exercises are not provided, but it is verified that the exercises are doable and free of errors. More difficult examples and exercises are marked by an asterisk (*).

A software engineer needs to know only a few fragments of mathematical theory; namely, the fragments that answer questions arising in the practice of functional programming. So this book keeps theoretical material at the minimum; *ars longa, vita brevis*. (Chapter 18 gives more discussion of the scope of the required theory.) Mathematical generalizations are not pursued beyond proven practical relevance or immediate pedagogical usefulness. This reduces the required mathematical knowledge to first notions of category theory, type theory, and formal logic. Concepts such as functors or natural transformations arise organically from the practice of reasoning about code and are introduced without reference to category theory. This book does not mention "introduction/elimination rules", "strong normalization", "complete partial order domains", "adjoint functors", "pullbacks", or "topoi", because learning these concepts will not help a programmer write code.

This book is also not an introduction to current theoretical research in functional programming. Instead, the focus is on material known to be practically useful — including some rarely mentioned constructions, such as the "filterable functor" or "applicative contrafunctor" typeclasses, but excluding a number of theoretical developments that are still in the experimental phase.

All code examples are intended only for explanation and illustration, and are not optimized for performance or stack safety.

The author thanks Joseph Kim and Jim Kleck for going through the exercises and reporting some errors in earlier versions of this book.

Formatting conventions used in this book

- Text in boldface indicates a new concept or term that is being defined. Text in italics is a logical emphasis. Example:

  An **aggregation** is a function from a list of values to a *single* value.

- Sample Scala code is written inline using a small monospaced font, such as this: `val a = "xyz"`. Longer code examples are written in separate code blocks, which may also show the output from the Scala interpreter for certain lines of code:

```
val s = (1 to 10).toList

scala> s.product
res0: Int = 3628800
```

- In the introductory chapters, type expressions and code examples are written in the syntax of Scala. Starting from Chapters 4–5, the book introduces a new notation for types where e.g. the Scala type expression `((A, B)) => Option[A]` is written as $A \times B \to \mathbb{1} + A$. Also, a new notation for code is introduced and developed in Chapters 5–6 for easier reasoning about typeclass laws. For example, the functor composition law is written in the code notation as

$$f^{\uparrow L} \fatsemi g^{\uparrow L} = (f \fatsemi g)^{\uparrow L} \quad ,$$

  where L is a functor and $f^{:A \to B}$ and $g^{:B \to C}$ are arbitrary functions of the specified types. The notation $f^{\uparrow L}$ denotes the function f lifted to the functor L and replaces Scala's syntax `x.map(f)` where `x` is of type `L[A]`. The symbol $\fatsemi$ denotes the forward composition of functions (Scala's `andThen` method). Appendix A summarizes the notation conventions for types and code.

- Frequently used methods of standard typeclasses, such as `pure`, `flatMap`, `flatten`, `filter`, etc., are denoted by shorter words and are labeled by the type constructor they belong to. For instance, the methods `pure`, `flatten`, and `flatMap` for a monad M are denoted pu_M, ftn_M, and flm_M.

- Derivations of laws are written in a two-column format where the right column contains formulas in the code notation and the left column gives an explanation or indicates the property or law used to derive the expression at right. A green underline in an expression shows the parts to be rewritten using the law or equation indicated in the *next* line:

$$\begin{aligned} \text{expect to equal } \text{pu}_M : \quad & \underline{\text{pu}_M^{\uparrow \text{Id}}} \fatsemi \text{pu}_M \fatsemi \text{ftn}_M \\ \text{lifting to the identity functor} : \quad & = \text{pu}_M \fatsemi \underline{\text{pu}_M \fatsemi \text{ftn}_M} \\ \text{left identity law for } M : \quad & = \text{pu}_M \quad . \end{aligned}$$

  A green underline is sometimes also used at the *last* step of the derivation, to indicate the part of the expression that resulted from the most recent rewriting.

Part I

Beginner level

1 Mathematical formulas as code. I. Nameless functions

1.1 Translating mathematics into code

1.1.1 First examples

We begin by writing Scala code for some computational tasks.

Example 1.1.1.1: Factorial of 10 Find the product of integers from 1 to 10 (the **factorial** of 10).

First, we write a mathematical formula for the result:

$$\prod_{k=1}^{10} k \quad .$$

We can then write Scala code in a way that resembles this formula:

```
scala> (1 to 10).product
res0: Int = 3628800
```

The Scala interpreter indicates that the result is the value 3628800 of type `Int`. To define a name for this value, we use the "`val`" syntax:

```
scala> val fac10 = (1 to 10).product
fac10: Int = 3628800

scala> fac10 == 3628800
res1: Boolean = true
```

The code `(1 to 10).product` is an **expression**, which means that (1) the code can be evaluated (e.g. using the Scala interpreter) and yields a value, and (2) the code can be inserted as a part of a larger expression. For example, we could write

```
scala> 100 + (1 to 10).product + 100
res0: Int = 3629000
```

Example 1.1.1.2: Factorial as a function Define a function that takes an integer n and computes the factorial of n.

A mathematical formula for this function can be written as

$$f(n) = \prod_{k=1}^{n} k \quad .$$

The corresponding Scala code is

```
def f(n:Int) = (1 to n).product
```

In Scala's `def` syntax, we need to specify the type of a function's argument; in this case, we write `n:Int`. In the usual mathematical notation, types of arguments are either not written at all, or written separately from the formula:

$$f(n) = \prod_{k=1}^{n} k, \quad \forall n \in \mathbb{N} \quad .$$

This indicates that n must be from the set of non-negative integers (denoted by $\mathbb{N}$ in mathematics). This is similar to specifying the type `Int` in the Scala code. So, the argument's type in the code specifies the *domain* of a function.

Having defined the function `f`, we can now apply it to an integer argument:

```
scala> f(10)
res6: Int = 3628800
```

It is an error to apply `f` to a non-integer value, e.g. to a string:

```
scala> f("abc")
<console>:13: error: type mismatch;
 found   : String("abc")
 required: Int
       f("abc")
         ^
```

1.1.2 Nameless functions

The formula and the code, as written above, both involve *naming* the function as "f". Sometimes a function does not really need a name, – for instance, if the function is used only once. "Nameless" mathematical functions are denoted using the symbol $\mapsto$ (pronounced "mapped to") like this:

$$x \mapsto \text{(some formula)} \quad .$$

So the mathematical notation for the nameless factorial function is

$$n \mapsto \prod_{k=1}^{n} k \quad .$$

This reads as "a function that maps n to the product of all k where k goes from 1 to n". The Scala expression implementing this mathematical formula is

```
(n: Int) => (1 to n).product
```

This expression shows Scala's syntax for a **nameless** function. Here,

```
n: Int
```

is the function's **argument**, while

```
(1 to n).product
```

is the function's **body**. The arrow symbol `=>` separates the argument from the body.[1]

Functions in Scala (whether named or nameless) are treated as values, which means that we can also define a Scala value as

```
scala> val fac = (n: Int) => (1 to n).product
fac: Int => Int = <function1>
```

We see that the value `fac` has the type `Int => Int`, which means that the function takes an integer (`Int`) argument and returns an integer result value. What is the value of the function `fac` *itself*? As we have just seen, the Scala interpreter prints `<function1>` as the "value" of `fac`. An alternative Scala interpreter[2] called `ammonite` prints something like this,

```
scala@ val fac = (n: Int) => (1 to n).product
fac: Int => Int = ammonite.$sess.cmd0$$$Lambda$1675/2107543287@1e44b638
```

[1] This book uses the symbol $\to$ for symbolic calculations and `=>` for Scala code. Several programming languages, such as OCaml and Haskell, use the notation `->` for the function arrow.

[2] https://ammonite.io/

This seems to indicate some identifying number, or perhaps a memory location.

One may imagine that a "function value" represents a block of compiled code, — code that will actually run and evaluate the function's body when the function is applied to its argument.

Once defined, a function can be applied to an argument like this:

```
scala> fac(10)
res1: Int = 3628800
```

However, functions can be used without naming them. We can directly apply a nameless factorial function to an integer argument `10` instead of writing `fac(10)`:

```
scala> ((n: Int) => (1 to n).product)(10)
res2: Int = 3628800
```

One would not often write code like this because there is no advantage in creating a nameless function and then applying it right away to an argument. This is so because we can evaluate the expression

```
((n: Int) => (1 to n).product)(10)
```

by substituting `10` instead of `n` in the function body, which gives us

```
(1 to 10).product
```

If a nameless function uses the argument several times, for example

```
((n: Int) => n*n*n + n*n)(12345)
```

it is still better to substitute the argument and to eliminate the nameless function. We could write

```
12345*12345*12345 + 12345*12345
```

but, of course, it is better to avoid repeating the value `12345`. To achieve that, we may define `n` as a value in an **expression block** like this:

```
scala> { val n = 12345; n*n*n + n*n }
res3: Int = 322687002
```

Defined in this way, the value `n` is visible only within the expression block. Outside the block, another value named `n` could be defined independently of this `n`. For this reason, the definition of `n` is called a **locally scoped** definition.

Nameless functions are convenient when they are themselves arguments of other functions, as we will see next.

Example 1.1.2.1: prime numbers Define a function that takes an integer argument n and determines whether n is a prime number.

A simple mathematical formula for this function can be written as

$$\text{is\_prime}\,(n) = \forall k \in [2, n-1]\,.\, n \neq 0 \bmod k \quad . \tag{1.1}$$

This formula has two clearly separated parts: first, a range of integers from 2 to $n-1$, and second, a requirement that all these integers should satisfy a given condition, $n \neq 0 \bmod k$. Formula (1.1) is translated into Scala code as

```
def is_prime(n: Int) = (2 to n-1).forall(k => n % k != 0)
```

In this code, the two parts of the mathematical formula are implemented in a way that is closely similar to the mathematical notation, except for the arrow after k.

We can now apply the function `is_prime` to some integer values:

```
scala> is_prime(12)
res3: Boolean = false

scala> is_prime(13)
res4: Boolean = true
```

As we can see from the output above, the function `is_prime` returns a value of type `Boolean`. Therefore, the function `is_prime` has type `Int => Boolean`.

A function that returns a `Boolean` value is called a **predicate**.

In Scala, it is optional – but strongly recommended – to specify the return type of named functions. The required syntax looks like this,

```
def is_prime(n: Int): Boolean = (2 to n-1).forall(k => n % k != 0)
```

However, we do not need to specify the type `Int` for the argument `k` of the nameless function `k => n % k != 0`. The Scala compiler knows that `k` is going to iterate over the *integer* elements of the range `(2 to n-1)`, which effectively forces `k` to be of type `Int`.

1.1.3 Nameless functions and bound variables

The code for `is_prime` differs from the mathematical formula (1.1) in two ways.

One difference is that the interval $[2, n-1]$ is in front of `forall`. Another is that the Scala code uses a nameless function `(k => n % k != 0)`, while Eq. (1.1) does not seem to involve any functions.

To understand the first difference, we need to keep in mind that the Scala syntax such as `(2 to n-1).forall(k => ...)` means to apply a function called `forall` to *two* arguments: the first argument is the range `(2 to n-1)`, and the second argument is the nameless function `(k => ...)`. In Scala, the **infix** syntax `x.f(z)`, or equivalently `x f z`, means that a function `f` is applied to its *two* arguments, `x` and `z`. In the ordinary mathematical notation, this would be $f(x, z)$. Infix notation is often easier to read and is widely used, e.g. when we write $x + y$ rather than something like $plus(x, y)$.

A single-argument function could be also defined with infix notation, and then the syntax is `x.f`, as in the expression `(1 to n).product` we have seen before.

The infix methods `.product` and `.forall` are already provided in the Scala standard library, so it is natural to use them. If we want to avoid the infix syntax, we could define a function `for_all` with two arguments and write code like this,

```
for_all(2 to n-1, k => n % k != 0)
```

This would have brought the syntax somewhat closer to the formula (1.1).

However, there still remains the second difference: The symbol k is used as an *argument* of a nameless function `(k => n % k != 0)` in the Scala code, – while the mathematical formula

$$\forall k \in [2, n-1] \,.\, n \neq 0 \bmod k \tag{1.2}$$

does not seem to use any functions but defines the symbol k that goes over the range $[2, n-1]$. The variable k is then used for writing the predicate $n \neq 0 \bmod k$.

Let us investigate the role of k more closely. The mathematical variable k is actually defined *only inside* the expression "$\forall k$: ..." and makes no sense outside that expression. This becomes clear by looking at Eq. (1.1): The variable k is not present in the left-hand side and could not possibly be used there. The name "k" is defined only in the right-hand side, where it is first mentioned as the arbitrary element $k \in [2, n-1]$ and then used in the sub-expression "... mod k".

So, the mathematical notation in Eq. (1.2) says two things: First, we use the name k for integers from 2 to $n-1$. Second, for each of those k we evaluate the expression $n \neq 0 \bmod k$, which can be viewed as a certain given *function of* k that returns a `Boolean` value. Translating the mathematical notation into code, it is therefore natural to use the nameless function

$$k \mapsto n \neq 0 \bmod k$$

and to write Scala code applying this nameless function to each element of the range $[2, n-1]$ and checking that all result values be `true`:

```
(2 to n-1).forall(k => n % k != 0)
```

Just as the mathematical notation defines the variable k only in the right-hand side of Eq. (1.1), the argument `k` of the nameless Scala function `k => n % k != 0` is defined only within that function's body and cannot be used in any code outside the expression `n % k != 0`.

Variables that are defined only inside an expression and are invisible outside are called **bound variables**, or "variables bound in an expression". Variables that are used in an expression but are defined outside it are called **free variables**, or "variables occurring free in an expression". These concepts apply equally well to mathematical formulas and to Scala code. For example, in the mathematical expression $k \to n \neq 0 \bmod k$ (which is a nameless function), the variable k is bound (it is defined only within that expression) but the variable n is free (it is defined outside that expression).

The main difference between free and bound variables is that bound variables can be *locally renamed* at will, unlike free variables. To see this, consider that we could rename k to z and write instead of Eq. (1.1) an equivalent definition

$$\text{is\_prime}\,(n) = \forall z \in [2, n-1]\,.\, n \neq 0 \bmod z \quad ,$$

or in Scala code,

```
(2 to n-1).forall(z => n % z != 0)
```

The argument `z` in the nameless function `z => n % z != 0` may be renamed without changing the result of the entire program. No code outside that function needs to be changed after renaming `z`. But the value `n` is defined outside and cannot be renamed "locally" (i.e. only within the sub-expression). If we wanted to rename `n` in the sub-expression `z => n % z != 0`, we would also need to change every place in the code that defines and uses `n` *outside* that expression, or else the program would become incorrect.

Mathematical formulas use bound variables in various constructions such as $\forall k : p(k)$, $\exists k : p(k)$, $\sum_{k=a}^{b} f(k)$, $\int_0^1 k^2 dk$, $\lim_{n\to\infty} f(n)$, and $\text{argmax}_k f(k)$. When translating mathematical expressions into code, we need to recognize the presence of bound variables, which the mathematical notation does not make quite so explicit. For each bound variable, we need to create a nameless function whose argument is that variable, e.g. `k=>p(k)` or `k=>f(k)` for the examples just shown. Only then will our code correctly reproduce the behavior of bound variables in mathematical expressions.

As an example, the mathematical formula $\forall k \in [1, n] : p(k)$ has a bound variable k and is translated into Scala code as

```
(1 to n).forall(k => p(k))
```

At this point we can apply a simplification trick to this code. The nameless function $k \to p(k)$ does exactly the same thing as the (named) function p: It takes an argument, which we may call k, and returns $p(k)$. So, we can simplify the Scala code above to

```
(1 to n).forall(p)
```

The simplification of $x \to f(x)$ to just f is always possible for functions f of a single argument.[3]

1.2 Aggregating data from sequences

Consider the task of counting how many even numbers there are in a given list L of integers. For example, the list $[5, 6, 7, 8, 9]$ contains *two* even numbers: 6 and 8.

A mathematical formula for this task can be written like this,

$$\text{count\_even}\,(L) = \sum_{k \in L} \text{is\_even}\,(k) \quad ,$$

$$\text{is\_even}\,(k) = \begin{cases} 1 & \text{if } k = 0 \bmod 2 \\ 0 & \text{otherwise} \end{cases} \quad .$$

Here we defined a helper function `is_even` in order to write more easily a formula for `count_even`. In mathematics, complicated formulas are often split into simpler parts by defining helper expressions.

[3] Certain features of Scala allow programmers to write code that looks like `f(x)` but actually involves additional implicit or default arguments of the function `f`, or an implicit type conversion for its argument `x`. In those cases, replacing the code `x => f(x)` by `f` will fail to compile. But these complications do not arise when working with simple functions.

We can write the Scala code similarly. We first define the helper function `is_even`; the Scala code can be written in a style quite similar to the mathematical formula:

```
def is_even(k: Int): Int = (k % 2) match {
  case 0 => 1 // First, check if it is zero.
  case _ => 0 // The underscore matches everything else.
}
```

For such a simple computation, we could also write shorter code using a nameless function,

```
val is_even = (k: Int) => if (k % 2 == 0) 1 else 0
```

Given this function, we now need to translate into Scala code the expression $\sum_{k \in L} \text{is\_even}(k)$. We can represent the list L using the data type `List[Int]` from the Scala standard library.

To compute $\sum_{k \in L} \text{is\_even}(k)$, we must apply the function `is_even` to each element of the list L, which will produce a list of some (integer) results, and then we will need to add all those results together. It is convenient to perform these two steps separately. This can be done with the functions `.map` and `.sum`, defined in the Scala standard library as infix methods for the data type `List`.

The method `.sum` is similar to `.product` and is defined for any `List` of numerical types (`Int`, `Float`, `Double`, etc.). It computes the sum of all numbers in the list:

```
scala> List(1, 2, 3).sum
res0: Int = 6
```

The method `.map` needs more explanation. This method takes a *function* as its second argument, applies that function to each element of the list, and puts all the results into a *new* list, which is then returned as the result value:

```
scala> List(1, 2, 3).map(x => x*x + 100*x)
res1: List[Int] = List(101, 204, 309)
```

In this example, the argument of `.map` is the nameless function $x \mapsto x^2 + 100x$. This function will be used repeatedly by `.map` to transform each integer from `List(1, 2, 3)`, creating a new list as a result.

It is equally possible to define the transforming function separately, give it a name, and then use it as the argument to `.map`:

```
scala> def func1(x: Int): Int = x*x + 100*x
func1: (x: Int)Int

scala> List(1, 2, 3).map(func1)
res2: List[Int] = List(101, 204, 309)
```

Short functions are often defined inline, while longer functions are defined separately with a name.

An infix method, such as `.map`, can be also used with a "dotless" syntax:

```
scala> List(1, 2, 3) map func1
res3: List[Int] = List(101, 204, 309)
```

If the transforming function `func1` is used only once, and especially for a simple operation such as $x \mapsto x^2 + 100x$, it is easier to work with a nameless function.

We can now combine the methods `.map` and `.sum` to define `count_even`:

```
def count_even(s: List[Int]) = s.map(is_even).sum
```

This code can be also written using a nameless function instead of `is_even`:

```
def count_even(s: List[Int]): Int = s
    .map { k => if (k % 2 == 0) 1 else 0 }
    .sum
```

It is customary in Scala to use infix methods when chaining several operations. For instance `s.map(...).sum` means first apply `s.map(...)`, which returns a *new* list, and then apply `.sum` to that list. To make the code more readable, we put each of the chained methods on a new line.

To test this code, let us run it in the Scala interpreter. In order to let the interpreter work correctly

with code entered line by line, the dot character needs to be at the *end* of the line. (In a compiled code, the dots can be at the beginning of the lines since the compiler reads the entire code at once.)

```
scala> def count_even(s: List[Int]): Int = s .
         map { k => if (k % 2 == 0) 1 else 0 } .
         sum
count_even: (s: List[Int])Int

scala> count_even(List(1,2,3,4,5))
res0: Int = 2

scala> count_even( List(1,2,3,4,5).map(x => x * 2) )
res1: Int = 5
```

Note that the Scala interpreter prints the types differently for functions declared using `def`. It prints `(s: List[Int])Int` for a function of type `List[Int] => Int`.

1.3 Filtering and truncating a sequence

In addition to the methods `.sum`, `.product`, `.map`, `.forall` that we have already seen, the Scala standard library defines many other useful methods. We will now take a look at using the methods `.max`, `.min`, `.exists`, `.size`, `.filter`, and `.takeWhile`.

The methods `.max`, `.min`, and `.size` are self-explanatory:

```
scala> List(10, 20, 30).max
res2: Int = 30

scala> List(10, 20, 30).min
res3: Int = 10

scala> List(10, 20, 30).size
res4: Int = 3
```

The methods `.forall`, `.exists`, `.filter`, and `.takeWhile` require a predicate as an argument. The `.forall` method returns `true` if and only if the predicate returns `true` for all values in the list; the `.exists` method returns `true` if and only if the predicate holds (returns `true`) for at least one value in the list. These methods can be written as mathematical formulas like this:

$$\text{forall}\,(S, p) = \forall k \in S.\ p(k) = \text{true}$$
$$\text{exists}\,(S, p) = \exists k \in S.\ p(k) = \text{true}$$

However, there is no mathematical notation for operations such as "removing elements from a list", so we will focus on the Scala syntax for these functions.

The `.filter` method returns a list that contains only the values for which the predicate returns `true`:

```
scala> List(1, 2, 3, 4, 5).filter(k => k % 3 != 0)
res5: List[Int] = List(1, 2, 4, 5)
```

The `.takeWhile` method truncates a given list, returning a new list with the initial portion of values from the original list for which predicate keeps being `true`:

```
scala> List(1, 2, 3, 4, 5).takeWhile(k => k % 3 != 0)
res6: List[Int] = List(1, 2)
```

In all these cases, the predicate's argument `k` must be of the same type as the elements in the list. In the examples shown above, the elements are integers (i.e. the lists have type `List[Int]`), therefore `k` must be of type `Int`.

The methods `.max`, `.min`, `.sum`, and `.product` are defined on lists of *numeric types*, such as `Int`, `Double`, and `Long`. The other methods are defined on lists of all types.

Using these methods, we can solve many problems that involve transforming and aggregating data stored in lists (as well as in arrays, sets, or other similar data structures). In this context, a

transformation is a function taking a list of values and returning another list of values; examples of transformation functions are `.filter` and `.map`. An **aggregation** is a function taking a list of values and returning a *single* value; examples of aggregation functions are `.max` and `.sum`.

Writing programs by chaining together various methods of transformation and aggregation is known as programming in the **map/reduce style**.

1.4 Solved examples

1.4.1 Aggregation

Example 1.4.1.1 Improve the code for `is_prime` by limiting the search to $k^2 \leq n$:

$$\text{is\_prime}\,(n) = \forall k \in [2, n-1] \text{ such that } k^2 \leq n.\ n \neq 0 \bmod k \quad .$$

Solution: Use `.takeWhile` to truncate the initial list when $k^2 \leq n$ becomes false:

```
def is_prime(n: Int): Boolean =
  (2 to n-1)
    .takeWhile(k =>   k*k <= n)
    .forall(k =>    n % k != 0)
```

Example 1.4.1.2 Compute $\prod_{k\in[1,10]} |\sin(k+2)|$.
Solution

```
(1 to 10)
  .map(k => math.abs(math.sin(k + 2)))
  .product
```

Example 1.4.1.3 Compute $\sum_{k\in[1,10];\,\cos k>0} \sqrt{\cos k}$.
Solution

```
(1 to 10)
  .filter(k => math.cos(k) > 0)
  .map(k => math.sqrt(math.cos(k)))
  .sum
```

It is safe to compute $\sqrt{\cos k}$, because we have first filtered the list by keeping only values k for which $\cos k > 0$. Let us check that this is so:

```
scala> (1 to 10).toList.filter(k => math.cos(k) > 0).map(x => math.cos(x))
res0: List[Double] = List(0.5403023058681398, 0.28366218546322625, 0.9601702866503661,
    0.7539022543433046)
```

Example 1.4.1.4 Compute the average of a non-empty list of type `List[Double]`,

$$\text{average}\,(s) = \frac{1}{n}\sum_{i=0}^{n-1} s_i \quad .$$

Solution We need to divide the sum by the length of the list:

```
def average(s: List[Double]): Double = s.sum / s.size

scala> average(List(1.0, 2.0, 3.0))
res0: Double = 2.0
```

Example 1.4.1.5 Given n, compute the Wallis product[4] truncated up to $\frac{2n}{2n+1}$:

$$\text{wallis}\,(n) = \frac{2}{1}\frac{2}{3}\frac{4}{3}\frac{4}{5}\frac{6}{5}\frac{6}{7}\cdots\frac{2n}{2n+1} \quad .$$

[4] https://en.wikipedia.org/wiki/Wallis_product

Solution Define the helper function `wallis_frac(i)` that computes the i^{th} fraction. The method `.toDouble` converts integers to `Double` numbers.

```
def wallis_frac(i: Int): Double = (2*i).toDouble/(2*i - 1)*(2*i)/(2*i + 1)

def wallis(n: Int) = (1 to n).map(wallis_frac).product

scala> math.cos(wallis(10000))  // Should be close to 0.
res0: Double = 3.9267453954401036E-5

scala> math.cos(wallis(100000)) // Should be even closer to 0.
res1: Double = 3.926966362362075E-6
```

The limit of the Wallis product is $\frac{\pi}{2}$, so the cosine of `wallis(n)` tends to zero in the limit of large n.

Example 1.4.1.6 Check numerically that $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$. First, define a function of n that computes a partial sum of that series until $k = n$. Then compute the partial sum for a large value of n and compare with the limit value.

Solution

```
def euler_series(n: Int): Double = (1 to n).map(k => 1.0/k/k).sum

scala> euler_series(100000)
res0: Double = 1.6449240668982423

scala> val pi = 4*math.atan(1)
pi: Double = 3.141592653589793

scala> pi*pi/6
res1: Double = 1.6449340668482264
```

Example 1.4.1.7 Check numerically the infinite product formula

$$\prod_{k=1}^{\infty} \left(1 - \frac{x^2}{k^2}\right) = \frac{\sin \pi x}{\pi x} \quad .$$

Solution Compute this product up to $k = n$ for $x = 0.1$ with a large value of n, say $n = 10^5$, and compare with the right-hand side:

```
def sine_product(n: Int, x: Double): Double = (1 to n).map(k => 1.0 - x*x/k/k).product

scala> sine_product(n = 100000, x = 0.1) // Arguments may be named, for clarity.
res0: Double = 0.9836317414461351

scala> math.sin(pi*0.1)/pi/0.1
res1: Double = 0.9836316430834658
```

Example 1.4.1.8 Define a function p that takes a list of integers and a function `f: Int => Int`, and returns the largest value of $f(x)$ among all x in the list.

Solution

```
def p(s: List[Int], f: Int => Int): Int = s.map(f).max
```

Here is a test for this function:

```
scala> p(List(2, 3, 4, 5), x => 60 / x)
res0: Int = 30
```

1.4.2 Transformation

Example 1.4.2.1 Given a list of lists, `s: List[List[Int]]`, select the inner lists of size at least 3. The result must be again of type `List[List[Int]]`.

Mathematical notation	Scala code
$x \mapsto \sqrt{x^2+1}$	`x => math.sqrt(x*x + 1)`
list $[1, 2, ..., n]$	`(1 to n)`
list $[f(1), ..., f(n)]$	`(1 to n).map(k => f(k))`
$\sum_{k=1}^{n} k^2$	`(1 to n).map(k => k*k).sum`
$\prod_{k=1}^{n} f(k)$	`(1 to n).map(f).product`
$\forall k \in [1, ..., n].\ p(k)$ holds	`(1 to n).forall(k => p(k))`
$\exists k \in [1, ..., n].p(k)$ holds	`(1 to n).exists(k => p(k))`
$\sum_{k \in S \text{ such that } p(k) \text{ holds}} f(k)$	`s.filter(p).map(f).sum`

Table 1.1: Translating mathematics into code.

Solution To "select the inner lists" means to compute a *new* list containing only the desired inner lists. We use `.filter` on the outer list `s`. The predicate for the filter is a function that takes an inner list and returns `true` if the size of that list is at least 3. Write the predicate as a nameless function, `t => t.size >= 3`, where `t` is of type `List[Int]`:

```
def f(s: List[List[Int]]): List[List[Int]] = s.filter(t => t.size >= 3)

scala> f(List( List(1,2), List(1,2,3), List(1,2,3,4) ))
res0: List[List[Int]] = List(List(1, 2, 3), List(1, 2, 3, 4))
```

The Scala compiler deduces the type of `t` from the code; no other type would work since we apply `.filter` to a *list of lists* of integers.

Example 1.4.2.2 Find all integers $k \in [1, 10]$ such that there are at least three different integers j, where $1 \leq j \leq k$, each j satisfying the condition $j^2 > 2k$.

Solution

```
scala> (1 to 10).toList.filter(k => (1 to k).filter(j => j*j > 2*k).size >= 3)
res0: List[Int] = List(6, 7, 8, 9, 10)
```

The argument of the outer `.filter` is a nameless function that also uses a `.filter`. The inner expression (shown at left) computes the list of j's that satisfy the condition $j^2 > 2k$, and then compares the size of that list with 3. In this way, we impose the requirement that there should be at least 3 values of j. We can see how the Scala code closely follows the mathematical formulation of the task.

```
(1 to k).filter(j => j*j > 2*k).size >= 3
```

1.5 Summary

Functional programs are mathematical formulas translated into code. Table 1.1 shows how to implement some often used mathematical constructions in Scala.

What problems can one solve with this knowledge?

- Compute mathematical expressions involving sums, products, and quantifiers, based on integer ranges, such as $\sum_{k=1}^{n} f(k)$ etc.
- Transform and aggregate data from lists using `.map`, `.filter`, `.sum`, and other methods from the Scala standard library.

What are examples of problems that are *not* solvable with these tools?

- Example 1: Compute the smallest $n \geq 1$ such that

$$f(f(f(...f(0)...) > 1000 \quad ,$$

  where the given function f is applied n times.

- Example 2: Given a list s of numbers, compute the list r of running averages:

$$r_n = \frac{1}{n} \sum_{k=0}^{n-1} s_k \quad .$$

- Example 3: Perform binary search over a sorted list of integers.

These computations involve *mathematical induction*, which we have not yet learned to translate into code in the general case.

Library functions we have seen so far, such as `.map` and `.filter`, implement a restricted class of iterative operations on lists: namely, operations that process each element of a given list independently and accumulate results. For instance, when computing `s.map(f)`, the number of function applications is given by the size of the initial list. However, Example 1 requires applying a function f repeatedly until a given condition holds – that is, repeating for an *initially unknown* number of times. So it is impossible to write an expression containing `.map`, `.filter`, `.takeWhile`, etc., that solves Example 1. We could write the solution of Example 1 as a formula by using mathematical induction, but we have not yet seen how to implement that in Scala code.

Example 2 can be formulated as a definition of a new list r by induction,

$$r_0 = s_0 \quad ; \qquad r_i = s_i + r_{i-1} \text{ for } i = 1, 2, 3, ...$$

However, operations such as `.map` and `.filter` cannot compute r_i depending on the value of r_{i-1}.

Example 3 defines the search result by induction: the list is split in half, and search is performed by inductive hypothesis in the half that contains the required value. This computation requires an initially unknown number of steps.

Chapter 2 explains how to implement these tasks by translating mathematical induction into code using recursion.

1.6 Exercises

1.6.1 Aggregation

Exercise 1.6.1.1 Machin's formula[5] converges to π faster than Example 1.4.1.5:

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239} \quad ,$$
$$\arctan \frac{1}{n} = \frac{1}{n} - \frac{1}{3}\frac{1}{n^3} + \frac{1}{5}\frac{1}{n^5} - ... = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} n^{-2k-1} \quad .$$

Implement a function that computes the series for $\arctan \frac{1}{n}$ up to a given number of terms, and compute an approximation of π using this formula. Show that about 12 terms of the series are already sufficient for a full-precision `Double` approximation of π.

[5] http://turner.faculty.swau.edu/mathematics/materialslibrary/pi/machin.html

Exercise 1.6.1.2 Using the function `is_prime`, check numerically the Euler product formula[6] for the Riemann zeta function $\zeta(4)$; it is known[7] that $\zeta(4) = \frac{\pi^4}{90}$:

$$\prod_{k \geq 2;\ k \text{ is prime}} \frac{1}{1 - p^{-4}} = \frac{\pi^4}{90} \quad .$$

1.6.2 Transformation

Exercise 1.6.2.1 Define a function `add_20` of type `List[List[Int]] => List[List[Int]]` that adds 20 to every element of every inner list. A sample test:

```
scala> add_20( List( List(1), List(2, 3) ) )
res0: List[List[Int]] = List(List(21), List(22, 23))
```

Exercise 1.6.2.2 An integer n is called a "3-factor" if it is divisible by only three different integers j such that $2 \leq j < n$. Compute the set of all "3-factor" integers n among $n \in [1, ..., 1000]$.

Exercise 1.6.2.3 Given a function `f: Int => Boolean`, an integer n is called a "3-f" if there are only three different integers $j \in [1, ..., n]$ such that $f(j)$ returns `true`. Define a function that takes f as an argument and returns a sequence of all "3-f" integers among $n \in [1, ..., 1000]$. What is the type of that function? Implement Exercise 1.6.2.2 using that function.

Exercise 1.6.2.4 Define a function `see100` of type `List[List[Int]] => List[List[Int]]` that selects only those inner lists whose largest value is at least 100. Test with:

```
scala> see100( List( List(0, 1, 100), List(60, 80), List(1000) ) )
res0: List[List[Int]] = List(List(0, 1, 100), List(1000))
```

Exercise 1.6.2.5 Define a function of type `List[Double] => List[Double]` that "normalizes" the list: finds the element having the largest absolute value and, if that value is nonzero, divides all elements by that factor and returns a new list; otherwise returns the original list.

1.7 Discussion

1.7.1 Functional programming as a paradigm

Functional programming (FP) is a **paradigm** of programming, – that is, an approach that guides programmers to write code in specific ways, for a wide range of programming tasks.

The main principle of FP is to write code *as a mathematical expression or formula*. This approach allows programmers to derive code through logical reasoning rather than through guessing, – similarly to how books on mathematics reason about mathematical formulas and derive results systematically, without guessing or "debugging." Similarly to mathematicians and scientists who reason about formulas, functional programmers can *reason about code* systematically and logically, based on rigorous principles. This is possible only because code is written as a mathematical formula.

Mathematical intuition is backed by the vast experience accumulated while working with data over millennia of human history. It took centuries to invent flexible and powerful notation such as $\sum_{k \in S} p(k)$ and to develop the corresponding rules of calculation. Functional programmers are fortunate to have these reasoning tools at their disposal.

As we have seen, the Scala code for certain computational tasks corresponds quite closely to mathematical formulas. (Scala conventions and syntax, of course, require programmers to spell out certain things that are omitted in the mathematical notation.) Just as in mathematics, large code expressions may be split into smaller expressions when needed. Expressions can be easily reused,

[6] https://en.wikipedia.org/wiki/Proof_of_the_Euler_product_formula_for_the_Riemann_zeta_function
[7] https://tinyurl.com/yxey4tsd

flexibly composed together, and written independently from each other. The FP community has developed a toolkit of functions (such as `.map`, `.filter`, etc.) that proved to be especially useful in real-life programming, although many of them are not standard in mathematical literature.

Mastering FP involves practicing to reason about programs as formulas, building up the specific kind of applied mathematical intuition, familiarizing oneself with concepts adapted to a programmer's needs, and learning how to translate the mathematics into code in various cases. The FP community has discovered a number of specific design patterns founded on mathematical principles but driven by practical necessities of programming (rather than by the needs of academic mathematics). This book explains the required mathematical design patterns in detail, developing them through intuition and examples of Scala code.

1.7.2 Functional programming languages

It is possible to apply the FP paradigm while writing code in any programming language. However, some languages lack certain features that make FP techniques much easier to use in practice. For example, in a language such as Python or Ruby, one can productively use only a limited number of FP idioms, such as the map/reduce operations. More advanced FP constructions are impractical in these languages because the required code becomes too hard to read and to write without errors, which negates the advantages of rigorous reasoning about functional programs.

Some programming languages, such as Haskell and OCaml, were designed specifically for advanced use in the FP paradigm. Other languages, such as ML, F#, Scala, Swift, Elm, and PureScript, have different design goals but still support enough FP features to be considered FP languages. This book uses Scala, but the same constructions may be implemented in other FP languages in a similar way. At the level of detail needed in this book, the differences between languages such as ML, OCaml, Haskell, F#, Scala, Swift, Elm, or PureScript do not play a significant role.

1.7.3 The mathematical meaning of "variables"

The usage of variables in functional programming is similar to how mathematical literature uses variables. In mathematics, **variables** are used first of all as *arguments* of functions; e.g. the formula

$$f(x) = x^2 + x$$

contains the variable x and defines a function f that takes x as its argument (to be definite, let us assume that x is an integer) and computes the value $x^2 + x$. The body of the function is the expression $x^2 + x$.

Mathematics has the convention that a variable, such as x, does not change its value within a formula. Indeed, there is no mathematical notation even to talk about "changing" the value of x *inside* the formula $x^2 + x$. It would be quite confusing if a mathematics textbook said "before adding the last x in the formula $x^2 + x$, we change that x by adding 4 to it". If the "last x" in $x^2 + x$ needs to have a 4 added to it, a mathematics textbook will just write the formula $x^2 + x + 4$.

Arguments of nameless functions are also immutable. Consider, for example,

$$f(n) = \sum_{k=0}^{n} (k^2 + k) \quad .$$

Here, n is the argument of the function f, while k is the argument of the nameless function $k \mapsto k^2 + k$. Neither n nor k can be "modified" in any sense within the expressions where they are used. The symbols k and n stand for some integer values, and these values are immutable. Indeed, it is meaningless to say that we "modified the integer 4". In the same way, we cannot modify k.

So, a variable in mathematics remains constant *within the expression* where it is defined; in that expression, a variable is essentially a "named constant". Of course, a function f can be applied to different values x, to compute a different result $f(x)$ each time. However, a given value of x will remain unmodified within the body of the function f while $f(x)$ is being computed.

Functional programming adopts this convention from mathematics: variables are immutable named constants. (Scala also has *mutable* variables, but we will not consider them in this book.)

In Scala, function arguments are immutable within the function body:

```
def f(x: Int) = x * x + x // Cannot modify `x` here.
```

The *type* of each mathematical variable (such as integer, vector, etc.) is also fixed. Each variable is a value from a specific set (e.g. the set of all integers, the set of all vectors, etc.). Mathematical formulas such as $x^2 + x$ do not express any "checking" that x is indeed an integer and not, say, a vector, in the middle of evaluating $x^2 + x$. The types of all variables are checked in advance.

Functional programming adopts the same view: Each argument of each function must have a **type** that represents *the set of possible allowed values* for that function argument. The programming language's compiler will automatically check the types of all arguments *before* the program runs. A program that calls functions on arguments of incorrect types will not compile.

The second usage of **variables** in mathematics is to denote expressions that will be reused. For example, one writes: let $z = \frac{x-y}{x+y}$ and now compute $\cos z + \cos 2z + \cos 3z$. Again, the variable z remains immutable, and its type remains fixed.

In Scala, this construction (defining an expression to be reused later) is written with the "`val`" syntax. Each variable defined using "`val`" is a named constant, and its type and value are fixed at the time of definition. Type annotations for "`val`"s are optional in Scala: for instance we could write

```
val x: Int = 123
```

or we could omit the type annotation `:Int` and write more concisely

```
val x = 123
```

because it is clear that this `x` is an integer. However, it is often helpful to write out types. If we do so, the compiler will check that the types match correctly and give an error message whenever wrong types are used. For example, a type error is detected when using a `String` instead of an `Int`:

```
scala> val x: Int = "123"
<console>:11: error: type mismatch;
 found   : String("123")
 required: Int
       val x: Int = "123"
                    ^
```

1.7.4 Iteration without loops

Another distinctive feature of the FP paradigm is handling of iteration without writing loops.

Iterative computations are ubiquitous in mathematics. As an example, consider the formula for the standard deviation estimated from a sample,

$$\sigma(s) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\sum_{j=1}^{n} s_i s_j - \frac{1}{n\,(n-1)}\left(\sum_{i=1}^{n} s_i\right)^2} \quad .$$

These expressions are computed by iterating over values of i and j. And yet, no mathematics textbook uses "loops" or says "now repeat this formula ten times". Indeed, it would be pointless to evaluate a formula such as $x^2 + x$ ten times in a loop, or to "repeat" an equation such as

$$(x-1)\,(x^2 + x + 1) = x^3 - 1 \quad .$$

Instead of loops, mathematicians write *expressions* such as $\sum_{i=1}^{n} s_i$, where symbols such as $\sum_{i=1}^{n}$ or $\prod_{i=1}^{n}$ denote iterative computations. Such computations are defined using mathematical induction. The functional programming paradigm has developed rich tools for translating mathematical induction into code. In this chapter, we have seen methods such as `.map`, `.filter`, and `.sum`, which

implement certain kinds of iterative computations. These and other operations can be combined in very flexible ways, which allows programmers to write iterative code without loops.

The programmer can avoid writing loops because the iteration is delegated to the library functions `.map`, `.filter`, `.sum`, and so on. It is the job of the library and the compiler to translate these functions into machine code. The machine code most likely *will* contain loops; but the functional programmer does not need to see that code or to reason about it.

1.7.5 Nameless functions in mathematical notation

Functions in mathematics are mappings from one set to another. A function does not necessarily *need* a name; the mapping just needs to be defined. However, nameless functions have not been widely used in the conventional mathematical notation. It turns out that nameless functions are important in functional programming because, in particular, they allow programmers to write code with a straightforward and consistent syntax.

Nameless functions contain bound variables that are invisible outside the function's scope. This property is directly reflected by the prevailing mathematical conventions. Compare the formulas

$$f(x) = \int_0^x \frac{dx}{1+x} \quad ; \quad f(x) = \int_0^x \frac{dz}{1+z} \quad .$$

The mathematical convention is that one may rename the integration variable at will, and so these formulas define the same function f.

In programming, the only situation when a variable "may be renamed at will" is when the variable represents an argument of a function. It follows that the notations $\frac{dx}{1+x}$ and $\frac{dz}{1+z}$ correspond to a nameless function whose argument was renamed from x to z. In FP notation, this nameless function would be denoted as $z \to \frac{1}{1+z}$, and the integral rewritten as code such as

```
integration(0, x, { z => 1.0 / (1 + z) } )
```

Now consider the traditional mathematical notations for summation, e.g.

$$\sum_{k=0}^{x} \frac{1}{1+k} \quad .$$

In that sum, the bound variable k is introduced under the $\sum$ symbol; but in integrals, the bound variable follows the special symbol "d". This notational inconsistency could be removed if we were to use nameless functions explicitly, for example:

$$\text{denote summation by } \sum_0^x \left(k \mapsto \frac{1}{1+k} \right) \text{ instead of } \sum_{k=0}^{x} \frac{1}{1+k} \quad ,$$

$$\text{denote integration by } \int_0^x \left(z \mapsto \frac{1}{1+z} \right) \text{ instead of } \int_0^x \frac{dz}{1+z} \quad .$$

In this notation, the new summation symbol $\sum_0^x$ does not mention the name "k" but takes a function as an argument. Similarly, the new integration symbol $\int_0^x$ does not mention "z" and does not use the special symbol "d" but now takes a function as an argument. Written in this way, the operations of summation and integration become *functions* that take functions as arguments. The above summation may be written in a consistent and straightforward manner as a Scala function:

```
summation(0, x, { y => 1.0 / (1 + y) } )
```

We could implement `summation(a, b, g)` as

```
def summation(a: Int, b: Int, g: Int => Double): Double = (a to b).map(g).sum

scala> summation(1, 10, x => math.sqrt(x))
res0: Double = 22.4682781862041
```

Integration requires longer code since the formulas are more complicated. Simpson's rule[8] is an approximate algorithm for numerical integration that can be defined by the formula

$$\text{integration}\,(a,b,g,\varepsilon) = \frac{\delta}{3}(g(a)+g(b)+4s_1+2s_2) \quad ,$$
$$\text{where} \quad n = 2\left\lfloor\frac{b-a}{\varepsilon}\right\rfloor, \quad \delta_x = \frac{b-a}{n} \quad ,$$
$$s_1 = \sum_{k=1,3,\ldots,n-1} g(a+k\delta_x) \quad , \qquad s_2 = \sum_{k=2,4,\ldots,n-2} g(a+k\delta_x) \quad .$$

A straightforward line-by-line translation of this formula into Scala is

```
def integration(a: Double, b: Double, g: Double => Double, eps: Double): Double = {
  // First, we define some helper values and functions that replace
  // the definitions "where n = ..." in the mathematical formula.
   val n: Int = 2 * ((b - a) / eps).toInt
   val delta_x = (b - a) / n
   val s1 = (1 to (n - 1) by 2).map { k => g(a + k * delta_x) }.sum
   val s2 = (2 to (n - 2) by 2).map { k => g(a + k * delta_x) }.sum
   // Now we write the expression for the final result.
   delta_x / 3 * (g(a) + g(b) + 4 * s1 + 2 * s2)
}

scala> integration(0, 5, x => x*x*x*x, eps = 0.01)     // The exact answer is 625.
res0: Double = 625.0000000004167

scala> integration(0, 7, x => x*x*x*x*x*x, eps = 0.01) // The exact answer is 117649.
res1: Double = 117649.00000014296
```

The entire code is one large *expression*, with a few sub-expressions (`s1`, `s2`, etc.) defined for convenience in the **local scope** of the function. In other words, this code is written in the FP paradigm.

1.7.6 Named and nameless expressions and their uses

It is a significant advantage if a programming language supports unnamed (or "nameless") expressions. To see this, consider a familiar situation where we take the absence of names for granted.

In today's programming languages, we may directly write expressions such as `(x+123)*y/(4+x)`. Note that the entire expression does not need to have a name. Parts of that expression (e.g. the sub-expressions `x+123` or `4+x`) also do not have separate names. It would be inconvenient if we *needed* to assign a name to each sub-expression. The code for `(x+123)*y/(4+x)` would then look like this:

```
{
  val r0 = 123
  val r1 = x + r0
  val r2 = r1 * y
  val r3 = 4
  val r4 = r3 + x
  val r5 = r2 / r4     // Do we still remember what ‘r2‘ means?
  r5
}
```

This style of programming resembles assembly languages, where *every* sub-expression — that is, every step of every calculation, — must be named separately (and, in the assembly languages, assigned a memory address or a CPU register).

Programmers become more productive when their programming language supports nameless expressions. This is also common practice in mathematics; names are assigned when needed, but most expressions remain nameless.

It is similarly useful if data structures can be created without names. For instance, a **dictionary** (also called a "hashmap") is created in Scala as

[8] https://en.wikipedia.org/wiki/Simpson%27s_rule

```
Map("a" -> 1, "b" -> 2, "c" -> 3)
```

This code is a nameless expression whose value is a dictionary. In programming languages that do not have such a construction, programmers have to write repetitive code that creates an initially empty dictionary and then fills it step by step with values:

```
// Scala code creating a dictionary:
Map("a" -> 1, "b" -> 2, "c" -> 3)

/* Shortest Java code for the same:
new HashMap<String, Integer>() {{
   put("a", 1);
   put("b", 2);
   put("c", 3);
 }}; */
```

Nameless functions are useful for the same reason as nameless values of other types: they allow us to build larger programs from simpler parts in a uniform way.

1.7.7 Historical perspective on usage of nameless functions

Nameless functions were first used in 1936 in a theoretical programming language called "λ-calculus". In that language,[9] all functions are nameless and have a single argument. The letter λ is a syntax separator denoting function arguments in nameless functions. For example, the nameless function $x \mapsto x + 1$ could be written as $\lambda x.add\ x\ 1$ in λ-calculus, if it had a function add for adding integers (which it does not).

In most programming languages that were in use until around 1990, all functions required names. But by 2015, most languages added support for nameless functions, because programming in the map/reduce style (which invites frequent use of nameless functions) turned out to be immensely productive. Table 1.2 shows the year when nameless functions were introduced in each language.

Language	Year	Code for $k \mapsto k + 1$
λ-calculus	1936	$\lambda k.\ add\ k\ 1$
typed λ-calculus	1940	$\lambda k : int.\ add\ k\ 1$
LISP	1958	`(lambda (k) (+ k 1))`
Standard ML	1973	`fn (k:int) => k + 1`
Caml	1985	`fun k -> k + 1`
Haskell	1990	`\ k -> k + 1`
Oz	1991	`fun {$ K} K + 1`
R	1993	`function(k) k + 1`
Python 1.0	1994	`lambda k: k + 1`
JavaScript	1995	`function(k) { return k + 1; }`
Mercury	1995	`func(K) = K + 1`
Ruby	1995	`lambda { |k| k + 1 }`
Lua 3.1	1998	`function(k) return k + 1 end`
Scala	2003	`(k:Int) => k + 1`
F#	2005	`fun (k:int) -> k + 1`
C# 3.0	2007	`delegate(int k) { return k + 1; }`
Clojure	2009	`fn [k] (+ k 1)`
C++ 11	2011	`[] (int k) { return k + 1; }`
Go	2012	`func(k int) { return k + 1 }`
Julia	2012	`function(k::Int) k + 1 end`
Kotlin	2012	`{ k:Int -> k + 1 }`
Swift	2014	`{ (k:int) -> int in return k + 1 }`
Java 8	2014	`(int k) -> k + 1`
Rust	2015	`|k:i32| k + 1`

Table 1.2: Nameless functions in programming languages.

What this book calls a "nameless function" is also called anonymous function, function expression, function literal, closure, lambda function, lambda expression, or just a "lambda".

[9] Although called a "calculus," it is a (drastically simplified) programming language. It has nothing to do with "calculus" as known in mathematics, such as differential or integral calculus. Also, the letter λ has no particular significance; it plays a purely syntactic role in the λ-calculus. Practitioners of functional programming usually do not need to study any λ-calculus. All practically relevant knowledge related to λ-calculus is explained in Chapter 4 of this book.

2 Mathematical formulas as code. II. Mathematical induction

We will now study more flexible ways of working with data collections in the functional programming paradigm. The Scala standard library has methods for performing general iterative computations, that is, computations defined by induction. Translating mathematical induction into code is the focus of this chapter.

First, we need to become fluent in using tuple types with Scala collections.

2.1 Tuple types

2.1.1 Examples of using tuples

Many standard library methods in Scala work with tuple types. A simple example of a tuple is a *pair* of values, – such as, a pair of an integer and a string. The Scala syntax for this type of pair is

```
val a: (Int, String) = (123, "xyz")
```

The type expression `(Int, String)` denotes the type of this pair.

A **triple** is defined in Scala like this:

```
val b: (Boolean, Int, Int) = (true, 3, 4)
```

Pairs and triples are examples of tuples. A **tuple** can contain any number of values, which may be called **parts** of a tuple (they are also called **fields** of a tuple). The parts of a tuple can have different types, but the type of each part is fixed once and for all. Also, the number of parts in a tuple is fixed. It is a **type error** to use incorrect types in a tuple, or an incorrect number of parts of a tuple:

```
scala> val bad: (Int, String) = (1,2)
<console>:11: error: type mismatch;
 found   : Int(2)
 required: String
       val bad: (Int, String) = (1,2)
                                  ^
scala> val bad: (Int, String) = (1,"a",3)
<console>:11: error: type mismatch;
 found   : (Int, String, Int)
 required: (Int, String)
       val bad: (Int, String) = (1,"a",3)
                                ^
```

Parts of a tuple can be accessed by number, starting from 1. The Scala syntax for **tuple accessor** methods looks like `._1`, for example:

```
scala> val a = (123, "xyz")
a: (Int, String) = (123,xyz)

scala> a._1
res0: Int = 123

scala> a._2
res1: String = xyz
```

It is a type error to access a tuple part that does not exist:

```
scala> a._0
<console>:13: error: value _0 is not a member of (Int, String)
       a._0
         ^

scala> a._5
<console>:13: error: value _5 is not a member of (Int, String)
       a._5
         ^
```

Type errors are detected at compile time, before any computations begin.

Tuples can be **nested**: any part of a tuple can be itself a tuple:

```
scala> val c: (Boolean, (String, Int), Boolean) = (true, ("abc", 3), false)
c: (Boolean, (String, Int), Boolean) = (true,(abc,3),false)

scala> c._1
res0: Boolean = true

scala> c._2
res1: (String, Int) = (abc,3)
```

To define functions whose arguments are tuples, we could use the tuple accessors. An example of such a function is

```
def f(p: (Boolean, Int), q: Int): Boolean = p._1 && (p._2 > q)
```

The first argument, `p`, of this function, has a tuple type. The function body uses accessor methods (`._1` and `._2`) to compute the result value. Note that the second part of the tuple `p` is of type `Int`, so it is valid to compare it with an integer `q`. It would be a type error to compare the *tuple* `p` with an *integer* using the expression `p > q`. It would be also a type error to apply the function `f` to an argument `p` that has a wrong type, e.g. the type `(Int, Int)` instead of `(Boolean, Int)`.

2.1.2 Pattern matching for tuples

Instead of using accessor methods when working with tuples, it is often convenient to use **pattern matching**. Pattern matching occurs in two situations in Scala:

- destructuring definition: `val` *pattern* `= ...`
- `case` expression: `case` *pattern* `=> ...`

```
scala> val g = (1, 2, 3)
g: (Int, Int, Int) = (1,2,3)

scala> val (x, y, z) = g
x: Int = 1
y: Int = 2
z: Int = 3
```

An example of a **destructuring definition** is shown at left. The value `g` is a tuple of three integers. After defining `g`, we define the three variables `x`, `y`, `z` *at once* in a single `val` definition. We imagine that this definition "destructures" the data structure contained in `g` and decomposes it into three parts, then assigns the names `x`, `y`, `z` to these parts. The types of the new values are also assigned automatically.

In the example above, the left-hand side of the destructuring definition contains a tuple pattern `(x, y, z)` that looks like a tuple, except that its parts are names `x`, `y`, `z` that are so far *undefined*. These names are called **pattern variables**. The destructuring definition checks whether the structure of the value of `g` "matches" the three pattern variables. (If `g` does not contain a tuple with exactly three parts, the definition will fail.) This computation is called **pattern matching**.

Pattern matching is often used for working with tuples. The expression `{case (a, b, c) => ...}` called a **case expression** (shown at left) performs pattern matching on its argument. The pattern matching will

```
scala> (1, 2, 3) match { case (a, b, c) => a + b + c }
res0: Int = 6
```

"destructure" (i.e. decompose) a tuple and try to match it to the given pattern `(a, b, c)`. In this pattern, `a`, `b`, `c` are as yet undefined new variables, – that is, they are pattern variables. If the pattern matching succeeds, the pattern variables `a`, `b`, `c` are assigned their values, and the function body can proceed to perform its computation. In this example, the pattern variables `a`, `b`, `c` will be assigned values 1, 2, and 3, and so the expression evaluates to 6.

Pattern matching is especially convenient for nested tuples. Here is an example where a nested tuple `p` is destructured by pattern matching:

```
def t1(p: (Int, (String, Int))): String = p match {
  case (x, (str, y)) => str + (x + y).toString
}

scala> t((10, ("result is ", 2)))
res0: String = result is 12
```

The type structure of the argument `(Int, (String, Int))` is visually repeated in the pattern `(x, (str, y))`, making it clear that `x` and `y` become integers and `str` becomes a string after pattern matching.

If we rewrite the code of `t1` using the tuple accessor methods instead of pattern matching, the code will look like this:

```
def t2(p: (Int, (String, Int))): String = p._2._1 + (p._1 + p._2._2).toString
```

This code is shorter but harder to read. For example, it is not immediately clear that `p._2._1` is a string. It is also harder to modify this code: Suppose we want to change the type of the tuple `p` to `((Int, String), Int)`. Then the new code is

```
def t3(p: ((Int, String), Int)): String = p._1._2 + (p._1._1 + p._2).toString
```

It takes time to verify, by going through every accessor method, that the function `t3` computes the same expression as `t2`. In contrast, the code is changed easily when using the pattern matching expression instead of the accessor methods:

```
def t4(p: ((Int, String), Int)): String = p match {
  case ((x, str), y) => str + (x + y).toString
}
```

The only change in the function body, compared to `t1`, is in the pattern matcher. So it is visually clear that `t4` computes the same expression as `t1`.

Sometimes we do not need some of the tuple parts in a pattern match. The following syntax is used to make this intention clear:

```
scala> val (x, _, _, z) = ("abc", 123, false, true)
x: String = abc
z: Boolean = true
```

The underscore symbol _ denotes the parts of the pattern that we want to ignore. The underscore will always match any value regardless of its type.

A feature of Scala is a short syntax for functions such as `{case (x, y) => y}` that extract elements from tuples. The shorter syntax is `(t => t._2)` or even shorter, `(_._2)`, as illustrated here:

```
scala> val p: ((Int, Int )) => Int = { case (x, y) => y }
p: ((Int, Int)) => Int = <function1>

scala> p((1, 2))
res0: Int = 2

scala> val q: ((Int, Int )) => Int = (t => t._2)
q: ((Int, Int)) => Int = <function1>

scala> q((1, 2))
res1: Int = 2

scala> Seq( (1,10), (2,20), (3,30) ).map(_._2)
res2: Seq[Int] = List(10, 20, 30)
```

2.1.3 Using tuples with collections

Tuples can be combined with any other types without restrictions. For instance, we can define a tuple of functions,

```
val q: (Int => Int, Int => Int) = (x => x + 1, x => x - 1)
```

We can create a list of tuples,

```
val r: List[(String, Int)] = List(("apples", 3), ("oranges", 2), ("pears", 0))
```

We could define a tuple of lists of tuples of functions, or any other combination.

Here is an example of using the standard method `.map` to transform a list of tuples. The argument of `.map` must be a function taking a tuple as its argument. It is convenient to use pattern matching for writing such functions:

```
scala> val basket: List[(String, Int)] = List(("apples", 3), ("pears", 2), ("lemons", 0))
basket: List[(String, Int)] = List((apples,3), (pears,2), (lemons,0))

scala> basket.map { case (fruit, count) => count * 2 }
res0: List[Int] = List(6, 4, 0)

scala> basket.map { case (fruit, count) => count * 2 }.sum
res1: Int = 10
```

In this way, we can use the standard methods such as `.map`, `.filter`, `.max`, `.sum` to manipulate sequences of tuples. The names of the pattern variables "`fruit`", "`count`" are chosen to help us remember the meaning of the parts of tuples.

We can easily transform a list of tuples into a list of values of a different type:

```
scala> basket.map { case (fruit, count) =>
  val isAcidic = (fruit == "lemons")
  (fruit, isAcidic)
}
res2: List[(String, Boolean)] = List((apples,false), (pears,false), (lemons,true))
```

In the Scala syntax, a nameless function written with braces `{ ... }` can define local values in its body. The return value of the function is the last expression written in the function body. In this example, the return value of the nameless function is the tuple `(fruit, isAcidic)`.

2.1.4 Treating dictionaries (Scala's `Map`s) as collections

In the Scala standard library, tuples are frequently used as types of intermediate values. For instance, tuples are used when iterating over dictionaries. The Scala type `Map[K,V]` represents a dictionary with keys of type `K` and values of type `V`. Here `K` and `V` are **type parameters**. Type parameters represent unknown types that will be chosen later, when working with values having specific types.

In order to create a dictionary with given keys and values, we can write

```
Map(("apples", 3), ("oranges", 2), ("pears", 0))
```

The same result is obtained by first creating a sequence of key/value *pairs* and then converting that sequence into a dictionary via the method `.toMap`:

```
List(("apples", 3), ("oranges", 2), ("pears", 0)).toMap
```

The same method works for other collection types such as `Seq`, `Vector`, `Stream`, and `Array`.

The Scala library defines a special infix syntax for pairs via the arrow symbol `->`. The expression `x -> y` is equivalent to the pair `(x, y)`:

```
scala> "apples" -> 3
res0: (String, Int) = (apples,3)
```

With this syntax, the code for creating a dictionary is easier to read:

```
Map("apples" -> 3, "oranges" -> 2, "pears" -> 0)
```

The method `.toSeq` converts a dictionary into a sequence of pairs:

```
scala> Map("apples" -> 3, "oranges" -> 2, "pears" -> 0).toSeq
res20: Seq[(String, Int)] = ArrayBuffer((apples,3), (oranges,2), (pears,0))
```

The `ArrayBuffer` is one of the many list-like data structures in the Scala library. All these data structures are subtypes of the common "sequence" type `Seq`. The methods defined in the Scala standard library sometimes return different implementations of the `Seq` type for reasons of performance.

The standard library has several useful methods that use tuple types, such as `.map` and `.filter` (with dictionaries), `.toMap`, `.zip`, and `.zipWithIndex`. The methods `.flatten`, `.flatMap`, `.groupBy`, and `.sliding` also work with most collection types, including dictionaries and sets. It is important to become familiar with these methods, because it will help writing code that uses sequences, sets, and dictionaries. Let us now look at these methods one by one.

The `.map` and `.toMap` methods Chapter 1 showed how the `.map` method works on sequences: the expression `xs.map(f)` applies a given function `f` to each element of the sequence `xs`, gathering the results in a new sequence. In this sense, we can say that the `.map` method "iterates over" sequences. The `.map` method works similarly on dictionaries, except that iterating over a dictionary of type `Map[K, V]` when applying `.map` looks like iterating over a sequence of *pairs*, `Seq[(K,V)]`. If `d:Map[K,V]` is a dictionary, the argument `f` of `d.map(f)` must be a function operating on tuples of type `(K,V)`. Typically, such functions are written using `case` expressions:

```
val m1 = Map("apples" -> 3, "pears" -> 2, "lemons" -> 0)

scala> m1.map { case (fruit, count) => count * 2 }
res0: Seq[Int] = ArrayBuffer(6, 4, 0)
```

If we want to transform a dictionary into another dictionary, we first create a sequence of pairs, transform it, and then convert it to a dictionary with the `.toMap` method:

```
scala> m1.map { case (fruit, count) => (fruit, count * 2) }.toMap
res1: Map[String,Int] = Map(apples -> 6, pears -> 4, lemons -> 0)
```

The `.filter` method works on dictionaries by iterating on key/value pairs. The filtering predicate must be a function of type `((K, V)) => Boolean`. For example:

```
scala> m1.filter { case (fruit, count) => count > 0 }.toMap
res2: Map[String,Int] = Map(apples -> 3, pears -> 2)
```

The `.zip` and `.zipWithIndex` methods The `.zip` method takes *two* sequences and produces a sequence of pairs, taking one element from each sequence:

```
scala> val s = List(1, 2, 3)
s: List[Int] = List(1, 2, 3)

scala> val t = List(true, false, true)
t: List[Boolean] = List(true, false, true)

scala> s.zip(t)
res3: List[(Int, Boolean)] = List((1,true), (2,false), (3,true))

scala> s zip t
res4: List[(Int, Boolean)] = List((1,true), (2,false), (3,true))
```

In the last line, the equivalent "dotless" infix syntax (`s zip t`) is shown to illustrate a syntax convention of Scala that we will sometimes use.

The `.zip` method works equally well on dictionaries: in that case, dictionaries are automatically converted to sequences of pairs before applying `.zip`.

The `.zipWithIndex` method transforms a sequence into a sequence of pairs, where the second part of the pair is the zero-based index:

```
scala> List("a", "b", "c").zipWithIndex
res5: List[(String, Int)] = List((a,0), (b,1), (c,2))
```

The `.flatten` method converts nested sequences to "flattened" ones:

```
scala> List(List(1, 2), List(2, 3), List(3, 4)).flatten
res6: List[Int] = List(1, 2, 2, 3, 3, 4)
```

The "flattening" operation computes the concatenation of the inner sequences. In Scala, sequences are concatenated using the operation `++`, e.g.:

```
scala> List(1, 2, 3) ++ List(4, 5, 6) ++ List(0)
res7: List[Int] = List(1, 2, 3, 4, 5, 6, 0)
```

So the `.flatten` method inserts the operation `++` between all the inner sequences.

Keep in mind that `.flatten` removes *only one* level of nesting, which is at the "outside" of the data structure. If applied to a `List[List[List[Int]]]`, the `.flatten` method returns a `List[List[Int]]`:

```
scala> List(List(List(1), List(2)), List(List(2), List(3))).flatten
res8: List[List[Int]] = List(List(1), List(2), List(2), List(3))
```

The `.flatMap` method is closely related to `.flatten` and can be seen as a shortcut, equivalent to first applying `.map` and then `.flatten`:

```
scala> List(1,2,3,4).map(n => (1 to n).toList)
res9: List[List[Int]] = List(List(1), List(1, 2), List(1, 2, 3), List(1, 2, 3, 4))

scala> List(1,2,3,4).map(n => (1 to n).toList).flatten
res10: List[Int] = List(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)

scala> List(1,2,3,4).flatMap(n => (1 to n).toList)
res11: List[Int] = List(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)
```

The `.flatMap` operation transforms a sequence by mapping each element to a potentially different number of new elements.

At first sight, it may be unclear why `.flatMap` is useful. (Should we perhaps combine `.filter` and `.flatten` into a `.flatFilter`, or combine `.zip` and `.flatten` into a `.flatZip`?) However, we will see later in this book that the use of `.flatMap`, which is related to "monads", is one of the most versatile and powerful design patterns in functional programming. In this chapter, several examples and exercises will illustrate the use of `.flatMap` for working on sequences.

The `.groupBy` method rearranges a sequence into a dictionary where some elements of the original sequence are grouped together into subsequences. For example, given a sequence of words, we can group all words that start with the letter `"y"` into one subsequence, and all other words into another subsequence. This is accomplished by the following code,

```
scala> Seq("wombat", "xanthan", "yoghurt", "zebra").
  groupBy(s => if (s startsWith "y") 1 else 2)
res12: Map[Int,Seq[String]] = Map(1 -> List(yoghurt), 2 -> List(wombat, xanthan, zebra))
```

The argument of the `.groupBy` method is a *function* that computes a "key" out of each sequence element. The key can have an arbitrarily chosen type. (In the current example, that type is `Int`.) The result of `.groupBy` is a dictionary that maps each key to the sub-sequence of values that have that key. (In the current example, the type of the dictionary is therefore `Map[Int, Seq[String]]`.) The order of elements in the sub-sequences remains the same as in the original sequence.

As another example of using `.groupBy`, the following code will group together all numbers that have the same remainder after division by 3:

```
scala> List(1,2,3,4,5).groupBy(k => k % 3)
res13: Map[Int,List[Int]] = Map(2 -> List(2, 5), 1 -> List(1, 4), 0 -> List(3))
```

The `.sliding` method creates a sequence of sliding windows of a given width:

```
scala> (1 to 10).sliding(4).toList
res14: List[IndexedSeq[Int]] = List(Vector(1, 2, 3, 4), Vector(2, 3, 4, 5), Vector(3, 4, 5, 6),
    Vector(4, 5, 6, 7), Vector(5, 6, 7, 8), Vector(6, 7, 8, 9), Vector(7, 8, 9, 10))
```

Usually, this method is used together with an aggregation operation on the inner sequences. For example, the following code computes a sliding-window average with window width 50 over an array of 100 numbers:

```
scala> (1 to 100).map(x => math.cos(x)).sliding(50).
  map(_.sum / 50).take(5).toList
res15: List[Double] = List(-0.005153079196990285, -0.0011160413780774369, 0.003947079736951305,
    0.005381273944717851, 0.0018679497047270743)
```

The `.sortBy` method sorts a sequence according to a sorting key. The argument of `.sortBy` is a *function* that computes the sorting key from a sequence element. In this way, we can sort elements in an arbitrary way:

```
scala> Seq(1, 2, 3).sortBy(x => -x)
res0: Seq[Int] = List(3, 2, 1)

scala> Seq("xx", "z", "yyy").sortBy(word => word)        // Sort alphabetically.
res1: Seq[String] = List(xx, yyy, z)

scala> Seq("xx", "z", "yyy").sortBy(word => word.length) // Sort by word length.
res2: Seq[String] = List(z, xx, yyy)
```

Sorting by the elements themselves, as we have done here with `.sortBy(word => word)`, is only possible if the element's type has a well-defined ordering. For strings, this is the alphabetic ordering, and for integers, the standard arithmetic ordering. For such types, a convenience method `.sorted` is defined, and works equivalently to `.sortBy(x => x)`:

```
scala> Seq("xx", "z", "yyy").sorted
res3: Seq[String] = List(xx, yyy, z)
```

2.1.5 Solved examples: Tuples and collections

Example 2.1.5.1 For a given sequence x_i, compute the sequence of pairs $b_i = (\cos x_i, \sin x_i)$.

Hint: use `.map`, assume `xs:Seq[Double]`.

Solution We need to produce a sequence that has a pair of values corresponding to each element of the original sequence. This transformation is exactly what the `.map` method does. So the code is

```
xs.map { x => (math.cos(x), math.sin(x)) }
```

Example 2.1.5.2 Count how many times $\cos x_i > \sin x_i$ occurs in a sequence x_i.

Hint: use `.count`, assume `xs:Seq[Double]`.

Solution The method `.count` takes a predicate and returns the number of times the predicate was `true` while evaluated on the elements of the sequence:

```
xs.count { x => math.cos(x) > math.sin(x) }
```

We could also reuse the solution of Exercise 2.1.5.1 that computed the cosine and the sine values. The code would then become

```
xs.map { x => (math.cos(x), math.sin(x)) }
  .count { case (cosine, sine) => cosine > sine }
```

Example 2.1.5.3 For given sequences a_i and b_i, compute the sequence of differences $c_i = a_i - b_i$.

Hint: use `.zip`, `.map`, and assume `as` and `bs` are of type `Seq[Double]`.

Solution We can use `.zip` on `as` and `bs`, which gives a sequence of pairs,

```
as.zip(bs) : Seq[(Double, Double)]
```

We then compute the differences $a_i - b_i$ by applying `.map` to this sequence:

```
as.zip(bs).map { case (a, b) => a - b }
```

Example 2.1.5.4 In a given sequence p_i, count how many times $p_i > p_{i+1}$ occurs.

Hint: use `.zip` and `.tail`.

Solution Given `ps:Seq[Double]`, we can compute `ps.tail`. The result is a sequence that is 1 element shorter than `ps`, for example:

```
scala> val ps = Seq(1,2,3,4)
ps: Seq[Int] = List(1, 2, 3, 4)

scala> ps.tail
res0: Seq[Int] = List(2, 3, 4)
```

Taking a `.zip` of the two sequences `ps` and `ps.tail`, we get a sequence of pairs:

```
scala> ps.zip(ps.tail)
res1: Seq[(Int, Int)] = List((1,2), (2,3), (3,4))
```

Note that `ps.tail` is 1 element shorter than `ps`, and the resulting sequence of pairs is also 1 element shorter than `ps`. In other words, it is not necessary to truncate `ps` before computing `ps.zip(ps.tail)`. Now apply the `.count` method:

```
ps.zip(ps.tail).count { case (a, b) => a > b }
```

Example 2.1.5.5 For a given $k > 0$, compute the sequence $c_i = \max(b_{i-k}, ..., b_{i+k})$.

Solution Applying the `.sliding` method to a list gives a list of nested lists:

```
scala> val bs = List(1,2,3,4,5)
bs: List[Int] = List(1, 2, 3, 4, 5)

scala> bs.sliding(3).toList
res0: List[List[Int]] = List(List(1, 2, 3), List(2, 3, 4), List(3, 4, 5))
```

For each b_i, we need to obtain a list of $2k+1$ nearby elements $(b_{i-k}, ..., b_{i+k})$. So we need to use `.sliding(2*k+1)` to obtain a window of the required size. Now we can compute the maximum of each of the nested lists by using the `.map` method on the outer list, with the `.max` method applied to the nested lists. So the argument of the `.map` method must be the function `nested => nested.max`:

```
bs.sliding(2 * k + 1).map(nested => nested.max)
```

In Scala, this code can be written more concisely using the syntax

```
bs.sliding(2 * k + 1).map(_.max)
```

because `_.max` means the nameless function `x => x.max`.

Example 2.1.5.6 Create a 10×10 multiplication table as a dictionary of type `Map[(Int, Int), Int]`. For example, a 3×3 multiplication table would be given by this dictionary,

```
Map( (1, 1) -> 1, (1, 2) -> 2, (1, 3) -> 3, (2, 1) -> 2,
  (2, 2) -> 4, (2, 3) -> 6, (3, 1) -> 3, (3, 2) -> 6, (3, 3) -> 9 )
```

Hint: use `.flatMap` and `.toMap`.

Solution We are required to make a dictionary that maps pairs of integers `(x, y)` to `x * y`. Begin by creating the list of *keys* for that dictionary, which must be a list of pairs `(x, y)` of the form `List((1,1), (1,2), ..., (2,1), (2,2), ...)`. We need to iterate over a sequence of values of `x`; and for each `x`, we then need to iterate over another sequence to provide values for `y`. Try this computation:

```
scala> val s = List(1, 2, 3).map(x => List(1, 2, 3))
s: List[List[Int]] = List(List(1, 2, 3), List(1, 2, 3), List(1, 2, 3))
```

We would like to get `List((1,1), (1,2), 1,3))` etc., and so we use `.map` on the inner list with a nameless function `y => (1, y)` that converts a number into a tuple,

```
scala> List(1, 2, 3).map { y => (1, y) }
res0: List[(Int, Int)] = List((1,1), (1,2), (1,3))
```

The curly braces in `{y => (1, y)}` are only for clarity; we could also use parentheses and write `(y => (1, y))`.

Now, we need to have `(x, y)` instead of `(1, y)` in the argument of `.map`, where `x` iterates over `List(1, 2, 3)` in the outside scope. Using this `.map` operation, we obtain

```
scala> val s = List(1, 2, 3).map(x => List(1, 2, 3).map { y => (x, y) })
s: List[List[(Int, Int)]] = List(List((1,1), (1,2), (1,3)), List((2,1), (2,2), (2,3)), List((3,1),
    (3,2), (3,3)))
```

This is almost what we need, except that the nested lists need to be concatenated into a single list. This is exactly what `.flatten` does:

```
scala> val s = List(1, 2, 3).map(x => List(1, 2, 3).map { y => (x, y) }).flatten
s: List[(Int, Int)] = List((1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3))
```

It is shorter to write `.flatMap(...)` instead of `.map(...).flatten`:

```
scala> val s = List(1, 2, 3).flatMap(x => List(1, 2, 3).map { y => (x, y) })
s: List[(Int, Int)] = List((1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3))
```

This is the list of keys for the required dictionary. The dictionary needs to map each *pair* of integers `(x,y)` to `x*y`. To create that dictionary, we will apply `.toMap` to a sequence of pairs `(key,value)`, which in our case needs to be of the form of a nested tuple `((x,y), x*y)`. To achieve this, we use `.map` with a function that computes the product and creates these nested tuples:

```
scala> val s = List(1, 2, 3).flatMap(x => List(1, 2, 3).map { y => (x, y) }).
  map { case (x, y) => ((x, y), x * y) }
s: List[((Int, Int), Int)] = List(((1,1),1), ((1,2),2), ((1,3),3), ((2,1),2), ((2,2),4), ((2,3),6),
    ((3,1),3), ((3,2),6), ((3,3),9))
```

We can simplify this code if we notice that we are first mapping each `y` to a tuple `(x,y)`, and later map each tuple `(x,y)` to a nested tuple `((x,y), x*y)`. Instead, the entire computation can be done in the inner `.map` operation:

```
scala> val s = List(1, 2, 3).flatMap(x => List(1, 2, 3).map { y => ((x,y), x*y) } )
s: List[((Int, Int), Int)] = List(((1,1),1), ((1,2),2), ((1,3),3), ((2,1),2), ((2,2),4), ((2,3),6),
    ((3,1),3), ((3,2),6), ((3,3),9))
```

It remains to convert this list of tuples to a dictionary with `.toMap`. Also, for better readability, we can use Scala's pair syntax `key -> value`, which is completely equivalent to writing the tuple `(key,value)`. The final code is

```
(1 to 10).flatMap(x => (1 to 10).map { y => (x,y) -> x*y }).toMap
```

Example 2.1.5.7 For a given sequence x_i, compute the maximum of all of the numbers x_i, x_i^2, $\cos x_i$, $\sin x_i$. Hint: use `.flatMap`, `.max`.

Solution We will compute the required value if we take `.max` of a list containing all of the numbers. To do that, first map each element of the list `xs:Seq[Double]` into a sequence of three numbers:

```
scala> val xs = List(0.1, 0.5, 0.9)         // An example list of some ‘Double‘ numbers.
xs: List[Double] = List(0.1, 0.5, 0.9)

scala> xs.map { x => Seq(x, x*x, math.cos(x), math.sin(x)) }
res0: List[Seq[Double]] = List(List(0.1, 0.010000000000000002, 0.9950041652780258,
    0.09983341664682815), List(0.5, 0.25, 0.8775825618903728, 0.479425538604203), List(0.9, 0.81,
    0.6216099682706644, 0.7833269096274834))
```

This list is almost what we need, except we need to `.flatten` it:

```
scala> res0.flatten
res1: List[Double] = List(0.1, 0.010000000000000002, 0.9950041652780258, 0.09983341664682815, 0.5,
    0.25, 0.8775825618903728, 0.479425538604203, 0.9, 0.81, 0.6216099682706644, 0.7833269096274834)
```

It remains to take the maximum of the resulting numbers:

```
scala> res1.max
res2: Double = 0.9950041652780258
```

The final code (starting from a given sequence `xs`) is

```
xs.flatMap { x => Seq(x, math.cos(x), math.sin(x)) }.max
```

Example 2.1.5.8 From a dictionary of type `Map[String, String]` mapping names to addresses, and assuming that the addresses do not repeat, compute a dictionary of type `Map[String, String]` mapping the addresses back to names.

Hint: use `.map` and `.toMap`.

Solution Keep in mind that iterating over a dictionary looks like iterating over a list of `(key, value)` pairs, and use `.map` to reverse each pair:

```
dict.map{ case (name, addr) => (addr, name) }.toMap
```

Example 2.1.5.9 Write the solution of Example 2.1.5.8 as a function with type parameters `Name` and `Addr` instead of the fixed type `String`.

Solution In Scala, the syntax for type parameters in a function definition is

```
def rev[Name, Addr](...) = ...
```

The type of the argument is `Map[Name, Addr]`, while the type of the result is `Map[Addr, Name]`. So we use the type parameters `Name` and `Addr` in the type signature of the function. The final code is

```
def rev[Name, Addr](dict: Map[Name, Addr]): Map[Addr, Name] =
  dict.map { case (name, addr) => (addr, name) }.toMap
```

The body of the function `rev` remains the same as in Example 2.1.5.8; only the type signature changed. This is because the procedure for reversing a dictionary works in the same way for dictionaries of any type. So the body of the function `rev` does not actually need to know the types of the keys and values in the dictionary. For this reason, it was easy for us to change the specific type `String` into type parameters in that function.

When the function `rev` is applied to a dictionary of a specific type, the Scala compiler will automatically set the type parameters `Name` and `Addr` that fit the required types of the dictionary's keys and values. For example, if we apply `rev` to a dictionary of type `Map[Boolean, Seq[String]]`, the type parameters will be set automatically as `Name = Boolean` and `Addr = Seq[String]`:

```
scala> val d = Map(true -> Seq("x", "y"), false -> Seq("z", "t"))
d: Map[Boolean, Seq[String]] = Map(true -> List(x, y), false -> List(z, t))

scala> rev(d)
res0: Map[Seq[String], Boolean] = Map(List(x, y) -> true, List(z, t) -> false)
```

Type parameters can be also set explicitly when using the function `rev`. If the type parameters are chosen incorrectly, the program will not compile:

```
scala> rev[Boolean, Seq[String]](d)
res1: Map[Seq[String],Boolean] = Map(List(x, y) -> true, List(z, t) -> false)

scala> rev[Int, Double](d)
<console>:14: error: type mismatch;
 found   : Map[Boolean,Seq[String]]
 required: Map[Int,Double]
       rev[Int, Double](d)
                        ^
```

Example 2.1.5.10* Given a sequence `words:Seq[String]` of words, compute a sequence of type `Seq[(Seq[String], Int)]`, where each inner sequence contains all the words having the same length, paired with the integer value showing that length. So, the input `Seq("the", "food", "is", "good")` should produce the output

```
Seq( (Seq("is"), 2), (Seq("the"), 3), (Seq("food", "good"), 4) )
```

The resulting sequence must be ordered by increasing length of words.

Solution It is clear that we need to begin by grouping the words by length. The library method `.groupBy` takes a function that computes a grouping key from each element of a sequence. In our case, we need to group by word length, which is computed with the method `.length` if applied to a string. So the first step is to write

```
words.groupBy{ word => word.length }
```

or, more concisely, `words.groupBy(_.length)`. The result of this expression is a dictionary that maps each length to the list of words having that length:

```
scala> words.groupBy(_.length)
res0: scala.collection.immutable.Map[Int,Seq[String]] = Map(2 -> List(is), 4 -> List(food, good), 3
    -> List(the))
```

This is close to what we need. If we convert this dictionary to a sequence, we will get a list of pairs

```
scala> words.groupBy(_.length).toSeq
res1: Seq[(Int, Seq[String])] = ArrayBuffer((2,List(is)), (4,List(food, good)), (3,List(the)))
```

It remains to swap the length and the list of words and to sort the result by increasing length. We can do this in any order: first sort, then swap; or first swap, then sort. The final code is

```
words
  .groupBy(_.length)
  .toSeq
  .sortBy { case (len, words) => len }
  .map { case (len, words) => (words, len) }
```

This can be written somewhat shorter if we use the code `_._1` for selecting the first parts from pairs and `.swap` for swapping the two elements of a pair:

```
words.groupBy(_.length).toSeq.sortBy(_._1).map(_.swap)
```

However, the program may now be harder to read and to modify.

2.1.6 Reasoning about type parameters in collections

In Example 2.1.5.10 we have applied a chain of operations to a sequence. Let us add comments showing the type of the intermediate result after each operation:

```
words // Seq[String]
  .groupBy(_.length)                          // Map[Int, Seq[String]]
  .toSeq                                      // Seq[ (Int, Seq[String]) ]
  .sortBy { case (len, words) => len }        // Seq[ (Int, Seq[String]) ]
  .map { case (len, words) => (words, len) } // Seq[ (Seq[String], Int) ]
```

In computations like this, the Scala compiler verifies at each step that the operations are applied to values of the correct type.

For instance, `.sortBy` is defined for sequences but not for dictionaries, so it would be a type error to apply `.sortBy` to a dictionary without first converting it to a sequence using `.toSeq`. The type of the intermediate result after `.toSeq` is `Seq[ (Int, Seq[String]) ]`, and the `.sortBy` operation is applied to that sequence. So the sequence element matched by `{ case (len, words) => len }` is a tuple `(Int, Seq[String])`, which means that the pattern variables `len` and `words` must have types `Int` and `Seq[String]` respectively. It would be a type error to use the sorting key function `{ case (len, words) => words }`: the sorting key can be an integer `len`, but not a string sequence `words` (because sorting by string sequences is not defined).

If we visualize how the type of the sequence should change at every step, we can more quickly understand how to implement the required task. Begin by writing down the intermediate types that would be needed during the computation:

```
words: Seq[String] // Need to group by word length.
Map[Int, Seq[String]]  // Need to sort by word length; can't sort a dictionary!
// Need to convert this dictionary to a sequence:
```

```
Seq[ (Int, Seq[String]) ] // Now sort this by the ‘Int‘ value. Sorting does not change the types.
// It remains to swap the parts of all tuples in the sequence:
Seq[ (Seq[String], Int) ] // We are done.
```

Having written down these types, we are better assured that the computation can be done correctly. Writing the code becomes straightforward, since we are guided by the already known types of the intermediate results:

```
words.groupBy(_.length).toSeq.sortBy(_._1).map(_.swap)
```

This example illustrates the main benefits of reasoning about types: it gives direct guidance about how to organize the computation, together with a greater assurance in the correctness of the code.

2.1.7 Exercises: Tuples and collections

Exercise 2.1.7.1 Find all pairs i, j within $(0, 1, ..., 9)$ such that $i + 4 * j > i * j$.
Hint: use `.flatMap` and `.filter`.

Exercise 2.1.7.2 Same task as in Exercise 2.1.7.1, but for i, j, k and the condition $i + 4 * j + 9 * k > i * j * k$.

Exercise 2.1.7.3 Given two sequences `p: Seq[String]` and `q: Seq[Boolean]` of equal length, compute a `Seq[String]` with those elements of `p` for which the corresponding element of `q` is `true`.
Hint: use `.zip`, `.map`, `.filter`.

Exercise 2.1.7.4 Convert a `Seq[Int]` into a `Seq[(Int, Boolean)]` where the `Boolean` value is `true` when the element is followed by a larger value. For example, the input sequence `Seq(1,3,2,4)` is to be converted into `Seq((1,true),(3,false),(2,true),(4,false))`. (The last element, 4, has no following element.)

Exercise 2.1.7.5 Given `p:Seq[String]` and `q:Seq[Int]` of equal length and assuming that values in `q` do not repeat, compute a `Map[Int, String]` mapping numbers from `q` to the corresponding strings from `p`.

Exercise 2.1.7.6 Write the solution of Exercise 2.1.7.5 as a function with type parameters `P` and `Q` instead of the fixed types `String` and `Int`. Test it with `P = Boolean` and `Q = Set[Int]`.

Exercise 2.1.7.7 Given `p: Seq[String]` and `q: Seq[Int]` of equal length, compute a `Seq[String]` that contains the strings from `p` ordered according to the corresponding numbers from `q`. For example, if `p = Seq("a", "b", "c")` and `q = Seq(10, -1, 5)` then the result must be `Seq("b", "c", "a")`.

Exercise 2.1.7.8 Write the solution of Exercise 2.1.7.7 as a function with type parameter `S` instead of the fixed type `String`. The required type signature and a sample test:

```
def reorder[S](p: Seq[S], q: Seq[Int]): Seq[S] = ???

scala> reorder(Seq(6.0,2.0,8.0,4.0), Seq(20,10,40,30))
res0: Seq[Double] = List(2.0, 6.0, 4.0, 8.0)
```

Exercise 2.1.7.9 Given a `Seq[(String, Int)]` showing a list of purchased items (where item names may repeat), compute a `Map[String, Int]` showing the total counts: e.g. for the input

```
Seq(("apple", 2), ("pear", 3), ("apple", 5))
```

the output must be

```
Map("apple" -> 7, "pear" -> 3)
```

Implement this computation as a function with type parameter `S` instead of `String`.
Hint: use `.groupBy`, `.map`, `.sum`.

Exercise 2.1.7.10 Given a `Seq[List[Int]]`, compute a new `Seq[List[Int]]` where each inner list contains *three* largest elements from the initial inner list (or fewer than three if the initial inner list is shorter).
Hint: use `.map`, `.sortBy`, `.take`.

Exercise 2.1.7.11 **(a)** Given two sets `p:Set[Int]` and `q:Set[Int]`, compute a set of type `Set[(Int, Int)]` as the Cartesian product of the sets `p` and `q`; that is, the set of all pairs `(x, y)` where `x` is an element from the set `p` and `y` is an element from the set `q`.

(b) Implement this computation as a function with type parameters `I`, `J` instead of `Int`. The required type signature and a sample test:

```
def cartesian[I,J](p: Set[I], q: Set[J]): Set[(I, J)] = ???

scala> cartesian(Set("a", "b"), Set(10, 20))
res0: Set[(String, Int)] = Set((a,10), (a,20), (b,10), (b,20))
```

Hint: use `.flatMap` and `.map` on sets.

Exercise 2.1.7.12* Given a `Seq[Map[Person, Amount]]`, showing the amounts various people paid on each day, compute a `Map[Person, Seq[Amount]]`, showing the sequence of payments for each person. Assume that `Person` and `Amount` are type parameters. The required type signature and a sample test:

```
def payments[Person, Amount](data: Seq[Map[Person, Amount]]): Map[Person, Seq[Amount]] = ???
// On day 1, Tarski paid 10 and Gödel paid 20. On day 2, Church paid 100 and Gentzen paid 50, etc.
scala> payments(Seq(Map("Tarski" -> 10, "Gödel" -> 20), Map("Church" -> 100, "Gentzen" -> 50),
     Map("Tarski" -> 50), Map("Banach" -> 15, "Gentzen" -> 35)))
res0: Map[String, Seq[Int]] = Map(Genzten -> List(50, 35), Church -> List(100), Banach -> List(15),
     Tarski -> List(10, 50), Gödel -> List(20))
```

Hint: use `.flatMap`, `.groupBy`, `.mapValues` on dictionaries.

2.2 Converting a sequence into a single value

Until this point, we have been working with sequences using methods such as `.map` and `.zip`. These techniques are powerful but still insufficient for solving certain problems.

A simple computation that is impossible to do using `.map` is obtaining the sum of a sequence of numbers. The standard library method `.sum` already does this; but we cannot re-implement `.sum` ourselves by using `.map`, `.zip`, or `.filter`. These operations always compute *new sequences*, while we need to compute a single value (the sum of all elements) from a sequence.

We have seen a few library methods such as `.count`, `.length`, and `.max` that compute a single value from a sequence; but we still cannot implement `.sum` using these methods. What we need is a more general way of converting a sequence to a single value, such that we could ourselves implement `.sum`, `.count`, `.max`, and other similar computations.

Another task not solvable with `.map`, `.sum`, etc., is to compute a floating-point number from a given sequence of decimal digits (including a "dot" character):

```
def digitsToDouble(ds: Seq[Char]): Double = ???

scala> digitsToDouble(Seq('2', '0', '4', '.', '5'))
res0: Double = 204.5
```

Why is it impossible to implement this function using `.map`, `.sum`, and other methods we have seen so far? In fact, the same task for *integer* numbers (instead of floating-point numbers) can be implemented via `.length`, `.map`, `.sum`, and `.zip`:

```
def digitsToInt(ds: Seq[Int]): Int = {
  val n = ds.length
  // Compute a sequence of powers of 10, e.g. [1000, 100, 100, 1].
  val powers: Seq[Int] = (0 to n - 1).map(k => math.pow(10, n - 1 - k).toInt)
  // Sum the powers of 10 with coefficients from `ds`.
  (ds zip powers).map { case (d, p) => d * p }.sum
}

scala> digitsToInt(Seq(2,4,0,5))
res0: Int = 2405
```

This task is doable because the required computation can be written as the formula

$$r = \sum_{k=0}^{n-1} d_k * 10^{n-1-k} \quad .$$

The sequence of powers of 10 can be computed separately and "zipped" with the sequence of digits d_k. However, for floating-point numbers, the sequence of powers of 10 depends on the position of the "dot" character. Methods such as `.map` or `.zip` cannot compute a sequence whose next elements depend on previous elements, and the dependence is described by some custom function.

2.2.1 Inductive definitions of aggregation functions

Mathematical induction is a general way of expressing the dependence of next values on previously computed values. To define a function from sequence to a single value (e.g. an aggregation function `f:Seq[Int] => Int`) via mathematical induction, we need to specify two computations:

- (The **base case** of the induction.) We need to specify what value the function `f` returns for an empty sequence, `Seq()`. If the function `f` is only defined for non-empty sequences, we need to specify what the function `f` returns for a one-element sequence such as `Seq(x)`, with any `x`.
- (The **inductive step**.) Assuming that the function `f` is already computed for some sequence `xs` (the **inductive assumption**), how to compute the function `f` for a sequence with one more element `x`? The sequence with one more element is written as `xs ++ Seq(x)`. So, we need to specify how to compute `f(xs ++ Seq(x))` assuming that `f(xs)` is already known.

Once these two computations are specified, the function `f` is defined (and can in principle be computed) for an arbitrary input sequence. This is how induction works in mathematics, and it works in the same way in functional programming. With this approach, the inductive definition of the method `.sum` looks like this:

- The sum of an empty sequence is 0. That is, `Seq().sum == 0`.
- If the result is already known for a sequence `xs`, and we have a sequence that has one more element `x`, the new result is equal to `xs.sum + x`. In code, this is `(xs ++ Seq(x)).sum == xs.sum + x`.

The inductive definition of the function `digitsToInt` is:

- For an empty sequence of digits, `Seq()`, the result is `0`. This is a convenient base case, even if we never call `digitsToInt` on an empty sequence.
- If `digitsToInt(xs)` is already known for a sequence `xs` of digits, and we have a sequence `xs ++ Seq(x)` with one more digit `x`, then

```
digitsToInt(xs ++ Seq(x)) = digitsToInt(xs) * 10 + x
```

Let us write inductive definitions for methods such as `.length`, `.max`, and `.count`:

- The length of a sequence:
  - for an empty sequence, `Seq().length == 0`
  - if `xs.length` is known then `(xs ++ Seq(x)).length == xs.length + 1`
- Maximum element of a sequence (undefined for empty sequences):
  - for a one-element sequence, `Seq(x).max == x`
  - if `xs.max` is known then `(xs ++ Seq(x)).max == math.max(xs.max, x)`
- Count the sequence elements satisfying a predicate `p`:

- for an empty sequence, `Seq().count(p) == 0`
- if `xs.count(p)` is known then `(xs ++ Seq(x)).count(p) == xs.count(p) + c`, where we set `c = 1` when `p(x) == true` and `c = 0` otherwise

There are two main ways of translating mathematical induction into code. The first way is to write a recursive function. The second way is to use a standard library function, such as `foldLeft` or `reduce`. Most often it is better to use the standard library functions, but sometimes the code is more transparent when using explicit recursion. So let us consider each of these ways in turn.

2.2.2 Implementing functions by recursion

A **recursive function** is any function that calls itself somewhere within its own body. The call to itself is the **recursive call**.

When the body of a recursive function is evaluated, it may repeatedly call itself with different arguments until the result value can be computed *without* any recursive calls. The last recursive call corresponds to the base case of the induction. It is an error if the base case is never reached, as in this example:

```
scala> def infiniteLoop(x: Int): Int = infiniteLoop(x+1)
infiniteLoop: (x: Int)Int

scala> infiniteLoop(2) // You will need to press Ctrl-C to stop this.
```

We translate mathematical induction into code by first writing a condition to decide whether we have the base case or the inductive step. As an example, let us define `.sum` by recursion. The base case returns `0`, and the inductive step returns a value computed from the recursive call:

```
def sum(s: Seq[Int]): Int = if (s == Seq()) 0 else {
  val x = s.head // To split s = Seq(x) ++ xs, compute x
  val xs = s.tail //  and xs.
  sum(prev) + next // Call sum(...) recursively.
}
```

In this example, we use the `if/else` expression to separate the base case from the inductive step. In the inductive step, we split the given sequence `s` into a single-element sequence `Seq(x)`, the "head" of `s`, and the remainder ("tail") sequence `xs`. So, we split `s` as `s = Seq(x) ++ xs` rather than as `s = xs ++ Seq(x)`.

For computing the sum of a numerical sequence, the order of summation does not matter. However, the order of operations *will* matter for many other computational tasks. We need to choose whether the inductive step should split the sequence as `s = Seq(x) ++ xs` or as `s = xs ++ Seq(x)`, according to the task at hand.

Consider the implementation of `digitsToInt` according to the inductive definition shown in the previous subsection:

```
def digitsToInt(s: Seq[Int]): Int = if (s == Seq()) 0 else {
  val x = s.last  // To split s = xs ++ Seq(x), compute x
  val xs = s.take(s.length - 1) //  and xs.
  digitsToInt(xs) * 10 + x // Call digitstoInt(...) recursively.
}
```

In this example, it is important to split the sequence into `s = xs ++ Seq(x)` in this order, and not in the order `Seq(x) ++ xs`. The reason is that digits increase their numerical value from right to left, so we need to multiply the value of the *left* subsequence, `digitsToInt(xs)`, by 10, in order to compute the correct result.

These examples show how mathematical induction is converted into recursive code. This approach often works but has two technical problems. The first problem is that the code will fail due to the "stack overflow" when the input sequence `s` is long enough. In the next subsection, we will see how this problem is solved (at least in some cases) using "tail recursion".

The second problem is that each inductively defined function repeats the code for checking the base case and the code for splitting the sequence `s` into the subsequence `xs` and the extra element `x`. This repeated common code can be put into a library function, and the Scala library provides such functions. We will look at using them in Section 2.2.4.

2.2.3 Tail recursion

The code of `lengthS` will fail for large enough sequences. To see why, consider an inductive definition of the `.length` method as a function `lengthS`:

```
def lengthS(s: Seq[Int]): Int =
  if (s == Seq()) 0
  else 1 + lengthS(s.tail)

scala> lengthS((1 to 1000).toList)
res0: Int = 1000

scala> val s = (1 to 100000).toList
s: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
     23, 24, 25, 26, 27, 28, 29, 30, 31, 32, ...

scala> lengthS(s)
java.lang.StackOverflowError
  at .lengthS(<console>:12)
  at .lengthS(<console>:12)
  at .lengthS(<console>:12)
  at .lengthS(<console>:12)
...
```

The problem is not due to insufficient main memory: we *are* able to compute and hold in memory the entire sequence `s`. The problem is with the code of the function `lengthS`. This function calls itself *inside* an expression `1 + lengthS(...)`. So we can visualize how the computer evaluates this code:

```
lengthS(Seq(1, 2, ..., 100000))
  = 1 + lengthS(Seq(2, ..., 100000))
  = 1 + (1 + lengthS(Seq(3, ..., 100000)))
  = ...
```

The function body of `lengthS` will evaluate the inductive step, that is, the "`else`" part of the "`if/else`", about 100000 times. Each time, the sub-expression with nested computations `1+(1+(...))` will get larger. This intermediate sub-expression needs to be held somewhere in memory, until at some point the function body goes into the base case and returns a value. When that happens, the entire intermediate sub-expression will contain about 100000 nested function calls still waiting to be evaluated. This sub-expression is held in a special area of memory called **stack memory**, where the not-yet-evaluated nested function calls are held in the order of their calls, as if on a "stack". Due to the way computer memory is managed, the stack memory has a fixed size and cannot grow automatically. So, when the intermediate expression becomes large enough, it causes an overflow of the stack memory and crashes the program.

A way to avoid stack overflows is to use a trick called **tail recursion**. Using tail recursion means rewriting the code so that all recursive calls occur at the end positions (at the "tails") of the function body. In other words, each recursive call must be *itself* the last computation in the function body, rather than placed inside other computations. Here is an example of tail-recursive code:

```
def lengthT(s: Seq[Int], res: Int): Int =
  if (s == Seq())
    res
  else
    lengthT(s.tail, 1 + res)
```

In this code, one of the branches of the `if/else` returns a fixed value without doing any recursive calls, while the other branch returns the result of recursive call to `lengthT(...)`. In the code of `lengthT`, recursive calls never occur within any sub-expressions.

It is not a problem that the recursive call to `lengthT` has some sub-expressions such as `1+res` as its arguments, because all these sub-expressions will be computed *before* `lengthT` is recursively called. The recursive call to `lengthT` is the *last* computation performed by this branch of the `if/else`. A tail-recursive function can have many `if/else` or `match/case` branches, with or without recursive calls; but all recursive calls must be always the last expressions returned.

The Scala compiler has a feature for checking automatically that a function's code is tail-recursive: the `@tailrec` annotation. If a function with a `@tailrec` annotation is not tail-recursive, or is not recursive at all, the program will not compile.

```
@tailrec def lengthT(s: Seq[Int], res: Int): Int =
```

```
  if (s == Seq()) res
  else lengthT(s.tail, 1 + res)
```

Let us trace the evaluation of this function on an example:

```
lengthT(Seq(1,2,3), 0)
  = lengthT(Seq(2, 3), 1 + 0) // = lengthT(Seq(2, 3), 1)
  = lengthT(Seq(3), 1 + 1)    // = lengthT(Seq(3), 2)
  = lengthT(Seq(), 1 + 2)     // = lengthT(Seq(), 3)
  = 3
```

All sub-expressions such as `1 + 1` and `1 + 2` are computed *before* recursive calls to `lengthT`. Because of that, sub-expressions do not grow within the stack memory. This is the main benefit of tail recursion.

How did we rewrite the code of `lengthS` to obtain the tail-recursive code of `lengthT`? An important difference between `lengthS` and `lengthT` is the additional argument, `res`, called the **accumulator argument**. This argument is equal to an intermediate result of the computation. The next intermediate result (`1 + res`) is computed and passed on to the next recursive call via the accumulator argument. In the base case of the recursion, the function now returns the accumulated result, `res`, rather than `0`, because at that time the computation is finished.

Rewriting code by adding an accumulator argument to achieve tail recursion is called the **accumulator technique** or the "accumulator trick".

One consequence of using the accumulator trick is that the function `lengthT` now always needs a value for the accumulator argument. However, our goal is to implement a function such as `length(s)` with just one argument, `s:Seq[Int]`. We can define `length(s) = lengthT(s, ???)` if we supply an initial accumulator value. The correct initial value for the accumulator is `0`, since in the base case (an empty sequence `s`) we need to return `0`.

So, a tail-recursive implementation of `lengthT` requires us to define *two* functions: the tail-recursive `lengthT` and an "adapter" function that will set the initial value of the accumulator argument. To emphasize that `lengthT` is a helper function, one could define it *inside* the adapter function:

```
def length[A](s: Seq[A]): Int = {
  @tailrec def lengthT(s: Seq[A], res: Int): Int = {
    if (s == Seq()) res
    else lengthT(s.tail, 1 + res)
  }
  lengthT(s, 0)
}
```

When `length` is implemented like that, users will not be able to call `lengthT` directly, because it is only visible within the body of the `length` function.

Another possibility in Scala is to use a **default value** for the `res` argument:

```
@tailrec def length[A](s: Seq[A], res: Int = 0): Int =
  if (s == Seq()) res
  else length(s.tail, 1 + res)
```

Giving a default value for a function argument is the same as defining *two* functions: one with that argument and one without. For example, the syntax

```
def f(x: Int, y: Boolean = false): Int = ... // Function body.
```

is equivalent to defining two functions (with the same name),

```
def f(x: Int, y: Boolean) = ... // Function body.
def f(x: Int): Int = f(Int, false)
```

Using a default argument value, we can define the tail-recursive helper function and the adapter function at once, making the code shorter.

The accumulator trick works in a large number of cases, but it may be far from obvious how to introduce the accumulator argument, what its initial value must be, and how to define the inductive step for the accumulator. In the example with the `lengthT` function, the accumulator trick works because of the following mathematical property of the expression being computed:

$$1 + (1 + (1 + (... + 1))) = (((1 + 1) + 1) + ...) + 1 \quad .$$

This is the **associativity law** of addition. Due to that law, the computation can be rearranged so that additions associate to the left. In code, it means that intermediate expressions are computed immediately before making recursive calls; this avoids the growth of the intermediate expressions.

Usually, the accumulator trick works because some associativity law is present. In that case, we are able to rearrange the order of recursive calls so that these calls always occur outside all other sub-expressions, – that is, in tail positions. However, not all computations obey a suitable associativity law. Even if a code rearrangement exists, it may not be immediately obvious how to find it.

As an example, consider a tail-recursive re-implementation of the function `digitsToInt` from the previous subsection, where the recursive call is within a sub-expression `digitsToInt(xs) * 10 + x`. To transform the code into a tail-recursive form, we need to rearrange the main computation,

$$r = d_{n-1} + 10 * (d_{n-2} + 10 * (d_{n-3} + 10 * (...d_0))) \quad ,$$

so that the operations group to the left. We can do this by rewriting r as

$$r = ((d_0 * 10 + d_1) * 10 + ...) * 10 + d_{n-1} \quad .$$

It follows that the digit sequence `s` must be split into the *leftmost* digit and the rest, `s = s.head ++ s.tail`. So, a tail-recursive implementation of the above formula is

```
@tailrec def fromDigits(s: Seq[Int], res: Int = 0):Int =
  // ‘res‘ is the accumulator.
  if (s == Seq()) res
  else fromDigits(s.tail, 10 * res + s.head)
```

Despite a certain similarity between this code and the code of `digitsToInt` from the previous subsection, the implementation `fromDigits` cannot be directly derived from the inductive definition of `digitsToInt`. One needs a separate proof that `fromDigits(s, 0)` computes the same result as `digitsToInt(s)`. The proof follows from the following property.

Statement 2.2.3.1 For any `xs: Seq[Int]` and `r: Int`, we have

```
fromDigits(xs, r) = digitsToInt(xs) + r * math.pow(10, s.length)
```

Proof We prove this by induction. To shorten the proof, denote sequences by $[1,2,3]$ instead of `Seq(1, 2, 3)` and temporarily write $d(s)$ instead of `digitsToInt(s)` and $f(s,r)$ instead of `fromDigitsT(s, r)`. Then an inductive definition of $f(s,r)$ is

$$f([],r) = r \quad , \qquad f([x]{+}{+}s,r) = f(s, 10 * r + x) \quad . \tag{2.1}$$

Denoting the length of a sequence s by $|s|$, we reformulate Statement 2.2.3.1 as

$$f(s,r) = d(s) + r * 10^{|s|} \quad , \tag{2.2}$$

We prove Eq. (2.2) by induction. To prove the base case $s = []$, we have $f([],r) = r$ and $d([]) + r * 10^0 = r$ since $d([]) = 0$ and $|s| = 0$. The resulting equality $r = r$ proves the base case.

To prove the inductive step, we assume that Eq. (2.2) holds for a given sequence s; then we need to prove that

$$f([x]{+}{+}s,r) = d([x]{+}{+}s) + r * 10^{|s|+1} \quad . \tag{2.3}$$

We will transform the left-hand side and the right-hand side separately, hoping to obtain the same expression. The left-hand side of Eq. (2.3):

$$\begin{aligned} & f([x]{+}{+}s,r) \\ \text{use Eq. (2.1)}: \quad & = f(s, 10 * r + x) \\ \text{use Eq. (2.2)}: \quad & = d(s) + (10 * r + x) * 10^{|s|} \quad . \end{aligned}$$

The right-hand side of Eq. (2.3) contains $d([x]{+}{+}s)$, which we somehow need to simplify. Assuming that $d(s)$ correctly calculates a number from its digits, we use a property of decimal notation: a digit x in front of n other digits has the value $x * 10^n$. This property can be formulated as an equation,

$$d([x]{+}{+}s) = x * 10^{|s|} + d(s) \quad . \tag{2.4}$$

So, the right-hand side of Eq. (2.3) can be rewritten as

$$\begin{aligned} & d([x]{+}{+}s) + r * 10^{|s|+1} \\ \text{use Eq. (2.4)}: \quad & = x * 10^{|s|} + d(s) + r * 10^{|s|+1} \\ \text{factor out } 10^{|s|}: \quad & = d(s) + (10 * r + x) * 10^{|s|} \quad . \end{aligned}$$

We have successfully transformed both sides of Eq. (2.3) to the same expression.

We have not yet proved that the function d satisfies the property in Eq. (2.4). The proof uses induction and begins by writing the code of d in a short notation,

$$d([]) = 0 \quad , \qquad d(s{+}{+}[y]) = d(s) * 10 + y \quad . \tag{2.5}$$

The base case is Eq. (2.4) with $s = []$. It is proved by

$$x = d([]{+}{+}[x]) = d([x]{+}{+}[]) = x * 10^0 + d([]) = x \quad .$$

The inductive step assumes Eq. (2.4) for a given x and a given sequence s, and needs to prove that for any y, the same property holds with $s{+}{+}[y]$ instead of s:

$$d([x]{+}{+}s{+}{+}[y]) = x * 10^{|s|+1} + d(s{+}{+}[y]) \quad . \tag{2.6}$$

The left-hand side of Eq. (2.6) is transformed into its right-hand side like this:

$$\begin{aligned} & d([x]{+}{+}s{+}{+}[y]) \\ \text{use Eq. (2.5)}: \quad & = d([x]{+}{+}s) * 10 + y \\ \text{use Eq. (2.4)}: \quad & = (x * 10^{|s|} + d(s)) * 10 + y \\ \text{expand parentheses}: \quad & = x * 10^{|s|+1} + d(s) * 10 + y \\ \text{use Eq. (2.5)}: \quad & = x * 10^{|s|+1} + d(s{+}{+}[y]) \quad . \end{aligned}$$

This demonstrates Eq. (2.6) and so concludes the proof.

2.2.4 Implementing general aggregation (`foldLeft`)

An **aggregation** converts a sequence of values into a single value. In general, the type of the result may be different from the type of sequence elements. To describe that general situation, we introduce type parameters, `A` and `B`, so that the input sequence is of type `Seq[A]` and the aggregated value is of type `B`. Then an inductive definition of any aggregation function `f: Seq[A] => B` looks like this:

- (Base case.) For an empty sequence, `f(Seq()) = b0` where `b0:B` is a given value.
- (Inductive step.) Assuming that `f(xs) = b` is already computed, we define `f(xs ++ Seq(x)) = g(x, b)` where `g` is a given function with type signature `g:(A, B) => B`.

The code implementing `f` is written using recursion:

```
def f[A, B](s: Seq[A]): B =
  if (s == Seq()) b0
  else g(s.last, f(s.take(s.length - 1)))
```

We can now refactor this code into a generic utility function, by making `b0` and `g` into parameters. A possible implementation is

```
def f[A, B](s: Seq[A], b: B, g: (A, B) => B): B =
  if (s == Seq()) b
  else g(s.last, f(s.take(s.length - 1), b, g)
```

However, this implementation is not tail-recursive. Applying `f` to a sequence of, say, three elements, `Seq(x, y, z)`, will create an intermediate expression `g(z, g(y, g(x, b)))`. This expression will grow with the length of `s`, which is not acceptable. To rearrange the computation into a tail-recursive form, we need to start the base case at the innermost call `g(x, b)`, then compute `g(y, g(x, b))` and continue. In other words, we need to traverse the sequence starting from its *leftmost* element `x`, rather than starting from the right. So, instead of splitting the sequence `s` into `s.last ++ s.take(s.length - 1)` as we did in the code of `f`, we need to split `s` into `s.head ++ s.tail`. Let us also exchange the order of the arguments of `g`, in order to be more consistent with the way this code is implemented in the Scala library. The resulting code is tail-recursive:

```
@tailrec def leftFold[A, B](s: Seq[A], b: B, g: (B, A) => B): B =
  if (s == Seq()) b
  else leftFold(s.tail, g(b, s.head), g)
```

We call this function a "left fold" because it aggregates (or "folds") the sequence starting from the leftmost element.

In this way, we have defined a general method of computing any inductively defined aggregation function on a sequence. The function `leftFold` implements the logic of aggregation defined via mathematical induction. Using `leftFold`, we can write concise implementations of methods such as `.sum`, `.max`, and many other aggregation functions. The method `leftFold` already contains all the code necessary to set up the base case and the inductive step. The programmer just needs to specify the expressions for the initial value `b` and for the updater function `g`.

As a first example, let us use `leftFold` for implementing the `.sum` method:

```
def sum(s: Seq[Int]): Int = leftFold(s, 0, { (x, y) => x + y })
```

To understand in detail how `leftFold` works, let us trace the evaluation of this function when applied to `Seq(1, 2, 3)`:

```
sum(Seq(1, 2, 3)) == leftFold(Seq(1, 2, 3), 0, g)
                // Here, g = { (x, y) => x + y }, so g(x, y) = x + y.
  == leftFold(Seq(2, 3), g(0, 1), g)                  // g (0, 1) = 1.
  == leftFold(Seq(2, 3), 1, g)   // Now expand the code of ‘leftFold‘.
  == leftFold(Seq(3), g(1, 2), g)     // g(1, 2) = 3; expand the code.
  == leftFold(Seq(), g(3, 3), g)      // g(3, 3) = 6; expand the code.
  == 6
```

The second argument of `leftFold` is the accumulator argument. The initial value of the accumulator is specified when first calling `leftFold`. At each iteration, the new accumulator value is computed by calling the updater function `g`, which uses the previous accumulator value and the value of the next sequence element. To visualize the process of recursive evaluation, it is convenient to write a table showing the sequence elements and the accumulator values as they are updated:

Current element `x`	Old accumulator value	New accumulator value
1	0	1
2	1	3
3	3	6

We implemented `leftFold` only as an illustration. Scala's library has a method called `.foldLeft` implementing the same logic using a slightly different type signature. To see this difference, compare the implementation of `sum` using our `leftFold` function and using the standard `.foldLeft` method:

```
def sum(s: Seq[Int]): Int = leftFold(s, 0, { (x, y) => x + y })

def sum(s: Seq[Int]): Int = s.foldLeft(0) { (x, y) => x + y }
```

The syntax of `.foldLeft` makes it more convenient to use a nameless function as the updater argument of `.foldLeft`, since curly braces separate that argument from others. We will use the standard

`.foldLeft` method from now on.

In general, the type of the accumulator value can be different from the type of the sequence elements. An example is an implementation of `count`:

```
def count[A](s: Seq[A], p: A => Boolean): Int =
  s.foldLeft(0) { (x, y) => x + (if (p(y)) 1 else 0) }
```

The accumulator is of type `Int`, while the sequence elements can have an arbitrary type, parameterized by `A`. The `.foldLeft` method works in the same way for all types of accumulators and all types of sequence elements.

The method `.foldLeft` is available in the Scala library for all collections, including dictionaries and sets. Since `.foldLeft` is tail-recursive, no stack overflows will occur even for very large sequences.

The Scala library contains several other methods similar to `.foldLeft`, such as `.foldRight` and `.reduce`. (However, `.foldRight` is not tail-recursive!)

2.2.5 Solved examples: using `foldLeft`

It is important to gain experience using the `.foldLeft` method.

Example 2.2.5.1 Use `.foldLeft` for implementing the `max` function for integer sequences. Return the special value `Int.MinValue` for empty sequences.

Solution Write an inductive formulation of the `max` function:

- (Base case.) For an empty sequence, return `Int.MinValue`.
- (Inductive step.) If `max` is already computed on a sequence `xs`, say `max(xs) = b`, the value of `max` on a sequence `xs ++ Seq(x)` is `math.max(b,x)`.

Now we can write the code:

```
def max(s: Seq[Int]): Int = s.foldLeft(Int.MinValue) { (b, x) => math.max(b, x) }
```

If we are sure that the function will never be called on empty sequences, we can implement `max` in a simpler way by using the `.reduce` method:

```
def max(s: Seq[Int]): Int = s.reduce { (x, y) => math.max(x, y) }
```

Example 2.2.5.2 Implement the `count` method on sequences of type `Seq[A]`.

Solution Using the inductive definition of the function `count` as shown in Section 2.2.1, we can write the code as

```
def count[A](s: Seq[A], p: A => Boolean): Int =
  s.foldLeft(0){ (b, x) => b + (if (p(x)) 1 else 0) }
```

Example 2.2.5.3 Implement the function `digitsToInt` using `.foldLeft`.

Solution The inductive definition of `digitsToInt` is directly translated into code:

```
def digitsToInt(d: Seq[Int]): Int = d.foldLeft(0){ (n, x) => n * 10 + x }
```

Example 2.2.5.4 For a given non-empty sequence `xs: Seq[Double]`, compute the minimum, the maximum, and the mean as a tuple $(x_{\min}, x_{\max}, x_{\text{mean}})$. The sequence should be traversed only once, i.e. the code must be `xs.foldLeft(...)`, using `.foldLeft` only once.

Solution Without the requirement of using a single traversal, we would write

```
(xs.min, xs.max, xs.sum / xs.length)
```

However, this code traverses `xs` at least three times, since each of the aggregations `xs.min`, `xs.max`, and `xs.sum` iterates over `xs`. We need to combine the four inductive definitions of `min`, `max`, `sum`, and `length` into a single inductive definition of some function. What is the type of that function's return value? We need to accumulate intermediate values of *all four* numbers (`min`, `max`, `sum`, and `length`) in a tuple. So the required type of the accumulator is `(Double, Double, Double, Double)`. To avoid repeating a long type expression, we can define a type alias for it, say, `D4`:

```
scala> type D4 = (Double, Double, Double, Double)
defined type alias D4
```

The updater function must update each of the four numbers according to the definitions of their inductive steps:

```
def update(p: D4, x: Double): D4 = p match {
  case (min, max, sum, length) =>
   (math.min(x, min), math.max(x, max), x + sum, length + 1)
}
```

Now we can write the code of the required function:

```
def f(xs: Seq[Double]): (Double, Double, Double) = {
  val init: D4 = (Double.PositiveInfinity, Double.NegativeInfinity, 0, 0)
  val (min, max, sum, length) = xs.foldLeft(init)(update)
  (min, max, sum/length)
}

scala> f(Seq(1.0, 1.5, 2.0, 2.5, 3.0))
res0: (Double, Double, Double) = (1.0,3.0,2.0)
```

Example 2.2.5.5* Implement the function `digitsToDouble` using `.foldLeft`. The argument is of type `Seq[Char]`. As a test, the expression `digitsToDouble(Seq('3','4','.','2','5'))` must evaluate to `34.25`. Assume that all input characters are either digits or a dot (so, negative numbers are not supported).

Solution The evaluation of a `.foldLeft` on a sequence of digits will visit the sequence from left to right. The updating function should work the same as in `digitsToInt` until a dot character is found. After that, we need to change the updating function. So, we need to remember whether a dot character has been seen. The only way for `.foldLeft` to "remember" any data is to hold that data in the accumulator value. We can choose the type of the accumulator according to our needs. So, for this task we can choose the accumulator to be a *tuple* that contains, for instance, the floating-point result constructed so far and a `Boolean` flag showing whether we have already seen the dot character.

To see what `digitsToDouble` must do, let us consider how the evaluation of `digitsToDouble(Seq('3', '4', '.', '2', '5'))` should go. We can write a table showing the intermediate result at each iteration. This will hopefully help us figure out what the accumulator and the updater function $g(...)$ must be:

Current digit c	**Previous result** n	**New result** $n' = g(n, c)$
'3'	0.0	3.0
'4'	3.0	34.0
'.'	34.0	34.0
'2'	34.0	34.2
'5'	34.2	34.25

While the dot character was not yet seen, the updater function multiplies the previous result by 10 and adds the current digit. After the dot character, the updater function must add to the previous result the current digit divided by a factor that represents increasing powers of 10. In other words, the update computation $n' = g(n, c)$ must be defined by these formulas:

- Before the dot character: $g(n, c) = n * 10 + c$.
- After the dot character: $g(n, c) = n + \frac{c}{f}$, where f is 10, 100, 1000, ..., for each new digit.

The updater function g has only two arguments: the current digit and the previous accumulator value. So, the changing factor f must be *part of* the accumulator value, and must be multiplied by 10 at each digit after the dot. If the factor f is not a part of the accumulator value, the function g will

not have enough information for computing the next accumulator value correctly. So, the updater computation must be $n' = g(n, c, f)$, not $n' = g(n, c)$.

For this reason, we choose the accumulator type as a tuple `(Double, Boolean, Double)` where the first number is the result n computed so far, the `Boolean` flag indicates whether the dot was already seen, and the third number is f, that is, the power of 10 by which the current digit will be divided if the dot was already seen. Initially, the accumulator tuple will be equal to `(0.0, false, 10.0)`. Then the updater function is implemented like this:

```
def update(acc: (Double, Boolean, Double), c: Char): (Double, Boolean, Double) =
acc match { case (n, flag, factor) =>
  if (c == '.') (n, true, factor) // Set flag to ‘true‘ after a dot character was seen.
  else {
    val digit = c - '0'
    if (flag)                     // This digit is after the dot. Update ‘factor‘.
      (n + digit/factor, flag, factor * 10)
    else                          // This digit is before the dot.
      (n * 10 + digit, flag, factor)
  }
}
```

Now we can implement `digitsToDouble` as follows,

```
def digitsToDouble(d: Seq[Char]): Double = {
  val initAccumulator = (0.0, false, 10.0)
  val (n, _, _) =
     d.foldLeft(initAccumulator)(update)
  n
}

scala> digitsToDouble(Seq('3', '4', '.', '2',
    '5'))
res0: Double = 34.25
```

The result of calling `d.foldLeft` is a tuple `(n, flag, factor)`, in which only the first part, `n`, is needed. In Scala's pattern matching expressions, the underscore symbol is used to denote the pattern variables whose values are not needed in the subsequent code. We could extract the first part using the accessor method `._1`, but the code will be more readable if we show all parts of the tuple by writing `(n, _, _)`.

Example 2.2.5.6 Implement the `.map` method for sequences by using `.foldLeft`. The input sequence should be of type `Seq[A]` and the output sequence of type `Seq[B]`, where `A` and `B` are type parameters. The required type signature of the function and a sample test:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B] = ???

scala> map(List(1, 2, 3)) { x => x * 10 }
res0: Seq[Int] = List(10, 20, 30)
```

Solution The required code should build a new sequence by applying the function `f` to each element. How can we build a new sequence using `.foldLeft`? The evaluation of `.foldLeft` consists of iterating over the input sequence and accumulating some result value, which is updated at each iteration. Since the result of a `.foldLeft` is always equal to the last computed accumulator value, it follows that the new sequence should *be* the accumulator value. So, we need to update the accumulator by appending the value `f(x)`, where `x` is the current element of the input sequence. We can append elements to sequences using the `:+` operation:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B] =
  xs.foldLeft(Seq[B]()) { (acc, x) => acc :+ f(x) }
```

The operation `acc :+ f(x)` is equivalent to `acc ++ Seq(f(x))` but is shorter to write.

Example 2.2.5.7 Implement a function `toPairs` that converts a sequence of type `Seq[A]` to a sequence of pairs, `Seq[(A, A)]`, by putting together the adjacent elements pairwise. If the initial sequence has an odd number of elements, a given default value of type `A` is used to fill the last pair. The required type signature and an example test:

```
def toPairs[A](xs: Seq[A], default: A): Seq[(A, A)] = ???

scala> toPairs(Seq(1, 2, 3, 4, 5, 6), -1)
```

```
res0: Seq[(Int, Int)] = List((1,2), (3,4), (5,6))

scala> toPairs(Seq("a", "b", "c"), "<nothing>")
res1: Seq[(String, String)] = List((a,b), (c,<nothing>))
```

Solution We need to use `.foldLeft` to accumulate a sequence of pairs, and each pair needs two values. However, we iterate over values in the input sequence one by one. So, a new pair can be made only once every two iterations. The accumulator needs to hold the information about the current iteration being even or odd. For odd-numbered iterations, the accumulator also needs to store the previous element that is still waiting for its pair. Therefore, we choose the type of the accumulator to be a tuple `(Seq[(A, A)], Seq(A))`. The first sequence is the intermediate result, and the second sequence is the "remainder": it holds the previous element for odd-numbered iterations and is empty for even-numbered iterations. Initially, the accumulator should be empty. A trace of the accumulator updates is shown in this table:

Current element `x`	Previous accumulator	Next accumulator
`"a"`	`(Seq(), Seq())`	`(Seq(), Seq("a"))`
`"b"`	`(Seq(), Seq("a"))`	`(Seq(("a","b")), Seq())`
`"c"`	`(Seq(("a","b")), Seq())`	`(Seq(("a","b")), Seq("c"))`

Now it becomes clear how to implement the updater function.

```
type Acc = (Seq[(A, A)], Seq[A])     // Type alias, for brevity.
def updater(acc: Acc, x: A): Acc = acc match {
    case (result, Seq())      => (result, Seq(x))
    case (result, Seq(prev))  => (result ++ Seq((prev, x)), Seq())
  }
```

We will call `.foldLeft` with this updater and then perform some post-processing to make sure we create the last pair in case the last iteration is odd-numbered, i.e. when the "remainder" is not empty after `.foldLeft` is finished. In this implementation, we use pattern matching to decide whether a sequence is empty:

```
def toPairs[A](xs: Seq[A], default: A): Seq[(A, A)] = {
  type Acc = (Seq[(A, A)], Seq[A])     // Type alias, for brevity.
  def init: Acc = (Seq(), Seq())
  def updater(acc: Acc, x: A): Acc = acc match {
    case (result, Seq())      => (result, Seq(x))
    case (result, Seq(prev))  => (result ++ Seq((prev, x)), Seq())
  }
  val (result, remainder) = xs.foldLeft(init)(updater)
  remainder match {    // May need to append the last element to the result.
    case Seq()    => result
    case Seq(x)   => result ++ Seq((x, default))
  }
}
```

This code shows examples of partial functions that are applied safely. One of these partial functions is used in the expression

```
remainder match {
  case Seq()    => ...
  case Seq(a)   => ...
}
```

This code works when `remainder` is empty or has length 1 but would fail for longer sequences. Within the implementation of `toPairs`, the value of `remainder` will always be a sequence of length at most 1, so the code is safe.

2.2.6 Exercises: Using `foldLeft`

Exercise 2.2.6.1 Implement a function `fromPairs` that performs the inverse transformation to the `toPairs` function defined in Example 2.2.5.7. The required type signature and a sample test:

```
def fromPairs[A](xs: Seq[(A, A)]): Seq[A] = ???

scala> fromPairs(Seq((1, 2), (3, 4)))
res0: Seq[Int] = List(1, 2, 3, 4)
```

Hint: This can be done with `.foldLeft` or with `.flatMap`.

Exercise 2.2.6.2 Implement the `flatten` method for sequences by using `.foldLeft`. The required type signature and a sample test:

```
def flatten[A](xxs: Seq[Seq[A]]): Seq[A] = ???

scala> flatten(Seq(Seq(1, 2, 3), Seq(), Seq(4)))
res0: Seq[Int] = List(1, 2, 3, 4)
```

Exercise 2.2.6.3 Use `.foldLeft` to implement the `zipWithIndex` method for sequences. The required type signature and a sample test:

```
def zipWithIndex[A](xs: Seq[A]): Seq[(A, Int)] = ???

scala> zipWithIndex(Seq("a", "b", "c", "d"))
res0: Seq[String] = List((a, 0), (b, 1), (c, 2), (d, 3))
```

Exercise 2.2.6.4 Use `.foldLeft` to implement a function `filterMap` that combines `.map` and `.filter` for sequences. The required type signature and a sample test:

```
def filterMap[A, B](xs: Seq[A])(pred: A => Boolean)(f: A => B): Seq[B] = ???

scala> filterMap(Seq(1, 2, 3, 4)) { x => x > 2 } { x => x * 10 }
res0: Seq[Int] = List(30, 40)
```

Exercise 2.2.6.5* Split a sequence into subsequences ("batches") of length not larger than a given maximum length n. The required type signature and a sample test:

```
def byLength[A](xs: Seq[A], length: Int): Seq[Seq[A]] = ???

scala> byLength(Seq("a", "b", "c", "d"), 2)
res0: Seq[Seq[String]] = List(List(a, b), List(c, d))

scala> byLength(Seq(1, 2, 3, 4, 5, 6, 7), 3)
res1: Seq[Seq[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7))
```

Exercise 2.2.6.6* Split a sequence into batches by "weight" computed via a given function. The total weight of items in any batch should not be larger than a given maximum weight. The required type signature and a sample test:

```
def byWeight[A](xs: Seq[A], maxW: Double)(w: A => Double): Seq[Seq[A]] = ???

scala> byWeight((1 to 10).toList, 5.75){ x => math.sqrt(x) }
res0: Seq[Seq[Int]] = List(List(1, 2, 3), List(4, 5), List(6, 7), List(8), List(9), List(10))
```

Exercise 2.2.6.7* Use `.foldLeft` to implement a `groupBy` function. The type signature and a test:

```
def groupBy[A, K](xs: Seq[A])(by: A => K): Map[K, Seq[A]] = ???

scala> groupBy(Seq(1, 2, 3, 4, 5)){ x => x % 2 }
res0: Map[Int, Seq[Int]] = Map(1 -> List(1, 3, 5), 0 -> List(2, 4))
```

Hints: The accumulator should be of type `Map[K, Seq[A]]`. To work with dictionaries, you will need to use the methods `.getOrElse` and `.updated`. The method `.getOrElse` fetches a value from a dictionary by key, and returns the given default value if the dictionary does not contain that key:

```
scala> Map("a" -> 1, "b" -> 2).getOrElse("a", 300)
```

```
res0: Int = 1

scala> Map("a" -> 1, "b" -> 2).getOrElse("c", 300)
res1: Int = 300
```

The method `.updated` produces a new dictionary that contains a new value for the given key, whether or not that key already exists in the dictionary:

```
scala> Map("a" -> 1, "b" -> 2).updated("c", 300) // Key is new.
res0: Map[String,Int] = Map(a -> 1, b -> 2, c -> 300)

scala> Map("a" -> 1, "b" -> 2).updated("a", 400) // Key already exists.
res1: Map[String,Int] = Map(a -> 400, b -> 2)
```

2.3 Converting a single value into a sequence

An aggregation converts or "folds" a sequence into a single value; the opposite operation ("unfolding") converts a single value into a sequence. An example of this task is to compute the sequence of decimal digits for a given integer:

```
def digitsOf(x: Int): Seq[Int] = ???

scala> digitsOf(2405)
res0: Seq[Int] = List(2, 4, 0, 5)
```

We cannot implement `digitsOf` using `.map`, `.zip`, or `.foldLeft`, because these methods work only if we *already have* a sequence; but the function `digitsOf` needs to create a new sequence. We could create a sequence via the expression `(1 to n)` if the required length of the sequence were known in advance. However, the function `digitsOf` must produce a sequence whose length is determined by a condition that we cannot easily evaluate in advance.

A general "unfolding" operation requires us to build a sequence whose length is not determined in advance. This kind of sequence is called a **stream**. A stream is a sequence whose elements are computed only when necessary (unlike sequences such as a `List` or an `Array`, whose elements are all computed in advance and stored). The unfolding operation will keep computing the next element; this creates a stream. We can then apply `.takeWhile` to the stream, in order to stop it when a certain condition holds. Finally, if required, the truncated stream may be converted to a list or to another type of sequence. In this way, we can generate a sequence of initially unknown length according to any given requirements.

The Scala library has a general stream-producing function `Stream.iterate`. This function has two arguments, the initial value and a function that computes the next value from the previous one:

```
scala> Stream.iterate(2) { x => x + 10 }
res0: Stream[Int] = Stream(2, ?)
```

The stream is ready to start computing the next elements of the sequence (so far, only the first element, `2`, has been computed). In order to see the next elements, we need to stop the stream at a finite size and then convert the result to a list:

```
scala> Stream.iterate(2) { x => x + 10 }.take(6).toList
res1: List[Int] = List(2, 12, 22, 32, 42, 52)
```

If we try to evaluate `.toList` on a stream without first limiting its size via `.take` or `.takeWhile`, the program will keep producing more elements of the stream until it runs out of memory and crashes.

Streams are similar to sequences, and methods such as `.map`, `.filter`, `.flatMap` are also defined for streams. For instance, we can use the method `.drop` that skips a given number of initial elements:

```
scala> Seq(10, 20, 30, 40, 50).drop(3)
res2: Seq[Int] = List(40, 50)

scala> Stream.iterate(2) { x => x + 10 }.drop(3)
res3: Stream[Int] = Stream(32, ?)
```

This example shows that in order to evaluate `.drop(3)`, the stream had to compute its elements up to `32` (but the subsequent elements are still not computed).

To figure out the code for `digitsOf`, we first write this function as a mathematical formula. To compute the digits for, say, $n = 2405$, we need to divide n repeatedly by 10, getting a sequence n_k of intermediate numbers ($n_0 = 2405$, $n_1 = 240$, ...) and the corresponding sequence of last digits, $n_k \bmod 10$ (in this example: 5, 0, ...). The sequence n_k is defined using mathematical induction:

- Base case: $n_0 = n$, where n is the given initial integer.
- Inductive step: $n_{k+1} = \left\lfloor \frac{n_k}{10} \right\rfloor$ for $k = 1, 2, ...$

Here $\left\lfloor \frac{n_k}{10} \right\rfloor$ is the mathematical notation for the integer division by 10. Let us tabulate the evaluation of the sequence n_k for $n = 2405$:

$k =$	0	1	2	3	4	5	6
$n_k =$	2405	240	24	2	0	0	0
$n_k \bmod 10 =$	5	0	4	2	0	0	0

The numbers n_k will remain all zeros after $k = 4$. It is clear that the useful part of the sequence is before it becomes all zeros. In this example, the sequence n_k needs to be stopped at $k = 4$. The sequence of digits then becomes $[5, 0, 4, 2]$, and we need to reverse it to obtain $[2, 4, 0, 5]$. For reversing a sequence, the Scala library has the standard method `.reverse`. A complete implementation for `digitsOf` is thus

```
def digitsOf(n: Int): Seq[Int] =
  if (n == 0) Seq(0) else { // n == 0 is a special case.
    Stream.iterate(n) { nk => nk / 10 }
      .takeWhile { nk => nk != 0 }
      .map { nk => nk % 10 }
      .toList.reverse
  }
```

We can shorten the code by using the syntax such as `(_ % 10)` instead of `{ nk => nk % 10 }`,

```
def digitsOf(n: Int): Seq[Int] =
  if (n == 0) Seq(0) else { // n == 0 is a special case.
    Stream.iterate(n)(_ / 10)
      .takeWhile(_ != 0)
      .map(_ % 10)
      .toList.reverse
  }
```

The type signature of the method `Stream.iterate` can be written as

```
def iterate[A](init: A)(next: A => A): Stream[A]
```

and shows a close correspondence to a definition by mathematical induction. The base case is the first value, `init`, and the inductive step is a function, `next`, that computes the next element from the previous one. It is a general way of creating sequences whose length is not determined in advance.

2.4 Transforming a sequence into another sequence

We have seen methods such as `.map` and `.zip` that transform sequences into sequences. However, these methods cannot express a general transformation where the elements of the new sequence are defined by induction and depend on previous elements. An example of this kind is computing the partial sums of a given sequence x_i, say $b_k = \sum_{i=0}^{k-1} x_i$. This formula defines $b_0 = 0$, $b_1 = x_0$, $b_2 = x_0 + x_1$, $b_3 = x_0 + x_1 + x_2$, etc. A definition via mathematical induction may be written like this:

- (Base case.) $b_0 = 0$.
- (Inductive step.) Given b_k, we define $b_{k+1} = b_k + x_k$ for $k = 0, 1, 2, ...$

The Scala library method `.scanLeft` implements a general sequence-to-sequence transformation defined in this way. The code implementing the partial sums is

```
def partialSums(xs: Seq[Int]): Seq[Int] = xs.scanLeft(0){ (x, y) => x + y }
```

```
scala> partialSums(Seq(1, 2, 3, 4))
res0: Seq[Int] = List(0, 1, 3, 6, 10)
```

The first argument of `.scanLeft` is the base case, and the second argument is an updater function describing the inductive step. In general, the type of elements of the second sequence is different from that of the first sequence. The updater function takes an element of the first sequence and a previous element of the second sequence, and returns the next element of the second sequence. Note that the result of `.scanLeft` is one element longer than the original sequence, because the base case provides an initial value.

Until now, we have seen that `.foldLeft` is sufficient to re-implement almost every method that work on sequences, such as `.map`, `.filter`, or `.flatten`. Let us show, as an illustration, how to implement the method `.scanLeft` via `.foldLeft`. In the implementation, the accumulator contains the previous element of the second sequence together with a growing fragment of that sequence, which is updated as we iterate over the first sequence. The code is

```
1 def scanLeft[A, B](xs: Seq[A])(b0: B)(next: (B, A) => B)
2     : Seq[B] = {
3   val init: (B, Seq[B]) = (b0, Seq(b0))
4   val (_, result) = xs.foldLeft(init){ case ((b, seq), x) =>
5     val newB = next(b, x)
6     (newB, seq ++ Seq(newB))
7   }
8   result
9 }
```

To implement the (nameless) updater function for `.foldLeft` in lines 4–7, we used the Scala feature that makes it easier to define functions with several arguments containing tuples. In our case, the updater function in `.foldLeft` has two arguments: the first is a tuple `(B, Seq[B])`, the second is a value of type `A`. The pattern matching expression `{case ((b, seq), x) => ...}` appears to match a nested tuple. In reality, this expression matches the two arguments of the updater function and, at the same time, destructures the tuple argument as `(b, seq)`.

2.5 Summary

We have seen a broad overview of translating mathematical induction into Scala code.

What problems can we solve now?

- Compute mathematical expressions involving arbitrary recursion.
- Use the accumulator trick to enforce tail recursion.
- Implement functions with type parameters.
- Use arbitrary inductive (i.e. recursive) formulas to:
  - convert sequences to single values (aggregation or "folding");
  - create new sequences from single values ("unfolding");
  - transform existing sequences into new sequences.

Definition by induction	Scala code example
$f([]) = b\,;\ \ f(s{+}{+}[x]) = g(f(s), x)$	`f(xs) = xs.foldLeft(b)(g)`
$x_0 = b\,;\ \ x_{k+1} = g(x_k)$	`xs = Stream.iterate(b)(g)`
$y_0 = b\,;\ \ y_{k+1} = g(y_k, x_k)$	`ys = xs.scanLeft(b)(g)`

Table 2.1: Implementing mathematical induction.

Table 2.1 shows Scala code implementing those tasks. Iterative calculations are implemented by translating mathematical induction directly into code. In the functional programming paradigm, the programmer does not need to write any loops or use array indices. Instead, the programmer reasons about sequences as mathematical values: "Starting from this value, we get that sequence, then transform it into this other sequence,"

etc. This is a powerful way of working with sequences, dictionaries, and sets. Many kinds of programming errors (such as an incorrect array index) are avoided from the outset, and the code is shorter and easier to read than conventional code written using loops.

What tasks are not possible with these tools? We cannot implement a non-tail-recursive function without stack overflow (i.e. without unlimited growth of intermediate expressions). The accumulator trick does not always work! In some cases, it is impossible to implement tail recursion in a given recursive computation. An example of such a computation is the "merge-sort" algorithm where the function body must contain two recursive calls within a single expression. (It is impossible to rewrite *two* recursive calls as one.)

What if our recursive code cannot be transformed into tail-recursive code via the accumulator trick, but the recursion depth is so large that stack overflows occur? There exist special tricks (e.g. the "continuation-passing" or "trampolines") that convert non-tail-recursive code into iterative code without stack overflows. Those advanced tricks are beyond the scope of this chapter.

2.5.1 Solved examples

Example 2.5.1.1 Compute the smallest n such that $f(f(f(...f(1)...) \geq 1000$, where the function f is applied n times. Write this as a function taking f, 1, and 1000 as arguments. Test with $f(x) = 2x + 1$.

Solution We define a stream of values $[1, f(1), f(f(1)), ...]$ and use `.takeWhile` to stop the stream when the given condition holds. The `.length` method then gives the length of the resulting sequence:

```
scala> Stream.iterate(1)(x => 2*x+1).takeWhile(x => x < 1000).toList
res0: List[Int] = List(1, 3, 7, 15, 31, 63, 127, 255, 511)

scala> Stream.iterate(1)(x => 2*x+1).takeWhile(x => x < 1000).length
res1: Int = 9
```

Example 2.5.1.2 **(a)** For a given `Stream[Int]`, compute the stream of the largest values seen so far.

(b) Compute the stream of k largest values seen so far (k is a given integer parameter).

Solution: We cannot use `.max` or sort the entire stream, since the length of the stream is not known in advance. So we need to use `.scanLeft`, which will build the output stream one element at a time.

(a) Maintain the largest value seen so far in the accumulator of the `.scanLeft`:

```
def maxSoFar(xs: Stream[Int]): Stream[Int] =
  xs.scanLeft(xs.head){ case (max, x) => math.max(max, x) }
    .drop(1)
```

We use `.drop(1)` to remove the initial value, `xs.head`, because it is not useful for our result but is always produced by `.scanLeft`.

To test this function, let us define a stream whose values go up and down:

```
val s = Stream.iterate(0)(x => 1 - 2*x)

scala> s.take(10).toList
res0: List[Int] = List(0, 1, -1, 3, -5, 11, -21, 43, -85, 171)

scala> maxSoFar(s).take(10).toList
res1: List[Int] = List(0, 1, 1, 3, 3, 11, 11, 43, 43, 171)
```

(b) We again use `.scanLeft`, where now the accumulator needs to keep the largest k values seen so far. There are two ways of maintaining this accumulator: First, to have a sequence of k values that we sort and truncate each time. Second, to use a specialized data structure such as a priority queue that automatically keeps values sorted and its length bounded. For the purposes of this tutorial, let us avoid using specialized data structures:

```
def maxKSoFar(xs: Stream[Int], k: Int): Stream[Seq[Int]] = {
  // The initial value of the accumulator is an empty Seq() of type Seq[Int].
  xs.scanLeft(Seq[Int]()) { case (seq, x) =>
  // Sort in the descending order, and take the first k values.
```

```
    (seq :+ x).sorted.reverse.take(k)
  }.drop(1) // Skip the useless first value.
}

scala> maxKSoFar(s, 3).take(10).toList
res2: List[Seq[Int]] = List(List(0), List(1, 0), List(1, 0, -1), List(3, 1, 0), List(3, 1, 0),
     List(11, 3, 1), List(11, 3, 1), List(43, 11, 3), List(43, 11, 3), List(171, 43, 11))
```

Example 2.5.1.3 Find the last element of a non-empty sequence. (Hint: use `.reduce`.)

Solution This function is available in the Scala library as the standard method `.last` on sequences. Here we need to re-implement it using `.reduce`. Begin by writing an inductive definition:

- (Base case.) `last(Seq(x)) == x`.
- (Inductive step.) `last(Seq(x) ++ xs) == last(xs)` assuming `xs` is non-empty.

The `.reduce` method implements an inductive aggregation similarly to `.foldLeft`, except that for `.reduce` the base case is fixed – it always returns `x` for a 1-element sequence `Seq(x)`. This is exactly what we need here, so the inductive definition is directly translated into code, with the updater function $g(x, y) = y$:

```
def last[A](xs: Seq[A]): A = xs.reduce { case (x, y) => y }
```

Example 2.5.1.4 **(a)** Count the occurrences of each distinct word in a string:

```
def countWords(s: String): Map[String, Int] = ???

scala> countWords("a quick a quick a fox")
res0: Map[String, Int] = Map("a" -> 3, "quick" -> 2, "fox" -> 1)
```

(b) Count the occurrences of each distinct element in a sequence of type `Seq[A]`.

Solution **(a)** We split the string into an array of words via `s.split(" ")`, and apply a `.foldLeft` to that array, since the computation is a kind of aggregation over the array of words. The accumulator of the aggregation will be the dictionary of word counts for all the words seen so far:

```
def countWords(s: String): Map[String, Int] = {
  val init: Map[String, Int] = Map()
  s.split(" ").foldLeft(init) { (dict, word) =>
    val newCount = dict.getOrElse(word, 0) + 1
    dict.updated(word, newCount)
  }
}
```

An alternative, shorter implementation of the same function is

```
def countWords(s: String): Map[String, Int] = s.split(" ").groupBy(w => w).mapValues(_.length)
```

The `.groupBy` creates a dictionary in one function call rather than one entry at a time. But the resulting dictionary contains word lists instead of word counts, so we use `.mapValues`:

```
scala> "a a b b b c".split(" ").groupBy(w => w)
res0: Map[String,Array[String]] = Map(b -> Array(b, b, b), a -> Array(a, a), c -> Array(c))

scala> res0.mapValues(_.length)
res1: Map[String,Int] = Map(b -> 3, a -> 2, c -> 1)
```

(b) The main code of `countWords` does not depend on the fact that words are of type `String`. It will work in the same way for any other type of keys for the dictionary. So we keep the same code and define the type signature of the function to contain a type parameter `A` instead of `String`:

```
def countValues[A](xs: Seq[A]): Map[A, Int] =
  xs.foldLeft(Map[A, Int]()) { (dict, word) =>
    val newCount = dict.getOrElse(word, 0) + 1
    dict.updated(word, newCount)
  }
```

```
scala> countValues(Seq(100, 100, 200, 100, 200, 200, 100))
res0: Map[Int,Int] = Map(100 -> 4, 200 -> 3)
```

Example 2.5.1.5 **(a)** Implement the binary search algorithm[1] for a sorted sequence `xs: Seq[Int]` as a function returning the index of the requested value `goal` (assume that `xs` always contains `goal`):

```
@tailrec def binSearch(xs: Seq[Int], goal: Int): Int = ???

scala> binSearch(Seq(1, 3, 5, 7), 5)
res0: Int = 2
```

(b) Re-implement `binSearch` using `Stream.iterate` without writing explicitly recursive code.

Solution **(a)** The well-known binary search algorithm splits the array into two halves and may continue the search recursively in one of the halves. We need to write the solution as a tail-recursive function with an additional accumulator argument. So we expect that the code should look like this,

```
@tailrec def binSearch(xs: Seq[Int], goal: Int, acc: _ = ???): Int = {
  if (???are we done???) acc
  else {
    // Determine which half of the sequence contains ‘goal‘.
    // Then update the accumulator accordingly.
    val newAcc = ???
    binSearch(xs, goal, newAcc) // Tail-recursive call.
  }
}
```

We now decide the type and the initial value of the accumulator, and implement the updater for it.

The information required for the recursive call must show the segment of the sequence where the target number is present. That segment is defined by two indices i, j representing the left and the right bounds of the sub-sequence, such that the target element is x_n with $x_i \le x_n < x_{j-1}$. It follows that the accumulator should be a pair of two integers (i, j). The initial value of the accumulator is the pair $(0, N)$, where N is the length of the entire sequence. The search is finished when $i + 1 = j$. We now write the code, introducing for convenience *two* accumulator values representing (i, j):

```
@tailrec def binSearch(xs: Seq[Int], goal: Int)(left: Int = 0, right: Int = xs.length): Int = {
  // Check whether ‘goal‘ is at one of the boundaries.
  if (right - left <= 1 || xs(left) == goal) left
  else {
    val middle = (left + right) / 2
    // Determine which half of the array contains ‘target‘.
    // Update the accumulator accordingly.
    val (newLeft, newRight) =
      if (goal < xs(middle)) (left, middle)
      else (middle, right)
    binSearch(xs, goal)(newLeft, newRight) // Tail-recursive call.
  }
}

scala> binSearch(0 to 10, 3)() // Default accumulator values.
res0: Int = 3
```

Here we used a feature of Scala that allows us to set `xs.length` as a default value for the argument `right` of `binSearch`. This works because `right` is in a different **argument group** from `xs`. Default values in an argument group may depend on arguments in a *previous* argument group. However, the code

```
def binSearch(xs: Seq[Int], goal: Int, left: Int = 0, right: Int = xs.length)
```

will generate an error: the arguments in the same argument group cannot depend on each other. (The error will say `not found: value xs`.)

(b) We can visualize the binary search as a procedure that generates a sequence of progressively

[1] https://en.wikipedia.org/wiki/Binary_search_algorithm

tighter bounds for the location of `goal`. The initial bounds are `(0, xs.length)`, and the final bounds are `(k, k+1)` for some `k`. We can generate the sequence of bounds using `Stream.iterate` and stop the sequence when the bounds become sufficiently tight. To make the use of `.takeWhile` more convenient, we add an extra sequence element where the bounds `(k, k)` are equal. The code becomes

```
def binSearch(xs: Seq[Int], goal: Int): Int = {
  type Acc = (Int, Int)
  val init: Acc = (0, xs.length)
  val updater: Acc => Acc = { case (left, right) =>
    if (right - left <= 1) (left, left) // Extra element (k, k) in the stream.
    else if (xs(left) == goal) (left, left + 1)
    else {
      val middle = (left + right) / 2
      // Determine which half of the array contains ‘target‘.
      // Update the accumulator accordingly.
      if (goal < xs(middle)) (left, middle)
      else (middle, right)
    }
  }
  Stream.iterate(init)(updater)
    .takeWhile{ case (left, right) => right > left }
    .last._1 // Take the ‘left‘ boundary from the last element.
}
```

This code is clearer because recursion is delegated to `Stream.iterate` and cleanly separated from the "business logic" (i.e. implementing the base case, the inductive step, and the post-processing).

Example 2.5.1.6 For a given positive `n:Int`, compute the sequence $[s_0, s_1, s_2, ...]$ defined by $s_0 = SD(n)$ and $s_k = SD(s_{k-1})$ for $k > 0$, where $SD(x)$ is the sum of the decimal digits of the integer x, e.g. $SD(123) = 6$. Stop the sequence s_i when the numbers begin repeating. For example, $SD(99) = 18$, $SD(18) = 9$, $SD(9) = 9$. So, for $n = 99$, the sequence s_i must be computed as $[99, 18, 9]$.

Hint: use `Stream.iterate`; compute the digits in the reverse order since their sum will be the same.

Solution We need to implement a function `sdSeq` having the type signature

```
def sdSeq(n: Int): Seq[Int]
```

First we need to implement $SD(x)$. The sum of digits is obtained by almost the same code as in Section 2.3:

```
def SD(n: Int): Int = if (n == 0) 0 else
  Stream.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).sum
```

Now we can try evaluating SD on some numbers to see its behavior:

```
scala> (1 to 15).toList.map(SD)
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6)
```

It is clear that $SD(n) < n$ as long as $n \geq 10$. So the sequence elements s_i will not repeat until they become smaller than 10, and then they will always repeat. This seems to be an easy way of stopping the sequence. Let us try that:

```
scala> Stream.iterate(99)(SD).takeWhile(x => x >= 10).toList
res1: List[Int] = List(99, 18)
```

We are missing the last element of the sequence, $SD(18) = 9$, because `.takeWhile` stops the sequence too early. In order to obtain the correct sequence, we need to compute one more element. To fix this, we can generate a stream of *pairs*:

```
scala> Stream.iterate((0, 99)){ case (prev, x) => (x, SD(x)) }.
  takeWhile{ case (prev, x) => prev >= 10 || x >= 10 }.toList
res2: List[(Int, Int)] = List((0,99), (99,18), (18,9))
```

This looks right; it remains to remove the first parts of the tuples:

```
def sdSeq(n: Int): Seq[Int] =
```

```
  Stream.iterate((0, n)){ case (prev, x) => (x, SD(x)) }  // Stream[(Int, Int)]
    .takeWhile{ case (prev, x) => prev >= 10 || x >= 10 } // Stream[(Int, Int)]
    .map(_._2)                                            // Stream[Int]
    .toList                                               // List[Int]

scala> sdSeq(99)
res3: Seq[Int] = List(99, 18, 9)
```

Example 2.5.1.7 Implement a function `unfold` with the type signature

```
def unfold[A](init: A)(next: A => Option[A]): Stream[A]
```

The function should create a stream of values of type `A` with the initial value `init`. Next elements are computed from previous ones via the function `next` until it returns `None`. An example test:

```
scala> unfold(0) { x => if (x > 5) None else Some(x + 2) }
res0: Stream[Int] = Stream(0, ?)

scala> res0.toList
res1: List[Int] = List(0, 2, 4, 6)
```

Solution We can formulate the task as an inductive definition of a stream. If `next(init) == None`, the stream must stop at `init`. (This is the base case of the induction). Otherwise, `next(init) == Some(x)` yields a new value `x` and indicates that we need to continue to "unfold" the stream with `x` instead of `init`. (This is the inductive step.) Streams can be created from individual values via the Scala standard library method `Stream.cons` that constructs a stream from a single value and a tail:

```
def unfold[A](init: A)(next: A => Option[A]): Stream[A] = next(init) match {
  case None      => Stream(init)                    // A stream containing a single value ‘init‘.
  case Some(x)   => Stream.cons(init, unfold(x)(next)) // ‘init‘ followed by the tail of stream.
}
```

Example 2.5.1.8 For a given stream $[s_0, s_1, s_2, ...]$ of type `Stream[T]`, compute the "half-speed" stream $h = [s_0, s_0, s_1, s_1, s_2, s_2, ...]$. The half-speed sequence h is defined as $h_{2k} = h_{2k+1} = s_k$ for $k = 0, 1, 2, ...$

Solution We use `.map` to replace each element s_i by a sequence containing two copies of s_i. Let us try this on a sample sequence:

```
scala> Seq(1,2,3).map( x => Seq(x, x))
res0: Seq[Seq[Int]] = List(List(1, 1), List(2, 2), List(3, 3))
```

The result is almost what we need, except we need to `.flatten` the nested list:

```
scala> Seq(1,2,3).map( x => Seq(x, x)).flatten
res1: Seq[Seq[Int]] = List(1, 1, 2, 2, 3, 3)
```

The composition of `.map` and `.flatten` is `.flatMap`, so the final code is

```
def halfSpeed[T](str: Stream[T]): Stream[T] = str.flatMap(x => Seq(x, x))

scala> halfSpeed(Seq(1,2,3).toStream)
res2: Stream[Int] = Stream(1, ?)

scala> halfSpeed(Seq(1,2,3).toStream).toList
res3: List[Int] = List(1, 1, 2, 2, 3, 3)
```

Example 2.5.1.9 (The **loop detection** problem.) Stop a given stream $[s_0, s_1, s_2, ...]$ at a place k where the sequence repeats itself; that is, an element s_k equals some earlier element s_i with $i < k$.

Solution The trick is to create a half-speed sequence h_i out of s_i and then find an index $k > 0$ such that $h_k = s_k$. (The condition $k > 0$ is needed because we will always have $h_0 = s_0$.) If we find such an index k, it would mean that either $s_k = s_{k/2}$ or $s_k = s_{(k-1)/2}$; in either case, we will have found an element s_k that equals an earlier element.

As an example, for an input sequence $s = [1, 3, 5, 7, 9, 3, 5, 7, 9, ...]$ we obtain the half-speed sequence $h = [1, 1, 3, 3, 5, 5, 7, 7, 9, 9, 3, 3, ...]$. Looking for an index $k > 0$ such that $h_k = s_k$, we find that $s_7 =$

$h_7 = 7$. The element s_7 indeed repeats an earlier element (although s_7 is not the first such repetition).

There are in principle two ways of finding an index $k > 0$ such that $h_k = s_k$: First, to iterate over a list of indices $k = 1, 2, ...$ and evaluate the condition $h_k = s_k$ as a function of k. Second, to build a sequence of pairs (h_i, s_i) and use `.takeWhile` to stop at the required index. In the present case, we cannot use the first way because we do not have a fixed set of indices to iterate over. Also, the condition $h_k = s_k$ cannot be directly evaluated as a function of k because s and h are streams that compute elements on demand, not lists whose elements are computed in advance and ready for use.

So the code must iterate over a stream of pairs (h_i, s_i):

```
def stopRepeats[T](str: Stream[T]): Stream[T] = {
  val halfSpeed = str.flatMap(x => Seq(x, x))
  val result = halfSpeed.zip(str) // Stream[(T, T)]
  .drop(1) // Enforce the condition k > 0.
  .takeWhile { case (h, s) => h != s } // Stream[(T, T)]
  .map(_._2) // Stream[T]
  str.head +: result // Prepend the first element that was dropped.
}

scala> stopRepeats(Seq(1, 3, 5, 7, 9, 3, 5, 7, 9).toStream).toList
res0: List[Int] = List(1, 3, 5, 7, 9, 3, 5)
```

Example 2.5.1.10 Reverse each word in a string, but keep the order of words:

```
def revWords(s: String): String = ???

scala> revWords("A quick brown fox")
res0: String = A kciuq nworb xof
```

Solution The standard method `.split` converts a string into an array of words:

```
scala> "pa re ci vo mu".split(" ")
res0: Array[String] = Array(pa, re, ci, vo, mu)
```

Each word is reversed with `.reverse`; the resulting array is concatenated into a string with `.mkString`:

```
def revWords(s: String): String = s.split(" ").map(_.reverse).mkString(" ")
```

Example 2.5.1.11 Remove adjacent repeated characters from a string:

```
def noDups(s: String): String = ???

scala> noDups("abbcdeeeeefddgggggh")
res0: String = abcdefdgh
```

Solution A string is automatically converted into a sequence of characters when we use methods such as `.map` or `.zip` on it. So, we can use `s.zip(s.tail)` to get a sequence of pairs (s_k, s_{k+1}) where c_k is the k-th character of the string s. A `.filter` will then remove elements s_k for which $s_{k+1} = s_k$:

```
scala> val s = "abbcd"
s: String = abbcd

scala> s.zip(s.tail).filter { case (sk, skPlus1) => sk != skPlus1 }
res0: IndexedSeq[(Char, Char)] = Vector((a,b), (b,c), (c,d))
```

It remains to convert this sequence of pairs into the string `"abcd"`. One way of doing this is to project the sequence of pairs onto the second parts of the pairs,

```
scala> res0.map(_._2).mkString
res1: String = bcd
```

We just need to add the first character, `'a'`. The resulting code is

```
def noDups(s: String): String = if (s == "") "" else {
  val pairs = s.zip(s.tail).filter { case (x, y) => x != y }
  pairs.head._1 +: pairs.map(_._2).mkString
```

```
}
```

The method `+:` prepends an element to a sequence, so `x +: xs` is equivalent to `Seq(x) ++ xs`.

Example 2.5.1.12 For a given sequence of type `Seq[A]`, find the longest subsequence that does not contain any adjacent duplicate values.

```
def longestNoDups[A](xs: Seq[A]): Seq[A] = ???

scala> longestNoDups(Seq(1, 2, 2, 5, 4, 4, 4, 8, 2, 3, 3))
res0: Seq[Int] = List(4, 8, 2, 3)
```

Solution This is a dynamic programming[2] problem. Many such problems are solved with a single `.foldLeft`. The accumulator represents the current "state" of the dynamic programming solution, and the "state" is updated with each new element of the input sequence.

We first need to determine the type of the accumulator value, or the "state". The task is to find the longest subsequence without adjacent duplicates. So the accumulator should represent the longest subsequence found so far, as well as any required extra information about other subsequences that might grow as we iterate over the elements of `xs`. What is that extra information in our case?

Imagine creating the set of *all* subsequences that have no adjacent duplicates. For the input sequence $[1,2,2,5,4,4,4,8,2,3,3]$, this set of all subsequences will be $\{[1,2],[2,5,4],[4,8,2,3]\}$. We can build this set incrementally in the accumulator value of a `.foldLeft`. To visualize how this set would be built, consider the partial result after seeing the first 8 elements of the input sequence, $[1,2,2,5,4,4,4,8]$. The partial set of non-repeating subsequences is $\{[1,2],[2,5,4],[4,8]\}$. As we add another element, 2, we update the partial set to $\{[1,2],[2,5,4],[4,8,2]\}$.

It is now clear that the subsequence $[1,2]$ has no chance of being the longest subsequence, since $[2,5,4]$ is already longer. However, we do not yet know whether $[2,5,4]$ or $[4,8,2]$ is the winner, because the subsequence $[4,8,2]$ could still grow and become the longest one (and it does become $[4,8,2,3]$ later). At this point, we need to keep both of these two subsequences in the accumulator, but we may already discard $[1,2]$.

We have deduced that the accumulator needs to keep only *two* sequences: the first sequence is already terminated and will not grow, the second sequence ends with the current element and may yet grow. The initial value of the accumulator is empty. The first subsequence is discarded when it becomes shorter than the second. The code can be written now:

```
def longestNoDups[A](xs: Seq[A]): Seq[A] = {
  val init: (Seq[A], Seq[A]) = (Seq(), Seq())
  val (first, last) = xs.foldLeft(init) { case ((first, current), x) =>
    // If ‘current‘ is empty, ‘x‘ cannot be repeated.
    val xWasRepeated = current != Seq() && current.last == x
    val firstIsLongerThanCurrent = first.length > current.length
    // Compute the new pair ‘(first, current)‘.
    // Keep ‘first‘ only if it is longer; otherwise replace it by ‘current‘.
    val newFirst = if (firstIsLongerThanCurrent) first else current
    // Append ‘x‘ to ‘current‘ if ‘x‘ is not repeated.
    val newCurrent = if (xWasRepeated) Seq(x) else current :+ x
    (newFirst, newCurrent)
  }
  // Return the longer of the two subsequences; prefer ‘first‘.
  if (first.length >= last.length) first else last
}
```

2.5.2 Exercises

Exercise 2.5.2.1 Compute the sum of squared digits of a given integer; e.g., `dsq(123) = 14` (see Example 2.5.1.6). Generalize the solution to take as an argument an function `f: Int => Int` replacing the squaring operation. The required type signature and a sample test:

[2] https://en.wikipedia.org/wiki/Dynamic_programming

```
def digitsMapSum(x: Int)(f: Int => Int): Int = ???

scala> digitsMap(123){ x => x * x }
res0: Int = 14

scala> digitsMap(123){ x => x * x * x }
res1: Int = 36
```

Exercise 2.5.2.2 Compute the **Collatz sequence** c_i as a stream defined by

$$c_0 = n \quad ; \qquad c_{k+1} = \begin{cases} \frac{c_k}{2} & \text{if } c_k \text{ is even,} \\ 3c_k + 1 & \text{if } c_k \text{ is odd.} \end{cases}$$

Stop the stream when it reaches 1 (as one would expect[3] it will).

Exercise 2.5.2.3 For a given integer n, compute the sum of cubed digits, then the sum of cubed digits of the result, etc.; stop the resulting sequence when it repeats itself, and so determine whether it ever reaches 1. (Use Exercise 2.5.2.1.)

```
def cubes(n: Int): Stream[Int] = ???

scala> cubes(123).take(10).toList
res0: List[Int] = List(123, 36, 243, 99, 1458, 702, 351, 153, 153, 153)

scala> cubes(2).take(10).toList
res1: List[Int] = List(2, 8, 512, 134, 92, 737, 713, 371, 371, 371)

scala> cubes(4).take(10).toList
res2: List[Int] = List(4, 64, 280, 520, 133, 55, 250, 133, 55, 250)

def cubesReach1(n: Int): Boolean = ???

scala> cubesReach1(10)
res3: Boolean = true

scala> cubesReach1(4)
res4: Boolean = false
```

Exercise 2.5.2.4 For `a`, `b`, `c` of type `Set[Int]`, compute the set of all sets of the form `Set(x, y, z)` where `x` is from `a`, `y` from `b`, and `z` from `c`. The required type signature and a sample test:

```
def prod3(a: Set[Int], b: Set[Int], c: Set[Int]): Set[Set[Int]] = ???

scala> prod3(Set(1,2), Set(3), Set(4,5))
res0: Set[Set[Int]] = Set(Set(1,3,4), Set(1,3,5), Set(2,3,4), Set(2,3,5))
```

Hint: use `.flatMap`.

Exercise 2.5.2.5* Same task as in Exercise 2.5.2.4 for a set of sets, i.e. given a `Set[Set[Int]]` instead of just three sets `a`, `b`, `c`. The required type signature and a sample test:

```
def prodSet(si: Set[Set[Int]]): Set[Set[Int]] = ???

scala> prodSet(Set(Set(1,2), Set(3), Set(4,5), Set(6)))
res0: Set[Set[Int]] = Set(Set(1,3,4,6),Set(1,3,5,6),Set(2,3,4,6),Set(2,3,5,6))
```

Hint: use `.foldLeft` and `.flatMap`.

Exercise 2.5.2.6* In a sorted array `xs:Array[Int]` where no values are repeated, find all pairs of values whose sum equals a given number n. Use tail recursion. A type signature and a sample test:

[3] https://en.wikipedia.org/wiki/Collatz_conjecture

```
def pairs(goal: Int, xs: Array[Int]): Set[(Int, Int)] = ???

scala> pairs(10, Array(1, 2, 3, 4, 5, 6, 7, 8))()
res0: Set[(Int, Int)] = Set((2,8), (3,7), (4,6), (5,5))
```

Exercise 2.5.2.7 Reverse a sentence's word order, but keep the words unchanged:

```
def revSentence(s: String): String = ???

scala> revSentence("A quick brown fox")
res0: String = "fox brown quick A"
```

Exercise 2.5.2.8 **(a)** Reverse an integer's digits (see Example 2.5.1.6) as shown:

```
def revDigits(n: Int): Int = ???

scala> revDigits(12345)
res0: Int = 54321
```

(b) A **palindrome integer** is an integer number `n` such that `revDigits(n) == n`. Write a predicate function of type `Int => Boolean` that checks whether a given positive integer is a palindrome.

Exercise 2.5.2.9 Define a function `findPalindrome: Long => Long` performing the following computation: First define `f(n) = revDigits(n) + n` for a given integer `n`, where the function `revDigits` was defined in Exercise 2.5.2.8. If `f(n)` is a palindrome integer, `findPalindrome` returns that integer. Otherwise, it keeps applying the same transformation and computes `f(n)`, `f(f(n))`, ..., until a palindrome integer is eventually found (this is mathematically guaranteed). A sample test:

```
scala> findPalindrome(10101)
res0: Long = 10101

scala> findPalindrome(123)
res0: Long = 444

scala> findPalindrome(83951)
res1: Long = 869363968
```

Exercise 2.5.2.10 Transform a given sequence `xs: Seq[Int]` into a sequence `Seq[(Int, Int)]` of pairs that skip one neighbor. Implement this transformation as a function `skip1` with a type parameter `A` instead of the type `Int`. The required type signature and a sample test:

```
def skip1[A](xs: Seq[A]): Seq[(A, A)] = ???

scala> skip1(List(1,2,3,4,5))
res0: List[Int] = List((1,3), (2,4), (3,5))
```

Exercise 2.5.2.11 **(a)** For a given integer interval $[n_1, n_2]$, find the largest integer $k \in [n_1, n_2]$ such that the decimal representation of k does *not* contain any of the digits 3, 5, or 7. **(b)** For a given integer interval $[n_1, n_2]$, find the integer $k \in [n_1, n_2]$ with the largest sum of decimal digits. **(c)** A positive integer n is called a **perfect number** if it is equal to the sum of its divisors (other integers k such that $k < n$ and n/k is an integer). For example, 6 is a perfect number because its divisors are 1, 2, and 3, and $1 + 2 + 3 = 6$, while 8 is not a perfect number because its divisors are 1, 2, and 4, and $1 + 2 + 4 = 7 \neq 8$. Write a function that determines whether a given number n is perfect. Determine all perfect numbers up to one million.

Exercise 2.5.2.12 Remove adjacent repeated elements from a sequence of type `Seq[A]` when they are repeated more than k times. Repetitions up to k times should remain unchanged. The required type signature and a sample test:

```
def removeDups[A](s: Seq[A], k: Int): Seq[A] = ???

scala> removeDups(Seq(1, 1, 1, 1, 5, 2, 2, 5, 5, 5, 5, 5, 1), 3)
```

```
res0: Seq[Int] = List(1, 1, 1, 5, 2, 2, 5, 5, 5, 1)
```

Exercise 2.5.2.13 Implement a function `unfold2` with the type signature

```
def unfold2[A,B](init: A)(next: A => Option[(A,B)]): Stream[B]
```

The function should create a stream of values of type `B` by repeatedly applying the given function `next` until it returns `None`. At each iteration, `next` should be applied to the value of type `A` returned by the previous call to `next`. An example test:

```
scala> unfold2(0) { x => if (x > 5) None else Some((x + 2, s"had $x")) }
res0: Stream[String] = Stream(had 0, ?)

scala> res0.toList
res1: List[String] = List(had 0, had 2, had 4)
```

Exercise 2.5.2.14* **(a)** Remove repeated elements (whether adjacent or not) from a sequence of type `Seq[A]`. (This re-implements the standard library's method `.distinct`.)

(b) For a sequence of type `Seq[A]`, remove all elements that are repeated (whether adjacent or not) more than k times:

```
def removeK[A](k: Int, xs: Seq[A]): Seq[A] = ???

scala> removeK(2, Seq("a", "b", "a", "b", "b", "c", "b", "a"))
res0: Seq[String] = List(a, b, a, b, c)
```

Exercise 2.5.2.15* For a given sequence `xs:Seq[Double]`, find a subsequence that has the largest sum of values. The sequence `xs` is not sorted, and its values may be positive or negative. The required type signature and a sample test:

```
def maxsub(xs: Seq[Double]): Seq[Double] = ???

scala> maxsub(Seq(1.0, -1.5, 2.0, 3.0, -0.5, 2.0, 1.0, -10.0, 2.0))
res0: Seq[Double] = List(2.0, 3.0, -0.5, 2.0, 1.0)
```

Hint: use dynamic programming and `.foldLeft`.

Exercise 2.5.2.16* Using tail recursion, find all common integers between two *sorted* sequences:

```
@tailrec def commonInt(xs: Seq[Int], ys: Seq[Int]): Seq[Int] = ???

scala> commonInt(Seq(1, 3, 5, 7), Seq(2, 3, 4, 6, 7, 8))
res0: Seq[Int] = List(3, 7)
```

2.6 Discussion

2.6.1 Total and partial functions

In Scala, functions can be total or partial. A **total** function will always compute a result value, while a **partial** function may fail to compute its result for certain values of its arguments.

A simple example of a partial function in Scala is the `.max` method: it only works for non-empty sequences. Trying to evaluate it on an empty sequence generates an error called an "exception":

```
scala> Seq(1).tail
res0: Seq[Int] = List()
scala> res0.max
java.lang.UnsupportedOperationException: empty.max
  at scala.collection.TraversableOnce$class.max(TraversableOnce.scala:229)
  at scala.collection.AbstractTraversable.max(Traversable.scala:104)
  ... 32 elided
```

This kind of error may crash the entire program at run time. Unlike the type errors we saw before, which occur at compilation time (i.e. before the program can start), **run-time errors** occur while the program is running, and only when some partial function happens to get an incorrect input. The incorrect input may occur at any point after the program started running, which may crash the entire program in the middle of a long computation.

So, it seems clear that we should write code that does not generate such errors. For instance, it is safe to apply `.max` to a sequence if we know that it is non-empty.

Sometimes, a function that uses pattern matching turns out to be a partial function because its pattern matching code fails on certain input data.

If a pattern matching expression fails, the code will throw an exception and stop running. In functional programming, we usually want to avoid this situation because it makes it much harder to reason about program correctness. In most cases, programs can be written to avoid the possibility of match errors. An example of an unsafe pattern matching expression is

```
def h(p: (Int, Int)): Int = p match { case (x, 0) => x }

scala> h( (1,0) )
res0: Int = 1

scala> h( (1,2) )
 scala.MatchError: (1,2) (of class scala.Tuple2$mcII$sp)
  at .h(<console>:12)
  ... 32 elided
```

Here the pattern contains a pattern variable `x` and a constant `0`. This pattern only matches tuples whose second part is equal to `0`. If the second argument is nonzero, a match error occurs and the program crashes. So, `h` is a partial function.

Pattern matching failures never happen if we match a tuple of correct size with a pattern such as `(x, y, z)`, because a pattern variable will always match a value. So, pattern matching with a pattern such as `(x, y, z)` is **infallible** (never fails at run time) when applied to a tuple with 3 elements.

Another way in which pattern matching can be made infallible is by including a pattern that matches everything:

```
p match {
  case (x, 0)   => ...   // This only matches some tuples.
  case _        => ...   // This matches everything.
}
```

If the first pattern `(x, 0)` fails to match the value `p`, the second pattern will be tried (and will always succeed). The `case` patterns in a `match` expression are tried in the order they are written. So, a `match` expression may be made infallible by adding a "match-all" underscore pattern.

2.6.2 Scope and shadowing of pattern matching variables

Pattern matching introduces **locally scoped** variables – that is, variables defined only on the right-hand side of the pattern match expression. As an example, consider this code:

```
def f(x: (Int, Int)): Int = x match { case (x, y) => x + y }

scala> f( (2,4) )
res0: Int = 6
```

The argument of `f` is the variable `x` of a tuple type `(Int,Int)`, but there is also a pattern variable `x` in the case expression. The pattern variable `x` matches the first part of the tuple and has type `Int`. Because variables are locally scoped, the pattern variable `x` is only defined within the expression `x + y`. The argument `x:(Int,Int)` is a completely different variable whose value has a different type.

The code works correctly but is confusing to read because of the name clash between the two quite different variables, both named `x`. Another negative consequence of the name clash is that the argument `x:(Int,Int)` *is invisible* within the case expression: if we write "`x`" in that expression, we

will get the pattern variable `x:Int`. One says that the argument `x:(Int,Int)` has been **shadowed** by the pattern variable `x`.

The problem is easy to correct: we can give the pattern variable some other name. Since the pattern variable is locally scoped, it can be renamed within its scope without having to change any other code. An equivalent code is

```
def f(x: (Int, Int)): Int = x match { case (a, b) => a + b }

scala> f( (2,4) )
res0: Int = 6
```

2.6.3 Lazy values and sequences: Iterators and streams

We have used streams to create sequences whose length is not known in advance. An example is a stream containing a sequence of increasing positive integers:

```
scala> val p = Stream.iterate(1)(_ + 1)
p: Stream[Int] = Stream(1, ?)
```

At this point, we have not defined a stopping condition for this stream. In some sense, streams are "infinite" sequences, although in practice a stream is always finite because computers cannot run infinitely long. Also, computers cannot store infinitely many values in memory.

More precisely, streams are "partially computed" rather than "infinite". The main difference between arrays and streams is that a stream's elements are computed on demand and not all initially available, while an array's elements are all computed in advance and are available immediately.

Generally, there are four possible ways a value could be available:

Availability	Explanation	Example Scala code
"eager"	computed in advance	`val z = f(123)`
"lazy"	computed upon first request	`lazy val z = f(123)`
"on-call"	computed each time it is requested	`def z = f(123)`
"never"	cannot be computed due to errors	`val (z, x) = "abc"`

A **lazy value** (declared as `lazy val` in Scala) is computed only when used in some other expression. Once computed, a lazy value stays in memory and will not be re-computed.

An "on-call" value is re-computed every time it is used. In Scala, this is the behavior of a `def` declaration.

Most collection types in Scala (such as `List`, `Array`, `Set`, and `Map`) are **eager**: all elements of an eager collection are already evaluated.

A stream is a **lazy collection**. Elements of a stream are computed when first needed; after that, they remain in memory and will not be computed again:

```
scala> val str = Stream.iterate(1)(_ + 1)
str: Stream[Int] = Stream(1, ?)

scala> str.take(10).toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> str
res1: Stream[Int] = Stream(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ?)
```

In many cases, it is not necessary to keep previous values of a sequence in memory. For example, consider the computation

```
scala> (1L to 1000000000L).sum
res0: Long = 500000000500000000
```

We do not actually need to keep a billion numbers in memory if we only want to

compute their sum. Indeed, the computation just shown does *not* keep all the numbers in memory. The same computation fails if we use a list or a stream:

```
scala> (1L to 1000000000L).toStream.sum
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

The code `(1L to 1000000000L).sum` works because `(1 to n)` produces a sequence whose elements are computed whenever needed but do not remain in memory. This can be seen as a sequence with the "on-call" availability of elements. Sequences of this sort are called **iterators**:

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> 1 until 5
res1: scala.collection.immutable.Range = Range(1, 2, 3, 4)
```

The types `Range` and `Range.Inclusive` are defined in the Scala standard library and are iterators. They behave as collections and support the usual methods (`.map`, `.filter`, etc.), but they do not store previously computed values in memory.

The `.view` method Eager collections such as `List` or `Array` can be converted to iterators by using the `.view` method. This is necessary when intermediate collections consume too much memory when fully evaluated. For example, consider the computation of Example 2.1.5.7 where we used `.flatMap` to replace each element of an initial sequence by three new numbers before computing `.max` of the resulting collection. If instead of three new numbers we wanted to compute *three million* new numbers each time, the intermediate collection created by `.flatMap` would require too much memory, and the computation would crash:

```
scala> (1 to 10).flatMap(x => 1 to 3000000).max
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Even though the range `(1 to 10)` is an iterator, a subsequent `.flatMap` operation creates an intermediate collection that is too large for our computer's memory. We can use `.view` to avoid this:

```
scala> (1 to 10).view.flatMap(x => 1 to 3000000).max
res0: Int = 3000000
```

The choice between using streams and using iterators is dictated by memory constraints. Except for that, streams and iterators behave similarly to other sequences. We may write programs in the map/reduce style, applying standard methods such as `.map`, `.filter`, etc., to streams and iterators. Mathematical reasoning about transforming a sequence is the same, whether the sequence is eager, lazy, or on-call.

The broken `Iterator` class The Scala library contains a class called `Iterator`, which has methods such as `Iterator.iterate` and other methods similar to `Stream`. However, `Iterator` actually not a correct iterator because it cannot be treated as a *value* in the mathematical sense:

```
scala> val iter = (1 until 10).toIterator
iter: Iterator[Int] = non-empty iterator

scala> iter.toList // Look at the elements of ‘iter‘.
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> iter.toList // Look at those elements again...??
res1: List[Int] = List()

scala> iter
res2: Iterator[Int] = empty iterator
```

Evaluating the expression `iter.toList` two times produces a *different* result the second time. As we see, the value `iter` becomes "empty" after the first use.

This situation is impossible in mathematics: if x is some value, such as 100, and f is some function, such as $f(x) = \sqrt{x}$, then $f(x)$ will be the same, $f(100) = \sqrt{100} = 10$, no matter how many times we compute $f(x)$. For instance, we can compute $f(x) + f(x) = 20$ and obtain the correct result. We could also set $y = f(x)$ and compute $y + y = 20$, with the same result. This property is called **referential transparency** or **purity** of the function f.

When we set $x = 100$ and compute $f(x) + f(x)$, the number 100 does not "become empty" after the first use; its value remains the same. This behavior is called the **value semantics** of numbers. One says that integers "are values" in the mathematical sense. Alternatively, one says that numbers are **immutable,** i.e. cannot be changed. (What would it mean to "modify" the number 10?)

In programming, a type has value semantics if a given computation applied to it always gives the same result. Usually, this means that the type contains immutable data, and the computation is referentially transparent. We can see that Scala's `Range` has value semantics and is immutable:

```
scala> val x = 1 until 10
x: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> x.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> x.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Collections such as `List`, `Map`, or `Stream` are immutable. Some elements of a `Stream` may not be evaluated yet, but this does not affect its value semantics:

```
scala> val str = (1 until 10).toStream
str: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> str.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Iterators produced by applying `.view` also have value semantics:

```
scala> val v = (1 until 10).view
v: scala.collection.SeqView[Int,IndexedSeq[Int]] = SeqView(...)

scala> v.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> v.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Due to the lack of value semantics, programs written using `Iterator` may not obey the usual rules of mathematical reasoning. This makes it easy to write wrong code that looks correct.

To illustrate the problem, let us re-implement Example 2.5.1.9 by keeping the same code but using `Iterator` instead of `Stream`:

```
def stopRepeatsBad[T](iter: Iterator[T]): Iterator[T] = {
  val halfSpeed = iter.flatMap(x => Seq(x, x))
  halfSpeed.zip(iter) // Do not prepend the first element. It won't help.
  .drop(1)
  .takeWhile { case (h, s) => h != s }
  .map(_._2)
}

scala> stopRepeatsBad(Seq(1, 3, 5, 7, 9, 3, 5, 7, 9).toIterator).toList
res0: List[Int] = List(5, 9, 3, 7, 9)
```

The result $[5, 9, 3, 7, 9]$ is incorrect, but not in an obvious way: the sequence *was* stopped at a repetition, as we expected, but some of the elements of the given sequence are missing (while other elements are present). It is difficult to debug a program when it produces *partially* correct numbers.

The error in this code occurs in the expression `halfSpeed.zip(iter)` due to the fact that `halfSpeed` was itself defined via `iter`. The result is that `iter` is *used twice* in this code, which leads to errors because `iter` is mutable and does not behave as a value. Creating an `Iterator` and using it twice in the same expression can give wrong results or even fail with an exception:

```
scala> val s = (1 until 10).toIterator
s: Iterator[Int] = non-empty iterator

scala> val t = s.zip(s).toList
java.util.NoSuchElementException: next on empty iterator
```

It is surprising and counter-intuitive that a variable cannot be used twice in some expression. Intuitively, we expect code such as `s.zip(s)` to work correctly even though the variable `s` is used twice. When we read the expression `s.zip(s)`, we imagine a given sequence `s` being "zipped" with itself. So we reason that `s.zip(s)` should produce a sequence of pairs. But Scala's `Iterator` is mutable, which breaks the usual ways of mathematical reasoning about code.

An `Iterator` can be converted to a `Stream` using the `.toStream` method. This restores the value semantics, since streams are values:

```
scala> val iter = (1 until 10).toIterator
iter: Iterator[Int] = non-empty iterator

scala> val str = iter.toStream
str: Stream[Int] = Stream(1, ?)

scala> str.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.zip(str).toList
res2: List[(Int, Int)] = List((1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7), (8,8), (9,9))
```

Instead of `Iterator`, we can use `Stream` and `.view` when lazy or on-call collections are required.

3 The logic of types. I. Disjunctive types

Disjunctive types describe values that belong to a disjoint set of alternatives.

To see how Scala implements disjunctive types, we need to begin by looking at "case classes".

3.1 Scala's case classes

3.1.1 Tuple types with names

It is often helpful to use names for the different parts of a tuple. Suppose that some program represents the size and the color of socks with the tuple type `(Double, String)`. What if the same tuple type `(Double, String)` is used in another place in the program to mean the amount paid and the payee? A programmer could mix the two values by mistake, and it would be hard to find out why the program incorrectly computes, say, the total amount paid.

```
def totalAmountPaid(ps: Seq[(Double, String)]): Double = ps.map(_._1).sum
val x = (10.5, "white")       // Sock size and color.
val y = (25.0, "restaurant")  // Payment amount and payee.

scala> totalAmountPaid(Seq(x, y)) // Nonsense.
res0: Double = 35.5
```

We would prevent this kind of mistake if we could use two *different* types, with names such as `MySock` and `Payment`, for the two kinds of data. There are three basic ways of defining a new named type in Scala: using a type alias, using a class (or "trait"), and using an opaque type.

Opaque types (hiding a type under a new name) is a feature of a future version of Scala 3; so we focus on type aliases and case classes.

A **type alias** is an alternative name for an existing (already defined) type. We could use type aliases in our example to add clarity to the code:

```
type MySockTuple = (Double, String)
type PaymentTuple = (Double, String)

scala> val s: MySockTuple = (10.5, "white")
s: MySockTuple = (10.5,white)

scala> val p: PaymentTuple = (25.0, "restaurant")
p: PaymentTuple = (25.0,restaurant)
```

But type aliases do not prevent mix-up errors:

```
scala> totalAmountPaid(Seq(s, p)) // Nonsense again.
res1: Double = 35.5
```

Scala's **case classes** can be seen as "tuples with names". A case class is equivalent to a tuple type that has a name chosen when we define the case class. Also, each part of the case class will have a separate name that we must choose. This is how to define case classes for the example with socks and payments:

```
case class MySock(size: Double, color: String)
case class Payment(amount: Double, name: String)

scala> val sock = MySock(10.5, "white")
sock: MySock = MySock(10.5,white)
```

```
scala> val paid = Payment(25.0, "restaurant")
paid: Payment = Payment(25.0,restaurant)                                    ^
```

This code defines new types named `MySock` and `Payment`. Values of type `MySock` are written as `MySock(10.5, "white")`, which is similar to writing the tuple `(10.5, "white")` except for adding the name `MySock` in front of the tuple.

To access the parts of a case class, we use the part names:

```
scala> sock.size
res2: Double = 10.5

scala> paid.amount
res3: Double = 25.0
```

The mix-up error is now a type error detected by the compiler:

```
def totalAmountPaid(ps: Seq[Payment]): Double = ps.map(_.amount).sum

scala> totalAmountPaid(Seq(paid, paid))
res4: Double = 50.0

scala> totalAmountPaid(Seq(sock, paid))
<console>:19: error: type mismatch;
 found   : MySock
 required: Payment
       totalAmountPaid(Seq(sock, paid))
                           ^
```

A function whose argument is of type `MySock` cannot be applied to an argument of type `Payment`. Case classes with different names are *different types*, even if they contain the same parts.

Just as tuples can have any number of parts, case classes can have any number of parts, but the part names must be distinct, for example:

```
case class Person(firstName: String, lastName: String, age: Int)

scala> val noether = Person("Emmy", "Noether", 137)
einstein: Person = Person(Emmy,Noether,137)

scala> noether.firstName
res5: String = Emmy

scala> noether.age
res6: Int = 137
```

This data type carries the same information as a tuple `(String, String, Int)`. However, the declaration of a `case class Person` gives the programmer several features that make working with the tuple's data more convenient and less error-prone.

Some (or all) part names may be specified when creating a case class value:

```
scala> val poincaré = Person(firstName = "Henri", lastName = "Poincaré", 165)
poincaré: Person = Person(Henri,Poincaré,165)
```

It is a type error to use wrong types with a case class:

```
scala> val p = Person(140, "Einstein", "Albert")
<console>:13: error: type mismatch;
 found   : Int(140)
 required: String
       val p = Person(140, "Einstein", "Albert")
                      ^
<console>:13: error: type mismatch;
 found   : String("Albert")
 required: Int
       val p = Person(140, "Einstein", "Albert")
                                       ^
```

This error is due to an incorrect order of parts when creating a case class value. However, parts can be specified in any order when using part names:

```
scala> val p = Person(age = 137, lastName = "Noether", firstName = "Emmy")
p: Person = Person(Emmy,Noether,137)
```

A part of a case class can have the type of another case class, creating a type similar to a nested tuple:

```
case class BagOfSocks(sock: MySock, count: Int)
val bag = BagOfSocks(MySock(10.5, "white"), 6)

scala> bag.sock.size
res7: Double = 10.5
```

3.1.2 Case classes with type parameters

Type classes can be defined with type parameters. As an example, consider a generalization of `MySock` where, in addition to the size and color, an "extended sock" holds another value. We could define several specialized case classes,

```
case class MySock_Int(size: Double, color: String, value: Int)
case class MySock_Boolean(size: Double, color: String, value: Boolean)
```

but it is better to define a single parameterized case class

```
case class MySockX[A](size: Double, color: String, value: A)
```

This case class can accommodate every type `A`. We may now create values of `MySockX` containing a `value` of any given type,

```
scala> val s = MySockX(10.5, "white", 123)
s: MySockX[Int] = MySockX(10.5,white,123)
```

Because `123` has type `Int`, the type parameter `A` in `MySockX[A]` was automatically set to the type `Int`.

Each time we create a value of type `MySockX`, a specific type will have to be used instead of the type parameter `A`. In other words, we can only create values of types `MySockX[Int]`, `MySockX[String]`, etc. If we want to be explicit, we may write

```
scala> val s = MySockX[String](10.5, "white", "last pair")
s: MySockX[String] = MySockX(10.5,white,last pair)
```

However, we can write code working with `MySockX[A]` **parametrically**, that is, keeping the type parameter `A` in the code. For example, a function that checks whether a sock of type `MySockX[A]` fits my foot can be written as

```
def fitsMe[A](sock: MySockX[A]): Boolean = (sock.size >= 10.5 && sock.size <= 11.0)
```

This function is defined for all types `A` at once, because its code works in the same way regardless of what `A` is. Scala will set the type parameter automatically when we apply `fitsMe` to an argument:

```
scala> fitsMe(MySockX(10.5, "blue", List(1,2,3))) // Type parameter A = List[Int].
res0: Boolean = true
```

This code forces the type parameter `A` to be `List[Int]`, and so we may omit the type parameter of `fitsMe`. When types become more complicated, it may be helpful to write out some type parameters. The compiler can detect a mismatch between the type parameter `A = List[Int]` used in the "sock" value and the type parameter `A = Int` in the function `fitsMe`:

```
scala> fitsMe[Int](MySockX(10.5, "blue", List(1,2,3)))
<console>:15: error: type mismatch;
 found   : List[Int]
 required: Int
       fitsMe[Int](MySockX(10.5, "blue", List(1,2,3)))
                                         ^
```

Case classes may have several type parameters, and the types of the parts may use these type parameters. Here is an artificial example of a case class using type parameters in different ways,

```
case class Complicated[A,B,C,D](x: (A, A), y: (B, Int) => A, z: C => C)
```

This case class contains parts of different types that use the type parameters `A`, `B`, `C` in tuples and functions. The type parameter `D` is not used at all; this is allowed (and occasionally useful).

A type with type parameters, such as `MySockX` or `Complicated`, is called a **type constructor**. A type constructor "constructs" a new type, such as `MySockX[Int]`, from a given type parameter `Int`. Values of type `MySockX` cannot be created without setting the type parameter. So, it is important to distinguish the type constructor, such as `MySockX`, from a type that can have values, such as `MySockX[Int]`.

3.1.3 Tuples with one part and with zero parts

Let us compare tuples and case classes more systematically.

Parts of a case class are accessed by name with a dot syntax, for example `sock.color`. Parts of a tuple are accessed with the accessors such as `x._1`. This syntax is the same as that for a case class whose parts have names `_1`, `_2`, etc. So, it appears that tuple parts *do* have names in Scala, although those names are always automatically chosen as `_1`, `_2`, etc. Tuple types are also automatically named in Scala as `Tuple2`, `Tuple3`, etc., and they are parameterized, since each part of the tuple may be of any chosen type. A tuple type expression such as `(Int, String)` is just a special syntax for the parameterized type `Tuple2[Int, String]`. One could define the tuple types as case classes like this,

```
case class Tuple2[A, B](_1: A, _2: B)
case class Tuple3[A, B, C](_1: A, _2: B, _3: C)  // And so on with Tuple4, Tuple5...
```

if these types were not already defined in the Scala library.

Proceeding systematically, we ask whether tuple types can have just one part or even no parts. Indeed, Scala defines `Tuple1[A]` as a tuple with a single part. (This type is occasionally used in practice.)

The tuple with zero parts also exists and is called `Unit` (rather than "`Tuple0`"). The syntax for the value of the `Unit` type is the empty tuple, `()`. It is clear that there is *only one* value, `()`, of this type; this explains the name "unit".

At first sight, the `Unit` type may appear to be completely useless: it is a tuple that contains *no data*. It turns out, however, that the `Unit` type is important in functional programming, and it is used as a type *guaranteed* to have only a single distinct value. This chapter will show some examples of using the `Unit` type.

Case classes may have one part or zero parts, similarly to the one-part and zero-part tuples:

```
case class B(z: Int)  // Tuple with one part.
case class C()        // Tuple with no parts.
```

Scala has a special syntax for empty case classes:

```
case object C // Similar to ‘case class C()‘.
```

There are two main differences between `case class C()` and `case object C`:

- A `case object` cannot have type parameters, while we may define, if needed, a `case class C[X, Y, Z]()` with type parameters `X`, `Y`, `Z`.
- A `case object` is allocated in memory only once, while new values of a `case class C()` will be allocated in memory each time `C()` is evaluated.

Other than that, `case class C()` and `case object C` have the same meaning: a named tuple with zero parts, which we may also view as a "named `Unit`" type. In this book, I will not use `case object`s because `case class`es are more general.

Let us summarize the correspondence between tuples and case classes:

Tuples	Case classes
`(123, "xyz"): Tuple2[Int, String]`	`case class A(x: Int, y: String)`
`(123,): Tuple1[Int]`	`case class B(z: Int)`
`(): Unit`	`case class C()`

3.1.4 Pattern matching for case classes

Scala performs pattern matching in two situations:

- destructuring definition: `val` *pattern* `= ...`
- `case` expression: `case` *pattern* `=> ...`

Case classes can be used in both situations. A destructuring definition can be used in a function whose argument is of case class type `BagOfSocks`:

```
case class MySock(size: Double, color: String)
case class BagOfSocks(sock: MySock, count: Int)

def printBag(bag: BagOfSocks): String = {
  val BagOfSocks(MySock(size, color), count) = bag // Destructure the ‘bag‘.
  s"bag has $count $color socks of size $size"
}

val bag = BagOfSocks(MySock(10.5, "white"), 6)

scala> printBag(bag)
res0: String = bag has 6 white socks of size 10.5
```

A `case` expression will destructure a value and compute a result:

```
def fitsMe(bag: BagOfSocks): Boolean = bag match {
  case BagOfSocks(MySock(size, _), _) => (size >= 10.5 && size <= 11.0)
}
```

In the code of this function, we match the `bag` value against the pattern `BagOfSocks(MySock(size, _), _)`. This pattern will always match and will define `size` as a pattern variable of type `Double`.

The syntax for pattern matching expressions with case classes is similar to the syntax for pattern matching of tuples, except for the presence of the *names* of the case classes. For example, removing the case class names from the pattern

```
case BagOfSocks(MySock(size, _), _) => ...
```

we obtain the nested tuple pattern

```
case ((size, _), _) => ...
```

that could be used for values of type `((Double, String), Int)`. We see that within pattern matching expressions, case classes behave exactly as tuple types with added names.

Scala's "case classes" got their name from their use in `case` expressions. It is usually more convenient to use `match` / `case` expressions with case classes than to use destructuring.

3.2 Disjunctive types

3.2.1 Motivation and first examples

In many situations, it is useful to have several different shapes of data within the same type. As a first example, suppose we are looking for real roots of a quadratic equation $x^2 + bx + c = 0$. There are three cases: no real roots, one real root, and two real roots. It is convenient to have a type that

represents "the real roots of a quadratic equation"; call it `RootsOfQ`. Inside that type, we distinguish between the three cases, but outside it looks like a single type.

Another example is the binary search algorithm that looks for an integer x in a sorted array. Either the algorithm finds the index of x, or it determines that the array does not contain x. It is convenient if the algorithm could return a single value of a type (call it `SearchResult`) that represents *either* an index at which x is found, *or* the absence of an index.

More generally, we may have computations that *either* return a value *or* generate an error and fail to produce a result. It is then convenient to return a value of type (call it `Result`) that represents either a correct result or an error message.

In certain computer games, one has different types of "rooms", each room having certain properties depending on its type. Some rooms are dangerous because of monsters, other rooms contain useful objects, certain rooms allow you to finish the game, and so on. We want to represent all the different kinds of rooms uniformly, as a type `Room`, so that a value of type `Room` automatically stores the correct properties in each case.

In all these situations, data comes in several mutually exclusive shapes. This data can be represented by a single type if that type is able to describe a mutually exclusive set of cases:

- `RootsOfQ` must be either the empty tuple `()`, or `Double`, or a tuple `(Double, Double)`
- `SearchResult` must be either `Int` or the empty tuple `()`
- `Result` must be either an `Int` value or a `String` message

We see that the empty tuple, also known as the `Unit` type, is natural to use in these situations. It is also helpful to assign names to each of the cases:

- `RootsOfQ` is "no roots" with value `()`, or "one root" with value `Double`, or "two roots" with value `(Double, Double)`
- `SearchResult` is "index" with value `Int`, or "not found" with value `()`
- `Result` is "value" of type `Int` or "error message" of type `String`

Scala's case classes provide exactly what we need here – named tuples with zero, one, two and more parts, and so it is natural to use case classes instead of tuples:

- `RootsOfQ` is a value of type `case class NoRoots()`, or a value of type `case class OneRoot(x: Double)`, or a value of type `case class TwoRoots(x: Double, y: Double)`
- `SearchResult` is a value of type `case class Index(Int)` or a value of type `case class NotFound()`
- `Result` is a value of type `case class Value(x: Int)` or a value of type `case class Error(message: String)`

Our three examples are now described as types that select one case class out of a given set. It remains to see how Scala defines such types. For instance, the definition of `RootsOfQ` needs to indicate that the case classes `NoRoots`, `OneRoot`, and `TwoRoots` are exactly the three alternatives described by the type `RootsOfQ`. The Scala syntax for that definition looks like this:

```
sealed trait RootsOfQ
final case class NoRoots()                    extends RootsOfQ
final case class OneRoot(x: Double)           extends RootsOfQ
final case class TwoRoots(x: Double, y: Double) extends RootsOfQ
```

In the definition of `SearchResult`, we have two cases:

```
sealed trait SearchResult
final case class Index(i: Int) extends SearchResult
final case class NotFound()    extends SearchResult
```

The definition of the `Result` type is parameterized, so that we can describe results of any type (while error messages are always of type `String`):

```
sealed trait Result[A]
final case class Value[A](x: A)            extends Result[A]
final case class Error[A](message: String) extends Result[A]
```

The "`sealed trait` / `final case class`" syntax defines a type that represents a choice of one case class from a fixed set of case classes. This kind of type is called a **disjunctive type** in this book.

3.2.2 Solved examples: Pattern matching for disjunctive types

Our first examples of disjunctive types are `RootsOfQ`, `SearchResult`, and `Result[A]` defined in the previous section. We will now look at the Scala syntax for creating values of disjunctive types and for using the created values.

Consider the disjunctive type `RootsOfQ` having three case classes (`NoRoots`, `OneRoot`, `TwoRoots`). The only way of creating a value of type `RootsOfQ` is to create a value of one of these case classes. This is done by writing expressions such as `NoRoots()`, `OneRoot(2.0)`, or `TwoRoots(1.0, -1.0)`. Scala will accept these expressions as having the type `RootsOfQ`:

```
scala> val x: RootsOfQ = OneRoot(2.0)
x: RootsOfQ = OneRoot(2.0)
```

Given a value `x:RootsOfQ`, how can we use it, say, as a function argument? The main tool for working with values of disjunctive types is pattern matching with `match` / `case` expressions. In Chapter 2, we used pattern matching to destructure tuples with syntax such as `{ case (x, y) => ... }`. To use `match` / `case` expressions with disjunctive types, we may have to write *more than one* `case` pattern in a `match` expression, because we need to match several possible cases of the disjunctive type:

```
def f(r: RootsOfQ): String = r match {
  case NoRoots()       => "no real roots"
  case OneRoot(r)      => s"one real root: $r"
  case TwoRoots(x, y)  => s"real roots: ($x, $y)"
}

scala> f(x)
res0: String = "one real root: 2.0"
```

If we only need to recognize a specific case of a disjunctive type, we can match all other cases with an underscore:

```
scala> x match {
  case OneRoot(r)   => s"one real root: $r"
  case _            => "have something else"
}
res1: String = one real root: 2.0
```

The `match` / `case` expression represents a choice over possible values of a given type. Note the similarity with this code:

```
def f(x: Int): Int = x match {
  case 0    => println(s"error: must be nonzero"); -1
  case 1    => println(s"error: must be greater than 1"); -1
  case _    => x
}
```

The values `0` and `1` are some possible values of type `Int`, just as `OneRoot(1.0)` is a possible value of type `RootsOfQ`. When used with disjunctive types, `match` / `case` expressions will usually contain a complete list of possibilities. If the list of cases is incomplete, the Scala compiler will print a warning:

```
scala> def g(x: RootsOfQ): String = x match {
         case OneRoot(r) => s"one real root: $r"
       }
```

```
<console>:14: warning: match may not be exhaustive.
It would fail on the following inputs: NoRoots(), TwoRoots(_, _)
       def g(x: RootsOfQ): String = x match {
                                    ^
```

This code defines a *partial* function `g` that can be applied only to values of the form `OneRoot(...)` and will fail for other values.

Let us look at more examples of using the disjunctive types we just defined.

Example 3.2.2.1 Given a sequence of quadratic equations, compute a sequence containing their real roots as values of type `RootsOfQ`.

Solution Define a case class representing a quadratic equation $x^2 + bx + c = 0$:

```
case class QEqu(b: Double, c: Double)
```

The following function determines how many real roots an equation has:

```
def solve(quadraticEqu: QEqu): RootsOfQ = {
   val QEqu(b, c) = quadraticEqu // Destructure QEqu.
   val d = b * b / 4 - c
   if (d > 0) {
     val s = math.sqrt(d)
     TwoRoots(b / 2 - s, b / 2 + s)
   } else if (d == 0.0) OneRoot(b / 2)
   else NoRoots()
}
```

Test the `solve` function:

```
scala> solve(QEqu(1,1))
res1: RootsOfQ = NoRoots()

scala> solve(QEqu(1,-1))
res2: RootsOfQ = TwoRoots(-0.6180339887498949,1.618033988749895)

scala> solve(QEqu(6,9))
res3: RootsOfQ = OneRoot(3.0)
```

We can now implement the function `findRoots`,

```
def findRoots(equs: Seq[QEqu]): Seq[RootsOfQ] = equs.map(solve)
```

If the function `solve` will not be used often, we may want to write it inline as a nameless function:

```
def findRoots(equs: Seq[QEqu]): Seq[RootsOfQ] = equs.map { case QEqu(b, c) =>
  (b * b / 4 - c) match {
    case d if d > 0   =>
      val s = math.sqrt(d)
      TwoRoots(b / 2 - s, b / 2 + s)
    case 0.0          => OneRoot(b / 2)
    case _            => NoRoots()
  }
}
```

This code depends on some features of Scala syntax. We can use the partial function `{ case QEqu(b, c) => ... }` directly as the argument of `.map` instead of defining this function separately. This avoids having to destructure `QEqu` at a separate step. The `if` / `else` expression is replaced by an "embedded" `if` within the `case` expression, which is easier to read. Test the final code:

```
scala> findRoots(Seq(QEqu(1,1), QEqu(2,1)))
res4: Seq[RootsOfQ] = List(NoRoots(), OneRoot(1.0))
```

Example 3.2.2.2 Given a sequence of values of type `RootsOfQ`, compute a sequence containing only the single roots. Example test:

```
def singleRoots(rs: Seq[RootsOfQ]): Seq[Double] = ???
```

```
scala> singleRoots(Seq(TwoRoots(-1, 1), OneRoot(3.0), OneRoot(1.0), NoRoots()))
res5: Seq[Double] = List(3.0, 1.0)
```

Solution We apply `.filter` and `.map` to the sequence of roots:

```
def singleRoots(rs: Seq[RootsOfQ]): Seq[Double] = rs.filter {
  case OneRoot(x) => true
  case _          => false
}.map { case OneRoot(x) => x }
```

In the `.map` operation, we need to cover only the one-root case because the other two possibilities have been "filtered out" by the preceding `.filter` operation.

Example 3.2.2.3 Implement binary search returning a `SearchResult`. We will modify the binary search implementation from Example 2.5.1.5(b) so that it returns a `NotFound` value when appropriate.

Solution The code from Example 2.5.1.5(b) will return *some* index even if the given number is not present in the array:

```
scala> binSearch(Array(1, 3, 5, 7), goal = 5)
res6: Int = 2

scala> binSearch(Array(1, 3, 5, 7), goal = 4)
res7: Int = 1
```

When the number is not present, the array's element at the computed index will not be equal to `goal`. We should return `NotFound()` in that case. The new code can be written as a `match` / `case` expression for clarity:

```
def safeBinSearch(xs: Seq[Int], goal: Int): SearchResult =
  binSearch(xs, goal) match {
    case n if xs(n) == goal   => Index(n)
    case _                    => NotFound()
  }
```

To test:

```
scala> safeBinSearch(Array(1, 3, 5, 7), 5)
res8: SearchResult = Index(2)

scala> safeBinSearch(Array(1, 3, 5, 7), 4)
res9: SearchResult = NotFound()
```

Example 3.2.2.4 Use the disjunctive type `Result[Int]` to implement "safe integer arithmetic", where a division by zero or a square root of a negative number will give an error message. Define arithmetic operations directly for values of type `Result[Int]`. When errors occur, abandon further computations.

Solution Begin by implementing the square root:

```
def sqrt(r: Result[Int]): Result[Int] = r match {
  case Value(x) if x >= 0  => Value(math.sqrt(x).toInt)
  case Value(x)            => Error(s"error: sqrt($x)")
  case Error(m)            => Error(m) // Keep the error message.
}
```

The square root is computed only if we have the `Value(x)` case, and only if $x \geq 0$. If the argument `r` was already an `Error` case, we keep the error message and perform no further computations.

To implement the addition operation, we need a bit more work:

```
def add(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) => Value(x + y)
  case (Error(m), _)        => Error(m) // Keep the error message.
  case (_, Error(m))        => Error(m)
}
```

This code illustrates nested patterns that match the tuple `(rx, ry)` against various possibilities. In this way, the code is clearer than code written with nested `if` / `else` expressions.

Implementing the multiplication operation results in almost the same code:

```
def mul(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) => Value(x * y)
  case (Error(m), _)        => Error(m)
  case (_, Error(m))        => Error(m)
}
```

To avoid repetition, we may define a general function that "lifts" operations on integers to operations on `Result[Int]` types:

```
def do2(rx: Result[Int], ry: Result[Int])(op: (Int, Int) => Int): Result[Int] =
  (rx, ry) match {
    case (Value(x), Value(y)) => Value(op(x, y))
    case (Error(m), _)        => Error(m)
    case (_, Error(m))        => Error(m)
  }
```

Now we can easily "lift" any binary operation on integers to a binary operation on `Result[Int]`, assuming that the operation never generates an error:

```
def sub(rx: Result[Int], ry: Result[Int]): Result[Int] = do2(rx, ry){ (x, y) => x - y }
```

Custom code is still needed for operations that *may* generate errors:

```
def div(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) if y != 0  => Value(x / y)
  case (Value(x), Value(y))            => Error(s"error: $x / $y")
  case (Error(m), _)                   => Error(m)
  case (_, Error(m))                   => Error(m)
}
```

We can now test the new "safe arithmetic" on simple calculations:

```
scala> add(Value(1), Value(2))
res10: Result[Int] = Value(3)

scala> div(add(Value(1), Value(2)), Value(0))
res11: Result[Int] = Error(error: 3 / 0)
```

We see that indeed all further computations are abandoned once an error occurs. An error message shows only the immediate calculation that generated the error. For instance, the error message for $20 + 1/0$ never mentions 20:

```
scala> add(Value(20), div(Value(1), Value(0)))
res12: Result[Int] = Error(error: 1 / 0)

scala> add(sqrt(Value(-1)), Value(10))
res13: Result[Int] = Error(error: sqrt(-1))
```

3.2.3 Standard disjunctive types: `Option`, `Either`, `Try`

The Scala library defines the disjunctive types `Option`, `Either`, and `Try` because they are used often. We now look at each of them in turn.

The `Option` type is a disjunctive type with two cases: the empty tuple and a one-element tuple. The names of the two case classes are `None` and `Some`. If the `Option` type were not already defined in the standard library, one could define it with the code

```
sealed trait Option[T]
final case object None         extends Option[Nothing]
final case class Some[T](t: T) extends Option[T]
```

This code is similar to the type `SearchResult` defined in Section 3.2.1, except that `Option` has a type parameter instead of a fixed type `Int`. Another difference is the use of a `case object` for the empty case instead of an empty case class, such as `None()`. Since Scala's `case object`s cannot have type parameters, the type parameter in the definition of `None` must be set to the special type `Nothing`, which is a type with *no* values (also called the **void type**).

An alternative (implemented in libraries such as `scalaz`) is to define the empty option value as

```
final case class None[T]() extends Option[T]
```

In that implementation, the empty option `None[T]()` has a type parameter.

Several consequences follow from the Scala library's decision to define `None` without a type parameter. One consequence is that `None` can be reused as a value of type `Option[A]` for any type `A`:

```
scala> val y: Option[Int] = None
y: Option[Int] = None

scala> val z: Option[String] = None
z: Option[String] = None
```

Typically, `Option` is used in situations where a value may be either present or missing, especially when a missing value is *not an error*. The missing-value case is represented by `None`, while `Some(x)` means that a value `x` is present.

Example 3.2.3.1 Suppose that information about subscribers to a certain online service must contain a name and an email address, but a telephone number is optional. To represent this information, we may define a case class like this,

```
case class Subscriber(name: String, email: String, phone: Option[Long])
```

What if we represent the missing telephone number by a special value such as `-1` and use the simpler type `Long` instead of `Option[Long]`? The disadvantage is that we would need to *remember* to check for the special value `-1` in all functions that take the telephone number as an argument. Looking at a function such as `sendSMS(phone: Long)` at a different place in the code, a programmer might forget that the telephone number is actually optional. In contrast, the type signature `sendSMS(phone: Option[Long])` unambiguously indicates that the telephone number might be missing and helps the programmer to remember to handle both cases.

Pattern-matching code involving `Option` can handle the two cases like this:

```
def getDigits(phone: Option[Long]): Option[Seq[Long]] = phone match {
  case None               => None       // Do nothing.
  case Some(number)       => Some(digitsOf(number))
}
```

Here we used the function `digitsOf` defined in Section 2.3.

At the two sides of `case None => None`, the value `None` has different types, namely `Option[Long]` and `Option[Seq[Long]]`. Since these types are declared in the type signature of the function `getDigits`, the Scala compiler is able to figure out the types of all expressions in the `match` / `case` construction. So, pattern-matching code can be written without explicit type annotations such as `(None: Option[Long])`.

If we now need to compute the number of digits, we can write

```
def numberOfDigits(phone: Option[Long]): Option[Long] = getDigits(phone) match {
  case None               => None       // Do nothing.
  case Some(digits)       => Some(digits.length)
}
```

These examples perform a computation when an `Option` value is non-empty, and leave it empty otherwise. This design pattern is used often. To avoid repeating the code, we can implement this design pattern as a function that takes the computation as an argument `f`:

```
def doComputation(x: Option[Long], f: Long => Long): Option[Long] = x match {
  case None               => None       // Do nothing.
  case Some(i)            => Some(f(i))
```

```
}
```

It is then natural to generalize this function to arbitrary types using type parameters instead of a fixed type `Long`. The resulting function is usually called `fmap`:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None               => None       // Do nothing.
  case Some(a)            => Some(f(a))
}

scala> fmap(digitsOf)(Some(4096))
res0: Option[Seq[Long]] = Some(List(4, 0, 9, 6))

scala> fmap(digitsOf)(None)
res1: Option[Seq[Long]] = None
```

We say that the `fmap` operation **lifts** a given function of type `A => B` to the type `Option[A] => Option[B]`.

It is important to keep in mind that the code `case Some(a) => Some(f(a))` changes the type of the option value. On the left side of the arrow, the type is `Option[A]`, while on the right side it is `Option[B]`. The Scala compiler knows this from the given type signature of `fmap`, so an explicit type parameter, `Some[B](f(a))`, is not needed.

The Scala library implements an equivalent function as a method on the `Option` class, with the syntax `x.map(f)` rather than `fmap(f)(x)`. We can concisely rewrite the previous code using the standard library methods as

```
def getDigits(phone: Option[Long]): Option[Seq[Long]] = phone.map(digitsOf)
def numberOfDigits(phone: Option[Long]): Option[Long] = phone.map(digitsOf).map(_.length)
```

We see that the `.map` operation for the `Option` type is analogous to the `.map` operation for sequences.

The similarity between `Option[A]` and `Seq[A]` is clearer if we view `Option[A]` as a special kind of "sequence" whose length is restricted to be either 0 or 1. So, `Option[A]` can have all the operations of `Seq[A]` except operations such as `.concat` that may grow the sequence beyond length 1. The standard operations defined on `Option` include `.map`, `.filter`, `.forall`, `.exists`, `.flatMap`, and `.foldLeft`.

Example 3.2.3.2 Given a phone number as `Option[Long]`, extract the country code if it is present. (Assume that the country code is any digits in front of the 10-digit number; for the phone number 18004151212, the country code is 1.) The result must be again of type `Option[Long]`.

Solution If the phone number is a positive integer n, we may compute the country code simply as `n/10000000000L`. However, if the result of that division is zero, we should return an empty `Option` (i.e. the value `None`) rather than `0`. To implement this logic, we may begin by writing this code,

```
def countryCode(phone: Option[Long]): Option[Long] = phone match {
  case None      => None
  case Some(n)   =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
}
```

We may notice that we have reimplemented the design pattern similar to `.map` in this code, namely "if `None` then return `None`, else do a computation". So we may try to rewrite the code as

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
} // Type error: the result is Option[Option[Long]], not Option[Long].
```

This code does not compile: we are returning an `Option[Long]` within a function lifted via `.map`, so the resulting type is `Option[Option[Long]]`. We may use `.flatten` to convert `Option[Option[Long]]` to the required type `Option[Long]`,

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
```

```
}.flatten // Types are correct now.
```

Since the `.flatten` follows a `.map`, we can rewrite the code using `.flatMap`:

```
def countryCode(phone: Option[Long]): Option[Long] = phone.flatMap { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
} // Types are correct now.
```

Another way of implementing this example is to notice the design pattern "if condition does not hold, return `None`, otherwise keep the value". For an `Option` type, this is equivalent to the `.filter` operation (recall that `.filter` returns an empty sequence if the predicate never holds). The code is

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map(_ / 10000000000L).filter(_ != 0L)
```

Test it:

```
scala> countryCode(Some(18004151212L))
res0: Option[Long] = Some(1)

scala> countryCode(Some(8004151212L))
res1: Option[Long] = None
```

Example 3.2.3.3 Add a new requirement to Example 3.2.3.2: if the country code is not present, we should return the default country code 1.

Solution This is an often used design pattern: "if empty, substitute a default value". The Scala library has the method `.getOrElse` for this purpose:

```
scala> Some(100).getOrElse(1)
res2: Int = 100

scala> None.getOrElse(1)
res3: Int = 1
```

So we can implement the new requirement as

```
scala> countryCode(Some(8004151212L)).getOrElse(1L)
res4: Long = 1
```

Using `Option` with collections Many Scala library methods return an `Option` as a result. The main examples are `.find`, `.headOption`, and `.lift` for sequences, and `.get` for dictionaries.

The `.find` method returns the first element satisfying a predicate:

```
scala> (1 to 10).find(_ > 5)
res0: Option[Int] = Some(6)

scala> (1 to 10).find(_ > 10) // No element is > 10.
res1: Option[Int] = None
```

The `.lift` method returns the element of a sequence at a given index:

```
scala> (10 to 100).lift(0)
res2: Option[Int] = Some(10)

scala> (10 to 100).lift(1000) // No element at index 1000.
res3: Option[Int] = None
```

The `.headOption` method returns the first element of a sequence, unless the sequence is empty. This is equivalent to `.lift(0)`:

```
scala> Seq(1,2,3).headOption
res4: Option[Int] = Some(1)

scala> Seq(1,2,3).filter(_ > 10).headOption
res5: Option[Int] = None
```

Applying `.find(p)` computes the same result as `.filter(p).headOption`, but `.find(p)` may be faster.

The `.get` method for a dictionary returns the value if it exists for a given key, and returns `None` if the key is not in the dictionary:

```
scala> Map(10 -> "a", 20 -> "b").get(10)
res6: Option[String] = Some(a)

scala> Map(10 -> "a", 20 -> "b").get(30)
res7: Option[String] = None
```

The `.get` method provides safe by-key access to dictionaries, unlike the direct access method that may fail:

```
scala> Map(10 -> "a", 20 -> "b")(10)
res8: String = a

scala> Map(10 -> "a", 20 -> "b")(30)
java.util.NoSuchElementException: key not found: 30
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  ... 32 elided
```

Similarly, `.lift` provides safe by-index access to collections, unlike the direct access that may fail:

```
scala> Seq(10,20,30)(0)
res9: Int = 10

scala> Seq(10,20,30)(5)
java.lang.IndexOutOfBoundsException: 5
  at scala.collection.LinearSeqOptimized$class.apply(LinearSeqOptimized.scala:65)
  at scala.collection.immutable.List.apply(List.scala:84)
  ... 32 elided
```

The `Either` type The standard disjunctive type `Either[A, B]` has two type parameters and is often used for computations that report errors. By convention, the *first* type (`A`) is the type of error, and the *second* type (`B`) is the type of the (non-error) result. The names of the two cases are `Left` and `Right`. A possible definition of `Either` may be written as

```
sealed trait Either[A, B]
final case class  Left[A, B](value: A) extends Either[A, B]
final case class Right[A, B](value: B) extends Either[A, B]
```

By convention, a value `Left(x)` represents an error, and a value `Right(y)` represents a valid result.

As an example, the following function substitutes a default value and logs the error information:

```
def logError(x: Either[String, Int], default: Int): Int = x match {
  case Left(error)  => println(s"Got error: $error"); default
  case Right(res)   => res
}
```

To test:

```
scala> logError(Right(123), -1)
res1: Int = 123

scala> logError(Left("bad result"), -1)
Got error: bad result
res2: Int = -1
```

Why use `Either` instead of `Option` for computations that may fail? A failing computation such as `"xyz".toInt` cannot return a result, and sometimes we might use `None` to indicate that a result is not available. However, when the result is a requirement for further calculations, we will usually need to know exactly *which* error prevented the result from being available. The `Either` type may provide detailed information about such errors, which `Option` cannot do.

The `Either` type generalizes the type `Result` defined in Section 3.2.1 to an arbitrary error type instead

of `String`. We have seen its usage in Example 3.2.2.4, where the design pattern was "if value is present, do a computation, otherwise keep the error". This design pattern is implemented by the `.map` method on `Either`:

```
scala> Right(1).map(_ + 1)
res0: Either[Nothing, Int] = Right(2)

scala> Left[String, Int]("error").map(_ + 1)
res1: Either[String, Int] = Left("error")
```

The type `Nothing` was filled in by the Scala compiler because we did not specify the first type parameter of `Right` in line 1.

The methods `.filter`, `.flatMap`, `.fold`, and `.getOrElse` are also defined for `Either`, with the same convention that a `Left` value represents an error.[1]

Exceptions and the `Try` type When computations fail for any reason, Scala generates an **exception** instead of returning a value. An exception means that the evaluation of some expression was stopped without returning a result.

As an example, exceptions are generated when the available memory is too small to store the resulting data (as we saw in Section 2.6.3), or if a stack overflow occurs during the computation (as we saw in Section 2.2.3). Exceptions may also occur due to programmer's error: when a pattern matching operation fails, when a requested key does not exist in a dictionary, or when the `.head` operation is applied to an empty list.

Motivated by these examples, we may distinguish "planned" and "unplanned" exceptions.

A **planned** exception is generated by programmer's code via the `throw` syntax:

```
scala> throw new Exception("This is a test... this is only a test.")
java.lang.Exception: This is a test... this is only a test.
```

The Scala library contains a `throw` operation in various places, such as in the code for applying the `.head` method to an empty sequence, as well as in other situations where exceptions are generated due to programmer's errors. These exceptions are generated deliberately and in well-defined situations. Although these exceptions indicate errors, these errors are anticipated in advance and so may be handled by the programmer.

For example, many Java libraries will generate exceptions when function arguments have unexpected values, when a network operation takes too long or a network connection is unexpectedly broken, when a file is not found or cannot be read due to access permissions, and in many other situations. All these exceptions are "planned" because they are generated explicitly by library code such as `throw new FileNotFoundException(...)`. The programmer's code is expected to catch these exceptions, to handle the error, and to continue the evaluation of the program.

An **unplanned** exception is generated by the Java runtime system when critical errors occur, such as an out-of-memory error. It is rare that a programmer writes `val y = f(x)` while *expecting* that an out-of-memory exception will sometimes occur at that point.[2] An unplanned exception indicates a serious and unforeseen problem with memory or another critically important resource, such as the operating system's threads or file handles. Such problems usually cannot be fixed and will prevent the program from running any further. It is reasonable that the program evaluation should abruptly stop (or "crash" as programmers say) after such an error.

The use of planned exceptions assumes that the programmer will write code to handle each exception. This assumption makes it significantly harder to write programs correctly: it is hard to figure out and to keep in mind all the possible exceptions that a given library function may `throw` in its code and in the code of all other libraries on which it depends. Instead of using exceptions for indicating errors, Scala programmers can write functions that return a disjunctive type such as `Either`, describing both possibilities (a correct result or an error condition). Users of these functions will *have* to do pattern matching on the result values. This helps programmers to remember and to

[1] These methods are available in Scala 2.12 or a later version.

[2] Just once in the author's experience, an out-of-memory condition had to be anticipated in an Android app.

handle all relevant error situations.

However, programmers will often need to use Java or Scala libraries that `throw` exceptions. To help write code for these situations, the Scala library contains a helper function called `Try()` and a disjunctive type also called `Try`. The type `Try[A]` can be seen as similar to `Either[Throwable, A]`, where `Throwable` is the general type of all exceptions (i.e. values to which a `throw` operation can be applied). The two parts of the disjunctive type `Try[A]` are called `Failure` and `Success[A]` (instead of `Left[Throwable]` and `Right[A]` in the `Either` type). The function `Try(expr)` will catch all exceptions thrown while the expression `expr` is evaluated. If the evaluation of `expr` succeeds and returns a value `x:A`, the value of `Try(expr)` will be `Success(x)`. Otherwise it will be `Failure(t)`, where `t:Throwable` is the value associated with the generated exception. Here is an example of using `Try`:

```
import scala.util.{Try,Success,Failure}

scala> Try("xyz".toInt)
res0: Try[Int] = Failure(java.lang.NumberFormatException: For input string: "xyz")

scala> Try("0002".toInt)
res1: Try[Int] = Success(2)
```

The code `Try("xyz".toInt)` does not generate any exceptions and will not crash the program. Any computation that may `throw` an exception can be enclosed in a `Try()`, and the exception will be caught and encapsulated within the disjunctive type as a `Failure(...)` value.

The methods `.map`, `.filter`, `.flatMap`, `.foldLeft` are defined for the `Try` class similarly to the `Either` type. One additional feature of `Try` is to catch exceptions generated by the function arguments of `.map`, `.filter`, `.flatMap`, and other standard methods:

```
scala> val y = x.map(y => throw new Exception("ouch"))
y: Try[Int] = Failure(java.lang.Exception: ouch)

scala> val z = x.filter(y => throw new Exception("huh"))
z: Try[Int] = Failure(java.lang.Exception: huh)
```

In this example, the values `y` and `z` were computed *successfully* even though exceptions were thrown while the function arguments of `.map` and `.filter` were evaluated. Other code can use pattern matching on the values `y` and `z` and examine those exceptions. However, it is important that these exceptions were caught and the program did not crash, so the other code is *able* to run.

While the standard types `Try` and `Either` will cover many use cases, programmers can also define custom disjunctive types in order to represent all the anticipated failures or errors in the business logic of a particular application. Representing all errors in the types helps assure that the program will not crash because of an exception that we forgot to handle or did not even know about.

3.3 Lists and trees: recursive disjunctive types

Consider this code defining a disjunctive type `NInt`:

```
sealed trait NInt
final case class N1(x: Int)  extends NInt
final case class N2(n: NInt) extends NInt
```

The type `NInt` has two disjunctive parts, `N1` and `N2`. But the definition of the case class `N1` refers to the type `NInt` as if it were already defined.

A type whose definition uses that same type is called a **recursive type**. The type `NInt` is an example of a recursive disjunctive type.

We might imagine defining a disjunctive type `X` whose parts recursively refer to the same type `X` (and/or to each other) in complicated ways. What would kind of data would be represented by such a type `X`, and in what situations would `X` be useful? In general, this question is not easy to answer. For instance, the simple definition

```
final case class Bad(x: Bad)
```

is useless: we can create a value of type `Bad` only if we already have a value of type `Bad`. This is an example of an infinite type recursion. We will never be able to create any values of type `Bad`, which

means that the type `Bad` is void (has no values, like the the special type `Nothing`).

Chapter 14 studies recursive types in more detail. For now, we will look at the main examples of recursive disjunctive types that are *known* to be useful. These examples are lists and trees.

3.3.1 Lists

A list of values of type `A` is either empty, or has one value of type `A`, or two values of type `A`, etc. We can visualize the type `List[A]` as a disjunctive type defined by

```
sealed trait List[A]
final case class List0[A]()             extends List[A]
final case class List1[A](x: A)         extends List[A]
final case class List2[A](x1: A, x2: A) extends List[A]
???             // Need an infinitely long definition.
```

However, this definition is not practical: we cannot define a separate case class for *each* possible length. Instead, we define the type `List[A]` via mathematical induction on the length of the list:

- Base case: empty list, `case class List0[A]()`.
- Inductive step: given a list of a previously defined length, say $\texttt{List}_{n-1}$, define a new case class $\texttt{List}_n$ describing a list with one more element of type `A`. So we could define $\texttt{List}_n = (\texttt{List}_{n-1}, \texttt{A})$.

Let us try to write this inductive definition as code:

```
sealed trait ListI[A]         // Inductive definition of a list.
final case class List0[A]()                     extends ListI[A]
final case class List1[A](prev: List0[A], x: A) extends ListI[A]
final case class List2[A](prev: List1[A], x: A) extends ListI[A]
???                // Still need an infinitely long definition.
```

To avoid writing an infinitely long type definition, we need to use a trick. Notice that all definitions of `List1`, `List2`, etc., have a similar form (while `List0` is not similar). We can replace all the definitions `List1`, `List2`, etc., by a single definition if we use the type `ListI[A]` recursively inside the case class:

```
sealed trait ListI[A]         // Inductive definition of a list.
final case class List0[A]()                     extends ListI[A]
final case class ListN[A](prev: ListI[A], x: A) extends ListI[A]
```

The type definition has become recursive. For this trick to work, it is important to use `ListI[A]` and not `ListN[A]` inside the definition `ListN[A]`; or else we would have created an infinite type recursion similar to `case class Bad` shown previously.

Since we obtained the type definition of `ListI` via a trick, let us verify that the code actually defines the disjunctive type we wanted.

To create a value of type `ListI[A]`, we must use one of the two available case classes. Using the first case class, we may create a value `List0()`. Since this empty case class does not contain any values of type `A`, it effectively represents an empty list (the base case of the induction). Using the second case class, we may create a value `ListN(prev, x)` where `x` is of type `A` and `prev` is some previously constructed value of type `ListI[A]`. This represents the inductive step, because the case class `ListN` is a named tuple containing `ListI[A]` and `A`. Now, the same consideration recursively applies to constructing the value `prev`, which must be either an empty list or a pair containing another list and an element of type `A`. The assumption that the value `prev:ListI[A]` is already constructed is equivalent to the inductive assumption that we already have a list of a previously defined length. So, we have verified that `ListI[A]` implements the inductive definition shown above.

Examples of values of type `ListI` are the empty list `List0()`, a one-element list `ListN(List0(), x)`, and a two-element list `ListN(ListN(List0(), x), y)`.

To illustrate writing pattern-matching code using this type, let us implement the method `headOption`:

```
@tailrec def headOption[A]: ListI[A] => Option[A] = {
  case List0()              => None
```

```
  case ListN(List0(), x)     => Some(x)
  case ListN(prev, _)        => headOption(prev)
}
```

The Scala library already defines the type `List[A]` in an equivalent but different way: its case classes are named differently, and the second case class (with the special name `::`) places the value of type `A` *before* the previously constructed list,

```
sealed trait List[A]
final case object Nil extends List[Nothing]
final case class ::[A](head: A, tail: List[A]) extends List[A]
```

Because "operator-like" case class names, such as `::`, support the infix syntax, we may write `head :: tail` instead of `::(head, tail)`. Pattern matching with the standard `List` class looks like this:

```
def headOption[A]: List[A] => Option[A] = {
  case Nil            => None
  case head :: tail   => Some(head)
}
```

Examples of values created using Scala's standard `List` type are the empty list `Nil`, a one-element list `x :: Nil`, and a two-element list `x :: y :: Nil`. The same syntax `x :: y :: Nil` is used both for creating values of type `List` and for pattern-matching on such values.

The Scala library also defines the helper function `List()`, so that `List()` is the same as `Nil` and `List(1, 2, 3)` is the same as `1 :: 2 :: 3 :: Nil`. Lists are easier to use in the syntax `List(1, 2, 3)`. Pattern matching can also use that syntax when convenient:

```
val x: List[Int] = List(1, 2, 3)

x match {
  case List(a)         => ...
  case List(a, b, c)   => ...
  case _               => ...
}
```

3.3.2 Tail recursion with `List`

Because the `List` type is defined by induction, it is straightforward to implement iterative computations with the `List` type using recursion.

A first example is the `map` function. We use reasoning by induction in order to figure out the implementation of `map`. The required type signature is

```
def map[A, B](xs: List[A])(f: A => B): List[B] = ???
```

The base case is an empty list, and we return again an empty list:

```
def map[A, B](xs: List[A])(f: A => B): List[B] = xs match {
  case Nil => Nil
  ...
```

In the inductive step, we have a pair `(head, tail)` in the case class `::`, with `head:A` and `tail:List[A]`. The pair can be pattern-matched with the syntax `head :: tail`. The `map` function should apply the argument `f` to the head value, which will give the first element of the resulting list. The remaining elements are computed by the induction assumption, i.e. by a recursive call to `map`:

```
def map[A, B](xs: List[A])(f: A => B): List[B] = xs match {
  case Nil           => Nil
  case head :: tail  => f(head) :: map(tail)(f) // Not tail-recursive.
```

While this implementation is straightforward and concise, it is not tail-recursive. This will be a problem for large enough lists.

Instead of implementing the often-used methods such as `.map` or `.filter` one by one, let us implement `foldLeft` because most of the other methods can be expressed via `foldLeft`.

The required type signature is

```
def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R = ???
```

Reasoning by induction, we start with the base case `xs == Nil`, where the only possibility is to return the value `init`:

```
def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R = xs match {
    case Nil            => init
    ...
```

The inductive step for `foldLeft` says that, given the values `head:A` and `tail:List[A]`, we need to apply the updater function to the previous accumulator value. That value is `init`. So we apply `foldLeft` recursively to the tail of the list once we have the updated accumulator value:

```
@tailrec def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R =
  xs match {
    case Nil            => init
    case head :: tail   =>
      val newInit = f(init, head) // Update the accumulator.
      foldLeft(tail)(newInit)(f)  // Recursive call to ‘foldLeft‘.
  }
```

This implementation is tail-recursive because the recursive call to `foldLeft` is the last expression returned in a `case` branch.

Another example is a function for reversing a list. The Scala library defines the `.reverse` method for this task, but we will show an implementation using `foldLeft`. The updater function *prepends* an element to a previous list:

```
def reverse[A](xs: List[A]): List[A] =
  xs.foldLeft(Nil: List[A])((prev, x) => x :: prev)

scala> reverse(List(1, 2, 3))
res0: List[Int] = List(3, 2, 1)
```

Without the explicit type annotation `Nil:List[A]`, the Scala compiler will decide that `Nil` has type `List[Nothing]`, and the types will not match later in the code. In Scala, one often finds that the initial value for `.foldLeft` needs an explicit type annotation.

The `reverse` function can be used to obtain a tail-recursive implementation of `map` for `List`. The idea is to first use `foldLeft` to accumulate transformed elements:

```
scala> Seq(1, 2, 3).foldLeft(Nil:List[Int])((prev, x) => x*x :: prev)
res0: List[Int] = List(9, 4, 1)
```

The result is a reversed `.map(x => x*x)`, so we need to apply `.reverse`:

```
def map[A, B](xs: List[A])(f: A => B): List[B] =
  xs.foldLeft(Nil: List[B])((prev, x) => f(x) :: prev).reverse

scala> map(List(1, 2, 3))(x => x*x)
res2: List[Int] = List(1, 4, 9)
```

This achieves stack safety at the cost of traversing the list twice. (This implementation is shown only as an example. The Scala library implements `.map` for `List` using low-level tricks in order to achieve better performance.)

Example 3.3.2.1 A definition of the **non-empty list** is similar to `List` except that the empty-list case is replaced by a 1-element case:

```
sealed trait NEL[A]
final case class Last[A](head: A)               extends NEL[A]
final case class More[A](head: A, tail: NEL[A]) extends NEL[A]
```

Values of a non-empty list look like this:

```
scala> val xs: NEL[Int] = More(1, More(2, Last(3)))  // [1, 2, 3]
xs: NEL[Int] = More(1,More(2,Last(3)))

scala> val ys: NEL[String] = Last("abc") // One element, ["abc"].
ys: NEL[String] = Last(abc)
```

To create non-empty lists more easily, we implement a conversion function `toNEL` from an ordinary list. To guarantee that a non-empty list can be created, we give `toNEL` *two* arguments:

```
def toNEL[A](x: A, rest: List[A]): NEL[A] = rest match {
  case Nil         => Last(x)
  case y :: tail   => More(x, toNEL(y, tail))
}   // Not tail-recursive: ‘toNEL()‘ is used inside ‘More(...)‘.
```

To test:

```
scala> toNEL(1, List()) // Result = [1].
res0: NEL[Int] = Last(1)

scala> toNEL(1, List(2, 3)) // Result = [1, 2, 3].
res1: NEL[Int] = More(1,More(2,Last(3)))
```

The `head` method is safe for non-empty lists, unlike `.head` for an ordinary `List`:

```
def head[A]: NEL[A] => A = {
    case Last(x)        => x
    case More(x, _)     => x
}
```

We can also implement a tail-recursive `foldLeft` function for non-empty lists:

```
@tailrec def foldLeft[A, R](n: NEL[A])(init: R)(f: (R, A) => R): R = n match {
    case Last(x)        => f(init, x)
    case More(x, tail)  => foldLeft(tail)(f(init, x))(f)
  }

scala> foldLeft(More(1, More(2, Last(3))))(0)(_ + _)
res2: Int = 6
```

Example 3.3.2.2 Use `foldLeft` to implement a `reverse` function for the type `NEL`. The required type signature and a sample test:

```
def reverse[A]: NEL[A] => NEL[A] = ???

scala> reverse(toNEL(10, List(20, 30))) // Result must be [30, 20, 10].
res3: NEL[Int] = More(30,More(20,Last(10)))
```

Solution We will use `foldLeft` to build up the reversed list as the accumulator value. It remains to choose the initial value of the accumulator and the updater function. We have already seen the code for reversing the ordinary list via the `.foldLeft` method (Section 3.3.2),

```
def reverse[A](xs: List[A]): List[A] = xs.foldLeft(Nil: List[A])((prev,x) => x :: prev)
```

However, we cannot reuse the same code for non-empty lists by writing `More(x, prev)` instead of `x :: prev`, because the `foldLeft` operation works with non-empty lists differently. Since lists are always non-empty, the updater function is always applied to an initial value, and the code works incorrectly:

```
def reverse[A](xs: NEL[A]): NEL[A] =
  foldLeft(xs)(Last(head(xs)):NEL[A])((prev,x) => More(x, prev))

scala> reverse(toNEL(10,List(20,30))) // Result = [30, 20, 10, 10].
res4: NEL[Int] = More(30,More(20,More(10,Last(10))))
```

The last element, `10`, should not have been repeated. It was repeated because the initial accumulator value already contained the head element `10` of the original list. However, we cannot set the initial

accumulator value to an empty list, since a value of type `NEL[A]` must be non-empty. It seems that we need to handle the case of a one-element list separately. So we begin by matching on the argument of `reverse`, and apply `foldLeft` only when the list is longer than 1 element:

```
def reverse[A]: NEL[A] => NEL[A] = {
    case Last(x)         => Last(x)    // ‘reverse‘ is trivial.
    case More(x, tail)   =>            // Use foldLeft on ‘tail‘.
      foldLeft(tail)(Last(x):NEL[A])((prev,x) => More(x, prev))
  }

scala> reverse(toNEL(10, List(20, 30))) // Result = [30, 20, 10].
res5: NEL[Int] = More(30,More(20,Last(10)))
```

Exercise 3.3.2.3 Implement a function `toList` that converts a non-empty list into an ordinary Scala `List`. The required type signature and a sample test:

```
def toList[A](nel: NEL[A]): List[A] = ???

scala> toList(More(1, More(2, Last(3)))) // This is [1, 2, 3].
res6: List[Int] = List(1, 2, 3)
```

Exercise 3.3.2.4 Implement a `map` function for the type `NEL`. Type signature and a sample test:

```
def map[A,B](xs: NEL[A])(f: A => B): NEL[B] = ???

scala> map[Int, Int](toNEL(10, List(20, 30)))(_ + 5) // Result = [15, 25, 35].
res7: NEL[Int] = More(15,More(25,Last(35)))
```

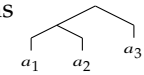
Exercise 3.3.2.5 Implement a function `concat` that concatenates two non-empty lists:

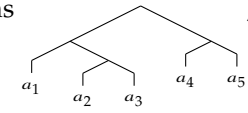
```
def concat[A](xs: NEL[A], ys: NEL[A]): NEL[A] = ???

scala> concat(More(1, More(2, Last(3))), More(4, Last(5))) // Result is [1, 2, 3, 4, 5].
res8: NEL[Int] = More(1,More(2,More(3,More(4,Last(5)))))
```

3.3.3 Binary trees

We will consider four kinds of trees defined as recursive disjunctive types: binary trees, rose trees, regular-shaped trees, and abstract syntax trees.

Examples of a **binary tree** with leaves of type `A` can be drawn as 

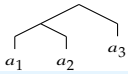
or as 

,

where a_i are some values of type `A`.

An inductive definition says that a binary tree is either a leaf with a value of type `A` or a branch containing *two* previously defined binary trees. Translating this definition into code, we get

```
sealed trait Tree2[A]
final case class Leaf[A](a: A)                       extends Tree2[A]
final case class Branch[A](x: Tree2[A], y: Tree2[A]) extends Tree2[A]
```

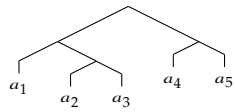
With this definition, the tree 

is created by the code expression

```
Branch(Branch(Leaf("a1"), Leaf("a2")), Leaf("a3"))
```

while the expression

```
Branch(Branch(Leaf("a1"),Branch(Leaf("a2"), Leaf("a3"))), Branch(Leaf("a4"), Leaf("a5")))
```

creates the tree 

.

Recursive functions on trees are translated into concise code. For instance, the function `foldLeft` for trees of type `Tree2` is implemented as

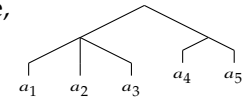
```
def foldLeft[A, R](t: Tree2[A])(init: R)(f: (R, A) => R): R = t match {
  case Leaf(a)          => f(init, a)
  case Branch(t1, t2)   =>
    val r1 = foldLeft(t1)(init)(f) // Fold the left branch.
    foldLeft(t2)(r1)(f) // Starting from ‘r1‘, fold the right branch.
}
```

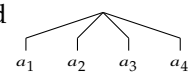
Note that this function *cannot* be made tail-recursive using the accumulator trick, because `foldLeft` needs to call itself twice in the `Branch` case. To test:

```
val t: Tree2[String] = Branch(Branch(Leaf("a1"), Leaf("a2")), Leaf("a3"))

scala> foldLeft(t)("")(_ + " " + _)
res0: String = " a1 a2 a3"
```

3.3.4 Rose trees

A **rose tree** is similar to the binary tree except the branches contain a non-empty list of trees. Because of that, a rose tree can fork into arbitrarily many branches at each node, rather than always into two branches as the binary tree does; for example, 

and 

are rose trees.

A possible definition of a data type for the rose tree is

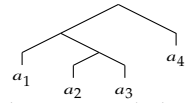
```
sealed trait TreeN[A]
final case class Leaf[A](a: A)                extends TreeN[A]
final case class Branch[A](ts: NEL[TreeN[A]]) extends TreeN[A]
```

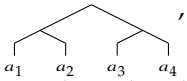
Exercise 3.3.4.1 Define the function `foldLeft` for a rose tree, using `foldLeft` for the type `NEL`. Type signature and a test:

```
def foldLeft[A, R](t: TreeN[A])(init: R)(f: (R, A) => R): R = ???

scala> foldLeft(Branch(More(Leaf(1), More(Leaf(2), Last(Leaf(3))))))(0)(_ + _)
res0: Int = 6
```

3.3.5 Regular-shaped trees

Binary trees and rose trees may choose to branch or not to branch at any given node, resulting in structures that may have different branching depths at different nodes, such as 

. A **regular-shaped tree** always branches in the same way at every node until a chosen total depth, e.g. 

, where all nodes at depth 0 and 1 always branch into two, while nodes at depth 2 do not branch. The branching number is fixed for a given type of a regular-shaped tree; in this example, the branching number is 2, so it is a regular-shaped *binary* tree.

How can we define a data type representing a regular-shaped binary tree? We need a tree that is either a single value, or a pair of values, or a pair of pairs, etc. Begin with the non-recursive (but, of course, impractical) definition

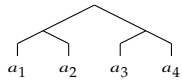
```
sealed trait RTree[A]
final case class Leaf[A](x: A)                    extends RTree[A]
final case class Branch1[A](xs: (A, A))           extends RTree[A]
final case class Branch2[A](xs: ((A, A),(A, A))) extends RTree[A]
???                         // Need an infinitely long definition.
```

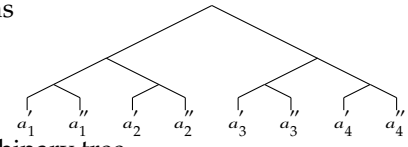
The case `Branch1` describes a regular-shaped tree with total depth 1, the case `Branch2` has total depth 2, and so on. Now, we cannot rewrite this definition as a recursive type because the case classes do not have the same structure. The non-trivial trick is to notice that each $\texttt{Branch}_n$ case class uses the previous case class's data structure *with the type parameter* set to `(A, A)` instead of `A`. So we can rewrite this definition as

```
sealed trait RTree[A]
final case class Leaf[A](x: A)                    extends RTree[A]
final case class Branch1[A](xs: Leaf[(A, A)])     extends RTree[A]
final case class Branch2[A](xs: Branch1[(A, A)]) extends RTree[A]
???                         // Need an infinitely long definition.
```

We can now apply the type recursion trick: replace the type $\texttt{Branch}_{n-1}\texttt{[(A, A)]}$ in the definition of $\texttt{Branch}_n$ by the type `RTree[(A, A)]` . This gives the type definition for a regular-shaped binary tree:

```
sealed trait RTree[A]
final case class Leaf[A](x: A)                 extends RTree[A]
final case class Branch[A](xs: RTree[(A, A)]) extends RTree[A]
```

Since we used some tricks to figure out the definition of `RTree[A]`, let us verify that this definition actually describes the recursive disjunctive type we wanted. The only way to create a structure of type `RTree[A]` is to create a `Leaf[A]` or a `Branch[A]`. A value of type `Leaf[A]` is a correct regularly-shaped tree. It remains to consider the case of `Branch[A]`. Creating a `Branch[A]` requires a previously created `RTree` with values of type `(A, A)` instead of `A`. By the inductive assumption, the previously created `RTree[A]` would have the correct shape. Now, it is clear that if we replace the type parameter `A` by the pair `(A, A)`, a regular-shaped tree such as 

remains regular-shaped but becomes one level deeper, which can be drawn (replacing each a_i by a pair a_i', a_i'') as 

.

We see that `RTree[A]` is the correct definition of a regular-shaped binary tree.

Example 3.3.5.1 Define a (non-tail-recursive) `map` function for a regular-shaped binary tree. The required type signature and a test:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = ???

scala> map(Branch(Branch(Leaf(((1,2),(3,4))))))(_ * 10)
res0: RTree[Int] = Branch(Branch(Leaf(((10,20),(30,40)))))
```

Solution Begin by pattern-matching on the tree:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => ???
  case Branch(xs)   => ???
}
```

In the base case, we have no choice but to return `Leaf(f(x))`.

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => ???
}
```

In the inductive step, we are given a previous tree value `xs:RTree[(A, A)]`. It is clear that we need to apply `map` recursively to `xs`. Let us try:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => Branch(map(xs)(f))    // Type error!
}
```

Here, `map(xs)(f)` has an incorrect type of the function `f`. Since `xs` has type `RTree[(A, A)]`, the recursive call `map(xs)(f)` requires `f` to be of type `((A, A)) => (B, B)` instead of `A => B`. So, we need to provide a function of the correct type instead of `f`. A function of type `((A, A)) => (B, B)` will be obtained out of `f: A => B` if we apply `f` to each part of the tuple `(A, A)`; the code for that function is `{ case (x, y) => (f(x), f(y)) }`. Therefore, we can implement `map` as

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => Branch(map(xs){ case (x, y) => (f(x), f(y)) })
}
```

Exercise 3.3.5.2 Using tail recursion, compute the depth of a regular-shaped binary tree of type `RTree`. (An `RTree` of depth n has 2^n leaf values.) The required type signature and a test:

```
@tailrec def depth[A](t: RTree[A]): Int = ???

scala> depth(Branch(Branch(Leaf((("a","b"),("c","d"))))))
res2: Int = 2
```

Exercise 3.3.5.3* Define a tail-recursive function `foldLeft` for a regular-shaped binary tree. The required type signature and a test:

```
@tailrec def foldLeft[A, R](t: RTree[A])(init: R)(f: (R, A) => R): R = ???

scala> foldLeft(Branch(Branch(Leaf(((1,2),(3,4))))))(0)(_ + _)
res0: Int = 10

scala> foldLeft(Branch(Branch(Leaf((("a","b"),("c","d"))))))("")(_ + _)
res1: String = abcd
```

3.3.6 Abstract syntax trees

Expressions in formal languages are represented by abstract syntax trees. An **abstract syntax tree** (or **AST** for short) is defined as either a leaf of one of the available leaf types, or a branch of one of the available branch types. All the available leaf and branch types must be specified as part of the definition of an AST. In other words, one must specify the data carried by leaves and branches, as well as the branching numbers.

To illustrate how ASTs are used, let us rewrite Example 3.2.2.4 via an AST. We view Example 3.2.2.4 as a small sub-language that deals with "safe integers" and supports the "safe arithmetic" operations `Sqrt`, `Add`, `Mul`, and `Div`. Example calculations in this sub-language are $\sqrt{16} * (1 + 2) = 12$; $20 + 1/0 =$ error; and $10 + \sqrt{-1} =$ error.

We can implement this sub-language in two stages. The first stage will create a data structure (an AST) that represents an unevaluated expression in the sub-language. The second stage will evaluate that AST to obtain either a number or an error message.

A straightforward way of defining a data structure for an AST is to use a disjunctive type whose cases describe all the possible operations of the sub-language. We will need one case class for each of `Sqrt`, `Add`, `Mul`, and `Div`. An additional operation, `Num`, will lift ordinary integers into "safe integers". So, we define the disjunctive type for "arithmetic sub-language expressions" as

```
sealed trait Arith
final case class Num(x: Int)             extends Arith
final case class Sqrt(x: Arith)          extends Arith
final case class Add(x: Arith, y: Arith) extends Arith
```

```
final case class Mul(x: Arith, y: Arith) extends Arith
final case class Div(x: Arith, y: Arith) extends Arith
```

A value of type `Arith` is either a `Num(x)` for some integer `x`, or an `Add(x, y)` where `x` and `y` are previously defined `Arith` expressions, or another operation.

This type definition is similar to the binary tree type

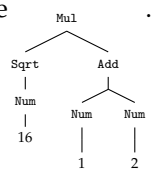
```
sealed trait Tree
final case class Leaf(x: Int)             extends Tree
final case class Branch(x: Tree, y: Tree) extends Tree
```

if we rename `Leaf` to `Num` and `Branch` to `Add`. However, the `Arith` type contains four different types of "branches", some with branching number 1 and others with branching number 2.

This example illustrates the structure of an AST: it is a tree of a general shape, where leaves and branches are chosen from a specified set of allowed possibilities. In this example, we have a single allowed type of leaf (`Num`) and four allowed types of branches (`Sqrt`, `Add`, `Mul`, and `Div`).

This completes the first stage of implementing the sub-language of "safe arithmetic". We may now use the disjunctive type `Arith` to create expressions in the sub-language. For example, $\sqrt{16} * (1 + 2)$ is represented by

```
scala> val x: Arith = Mul(Sqrt(Num(16)), Add(Num(1), Num(2)))
x: Arith = Mul(Sqrt(Num(16)),Add(Num(1),Num(2)))
```

We can visualize `x` as the abstract syntax tree 

.

The expressions $20 + 1/0$ and $10 * \sqrt{-1}$ are represented by

```
scala> val y: Arith = Add(Num(20), Div(Num(1), Num(0)))
y: Arith = Add(Num(20),Div(Num(1),Num(0)))

scala> val z: Arith = Add(Num(10), Sqrt(Num(-1)))
z: Arith = Add(Num(10),Sqrt(Num(-1)))
```

As we see, the expressions `x`, `y`, and `z` *remain unevaluated*; each of them is a data structure that encodes a tree of operations of the sub-language. These operations will be evaluated at the second stage of implementing the sub-language.

To evaluate expressions in the "safe arithmetic", we can implement a function with type signature `run: Arith => Either[String, Int]`. That function plays the role of an **interpreter** or "**runner**" for programs written in the sub-language. The runner will destructure the expression tree and execute all the operations, taking care of possible errors.

To implement `run`, we need to define required arithmetic operations on the type `Either[String, Int]`. For instance, we need to be able to add or multiply values of that type. Instead of custom code from Example 3.2.2.4, we can use the standard `.map` and `.flatMap` methods defined on `Either`. For example, addition and multiplication of two "safe integers" is implemented as

```
def add(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.map(r2 => r1 + r2) }
def mul(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.map(r2 => r1 * r2) }
```

while the "safe division" is

```
def div(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.flatMap(r2 =>
  if (r2 == 0) Left(s"error: $r1 / $r2") else Right(r1 / r2) )
}
```

With this code, we can implement the runner as

```
def run: Arith => Either[String, Int] = {
  case Num(x)     => Right(x)
  case Sqrt(x)    => run(x).flatMap { r =>
    if (r < 0) Left(s"error: sqrt($r)") else Right(math.sqrt(r).toInt)
  }
  case Add(x, y)  => add(run(x), run(y))
  case Mul(x, y)  => mul(run(x), run(y))
  case Div(x, y)  => div(run(x), run(y))
}
```

Test it with the values `x`, `y`, `z` defined previously:

```
scala> run(x)
res0: Either[String, Int] = Right(12)

scala> run(y)
res1: Either[String, Int] = Left("error: 1 / 0")

scala> run(z)
res2: Either[String, Int] = Left("error: sqrt(-1)")
```

3.4 Summary

What problems can we solve now?

- Represent values from a disjoint domain by a custom-defined disjunctive type.
- Use disjunctive types instead of exceptions to indicate failures.
- Use standard disjunctive types `Option`, `Try`, `Either` and methods defined for them.
- Define recursive disjunctive types (e.g. lists and trees) and implement recursive functions that work with them.

The following examples and exercises illustrate these tasks.

3.4.1 Solved examples

Example 3.4.1.1 Define a disjunctive type `DayOfWeek` representing the seven days.

Solution Since there is no information other than the label on each day, we use empty case classes:

```
sealed trait DayOfWeek
final case class Sunday()    extends DayOfWeek
final case class Monday()    extends DayOfWeek
final case class Tuesday()   extends DayOfWeek
final case class Wednesday() extends DayOfWeek
final case class Thursday()  extends DayOfWeek
final case class Friday()    extends DayOfWeek
final case class Saturday()  extends DayOfWeek
```

This data type is analogous to an enumeration type in C or C++:

```
typedef enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday } DayOfWeek;
```

Example 3.4.1.2 Modify `DayOfWeek` so that the values additionally represent a restaurant name and total amount for Fridays and a wake-up time on Saturdays.

Solution For the days where additional information is given, we use non-empty case classes:

```
sealed trait DayOfWeekX
final case class Sunday()                                  extends DayOfWeekX
final case class Monday()                                  extends DayOfWeekX
final case class Tuesday()                                 extends DayOfWeekX
final case class Wednesday()                               extends DayOfWeekX
final case class Thursday()                                extends DayOfWeekX
final case class Friday(restaurant: String, amount: Int)   extends DayOfWeekX
final case class Saturday(wakeUpAt: java.time.LocalTime)   extends DayOfWeekX
```

This data type is no longer equivalent to an enumeration type.

Example 3.4.1.3 Define a disjunctive type that describes the real roots of the equation $ax^2 + bx + c = 0$, where a, b, c are arbitrary real numbers.

Solution Begin by solving the equation and enumerating all the possible cases. It may happen that $a = b = c = 0$, and then all x are roots. If $a = b = 0$ but $c \neq 0$, the equation is $c = 0$, which has no roots. If $a = 0$ but $b \neq 0$, the equation becomes $bx + c = 0$, having a single root. If $a \neq 0$ and $b^2 > 4ac$, we have two distinct real roots. If $a \neq 0$ and $b^2 = 4ac$, we have one real root. If $b^2 < 4ac$, we have no real roots. The resulting type definition can be written as

```
sealed trait RootsOfQ2
final case class AllRoots()                       extends RootsOfQ2
final case class ConstNoRoots()                   extends RootsOfQ2
final case class Linear(x: Double)                extends RootsOfQ2
final case class NoRealRoots()                    extends RootsOfQ2
final case class OneRootQ(x: Double)              extends RootsOfQ2
final case class TwoRootsQ(x: Double, y: Double)  extends RootsOfQ2
```

This disjunctive type contains six parts, among which three parts are empty tuples and two parts are single-element tuples; but this is not a useless redundancy. We would lose information if we reused `Linear` for the two cases $a = 0, b \neq 0$ and $a \neq 0, b^2 = 4ac$, or if we reused `NoRoots()` for representing all three different no-roots cases.

Example 3.4.1.4 Define a function `rootAverage` that computes the average value of all real roots of a general quadratic equation, where the set of roots is represented by the type `RootsOfQ2` defined in Example 3.4.1.3. The required type signature is

```
val rootAverage: RootsOfQ2 => Option[Double] = ???
```

The function should return `None` if the average is undefined.

Solution The average is defined only in cases `Linear`, `OneRootQ`, and `TwoRootsQ`. In all other cases, we must return `None`. We implement this via pattern matching:

```
val rootAverage: RootsOfQ2 => Option[Double] = roots => roots match {
    case Linear(x)       => Some(x)
    case OneRootQ(x)     => Some(x)
    case TwoRootsQ(x, y) => Some((x + y) * 0.5)
    case _               => None
  }
```

We do not need to enumerate all other cases since the underscore (_) matches everything that the previous cases did not match.

The often-used code pattern of the form `x => x match { case ... }` can be shortened to the nameless function syntax `{ case ...}`. The code then becomes

```
val rootAverage: RootsOfQ2 => Option[Double] = {
  case Linear(x)       => Some(x)
  case OneRootQ(x)     => Some(x)
  case TwoRootsQ(x, y) => Some((x + y) * 0.5)
  case _               => None
}
```

Test it:

```
scala> Seq(NoRealRoots(),OneRootQ(1.0), TwoRootsQ(1.0, 2.0), AllRoots()).map(rootAverage)
res0: Seq[Option[Double]] = List(None, Some(1.0), Some(1.5), None)
```

Example 3.4.1.5 Generate 100 quadratic equations $x^2 + bx + c = 0$ with random coefficients b, c (uniformly distributed between −1 and 1) and compute the mean of the largest real roots from all these equations.

Solution Use the type `QEqu` and the `solve` function from Example 3.2.2.1. A sequence of equations with random coefficients is created by applying the method `Seq.fill`:

```
def random(): Double = scala.util.Random.nextDouble() * 2 - 1
val coeffs: Seq[QEqu] = Seq.fill(100)(QEqu(random(), random()))
```

Now we can use the `solve` function to compute all roots:

```
val solutions: Seq[RootsOfQ] = coeffs.map(solve)
```

For each set of roots, compute the largest root:

```
scala> val largest: Seq[Option[Double]] = solutions.map {
  case OneRoot(x)     => Some(x)
  case TwoRoots(x, y) => Some(math.max(x, y))
  case _              => None
}
largest: Seq[Option[Double]] = List(None, Some(0.9346072365885472), Some(1.1356234869160806),
    Some(0.9453181931646322), Some(1.1595052441078866), None, Some(0.5762252742788)...
```

It remains to remove the `None` values and to compute the mean of the resulting sequence. The Scala library defines the `.flatten` method that removes `None`s and transforms `Seq[Option[A]]` into `Seq[A]`:

```
scala> largest.flatten
res0: Seq[Double] = List(0.9346072365885472, 1.1356234869160806, 0.9453181931646322,
    1.1595052441078866, 0.5762252742788...
```

Now compute the mean of the last sequence. Since the `.flatten` operation is preceded by `.map`, we can replace it by a `.flatMap`. The final code is

```
val largest = Seq.fill(100)(QEqu(random(), random()))
  .map(solve)
  .flatMap {
    case OneRoot(x)     => Some(x)
    case TwoRoots(x, y) => Some(math.max(x, y))
    case _              => None
  }

scala> largest.sum / largest.size
res1: Double = 0.7682649774589514
```

Example 3.4.1.6 Implement a function with type signature

```
def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = ???
```

The function should preserve information as much as possible.

Solution Begin by pattern matching on the argument:

```
1 def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
2   case None                 => ???
3   case Some(eab:Either[A, B])  => ???
4 }
```

In line 3, we wrote the **type annotation** `:Either[A, B]` only for clarity; it is not required here because the Scala compiler can deduce the type of the pattern variable `eab` from the fact that we are matching a value of type `Option[Either[A, B]]`.

In the scope of line 2, we need to return a value of type `Either[A, Option[B]]`. A value of that type must be either a `Left(x)` for some `x:A`, or a `Right(y)` for some `y:Option[B]`, where `y` must be either `None` or `Some(z)` with a `z:B`. However, in our case the code is of the form `case None => ???`, and we cannot produce any values `x:A` or `z:B` since `A` and `B` are arbitrary, unknown types. The only remaining

possibility is to return `Right(y)` with `y = None`, and so the code must be

```
...
case None => Right(None) // No other choice here.
```

In the next scope, we can perform pattern matching on the value `eab`:

```
...
case Some(eab: Either[A, B]) = eab match {
  case Left(a)   => ???
  case Right(b)  => ???
}
```

It remains to figure out what expressions to write in each case. In the case `Left(a) => ???`, we have a value of type `A`, and we need to compute a value of type `Either[A, Option[B]]`. We execute the same argument as before: The return value must be `Left(x)` for some `x:A`, or `Right(y)` for some `y:Option[B]`. At this point, we have a value of type `A` but no values of type `B`. So we have two possibilities: to return `Left(a)` or to return `Right(None)`. If we decide to return `Left(a)`, the code is

```
1 def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
2   case None       => Right(None) // No other choice here.
3   case Some(eab)  => eab match {
4     case Left(a)  => Left(a)   // Could return Right(None) here.
5     case Right(b) => ???
6   }
7 }
```

Let us decide whether to return `Left(a)` or `Right(None)` in line 4. Both choices will satisfy the required return type `Either[A, Option[B]]`. However, if we return `Right(None)` in that line, we will ignore the given value `a:A`, i.e. we will lose information. So we return `Left(a)` in line 4.

Reasoning similarly for line 5, we find that we may return `Right(None)` or `Right(Some(b))`. The first choice ignores the given value of `b:B`. To preserve information, we make the second choice:

```
1 def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
2   case None       => Right(None)
3   case Some(eab)  => eab match {
4       case Left(a)    => Left(a)
5       case Right(b)   => Right(Some(b))
6   }
7 }
```

We can now refactor this code into a somewhat more readable form by using nested patterns:

```
def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
  case None             => Right(None)
  case Some(Left(a))    => Left(a)
  case Some(Right(b))   => Right(Some(b))
}
```

Example 3.4.1.7 Implement a function with the type signature

```
def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = ???
```

The function should preserve information as much as possible.

Solution Begin by pattern matching on the argument:

```
1 def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
2   case (Some(a), Some(b)) => ???
3   ...
```

In line 2, we have values `a:A` and `b:B`, and we need to compute a value of type `Option[(A, B)]`. A value of that type is either `None` or `Some((x, y))` where we would need to obtain values `x:A` and `y:B`. Since `A` and `B` are arbitrary types, we cannot produce new values `x` and `y` from scratch. The only way of obtaining `x:A` and `y:B` is to set `x = a` and `y = b`. So, our choices are to return `Some((a, b))` or `None`. We reject returning `None` since that would unnecessarily lose information. Thus, we continue writing code as

```
1 def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
2   case (Some(a), Some(b)) => Some((a, b))
3   case (Some(a), None)    => ???
4   ...
```

In line 3, we have a value `a:A` but no values of type `B`. Since the type `B` is arbitrary, we cannot produce any values of type `B` to return a value of the form `Some((x, y))`. So, `None` is the only computable value of type `Option[(A, B)]` in line 3. We continue to write the code:

```
1 def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
2   case (Some(a), Some(b)) => Some((a, b))
3   case (Some(a), None)    => None // No other choice here.
4   case (None, Some(b))    => ???
5   case (None, None)       => ???
6 }
```

In lines 4–5, we find that there is no choice other than returning `None`. So we can simplify the code:

```
def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
  case (Some(a), Some(b)) => Some((a, b))
  case _                  => None // No other choice here.
}
```

3.4.2 Exercises

Exercise 3.4.2.1 Define a disjunctive type `CellState` representing the visual state of one cell in the *Minesweeper*[3] game: A cell can be closed (showing nothing), or show a bomb, or be open and show the number of bombs in neighbor cells.

Exercise 3.4.2.2 Define a function from `Seq[Seq[CellState]]` to `Int`, counting the total number of cells with zero neighbor bombs shown.

Exercise 3.4.2.3 Define a disjunctive type `RootOfLinear` representing all possibilities for the solution of the equation $ax + b = 0$ for arbitrary real a, b. (The possibilities are: no roots; one root; all x are roots.) Implement the solution as a function `solve1` with type signature

```
def solve1: ((Double, Double)) => RootOfLinear = ???
```

Exercise 3.4.2.4 Given a `Seq[(Double, Double)]` containing pairs (a, b) of the coefficients of $ax + b = 0$, produce a `Seq[Double]` containing the roots of that equation when a unique root exists. Use the type `RootOfLinear` and the function `solve1` defined in Exercise 3.4.2.3.

Exercise 3.4.2.5 The case class `Subscriber` was defined in Example 3.2.3.1. Given a `Seq[Subscriber]`, compute the sequence of email addresses for all subscribers that did *not* provide a phone number.

Exercise 3.4.2.6* In this exercise, a "procedure" is a function of type `Unit => Unit`; an example of a procedure is `{ () => println("hello") }`. Define a disjunctive type `Proc` for an abstract syntax tree representing three operations on procedures: 1) `Func[A](f)`, create a procedure from a function `f` of type `Unit => A`, where `A` is a type parameter. (Note that the type `Proc` does not have any type parameters.) 2) `Sequ(p1, p2)`, execute two procedures sequentially. 3) `Para(p1, p2)`, execute two procedures in parallel. Then implement a "runner" that converts a `Proc` into a `Future[Unit]`, running the computations either sequentially or in parallel as appropriate. Test with this code:

```
sealed trait Proc; final case class ??? // etc.
def runner: Proc => Future[Unit] = ???
val proc1: Proc = Func{_ => Thread.sleep(200); println("hello1")}
val proc2: Proc = Func{_ => Thread.sleep(400); println("hello2")}

scala> runner(Sequ(Para(proc2, proc1), proc2))
hello1
hello2
hello2
```

[3] https://en.wikipedia.org/wiki/Minesweeper_(video_game)

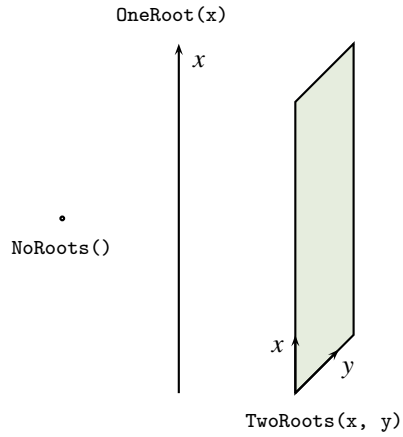


Figure 3.1: The disjoint domain represented by the `RootsOfQ` type.

Exercise 3.4.2.7 Implement functions that have a given type signature, preserving information as much as possible:

```
def f1[A, B]: Option[(A, B)] => (Option[A], Option[B]) = ???
def f2[A, B]: Either[A, B] => (Option[A], Option[B]) = ???
def f3[A,B,C]: Either[A, Either[B,C]] => Either[Either[A,B], C] = ???
```

Exercise 3.4.2.8 Define a parameterized type `EvenList[A]` representing a list of values of type `A` that is guaranteed to have an even number of elements (zero, two, four, etc.). Implement functions `foldLeft` and `map` for `EvenList`.

3.5 Discussion

3.5.1 Disjunctive types as mathematical sets

To understand the properties of disjunctive types from the mathematical point of view, consider a function whose argument is a disjunctive type, such as

```
def isDoubleRoot(r: RootsOfQ) = ...
```

The type of the argument `r:RootsOfQ` represents the mathematical domain of the function, that is, the set of admissible values of the argument `r`. What kind of domain is that? The set of real roots of a quadratic equation $x^2 + bx + c = 0$ can be empty, or it can contain a single real number x, or a pair of real numbers (x, y). Geometrically, a number x is pictured as a point on a line (a one-dimensional space), and pair of numbers (x, y) is pictured as a point on a Cartesian plane (a two-dimensional space). The no-roots case corresponds to a zero-dimensional space, which is pictured as a single point (see Figure 3.1). The point, the line, and the plane do not intersect (have no common points); together, they form the domain of the possible roots. Such domains are called **disjoint**. So, the set of real roots of a quadratic equation $x^2 + bx + c = 0$ is a disjoint domain containing three parts.

In the mathematical notation, a one-dimensional real space is denoted by $\mathbb{R}$, a two-dimensional space by $\mathbb{R}^2$, and a zero-dimensional space by $\mathbb{R}^0$. At first, we may think that the mathematical representation of the type `RootsOfQ` is a union of the three sets, $\mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2$. But an ordinary union

of sets would not always work correctly because we need to distinguish the parts of the union unambiguously, even if some parts have the same type. For instance, the disjunctive type shown in Example 3.4.1.3 cannot be correctly represented by the mathematical union

$$\mathbb{R}^0 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2$$

because $\mathbb{R}^0 \cup \mathbb{R}^0 = \mathbb{R}^0$ and $\mathbb{R}^1 \cup \mathbb{R}^1 = \mathbb{R}^1$, so

$$\mathbb{R}^0 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2 = \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2 \quad .$$

This representation has lost the distinction between e.g. `Linear(x)` and `OneRootQ(x)`.

In the Scala code, each part of a disjunctive type must be distinguished by a unique name such as `NoRoots`, `OneRoot`, and `TwoRoots`. To represent this mathematically, we can attach a distinct label to each part of the union. Labels are symbols without any special meaning, and we can just assume that labels are names of Scala case classes. Parts of the union are then represented by sets of pairs such as $(\texttt{OneRoot}, x)_{x \in \mathbb{R}^1}$. Then the domain `RootsOfQ` is expressed as

$$\texttt{RootsOfQ} = (\texttt{NoRoots}, u)_{u \in \mathbb{R}^0} \cup (\texttt{OneRoot}, x)_{x \in \mathbb{R}^1} \cup (\texttt{TwoRoots}, (x, y))_{(x,y) \in \mathbb{R}^2} \quad .$$

This is an ordinary union of mathematical sets, but each of the sets has a unique label, so no two values from different parts of the union could possibly be equal. This kind of labeled union is called a **disjoint union**. Each element of the disjoint union is a pair of the form `(label, data)`, where the label uniquely identifies the part of the union, and the data can have any chosen type such as $\mathbb{R}^1$. If we use disjoint unions, we cannot confuse different parts of the union even if their data have the same type, because labels are required to be distinct.

Disjoint unions are not often used in mathematics, but they are needed in software engineering because real-life data often belongs to disjoint domains.

Named `Unit` types At first sight, it may seem strange that the zero-dimensional space is represented by a set containing *one* point. Why should we not use an empty set (rather than a set with one point) to represent the case where the equation has no real roots? The reason is that we are required to represent not only the values of the roots but also the information *about* the existence of the roots. The case with no real roots needs to be represented by some *value* of type `RootsOfQ`. This value cannot be missing, which would happen if we used an empty set to represent the no-roots case. It is natural to use the named empty tuple `NoRoots()` to represent this case, just as we used a named 2-tuple `TwoRoots(x, y)` to represent the case of two roots.

Consider the value u used by the mathematical set $(\texttt{NoRoots}, u)_{u \in \mathbb{R}^0}$. Since $\mathbb{R}^0$ consists of a single point, there is only *one* possible value of u. Similarly, the `Unit` type in Scala has only one distinct value, written as `()`. A case class with no parts, such as `NoRoots`, has only one distinct value, written as `NoRoots()`. This Scala value is fully analogous to the mathematical notation $(\texttt{NoRoots}, u)_{u \in \mathbb{R}^0}$.

We see that case classes with no parts are quite similar to `Unit` except for an added name. For this reason, they can be viewed as "named unit" types.

3.5.2 Disjunctive types in other programming languages

Disjunctive types and the associated pattern matching turns out to be one of the defining features of functional programming languages. Languages that were not designed for functional programming do not support these features, while ML, OCaml, Haskell, F#, Scala, Swift, Elm, and PureScript support disjunctive types and pattern matching as part of the language design.

It is remarkable that the named tuple types (also called "structs" or "records") are provided in almost every programming language, while disjunctive types are almost never present except in languages designed for the FP paradigm. (Ada and Pascal are the only languages that have disjunctive types without other FP features.[4])

[4] https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(basic_instructions)#Other_types

The `union` types in C and C++ are not disjunctive types because it is not possible to determine which part of the union is represented by a given value. A `union` declaration in C looks like this,

```
union { int x; double y; long z; } di;
```

The problem is that we cannot determine whether a given value `di` represents an `int`, a `double`, or a `long`. This leads to errors that are hard to detect.

Programming languages of the C family (C, C++, Objective C, Java) support **enumeration** (`enum`) types, which are a limited form of disjunctive types. An `enum` type declaration in Java looks like this:

```
enum Color { RED, GREEN, BLUE; }
```

In Scala, this is equivalent to a disjunctive type containing three *empty* tuples,

```
sealed trait Color
final case class RED()   extends Color
final case class GREEN() extends Color
final case class BLUE()  extends Color
```

If `enum` types were "enriched" with extra data, so that the tuples could be non-empty, we would obtain the full functionality of disjunctive types. A definition of `RootsOfQ` could then look like this:

```
enum RootsOfQ {      // This is not valid in Java!
  NoRoots(), OneRoot(Double x), TwoRoots(Double x, Double y);
}
```

A future version of Scala 3 will have a shorter syntax for disjunctive types[5] that indeed looks like an "enriched `enum`":

```
enum RootsOfQ {
  case NoRoots
  case OneRoot(x: Double)
  case TwoRoots(x: Double, y: Double)
}
```

For comparison, the syntax for a disjunctive type equivalent to `RootsOfQ` in OCaml and Haskell is

```
(* OCaml *)
type RootsOfQ = NoRoots | OneRoot of float | TwoRoots of float*float
```

```
-- Haskell
data RootsOfQ = NoRoots | OneRoot Double | TwoRoots (Double, Double)
```

This is more concise than the Scala syntax. When reasoning about disjunctive types, it is inconvenient to write out long type definitions. Chapter 5 will define a mathematical notation designed for efficient reasoning about types.

3.5.3 Disjunctions and conjunctions in formal logic

In logic, a **proposition** is a logical formula that could be true or false. A **disjunction** of propositions A, B, C is denoted by $A \vee B \vee C$ and is true if and only if *at least one* of A, B, C is true. A **conjunction** of A, B, C is denoted by $A \wedge B \wedge C$ and is true if and only if *all* of the propositions A, B, C are true.

There is a similarity between a disjunctive data type and a logical disjunction of propositions. A value of the disjunctive data type `RootsOfQ` can be constructed only if we have one of the values `NoRoots()`, `OneRoot(x)`, or `TwoRoots(x, y)`. Let us now rewrite the previous sentence as a logical formula. Denote by $\mathcal{CH}(A)$ the logical proposition "this $\mathcal{C}$ode $\mathcal{H}$as a value of type `A`", where "this code" refers to a particular function in a program. So, the proposition "the function *can* return a value of type `RootsOfQ`" is denoted by $\mathcal{CH}(\texttt{RootsOfQ})$. We can then write the above sentence about `RootsOfQ` as the logical formula

$$\mathcal{CH}(\texttt{RootsOfQ}) = \mathcal{CH}(\texttt{NoRoots}) \vee \mathcal{CH}(\texttt{OneRoot}) \vee \mathcal{CH}(\texttt{TwoRoots}) \quad . \tag{3.1}$$

[5] https://dotty.epfl.ch/docs/reference/enums/adts.html

There is also a a similarity between logical *conjunctions* and tuple types. Consider the named tuple (i.e. a case class) `TwoRoots(x: Double, y: Double)`. When can we have a value of type `TwoRoots`? Only if we have two values of type `Double`. Rewriting this sentence as a logical formula, we get

$$\mathcal{CH}(\texttt{TwoRoots}) = \mathcal{CH}(\texttt{Double}) \wedge \mathcal{CH}(\texttt{Double}) \quad .$$

Formal logic admits the simplification

$$\mathcal{CH}(\texttt{Double}) \wedge \mathcal{CH}(\texttt{Double}) = \mathcal{CH}(\texttt{Double}) \quad .$$

However, no such simplification will be available in the general case, e.g.

```
case class Data3(x: Int, y: String, z: Double)
```

For this type, we will have the formula

$$\mathcal{CH}(\texttt{Data3}) = \mathcal{CH}(\texttt{Int}) \wedge \mathcal{CH}(\texttt{String}) \wedge \mathcal{CH}(\texttt{Double}) \quad . \tag{3.2}$$

We find that tuples are related to logical conjunctions in the same way as disjunctive types are related to logical disjunctions. This is the main reason for choosing the name "disjunctive types".[6]

The correspondence between disjunctions, conjunctions, and data types is explained in more detail in Chapter 5. For now, we note that the operations of conjunction and disjunction are not sufficient to produce all possible logical expressions. To obtain a complete logic, it is also necessary to have a logical negation $\neg A$ ("A is not true") or, equivalently, a logical implication $A \to B$ ("if A is true than B is true"). It turns out that the logical implication $A \to B$ is related to the function type `A => B`. In Chapter 4, we will study function types in depth.

[6] These types are also called "variants", "sum types", "co-product types", and "tagged union types".

Part II

Intermediate level

4 The logic of types. II. Higher-order functions

4.1 Functions that return functions

4.1.1 Motivation and first examples

Consider the task of preparing a logger function that prints messages with a configurable prefix.

A simple logger function can be a value of type `String => Unit`, such as

```
val logger: String => Unit = { message => println(s"INFO: $message") }

scala> logger("hello world")
INFO: hello world
```

This function prints any given message with the logging prefix `"INFO"`.

The standard library function `println(...)` always returns a `Unit` value after printing its arguments. As we already know, there is only a single value of type `Unit`, and that value is denoted by `()`. To see that `println` returns `Unit`, run this code:

```
scala> val x = println(123)
123
x: Unit = ()
```

The task is to make the logging prefix configurable. A simple solution is to implement a function `logWith` that takes a prefix as an argument and returns a new logger containing that prefix. Note that the function `logWith` returns a new *function*, i.e. a new value of type `String => Unit`:

```
def logWith(prefix: String): (String => Unit) = {
    message => println(s"$prefix: $message")
}
```

The body of `logWith` consists of a nameless function `message => println(...)`, which is a value of type `String => Unit`. This value will be returned when we evaluate `logWith("...")`.

We can now use `logWith` to create some logger functions:

```
scala> val info = logWith("INFO")
info: String => Unit = <function1>

scala> val warn = logWith("WARN")
warn: String => Unit = <function1>
```

The created logger functions are then usable as ordinary functions:

```
scala> info("hello")
INFO: hello

scala> warn("goodbye")
WARN: goodbye
```

The values `info` and `warn` can be used by any code that needs a logging function.

It is important that the prefix is "baked into" functions created by `logWith`. A logger such as `warn` will always print messages with the prefix `"WARN"`, and the prefix cannot be changed any more. This is so because the value `prefix` is treated as a local constant within the body of the nameless function computed and returned by `logWith`. For instance, the body of the function `warn` is equivalent to

```
{ val prefix = "WARN"; (message => s"$prefix: $message") }
```

Whenever a new function is created using `logWith(prefix)`, the (immutable) value of `prefix` is stored within the body of the newly created function. This is a general feature of nameless functions: the function's body keeps copies of all the outer-scope values it uses. One sometimes says that the function's body "closes over" those values; for this reason, nameless functions are also called "**closures**". It would be clearer to say that nameless functions "capture" values from outer scopes.

As another example of the capture of values, consider this code:

```
val f: Int => Int = {
  val p = 10
  val q = 20
  x => p + q * x
}
```

The body of the function `f` is equivalent to `{ x => 10 + 20 * x }` because the values `p = 10` and `q = 20` were captured.

4.1.2 Curried and uncurried functions

Reasoning mathematically about the code

```
val info = logWith("INFO")
info("hello")
```

we would expect that `info` is *the same value* as `logWith("INFO")`, and so the code `info("hello")` should have the same effect as the code `logWith("INFO")("hello")`. This is indeed so:

```
scala> logWith("INFO")("hello")
INFO: hello
```

The syntax `logWith("INFO")("hello")` looks like the function `logWith` applied to *two* arguments. Yet, `logWith` was defined as a function with a single argument of type `String`. This is not a contradiction because `logWith("INFO")` returns a function that accepts an additional argument. So, both function applications `logWith("INFO")` and `logWith("INFO")("hello")` are valid. In this sense, we are allowed to apply `logWith` to one argument at a time.

A function that can be applied to arguments in this way is called a **curried** function.

While a curried function can be applied to one argument at a time, an **uncurried** function must be applied to all arguments at once, e.g.

```
def prefixLog(prefix: String, message: String): Unit = println(s"$prefix: $message")
```

The type of the curried function `logWith` is `String => (String => Unit)`. By Scala's syntax conventions, the function arrow (`=>`) groups to the *right*. So the parentheses in the type expression `String => (String => Unit)` are not needed; the function's type can be written as `String => String => Unit`.

The type `String => String => Unit` is different from `(String => String) => Unit`, which is the type of a function returning `Unit` and having a function of type `String => String` as its argument. When an argument's type is a function type, e.g. `String => String`, it must be enclosed in parentheses.

In general, a curried function takes an argument and returns another function that again takes an argument and returns another function, and so on, until finally a non-function type is returned. So, the type signature of a curried function generally looks like `A => B => C => ... => R => S`, where `A`, `B`, ..., `R` are the **curried arguments** and `S` is the "final" result type.

For example, in the type expression `A => B => C => D` the types `A`, `B`, `C` are the types of curried arguments, and `D` is the final result type. It takes time to get used to reading this kind of syntax.

In Scala, functions defined with multiple argument groups (enclosed in multiple pairs of parentheses) are curried functions. We have seen examples of curried functions before:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B]
def fmap[A, B](f: A => B)(xs: Option[A]): Option[B]
```

```
def foldLeft[A, R](xs: Seq[A])(init: R)(update: (R, A) => R): R
```

The type signatures of these functions can be also written equivalently as

```
def map[A, B]: Seq[A] => (A => B) => Seq[B]
def fmap[A, B]: (A => B) => Option[A] => Option[B]
def foldLeft[A, R]: Seq[A] => R => ((R, A) => R) => R
```

Curried arguments of a function type, such as `(A => B)`, need parentheses.

To summarize, a curried function such as `logWith` can be defined in three equivalent ways in Scala:

```
1 def logWith1(prefix: String)(message: String): Unit = println(s"$prefix: $message")
2 def logWith2(prefix: String): String => Unit = { message => println(s"$prefix: $message") }
3 def logWith3: String => String => Unit = { prefix => message => println(s"$prefix: $message") }
```

We will sometimes enclose nameless functions in parentheses or curly braces to improves clarity.

Line 3 above shows that the arrow symbols `=>` group to the right within the *code* of nameless functions: `x => y => expr` means `{x => {y => expr}}`, a nameless function taking an argument `x` and returning a nameless function that takes an argument `y` and returns an expression `expr`. This syntax convention is adopted for two reasons. First, the code `x => y => z` visually corresponds to the curried function's type signature `A => B => C`, which uses the same syntax convention. Second, the syntax `(x => y) => z` could not work for a nameless function because `(x => y)` is not a valid pattern expression. Patterns cannot decompose nameless function values such as `x => y` (which represent compiled code) into partial code for `x` and `y`. So `x => (y => z)` is the only possible way of inserting parentheses into `x => y => z`.

Although the code `(x => y) => z` is invalid, a type expression `(A => B) => C` is valid. A nameless function of type `(A => B) => C` is written as `f => expr` where `f: A => B` is its argument and `expr` its body.

4.1.3 Equivalence of curried and uncurried functions

We defined the curried function `logWith` in order to be able to create logger functions such as `info` and `warn`. However, some curried functions, such as `foldLeft`, are almost always applied to all possible arguments. A curried function applied to all its possible arguments is equivalent to an uncurried function that takes all those arguments at once. Let us look at this equivalence in more detail.

Consider a curried function with type signature `Int => Int => Int`. This function takes an integer and returns an (uncurried) function taking an integer and returning an integer. An example of such a curried function is

```
def f1(x: Int): Int => Int = { y => x - y }
```

The function takes an integer `x` and returns the expression `y => x - y`, which is a function of type `Int => Int`. The code of `f1` can be written equivalently as

```
val f1: Int => Int => Int = { x => y => x - y }
```

Let us compare the function `f1` with a function that takes its two arguments at once, e.g.

```
def f2(x: Int, y: Int): Int = x - y
```

The function `f2` has type signature `(Int, Int) => Int`.

The syntax for using the functions `f1` and `f2` is different:

```
scala> f1(20)(4)
res0: Int = 16

scala> f2(20, 4)
res1: Int = 16
```

The main difference between the usage of `f1` and `f2` is that `f2` must be applied at once to both arguments, while `f1` applied to just the first argument, `20`. The result of evaluating `f1(20)` is a function that can be later applied to another argument:

```
scala> val r1 = f1(20)
r1: Int => Int = <function1>

scala> r1(4)
res2: Int = 16
```

Applying a curried function to some but not all possible arguments is called a **partial application**. Applying a curried function to all possible arguments is called a **saturated** application.

If we need to partially apply an *uncurried* function, we can use the underscore (_) symbol:

```
1 scala> val r2: Int => Int = f2(20, _)
2 r2: Int => Int = <function1>
3
4 scala> r2(4)
5 res3: Int = 16
```

(The type annotation `Int => Int` is required in line 1.) This code creates a function `r2` by partially applying `f2` to the first argument but not to the second. Other than that, `r2` is the same function as `r1` defined above; i.e. `r2` returns the same values for the same arguments as `r1`. A more straightforward syntax for a partial application is

```
scala> val r3: Int => Int = { x => f2(20, x) }    // Same as r2 above.
r3: Int => Int = <function1>

scala> r3(4)
res4: Int = 16
```

We can see that a curried function, such as `f1`, is better adapted for partial application than `f2`, because the syntax is shorter. However, the functions `f1` and `f2` are **computationally equivalent** in the sense that given `f1` we can reconstruct `f2` and vice versa:

```
def f2new(x: Int, y: Int): Int = f1(x)(y)             // f2new is equal to f2
def f1new: Int => Int => Int = { x => y => f2(x, y) } // f1new is equal to f1
```

It is clear that the function `f1new` computes the same results as `f1`, and that the function `f2new` computes the same results as `f2`. The computational equivalence of the functions `f1` and `f2` is not *equality* — these functions are *different*; but one of them can be reconstructed from the other if necessary.

More generally, a curried function has a type signature of the form `A => B => C => ... => R => S`, where `A`, `B`, `C`, ..., `S` are some types. A function with this type signature is computationally equivalent to an uncurried function with type signature `(A,B,C,...,R) => S`. The uncurried function takes all arguments at once, while the curried function takes one argument at a time. Other than that, these two functions compute the same results given the same arguments.

We have seen how a curried function can be converted to an equivalent uncurried one, and vice versa. The Scala library defines the methods `curried` and `uncurried` that convert between these forms of functions. To convert between `f2` and `f1`:

```
scala> val f1c = (f2 _).curried
f1c: Int => (Int => Int) = <function1>

scala> val f2u = Function.uncurried(f1c)
f2u: (Int, Int) => Int = <function2>
```

The syntax `(f2 _)` is needed in Scala to convert methods to function values. Recall that Scala has two ways of defining a function: one as a method (defined using `def`), another as a function value (defined using `val`). The extra underscore will become unnecessary in Scala 3.

The methods `.curried` and `.uncurried` are easy to implement, as we will show in Section 4.2.1.

4.2 Fully parametric functions

We have seen that some functions are declared with type parameters, which are set only when the function is applied to specific arguments. Examples of such functions are `map` and `filter`, written as

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B]
def filter[A](xs: Seq[A])(p: A => Boolean): Seq[A]
```

Such functions can be applied to arguments of different types without changing the function's code. It is clearly better to implement a single function with type parameters instead of writing several

functions with repeated code but working with different types.

When we apply the function `map` as `map(xs)(f)` to a specific value `xs` of type, e.g., `Seq[Int]`, and a specific function `f` of type, say, `Int => String`, the Scala compiler will automatically set the type parameters `A = Int` and `B = String` in the code of `map`. We may also set type parameters explicitly and write, for example, `map[Int, String](xs)(f)`. This syntax shows a certain similarity between type parameters such as `Int`, `String` and "value parameters" (arguments) `xs` and `f`. Setting type parameters (e.g. `map[Int, String]`) means substituting e.g. `A=Int`, `B=String` into the type signature of the function, similarly to how setting value parameters means substituting specific values into the function body.

In the functions `map` and `filter` as just shown, some types are parameters while others are specific types, such as `Seq` and `Boolean`. It is sometimes possible to replace *all* types in the type signature of a function by type parameters.

A function is **fully parametric** if its arguments have types described by type parameters, and the code of the function treats all types as type parameters. In other words, fully parametric functions do not use any values of specific types, such as `Int` or `String`, in their type signature or in their body. A fully parametric function does not use any information about its argument types, other than assuming that the arguments' types correctly match the type signature.

What kind of functions are fully parametric? To build up intuition, let us compare the following two functions that have the same type signature:

```
def cos_sin(p: (Double, Double)): (Double, Double) = p match {
  case (x, y) =>
    val r = math.sqrt(x * x + y * y)
    (x / r, y / r) // Return cos and sin of the angle.
}

def swap(p: (Double, Double)): (Double, Double) = p match {
  case (x, y) => (y, x)
}
```

We can introduce type parameters into the type signature of `swap` to make it fully parametric:

```
def swap[A, B](p: (A, B)): (B, A) = p match {
  case (x, y) => (y, x)
}
```

Converting `swap` into a fully parametric function is possible because the operation of swapping two parts of a tuple `(A, B)` works in the same way for all types `A`, `B`. No changes were made in the body of the function. The specialized version of `swap` working on `(Double, Double)` can be obtained from the fully parametric version of `swap` if we set the type parameters as `A = Double`, `B = Double`.

In contrast, the function `cos_sin` performs a computation that is specific to the type `Double` and cannot be generalized to an arbitrary type parameter `A` instead of `Double`. So, `cos_sin` cannot be generalized to a fully parametric function.

The `swap` operation for pairs is already defined in the Scala library:

```
scala> (1, "abc").swap
res0: (String, Int) = (abc,1)
```

Other swapping functions can be defined for tuples with more elements, e.g.

```
def swap12[A,B,C]: ((A, B, C)) => (B, A, C) = { case (x, y, z) => (y, x, z) }
```

The Scala syntax requires the double parentheses around tuple types of arguments but not around the tuple type of a function's result. So, the function `cos_sin` may be written as a value of type

```
val cos_sin: ((Double, Double)) => (Double, Double) = ...
```

4.2.1 Examples. Function composition

Further examples of fully parametric functions are the identity function, the constant function, the function composition methods, and the curry / uncurry conversions.

The identity function (available in the Scala library as `identity[T]`) is

```
def id[T]: T => T = (t => t)
```

The constant function (available in the Scala library as `Function.const`) takes an argument `c` and returns a new function that always returns `c`:

```
def const[C, X](c: C): X => C = (_ => c)
```

The syntax `_ => c` is used to emphasize that the function ignores its argument.

Function composition Consider two functions `f: Int => Double` and `g: Double => String`. We can apply `f` to an integer argument `x:Int` and get a result `f(x)` of type `Double`. Applying `g` to that result gives a `String` value `g(f(x))`. The transformation from the original integer `x:Int` to the final `String` value `g(f(x))` can be viewed as a new function of type `Int => String`. That new function is called the **forward composition** of the two functions `f` and `g`. In Scala, this operation is written as `f andThen g`:

```
val f: Int => Double = (x => 5.67 + x)
val g: Double => String = (x => f"Result x = $x%3.2f")

scala> val h = f andThen g
h: Int => String = <function1>

scala> h(40)
res36: String = Result x = 45.67
```

The Scala compiler derives the type of `h` automatically as `Int => String`.

The forward composition is denoted by $\fatsemi$ (pronounced "before") and can be defined as

$$f \fatsemi g \triangleq x \to g(f(x)) \quad . \tag{4.1}$$

The symbol $\triangleq$ means "is defined as".

We could implement the operation of forward composition as a fully parametric function,

```
def andThen[X, Y, Z](f: X => Y)(g: Y => Z): X => Z = { x => g(f(x)) }
```

This type signature requires the types of the function arguments to match in a certain way, or else the composition is undefined (and the code would produce a type error).

The method `andThen` is an example of a function that *both* returns a new function *and* takes other functions as arguments.

The **backward composition** of two functions f and g works in the opposite order: first g is applied and then f is applied to the result. Using the symbol $\circ$ (pronounced "after") for this operation, we can write

$$f \circ g \triangleq x \to f(g(x)) \quad . \tag{4.2}$$

In Scala, the backward composition is called `compose` and used as `f compose g`. This method may be implemented as a fully parametric function

```
def compose[X, Y, Z](f: Y => X)(g: Z => Y): Z => X = { z => f(g(z)) }
```

We have already seen the methods `curried` and `uncurried` defined by the Scala library. As an illustration, let us write our own code for converting curried functions to uncurried:

```
def uncurry[A, B, R](f: A => B => R): ((A, B)) => R = { case (a, b) => f(a)(b) }
```

We conclude from these examples that fully parametric functions perform operations so general that they work in the same way for all types. Some arguments of fully parametric functions may have complicated types such as `A => B => R`, which are type expressions made up from type parameters. But fully parametric functions do not use values of specific types such as `Int` or `String`.

Functions with type parameters are sometimes called "generic". This book uses the term "fully parametric" to refer to a certain restricted kind of generic functions.

4.2.2 Laws of function composition

The operations of function composition, introduced in Section 4.2.1, have three important properties or "laws" that follow directly from the definitions. These laws are:

- The two identity laws: the composition of any function f with the identity function will give again the function f.
- The associativity law: the consecutive composition of three functions f, g, h does not depend on the order in which the pairs are composed.

These laws hold equally for the forward and the backward composition, since those are just syntactic variants of the same mathematical operation. Let us write these laws rigorously as mathematical equations and prove them.

Proofs with forward composition The composition of the identity function with an arbitrary function f can be $\mathrm{id} \fatsemi f$ with the identity function to the left of f, or $f \fatsemi \mathrm{id}$ with the identity function to the right of f. In both cases, the result must be equal to the function f. The resulting two laws are

$$\begin{aligned} \text{left identity law of composition :} \quad & \mathrm{id} \fatsemi f = f \quad , \\ \text{right identity law of composition :} \quad & f \fatsemi \mathrm{id} = f \quad . \end{aligned}$$

To show that these laws always hold, we need to show that both sides of the laws, which are functions, give the same result when applied to an arbitrary value x. Let us first clarify how the type parameters must be set for the laws to have consistently matching types.

The laws must hold for an arbitrary function f. So we may assume that f has the type signature $A \to B$, where A and B are arbitrary type parameters. Consider the left identity law. The function $(\mathrm{id} \fatsemi f)$ is, by definition (4.1), a function that takes an argument x, applies id to that x, and then applies f to the result:

$$\mathrm{id} \fatsemi f = (x \to f(\mathrm{id}(x))) \quad .$$

If f has type $A \to B$, its argument must be of type A, or else the types will not match. Therefore, the identity function must have type $A \to A$, and the argument x must have type A. With these choices of the type parameters, the function $(x \to f(\mathrm{id}(x)))$ will have type $A \to B$, as it must since the right-hand side of the law is f. We add type annotations to the code as *superscripts*,

$$\mathrm{id}^{:A \to A} \fatsemi f^{:A \to B} = (x^{:A} \to f(\mathrm{id}(x)))^{:A \to B} \quad .$$

In the Scala syntax, this formula may be written as

```
id[A] andThen (f: A => B) == { x: A => f(id(x)) }: A => B
```

It is quicker to write code in the mathematical notation than in the Scala syntax.

We will follow the convention where type parameters are single uppercase letters, as is common in Scala code (although this convention is not enforced by the Scala compiler). The colon symbol (:) in the superscript $x^{:A}$ means a type annotation, as in Scala code `x:A`. Superscripts without a colon, such as id^A, denote type parameters, as in Scala code `identity[A]`. Since the function `identity[A]` has type `A => A`, we can write id^A or equivalently (but more verbosely) $\mathrm{id}^{:A \to A}$ to denote that function.

By definition of the identity function, we have $\mathrm{id}(x) = x$, and so

$$\mathrm{id} \fatsemi f = (x \to f(\mathrm{id}(x))) = (x \to f(x)) = f \quad .$$

The last step works since $x \to f(x)$ is a function taking an argument x and applying f to that argument; i.e. $x \to f(x)$ is an expanded form of the same function f.

Now consider the right identity law:

$$f \fatsemi \mathrm{id} = (x \to \mathrm{id}(f(x))) \quad .$$

To check that the types match, assume that $f^{:A\to B}$. Then x must have type A, and the identity function must have type $B \to B$. The result of $\mathrm{id}(f(x))$ will also have type B. With these choices of type parameters, all types match:

$$f^{:A\to B} \fatsemi \mathrm{id}^{:B\to B} = (x^{:A} \to \mathrm{id}(f(x)))^{:A\to B} \quad .$$

Since $\mathrm{id}(f(x)) = f(x)$, we find that

$$f \fatsemi \mathrm{id} = (x \to f(x)) = f \quad .$$

In this way, we have demonstrated that both identity laws hold.

The associativity law is written as an equation like this:

$$\text{associativity law of function composition :} \quad (f \fatsemi g) \fatsemi h = f \fatsemi (g \fatsemi h) \quad . \tag{4.3}$$

Let us verify that the types match here. The types of the functions f, g, and h must be such that all the function compositions exist. If f has type $A \to B$ for some type parameters A and B, then the argument of g must be of type B; so we must have $g^{:B\to C}$, where C is another type parameter. The composition $f \fatsemi g$ has type $A \to C$, so h must have type $C \to D$ for some type D. Assuming the types as $f^{:A\to B}$, $g^{:B\to C}$, and $h^{:C\to D}$, we find that the types in all the compositions $f \fatsemi g$, $g \fatsemi h$, $(f \fatsemi g) \fatsemi h$, and $f \fatsemi (g \fatsemi h)$ match. We can rewrite Eq. (4.3) with type annotations,

$$(f^{:A\to B} \fatsemi g^{:B\to C}) \fatsemi h^{:C\to D} = f^{:A\to B} \fatsemi (g^{:B\to C} \fatsemi h^{:C\to D}) \quad . \tag{4.4}$$

Having checked the types, we are ready to prove the associativity law. We note that both sides of the law (4.4) are functions of type $A \to D$. To prove that two functions are equal means to prove that they always return the same results when applied to the same arguments. So we need to apply both sides of Eq. (4.4) to an arbitrary value $x^{:A}$. Using the definition (4.1) of the forward composition,

$$(f \fatsemi g)(x) = g(f(x)) \quad ,$$

we find

$$\begin{aligned} ((f \fatsemi g) \fatsemi h)(x) &= h((f \fatsemi g)(x)) = h(g(f(x))) \quad , \\ (f \fatsemi (g \fatsemi h))(x) &= (g \fatsemi h)(f(x)) = h(g(f(x))) \quad . \end{aligned}$$

Both sides of the law are now equal when applied to an arbitrary value x.

Because of the associativity law, we do not need parentheses when writing the expression $f \fatsemi g \fatsemi h$. The function $(f \fatsemi g) \fatsemi h$ is the same as $f \fatsemi (g \fatsemi h)$.

In the proof, we have omitted the type annotations since we already checked that the types match. Checking the types beforehand allows us to write shorter proofs.

Proofs with backward composition This book uses the **forward composition** $f \fatsemi g$ rather than the backward composition $g \circ f$. If necessary, all equations can be converted from one notation to the other by reversing the order of compositions,

$$f \fatsemi g = g \circ f$$

for any functions $f^{:A\to B}$ and $g^{:B\to C}$. Let us see how to prove the composition laws in the backward notation. We will just need to reverse the order of function compositions in the proofs above.

The left identity and right identity laws are

$$f \circ \mathrm{id} = f \quad , \quad \mathrm{id} \circ f = f \quad .$$

To match the types, we need to choose the type parameters as

$$f^{:A\to B} \circ \mathrm{id}^{:A\to A} = f^{:A\to B} \quad , \quad \mathrm{id}^{B\to B} \circ f^{:A\to B} = f^{:A\to B} \quad .$$

We can apply both sides of the laws to an arbitrary value $x^{:A}$. For the left identity law, we find from definition (4.2) that

$$f \circ \mathrm{id} = (x \to f(\mathrm{id}(x))) = (x \to f(x)) = f \quad .$$

Similarly for the right identity law,

$$\mathrm{id} \circ f = (x \to \mathrm{id}\,(f\,(x))) = (x \to f\,(x)) = f \quad .$$

The associativity law,

$$h \circ (g \circ f) = (h \circ g) \circ f \quad ,$$

is proved by applying both sides to an arbitrary value x of a suitable type:

$$(h \circ (g \circ f))\,(x) = h\,((g \circ f)\,(x)) = h\,(g\,(f\,(x))) \quad ,$$
$$((h \circ g) \circ f)\,(x) = (h \circ g)\,(f(x)) = h\,(g\,(f\,(x))) \quad .$$

The types are checked by assuming that f has the type $f^{:A \to B}$. The types in $g \circ f$ match only when $g^{:B \to C}$, and then $g \circ f$ is of type $A \to C$. The type of h must be $h^{:C \to D}$ for the types in $h \circ (g \circ f)$ to match. We can write the associativity law with type annotations as

$$h^{:C \to D} \circ (g^{:B \to C} \circ f^{:A \to B}) = (h^{:C \to D} \circ g^{:B \to C}) \circ f^{:A \to B} \quad . \tag{4.5}$$

The associativity law allows us to omit parentheses in the expression $h \circ g \circ f$.

The length of calculations is the same in the forward and the backward notation. One difference is that types of function compositions are more visually clear in the forward notation: it is easier to check that types match in Eq. (4.4) than in Eq. (4.5).

4.2.3 Example: A function that is not fully parametric

Fully parametric functions should not make any decisions based on the actual types of arguments. As an example of an *incorrect* implementation of a fully parametric function, consider the following "fake identity" function:

```
def fakeId[A]: A => A = {    // Special code for A = Int:
  case x: Int   => (x - 1).asInstanceOf[A]
  case x        =>  x        // Common code for all other types A.
}
```

This function's type signature is the same as that of `id[A]`, and its behavior is the same for all types `A` except for `A = Int`:

```
scala> fakeId("abc")
res0: String = abc

scala> fakeId(true)
res1: Boolean = true

scala> fakeId(0)
res2: Int = -1
```

While Scala allows us to write this kind of code, the resulting function does not appear to be useful. In any case, `fakeId` is not a fully parametric function.

The identity laws of composition will not hold if we use `fakeId[A]` instead of the correct function `id[A]`. For example, consider the composition of `fakeId` with a simple function `f_1` defined by

```
def f_1: Int => Int = { x => x + 1 }
```

The composition (`f_1 andThen fakeId`) will have type `Int => Int`. Since `f_1` has type `Int => Int`, Scala will automatically set the type parameter `A = Int` in `fakeId[A]`,

```
scala> def f_2 = f_1 andThen fakeId
f_2: Int => Int
```

The identity law says that $f_2 = f_1 \fatsemi \mathrm{id} = f_1$. But we can check that `f_1` and `f_2` are not the same:

```
scala> f_1(0)
res3: Int = 1

scala> f_2(0)
res4: Int = 0
```

It is important that we are able to detect a non-fully parametric function by checking whether some equation holds, without need to examine the code of the function `fakeId`. In this book, we will always formulate the desired properties of functions through equations or "laws". To prove the laws, we will need to perform symbolic calculations similar to the proofs in Section 4.2.2. These calculations are **symbolic** in the sense that we were manipulating symbols such as x, f, g, and h without substituting any specific values for these symbols but using only the general properties of functions. In the next section, we will get more experience with such calculations.

4.3 Symbolic calculations with nameless functions

4.3.1 Calculations with curried functions

In mathematics, functions are evaluated by substituting their argument values into their body. Nameless functions are evaluated in the same way. For example, applying the nameless function $x \to x + 10$ to an integer 2, we substitute 2 instead of x in "$x + 10$" and get "$2 + 10$", which we then evaluate to 12. The computation is written like this,

$$(x \to x + 10)(2) = 2 + 10 = 12 \quad .$$

To run this computation in Scala, we need to add a type annotation:

```
scala> ((x: Int) => x + 10)(2)
res0: Int = 12
```

Curried function applications such as $f(x)(y)$ are rarely used in mathematics, so we need to gain some experience working with them.

Consider a curried nameless function being applied to arguments, such as `(x => y => x - y)(20)(4)`, and compute the result of this function application. Begin with the argument `20`; applying a nameless function of the form `(x => ...)` to `20` means to substitute `x = 20` into the body of the function. After that substitution, we obtain the expression `y => 20 - y`, which is again a nameless function. Applying that function to the remaining argument `(4)` means substituting `y = 4` into the body of `y => 20 - y`. We get the expression `20 - 4`, which equals `16`. Check the result with Scala:

```
scala> ((x: Int) => (y: Int) => x - y)(20)(4)
res1: Int = 16
```

Applying a curried function such as `x => y => z => expr(x,y,z)` to three curried arguments `10`, `20`, and `30` means to substitute `x = 10`, `y = 20`, and `z = 30` into the expression `expr(x,y,z)`. In this way, we can easily apply a curried function to any number of curried arguments.

This calculation is helped by the convention that `f(g)(h)` means first applying `f` to `g` and then applying the result to `h`. In other words, function application groups to the *left*: `f(g)(h) = (f(g))(h)`. It would be confusing if function application grouped to the right and `f(g)(h)` meant first applying `g` to `h` and then applying `f` to the result. If *that* were the syntax convention, it would be harder to reason about applying a curried function to its arguments.

We see that the right grouping of the function arrow `=>` is well adapted to the left grouping of function applications. All functional languages follow these syntactic conventions.

To make calculations shorter, we will write code in a mathematical notation rather than in the Scala syntax. Type annotations are written with a colon in the superscript, for example: $x^{:\text{Int}} \to x + 10$ instead of the code `((x:Int) => x + 10)`.

The symbolic evaluation of the Scala code `((x:Int) => (y:Int) => x - y)(20)(4)` can be written as

$$\begin{aligned} & (x^{:\text{Int}} \to y^{:\text{Int}} \to \underline{x} - y)\,(20)\,(4) \\ \text{apply function and substitute } x = 20: \quad & = (y^{:\text{Int}} \to 20 - \underline{y})\,(4) \\ \text{apply function and substitute } y = 4: \quad & = 20 - 4 = 16 \quad . \end{aligned}$$

In the above step-by-step calculation, the colored underlines and comments at left are added for clarity. A colored underline indicates a sub-expression that is going to be rewritten at the *next* step.

Here we performed calculations by substituting an argument into a function at each step. A compiled Scala program is evaluated in a similar way at run time.

Nameless functions are *values* and so can be used as part of larger expressions, just as any other values. For instance, nameless functions can be arguments of other functions (nameless or not). Here is an example of applying a nameless function $f \to f(2)$ to a nameless function $x \to x + 4$:

$$\begin{aligned} & (f \to \underline{f}(2))\,(x \to x + 4) \\ \text{substitute } f = (x \to x + 4): \quad & = (x \to \underline{x} + 4)(2) \\ \text{substitute } x = 2: \quad & = 2 + 4 = 6 \quad . \end{aligned}$$

In the nameless function $f \to f(2)$, the argument f has to be itself a function, otherwise the expression $f(2)$ would make no sense. The argument x of $f(x)$ must be an integer, or else we would not be able to compute $x + 4$. The result of computing $f(2)$ is 4, an integer. We conclude that f must have type Int → Int, or else the types will not match. To verify this result in Scala, we need to specify a type annotation for f:

```
scala> ((f: Int => Int) => f(2))(x => x + 4)
res2: Int = 6
```

No type annotation is needed for $x \to x + 4$ since the Scala compiler already knows the type of f and can figure out that x in $x \to x + 4$ must have type `Int`.

To summarize the syntax conventions for curried nameless functions:

- Function expressions group everything to the right: $x \to y \to z \to e$ means $x \to (y \to (z \to e))$.
- Function applications group everything to the left, so $f(x)(y)(z)$ means $((f(x))(y))(z)$.
- Function applications group stronger than infix operations, so $x + f(y)$ means $x + (f(y))$, as usual in mathematics.

Here are some more examples of performing function applications symbolically. Types are omitted for brevity; every non-function value is of type `Int`.

$$\begin{aligned} (x \to x * 2)\,(10) &= 10 * 2 = 20 \quad . \\ (p \to z \to z * p)\,(t) &= (z \to z * t) \quad . \\ (p \to z \to z * p)\,(t)(4) &= (z \to z * t)(4) = 4 * t \quad . \end{aligned}$$

Some results of these computation are integer values such as 20; other results are nameless functions such as $z \to z * t$. Verify this in Scala:

```
scala> ((x:Int) => x*2)(10)
res3: Int = 20

scala> ((p:Int) => (z:Int) => z*p)(10)
res4: Int => Int = <function1>

scala> ((p:Int) => (z:Int) => z*p)(10)(4)
res5: Int = 40
```

In the following examples, some arguments are themselves functions. Consider an expression that uses the nameless function $(g \to g(2))$ as an argument:

$$(f \to p \to \underline{f(p)})\,(g \to g(2)) \tag{4.6}$$

$$\text{substitute } f = (g \to g(2)): \quad = p \to (g \to \underline{g}(2))\,(p)$$

$$\text{substitute } g = p: \quad = p \to p(2) \quad . \tag{4.7}$$

The result of this expression is a function $p \to p(2)$ that will apply *its* argument p to the value 2. A possible value for p is the function $x \to x + 4$. So, let us apply Eq. (4.6) to that function:

$$\underline{(f \to p \to f(p))\,(g \to g(2))}\,(x \to x + 4)$$

$$\text{use Eq. (4.7)}: \quad = (p \to \underline{p}(2))\,(x \to x + 4)$$

$$\text{substitute } p = (x \to x + 4): \quad = (x \to \underline{x} + 4)\,(2)$$

$$\text{substitute } x = 2: \quad = 2 + 4 = 6 \quad .$$

To verify this calculation in Scala, we need to add appropriate type annotations for f and p. To figure out the types, we reason like this:

We know that the function $f \to p \to f(p)$ is being applied to the arguments $f = (g \to g(2))$ and $p = (x \to x + 4)$. So, the argument f in $f \to p \to f(p)$ must be a function that takes p as an argument.

The variable x in $x \to x + 4$ must be of type `Int`, or else we cannot add x to 4. Thus, the type of the expression $x \to x + 4$ is Int → Int, and so must be the type of the argument p. We write $p^{:\text{Int}\to\text{Int}}$.

Finally, we need to make sure that the types match in the function $f \to p \to f(p)$. Types match in $f(p)$ if the type of f's argument is the same as the type of p, which is Int → Int. So f's type must be (Int → Int) → A for some type A. Since in our example $f = (g \to g(2))$, types match only if g has type Int → Int. But then $g(2)$ has type Int, and so we must have A = Int. Thus, the type of f is (Int → Int) → Int. We know enough to write the Scala code now:

```
scala> ((f: (Int => Int) => Int) => p => f(p))(g => g(2))(x => x + 4)
res6: Int = 6
```

Type annotations for p, g, and x may be omitted because the Scala compiler will figure out the missing types from the given type of f. However, it is never an error to specify more type annotations when writing code; it just makes the code longer.

4.3.2 Solved examples: Deriving a function's type from its code

Checking that the types match is an important part of the functional programming paradigm, both in the practice of writing code and in theoretical derivations of laws for various functions. For instance, in the derivations of the composition laws (Section 4.2.2), we were able to deduce the possible type parameters for f, g, and h in the expression $f \,;\, g \,;\, h$. This worked because the composition operation `andThen` (denoted by the symbol ;) is fully parametric. Given a fully parametric function, it is often possible to derive the most general type signature that matches the body of that function. The same type-matching procedure may also help in converting a given function to a fully parametric form.

Let us look at some examples of doing this.

Example 4.3.2.1 The functions `const` and `id` were defined in Section 4.2.1. What is the value `const(id)` and what is its type? Determine the most general type parameters in the expression `const(id)`.

Solution We need to treat the functions `const` and `id` as values, since our goal is to apply `const` to `id`. Write the code of these functions in a short notation:

$$\text{const}^{C,X} \triangleq (c^{:C} \to \_^{:X} \to c) \quad ,$$

$$\text{id}^{A} \triangleq (a^{:A} \to a) \quad .$$

The types will match in the expression `const(id)` only if the argument of the function `const` has the same type as the type of `id`. Since `const` is a curried function, we need to look at its *first* curried

argument, which is of type C. The type of `id` is $A \to A$, where A is an arbitrary type so far. So, the type parameter C in $\text{const}^{C,X}$ must be equal to $A \to A$:

$$C = A \to A \quad .$$

The type parameter X in $\text{const}^{C,X}$ is not constrained, so we keep it as X. The result of applying `const` to `id` is of type $X \to C$, which equals $X \to A \to A$. In this way, we find

$$\text{const}^{A \to A, X}(\text{id}^A) : X \to A \to A \quad .$$

The types A and X can be arbitrary. The type $X \to A \to A$ is the most general type for the expression `const(id)` because we have not made any assumptions about the types except requiring that all functions must be always applied to arguments of the correct types.

To compute the value of `const(id)`, it remains to substitute the code of `const` and `id`. Since we already checked the types, we may omit all type annotations:

$$\begin{aligned} & \underline{\text{const}}\ (\text{id}) \\ \text{definition of const :} \quad & = (c \to x \to \underline{c})(\text{id}) \\ \text{apply function, substitute } c = \text{id} : \quad & = (x \to \underline{\text{id}}) \\ \text{definition of id :} \quad & = (x \to a \to a) \quad . \end{aligned}$$

The function $(x \to a \to a)$ takes an argument $x^{:X}$ and returns the identity function $a^{:A} \to a$. It is clear that the argument x is ignored by this function, so we can rewrite it equivalently as

$$\text{const}\ (\text{id}) = (\_^{:X} \to a^{:A} \to a) \quad .$$

Example 4.3.2.2 Implement a function `twice` that takes a function `f: Int => Int` as its argument and returns a function that applies `f` twice. For example, if the function `f` is `{ x => x + 3 }`, the result of `twice(f)` should be equal to the function `x => x + 6`. After implementing the function `twice`, generalize it to a fully parametric function.

Solution According to the requirements, the function `twice` must return a new function of type `Int => Int`. So the type signature of `twice` is

```
def twice(f: Int => Int): Int => Int = ???
```

Since `twice(f)` must be a new function with an integer argument, we begin the code of `twice` by writing a new nameless function `{ (x: Int) => ... }`,

```
def twice(f: Int => Int): Int => Int = { (x: Int) => ??? }
```

The new function must apply `f` twice to its argument, that is, it must return `f(f(x))`. We can finish the implementation now:

```
def twice(f: Int => Int): Int => Int = { x => f(f(x)) }
```

The type annotation `(x: Int)` can be omitted. To test:

```
scala> val g = twice(x => x + 3)
g: Int => Int = <function1>

scala> g(10)
res0: Int = 16
```

To transform `twice` into a fully parametric function means replacing its type signature by a fully parameterized type signature while keeping the function body unchanged,

```
def twice[A, B, ...](f: ...): ... = { x => f(f(x)) }
```

To determine the type signature and the possible type parameters A, B, ..., we need to determine the most general type that matches the function body. The function body is the expression $x \to f(f(x))$. Assume that x has type A; for types to match in the sub-expression $f(x)$, we need

f to have type $A \to B$ for some type B. The sub-expression $f(x)$ will then have type B. For types to match in $f(f(x))$, the argument of f must have type B; but we already assumed $f^{:A\to B}$. This is consistent only if $A = B$. In this way, $x^{:A}$ implies $f^{:A\to A}$, and the expression $x \to f(f(x))$ has type $A \to A$. We can now write the type signature of `twice`,

```
def twice[A](f: A => A): A => A = { x => f(f(x)) }
```

This fully parametric function has only one independent type parameter, A, and can be equivalently written in the code notation as

$$\text{twice}^A \triangleq f^{:A\to A} \to x^{:A} \to f(f(x)) = (f^{:A\to A} \to f \fatsemi f) \quad . \tag{4.8}$$

The procedure of deriving the most general type for a given code is called **type inference**. In Example 4.3.2.2, the presence of the type parameter A and the type signature $(A \to A) \to A \to A$ have been "inferred" from the code $f \to x \to f(f(x))$.

Example 4.3.2.3 Consider the fully parametric function `twice` defined in Example 4.3.2.2. What is the type of `twice(twice)`, and what computation does it perform? Test your answer on the expression `twice(twice[Int])(x => x+3)(10)`. What are the type parameters in that expression?

Solution Begin by figuring out the required type of `twice(twice)`. We introduce unknown type parameters as `twice[A](twice[B])`. The function `twice[A]` of type $(A \to A) \to A \to A$ can be applied to the argument `twice[B]` only if `twice[B]` has type $A \to A$. But `twice[B]` is of type $(B \to B) \to B \to B$. Since the symbol $\to$ groups to the right, we have

$$(B \to B) \to B \to B = (B \to B) \to (B \to B) \quad .$$

This can match with $A \to A$ only if we set $A = (B \to B)$. So the most general type of `twice(twice)` is

$$\text{twice}^{B\to B}(\text{twice}^B) : (B \to B) \to B \to B \quad . \tag{4.9}$$

After checking that types match, we may omit types from further calculations.

Example 4.3.2.2 defined `twice` with the `def` syntax. To use `twice` as an argument in the expression `twice(twice)`, it is convenient to define `twice` as a value, `val twice = ...` However, the function `twice` needs type parameters, and Scala 2 does not directly support `val` definitions with type parameters. Scala 3 will support type parameters appearing together with arguments in a nameless function:

```
val twice = [A] => (f: A => A) => (x: A) => f(f(x)) // Valid only in Scala 3.
```

Keeping this in mind, we will use the curried definition of `twice` given by Eq. (4.8), which shows that $\text{twice}(f) = f \fatsemi f$. Substituting that into the expression `twice(twice)`, we find

$$\begin{aligned} & \text{twice(twice)} = \text{twice} \fatsemi \text{twice} \\ \text{expand function composition :} \quad & = f \to \text{twice}(\underline{\text{twice}}(f)) \quad . \\ \text{definition of twice}(f) : \quad & = f \to \underline{\text{twice}}(f \fatsemi f) \\ \text{definition of twice} : \quad & = f \to f \fatsemi f \fatsemi f \fatsemi f \quad . \end{aligned}$$

So `twice(twice)` is a function that applies its (function-typed) argument *four* times.

The type parameters for the code `twice(twice[Int])` follow from Eq. (4.9) with $B = \text{Int}$ and can be written as $\text{twice}^{\text{Int}\to\text{Int}}(\text{twice}^{\text{Int}})$, or in Scala syntax, `twice[Int => Int](twice[Int])`. To test, we need to write at least one type parameter in the code, or else Scala cannot infer the types correctly:

```
scala> twice(twice[Int])(x => x + 3)(100) // x => x + 3 + 3 + 3 + 3
res0: Int = 112

scala> twice[Int => Int](twice)(x => x + 3)(100)
res1: Int = 112
```

Example 4.3.2.4 Infer the type signature for the fully parametric function

```
def p[...]:... = { f => f(2) }
```

Can the types possibly match in the expression `p(p)`?

Code notation	Scala syntax	Comments
$x^{:A}$	`x:A`	a value or an argument of type `A`
$f^{:A \to B}$	`f: A => B`	a function of type `A => B`
$x^{:\text{Int}} \to f(x)$	`{ x: Int => f(x) }`	nameless function with argument `x`
$f^{A,B} \triangleq \ldots$	`def f[A, B] = ...`	define a function with type parameters
id^A, also $\text{id}^{:A \to A}$	`identity[A]`	the standard identity function
$A \to B \to C$	`A => B => C`	type of a curried function
$f \fatsemi g$	`f andThen g`	forward composition of functions
$g \circ f$	`g compose f`	backward composition of functions

Table 4.1: Mathematical notation for symbolic computations with code.

Solution In the nameless function $f \to f(2)$, the argument f must be itself a function with an argument of type `Int`, otherwise the sub-expression $f(2)$ is ill-typed. So, types will match if f has type Int → Int or Int → String or similar. The most general case is when f has type Int → A, where A is an arbitrary type (i.e. a type parameter). The type A will then be the type of the value $f(2)$. Since nameless function $f \to f(2)$ has an argument f of type Int → A and the result of type A, we find that the type of p must be (Int → A) → A. With this type assignment, all types match. The type parameter A remains undetermined and is added to the type signature of the function `p`. The code is

```
def p[A]: (Int => A) => A = { f => f(2) }
```

To answer the question about the expression `p(p)`, we begin by writing that expression with new type parameters as `p[A](p[B])`. Then we try to choose `A` and `B` so that the types match in that expression. Does the type of `p[B]`, which is `(Int => B) => B`, match the type of the argument of `p[A]`, which is `Int => A`, with some choice of `A` and `B`? A function type `P => Q` matches `X => Y` only if `P = X` and `Q = Y`. So `(Int => B) => B` can match `Int => A` only if `Int => B` matches `Int` and if `B = A`. But it is impossible for `Int => B` to match `Int`, no matter how we choose `B`.

We conclude that types cannot be chosen consistently in `p[A](p[B])`. Such expressions contain a type error and are rejected by the Scala compiler. One also says that the expression `p(p)` is **not well-typed**, or does not **typecheck**.

We have seen how for code fragments containing only function expressions one can always find the most general types that makes all functions match their arguments, unless the code does not typecheck. The Damas-Hindley-Milner algorithm[1] performs type inference (or determines that there is a type error) for a large class of expressions containing functions, tuples, and disjunctive types.

4.4 Summary

Table 4.1 shows the code notations introduced in this chapter.

What can we do with this chapter's techniques?

- Implement functions that return new functions and/or take functions as arguments.
- Simplify function applications symbolically.
- Infer the most general type for a given code expression.
- Convert functions to a fully parametric form when possible.

[1] https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system#Algorithm_W

The following solved examples and exercises illustrate these techniques.

4.4.1 Solved examples

Example 4.4.1.1 Implement a function that applies a given function f repeatedly to an initial value x_0, until a given function `cond` returns `true`:

```
def converge[X](f: X => X, x0: X, cond: X => Boolean): X = ???
```

Solution We create an iterator that keeps applying the function f, and use `.find` to stop the sequence when the condition first holds:

```
def converge[X](f: X => X, x0: X, cond: X => Boolean): X =
  Stream.iterate(x0)(f)   // Type is Stream[X].
  .find(cond)             // Type is Option[X].
  .get                    // Type is X.
```

The method `.get` is a partial function that can be applied only to non-empty `Option` values. It is safe to call `.get` here, because the stream is unbounded and, if the condition `cond` never becomes `true`, the program will run out of memory (since `Stream.iterate` keeps all computed values in memory) or the user will run out of patience. So `.find(cond)` can never return an empty `Option` value. Of course, it is not satisfactory that the program crashes when the sequence does not converge. Exercise 4.4.2.7 will implement a safer version of this function by limiting the allowed number of iterations.

A tail-recursive implementation that works in constant memory is

```
@tailrec def converge[X](f: X => X, x0: X, cond: X => Boolean): X =
  if (cond(x0)) x0 else converge(f, f(x0), cond)
```

To test this code, compute an approximation to $\sqrt{q}$ by Newton's method. The iteration function f is

$$f(x) = \frac{1}{2}\left(x + \frac{q}{x}\right) \quad .$$

We iterate $f(x)$ starting with $x_0 = q/2$ until we obtain a given precision:

```
def approx_sqrt(q: Double, precision: Double): Double = {
      def cond(x: Double): Boolean = math.abs(x * x - q) <= precision
      def iterate_sqrt(x: Double): Double = 0.5 * (x + q / x)
      converge(iterate_sqrt, q / 2, cond)
}
```

Newton's method for $\sqrt{q}$ is guaranteed to converge when $q \geq 0$. Test it:

```
scala> approx_sqrt(25, 1.0e-8)
res0: Double = 5.000000000016778
```

Example 4.4.1.2 Using both `def` and `val`, define a Scala function that takes an integer `x` and returns a function that adds `x` to *its* argument.

Solution Let us first write down the required type signature: the function must take an integer argument `x:Int`, and the return value must be a function of type `Int => Int`.

```
def add_x(x: Int): Int => Int = ???
```

We are required to return a function that adds `x` to its argument. Let us call that argument `z`, to avoid confusion with the `x`. So, we are required to return the function `{ z => z + x }`. Since functions are values, we return a new function by writing a nameless function expression:

```
def add_x(x: Int): Int => Int = { z => z + x }
```

To implement the same function by using a `val`, we first convert the type signature of `add_x` to the equivalent curried type Int → Int → Int. Now we can write the Scala code of a function `add_x_v`:

```
val add_x_v: Int => Int => Int = { x => z => z + x }
```

The function `add_x_v` is equal to `add_x` except for using the `val` syntax instead of `def`. It is not necessary to specify the type of the arguments `x` and `z` because we already specified the type Int → Int → Int for the value `add_x_v`.

Example 4.4.1.3 Using both `def` and `val`, implement a curried function `prime_f` that takes a function f and an integer x, and returns `true` when $f(x)$ is a prime number. Use the function `is_prime` defined in Section 1.1.2.

Solution First, determine the required type signature of `prime_f`. The value $f(x)$ must have type `Int`, or else we cannot check whether it is prime. So, f must have type Int → Int. Since `prime_f` should be a curried function, we need to put each argument into its own set of parentheses:

```
def prime_f(f: Int => Int)(x: Int): Boolean = ???
```

To implement `prime_f`, we need to return the result of `is_prime` applied to `f(x)`. A simple solution is

```
def prime_f(f: Int => Int)(x: Int): Boolean = is_prime(f(x))
```

To implement the same function as a `val`, rewrite its type signature as

```
val prime_f: (Int => Int) => Int => Boolean = ???
```

(The parentheses around `Int => Int` are mandatory since `Int => Int => Int => Boolean` would be a completely different type.) The implementation is

```
val prime_f: (Int => Int) => Int => Boolean = { f => x => is_prime(f(x)) }
```

The code `is_prime(f(x))` is a forward composition of the functions `f` and `is_prime`, so we can write

```
val prime_f: (Int => Int) => Int => Boolean = (f => f andThen is_prime)
```

A nameless function of the form `f => f.something` is equivalent to a shorter Scala syntax `(_.something)`. So we finally rewrite the code of `prime_f` as

```
val prime_f: (Int => Int) => Int => Boolean = (_ andThen is_prime)
```

Example 4.4.1.4 Implement a function `choice(x,p,f,g)` that takes a value x, a predicate p, and two functions f and g. The return value must be $f(x)$ if $p(x)$ returns true; otherwise the return value must be $g(x)$. Infer the most general type for this function.

Solution The code of this function must be

```
def choice[...](x,p,f,g) = if (p(x)) f(x) else g(x)
```

To infer the most general type for this code, begin by assuming that x has type A, where A is a type parameter. Then the predicate p must have type `A => Boolean`. Since p is an arbitrary predicate, the value $p(x)$ will be sometimes `true` and sometimes `false`. So, `choice(x,p,f,g)` will sometimes compute $f(x)$ and sometimes $g(x)$. It follows that type A must be the argument type of both f and g, which means that the most general types so far are $f^{:A\to B}$ and $g^{:A\to C}$, yielding the type signature

$$\text{choice}(x^{:A}, p^{:A\to\text{Boolean}}, f^{:A\to B}, g^{:A\to C}) \quad .$$

What could be the return type of `choice(x,p,f,g)`? If $p(x)$ returns `true`, the function `choice` returns $f(x)$, which is of type B. Otherwise, `choice` returns $g(x)$, which is of type C. However, the type signature of `choice` must be fixed in advance (at compile time) and cannot depend on the value $p(x)$ computed at run time. So, the types of $f(x)$ and of $g(x)$ must be the same, $B = C$. The type signature of `choice` will thus have only two type parameters, A and B:

```
def choice[A, B](x: A, p: A => Boolean, f: A => B, g: A => B): B =  if (p(x)) f(x) else g(x)
```

Example 4.4.1.5 Infer the most general type for the fully parametric function

```
def q[...]: ... = { f => g => g(f) }
```

What types are inferred for the expressions `q(q)` and `q(q(q))`?

Solution Begin by assuming $f^{:A}$ with a type parameter A. In the sub-expression $g \to g(f)$, the curried argument g must itself be a function, because it is being applied to f as $g(f)$. So we assign types as $f^{:A} \to g^{:A\to B} \to g(f)$, where A and B are type parameters. Then the final returned value $g(f)$ has type B. Since there are no other constraints on the types, the types A and B remain arbitrary, so we add them to the type signature:

```
def q[A, B]: A => (A => B) => B = { f => g => g(f) }
```

To match types in the expression `q(q)`, we first assume arbitrary type parameters and write `q[A, B](q[C, D])`. We need to introduce new type parameters C, D because these type parameters may need to be set differently from A, B when we try to match the types in the expression `q(q)`.

The type of the first curried argument of `q[A, B]`, which is A, must match the entire type of `q[C, D]`, which is $C \to (C \to D) \to D$. So we must set the type parameter A as

$$A = C \to (C \to D) \to D \quad .$$

The type of `q(q)` becomes

$$q^{A,B}(q^{C,D}) : ((C \to (C \to D) \to D) \to B) \to B \quad ,$$
$$\text{where} \quad A = C \to (C \to D) \to D \quad .$$

There are no other constraints on the type parameters B, C, D.

We use this result to infer the most general type for `q(q(q))`. We may denote $r \triangleq q(q)$ for brevity; then, as we just found, r has type $((C \to (C \to D) \to D) \to B) \to B$. To infer types in the expression `q(r)`, we introduce new type parameters E, F and write `q[E, F](r)`. The type of the argument of `q[E, F]` is E, and this must be the same as the type of r. This gives the constraint

$$E = ((C \to (C \to D) \to D) \to B) \to B \quad .$$

Other than that, the type parameters are arbitrary. The type of the expression `q(q(q))` is $(E \to F) \to F$. We conclude that the most general type of `q(q(q))` is

$$q^{E,F}(q^{A,B}(q^{C,D})) : ((((C \to (C \to D) \to D) \to B) \to B) \to F) \to F \quad ,$$
$$\text{where} \quad A = C \to (C \to D) \to D$$
$$\text{and} \quad E = ((C \to (C \to D) \to D) \to B) \to B \quad .$$

It is clear from this derivation that expressions such as `q(q(q(q)))`, `q(q(q(q(q))))`, etc., are well-typed.

Let us test these results in Scala, renaming the type parameters for clarity to `A`, `B`, `C`, `D`:

```
scala> def qq[A, B, C]: ((A => (A => B) => B) => C) => C = q(q)
qq: [A, B, C]=> ((A => ((A => B) => B)) => C) => C

scala> def qqq[A, B, C, D]: ((((A => (A => B) => B) => C) => C) => D) => D = q(q(q))
qqq: [A, B, C, D]=> ((((A => ((A => B) => B)) => C) => C) => D) => D
```

We did not need to write any type parameters within the expressions `q(q)` and `q(q(q))` because the full type signature was declared for each of these expressions. Since the Scala compiler did not print any error messages, we are assured that the types match correctly.

Example 4.4.1.6 Infer types in the code expression

$$(f \to g \to g(f))\,(f \to g \to g(f))\,(f \to f(10))$$

and simplify the code through symbolic calculations.

Solution The given expression is a curried function $f \to g \to g(f)$ applied to two curried arguments. The plan is to consider each of these sub-expressions in turn, assigning types for them using type parameters, and then to figure out how to set the type parameters so that all types match.

Begin by renaming the shadowed variables (f and g) to remove shadowing:

$$(f \to g \to g(f))\,(x \to y \to y(x))\,(h \to h(10)) \quad . \tag{4.10}$$

As we have seen in Example 4.4.1.5, the sub-expression $f \to g \to g(f)$ is typed as $f^{:A} \to g^{:A\to B} \to g(f)$, where A and B are some type parameters. The sub-expression $x \to y \to y(x)$ is the same function as $f \to g \to g(f)$ but with possibly different type parameters, say, $x^{:C} \to y^{:C\to D} \to y(x)$. The types A, B, C, D are so far unknown.

Finally, the variable h in the sub-expression $h \to h(10)$ must have type $\text{Int} \to E$, where E is another type parameter. So, the sub-expression $h \to h(10)$ is a function of type $(\text{Int} \to E) \to E$.

The types must match in the entire expression (4.10):

$$(f^{:A} \to g^{:A\to B} \to g(f))(x^{:C} \to y^{:C\to D} \to y(x))(h^{:\text{Int}\to E} \to h(10)) \quad . \tag{4.11}$$

It follows that f must have the same type as $x \to y \to y(x)$, while g must have the same type as $h \to h(10)$. The type of g, which we know as $A \to B$, will match the type of $h \to h(10)$, which we know as $(\text{Int} \to E) \to E$, only if $A = (\text{Int} \to E)$ and $B = E$. It follows that f has type $\text{Int} \to E$. At the same time, the type of f must match the type of $x \to y \to y(x)$, which is $C \to (C \to D) \to D$. This can work only if $C = \text{Int}$ and $E = (C \to D) \to D = (\text{Int} \to D) \to D$.

In this way, we have found all the relationships between the type parameters A, B, C, D, E in Eq. (4.11). The type D remains arbitrary, while the type parameters A, B, C, E are expressed as

$$A = \text{Int} \to (\text{Int} \to D) \to D \quad , \tag{4.12}$$

$$B = E = (\text{Int} \to D) \to D \quad , \tag{4.13}$$

$$C = \text{Int} \quad .$$

The entire expression in Eq. (4.11) is a saturated application of a curried function, and thus has the same type as the "final" result expression $g(f)$, which has type B. So, the entire expression in Eq. (4.11) has type $B = (\text{Int} \to D) \to D$.

Having established that types match, we can now omit the type annotations and rewrite the code:

$$\begin{aligned}
& (f \to g \to \underline{g}(\underline{f}))\,(x \to y \to y(x))\,(h \to h(10)) \\
\text{substitute } f \text{ and } g: \quad & = (h \to \underline{h}(10))\,(x \to y \to y(x)) \\
\text{substitute } h: \quad & = (x \to y \to y(\underline{x}))(10) \\
\text{substitute } x: \quad & = y \to y(10) \quad .
\end{aligned}$$

The type of this expression is $(\text{Int} \to D) \to D$ with a type parameter D. Since the argument y is an arbitrary function, we cannot simplify $y(10)$ or $y \to y(10)$ any further. We conclude that the final simplified form of Eq. (4.10) is $y^{:\text{Int}\to D} \to y(10)$.

To test this, we first define the function $f \to g \to g(f)$ as in Example 4.4.1.5,

```
def q[A, B]: A => (A => B) => B = { f => g => g(f) }
```

We also define the function $h \to h(10)$ with a general type $(\text{Int} \to E) \to E$,

```
def r[E]: (Int => E) => E = { h => h(10) }
```

To help Scala evaluate Eq. (4.11), we need to set the type parameters for the first `q` function as `q[A, B]` where A and B are given by Eqs. (4.12)–(4.13):

```
scala> def s[D] = q[Int => (Int => D) => D, (Int => D) => D](q)(r)
s: [D]=> (Int => D) => D
```

To verify that the function s^D indeed equals $y^{:\text{Int}\to D} \to y(10)$, we apply s^D to some functions of type $\text{Int} \to D$, say, for $D = \text{Boolean}$ or $D = \text{Int}$:

```
scala> s(_ > 0)  // Set D = Boolean and evaluate (10 > 0).
res6: Boolean = true

scala> s(_ + 20) // Set D = Int and evaluate (10 + 20).
res7: Int = 30
```

Example 4.4.1.7 Compute $(x \to y \to x(x(y))) \fatsemi (p \to p(2)) \fatsemi (z \to z+3)$ symbolically and infer types.

Solution The forward composition $f \fatsemi g$ substitutes the *body* of f into the argument of g:

$$\text{substitute } y = f(x): \quad (x \to f(x)) \fatsemi (y \to \underline{g(y)}) = (x \to g(f(x))) \quad .$$

This allows us to compute the forward compositions left to right:

$$(x \to y \to x(x(y))) \fatsemi (p \to p(2)) = x \to (y \to x(x(y)))(2) = x \to x(x(2)) \quad .$$
$$(x \to x(x(2))) \fatsemi (z \to z+3) = x \to x(x(2)) + 3 \quad .$$

Computing the pairwise combinations in another order, we get:

$$(p \to p(2)) \fatsemi (z \to z+3) = p \to p(2) + 3 \quad .$$
$$(x \to y \to x(x(y))) \fatsemi (p \to p(2)+3) = x \to (y \to x(x(y)))\,(2) + 3 = x \to x(x(2)) + 3 \quad .$$

Types are inferred as $(x \to y \to x(x(y)))^{:(\text{Int}\to\text{Int})\to(\text{Int}\to\text{Int})} \fatsemi (p \to p(2))^{:(\text{Int}\to\text{Int})\to\text{Int}} \fatsemi (z \to z+3)^{:\text{Int}\to\text{Int}}$.

4.4.2 Exercises

Exercise 4.4.2.1 For `id` and `const` as defined above, what are the types of `id(id)`, `id(id)(id)`, `id(id(id))`, `id(const)`, and `const(const)`? Simplify these code expressions by symbolic calculations.

Exercise 4.4.2.2 For the function `twice` from Example 4.3.2.2, infer the most general type for the function `twice(twice(twice))`. What does that function do? Test your answer on an example.

Exercise 4.4.2.3 Define a function `thrice` similarly to `twice` except it should apply a given function 3 times. What does the function `thrice(thrice(thrice))` do?

Exercise 4.4.2.4 Define a function `ence` similarly to `twice` except it should apply a given function n times, where n is an additional curried argument.

Exercise 4.4.2.5 Define a fully parametric function `flip(f)` that swaps arguments for any given function `f` having two arguments. To test:

```
def f(x: Int, y: Int) = x - y // Expect f(10, 2) == 8.
val g = flip(f)  // Now expect g(2, 10) == 8.

scala> assert( f (10, 2) == 8 && g(2, 10) == 8 )
```

Exercise 4.4.2.6 Revise the function from Exercise 1.6.2.4, implementing it as a curried function and replacing the hard-coded number 100 by a *curried* first argument. The type signature should become `Int => List[List[Int]] => List[List[Int]]`.

Exercise 4.4.2.7 Implement the function `converge` from Example 4.4.1.1 as a curried function with an additional argument to set the maximum number of iterations, returning `Option[Double]` as the final result type. The new version of `converge` should return `None` if the convergence condition is not satisfied after the given maximum number of iterations. The type signature and an example test:

```
@tailrec def convergeN[X](cond: X => Boolean)(x0: X)(maxIter: Int)(f: X => X): Option[X] = ???

scala> convergeN[Int](_ < 0)(0)(10)(_ + 1) // This does not converge.
res0: Option[Int] = None

scala> convergeN[Double]{ x => math.abs(x * x - 25) < 1e-8 }(1.0)(10) { x => 0.5 * (x + 25 / x ) }
res1: Option[Double] = Some(5.000000000053722)
```

Exercise 4.4.2.8 Write a function `curry2` converting an uncurried function of type `(Int, Int) => Int` into an equivalent curried function of type `Int => Int => Int`.

Exercise 4.4.2.9 Apply the function $(x \to \_ \to x)$ to the value $(z \to z(q))$ where $q^{:Q}$ is a given value of type Q. Infer types in these expressions.

Exercise 4.4.2.10 Infer types in the following expressions and test in Scala:
(a) $p \to q \to p(t \to t(q))$.
(b) $p \to q \to q(x \to x(p(q)))$.

Exercise 4.4.2.11 Show that the following expressions cannot be well-typed:
(a) $p \to p(q \to q(p))$.
(b) $p \to q \to q(x \to p(q(x)))$.

Exercise 4.4.2.12 Infer types and simplify the following code expressions by symbolic calculations:
(a) $q \to (x \to y \to z \to x(z)(y(z)))\,(a \to a)\,(b \to b(q))$.
(b) $(f \to g \to h \to f(g(h)))\,(x \to x)$.
(c) $(x \to y \to x(y))\,(x \to y \to x)$.
(d) $(x \to y \to x(y))\,(x \to y \to y)$.
(e) $x \to (f \to y \to f(y)(x))\,(z \to \_ \to z)$.
(f) $z \to (x \to y \to x)\,(x \to x(z))\,(y \to y(z))$.

Exercise 4.4.2.13 Infer types and simplify the following code expressions by symbolic calculations:
(a) $(z \to z + 1)\ ;\ (x \to y \to x * y)\ ;\ (p \to p(2))$.
(b) $(p \to q \to p + q + 1)\ ;\ (f \to f\ ;\ f)\ ;\ (x \to x(1))$.

4.5 Discussion

4.5.1 Higher-order functions

The **order** of a function is the number of function arrows (`=>`) contained in the type signature of that function. If a function's type signature contains more than one arrow, the function is called a **higher-order** function. Higher-order functions take functions as arguments and/or return functions as result values.

The methods `andThen`, `compose`, `curried`, and `uncurried` are examples of higher-order functions that take other functions as arguments *and* return new functions.

The following examples illustrate the concept of a function's order. Consider the code

```
def f1(x: Int): Int = x + 10
```

The function `f1` has type signature `Int => Int` and order 1, so it is *not* a higher-order function.

```
def f2(x: Int): Int => Int = (z => z + x)
```

The function `f2` has type signature `Int => Int => Int` and is a higher-order function of order 2.

```
def f3(g: Int => Int): Int = g(123)
```

The function `f3` has type signature `(Int => Int) => Int` and is a higher-order function of order 2.

Although `f2` is a higher-order function, its "higher-orderness" comes from the fact that the return value is of a function type. An equivalent computation can be performed by an uncurried function that is not higher-order:

```
scala> def f2u(x: Int, z: Int): Int = z + x    // Type signature (Int, Int) => Int
```

Unlike `f2`, the function `f3` *cannot* be converted to a non-higher-order function because `f3` has an argument of a function type. Converting to an uncurried form cannot eliminate such arguments.

4.5.2 Name shadowing and the scope of bound variables

Bound variables are introduced in nameless functions whenever an argument is defined. For example, in the nameless function $x \to y \to x + y$, the bound variables are the curried arguments x and y. The variable y is only defined within the scope $(y \to x + y)$ of the inner function; the variable x is defined within the entire scope of $x \to y \to x + y$.

Another way of introducing bound variables in Scala is to write a `val` or a `def` within curly braces:

```
val x = {
  val y = 10          // Bound variable.
  y + y * y
}     // Same as ‘val x = 10 + 10 * 10‘.
```

A bound variable is invisible outside the scope that defines it. So, it is easy to rename a bound variable: no outside code could possibly use it and depend on its value.

However, outside code may define a variable that (by chance) has the same name as a bound variable inside the scope. Consider this example from calculus: In the integral

$$f(x) = \int_0^x \frac{dx}{1+x} \quad ,$$

two bound variables named x are defined in two scopes: one in the scope of f, another in the scope of the nameless function $x \mapsto \frac{1}{1+x}$. The convention in mathematics is to treat these two x's as two *completely different* variables that just happen to have the same name. In sub-expressions where both of these bound variables are visible, priority is given to the bound variable defined in the closest inner scope. The outer definition of x is then **shadowed**, i.e. hidden, by the inner definition of x. For this reason, evaluating $f(10)$ will give

$$f(10) = \int_0^{10} \frac{dx}{1+x} = \log_e(11) \approx 2.398 \quad ,$$

rather than $\int_0^{10} \frac{dx}{1+10} = \frac{10}{11}$, because the outer definition $x = 10$ is shadowed within the expression $\frac{1}{1+x}$ by the definition of x in the local scope of $x \mapsto \frac{1}{1+x}$.

Since this is the standard mathematical convention, the same convention is adopted in functional programming. A variable defined in a function scope (i.e. a bound variable) is invisible outside that scope but will shadow any outside definitions of a variable with the same name.

Name shadowing is not advisable in practical programming, because it usually decreases the clarity of code and so invites errors. Consider the nameless function

$$x \to x \to x \quad ,$$

and let us decipher this confusing syntax. The symbol $\to$ groups to the right, so $x \to x \to x$ is the same as $x \to (x \to x)$. It is a function that takes x and returns $x \to x$. Since the argument x in $(x \to x)$ may be renamed to y without changing the function, we can rewrite the code to

$$x \to (y \to y) \quad .$$

Having removed name shadowing, we can more easily understand this code and reason about it. For instance, it becomes clear that this function ignores its argument x and always returns the same value (the identity function $y \to y$). So we can rewrite $(x \to x \to x)$ as $(\_ \to y \to y)$, which is clearer.

4.5.3 Operator syntax for function applications

In mathematics, function applications are sometimes written without parentheses, for instance $\cos x$ or $\sin z$. Formulas such as $2 \sin x \cos x$ imply parentheses as $2 \cdot \sin(x) \cdot \cos(x)$. Functions such as $\cos x$ are viewed as "operators" that are applied to their arguments without parentheses, similar to the operators of summation $\sum_k f(k)$ and differentiation $\frac{d}{dx} f(x)$.

Many programming languages (such as ML, OCaml, F#, Haskell, Elm, PureScript) have adopted this "operator syntax", making parentheses optional for function arguments so that `f x` means the same as `f(x)`. Parentheses are still used where necessary to avoid ambiguity or for readability.[2]

The conventions for nameless functions in the operator syntax become:

- Function expressions group to the right, so $x \to y \to z \to e$ means $x \to (y \to (z \to e))$.
- Function applications group to the left, so $f\ x\ y\ z$ means $((f\ x)y)z$.
- Function applications group stronger than infix operations, so $x + f\ y$ means $x + (f\ y)$, just as in mathematics "$x + \cos y$" groups "$\cos y$" stronger than the infix "+" operation.

Thus, $x \to y \to a\,b\,c + p\,q$ means $x \to (y \to ((a\,b)\ c) + (p\,q))$. When this notation becomes hard to read correctly, one needs to add parentheses, e.g. to write $f(x \to g\ h)$ instead of $f\ x \to g\ h$.

This book will avoid using the "operator syntax" when reasoning about code. Scala does not support the parentheses-free operator syntax; parentheses are needed around each curried argument.

From the point of view of programming language theory, curried functions are "simpler" because they always have a *single* argument (and may return a function that will consume further arguments). From the point of view of programming practice, curried functions are often harder to read and to write.

In the operator syntax used e.g. in OCaml and Haskell, a curried function `f` is applied to curried arguments as, e.g., `f 20 4`. This departs further from the mathematical tradition and requires some getting used to. If the two arguments are more complicated than just 20 and 4, the resulting expression may become harder to read, compared with the syntax where commas are used to separate the arguments. (Consider, for instance, the expression `f (g 10) (h 20) + 30`.) To improve readability of code, programmers may prefer to first define short names for complicated expressions and then use these names as curried arguments.

In Scala, the choice of whether to use curried or uncurried function signatures is largely a matter of syntactic convenience. Most Scala code tends to be written with uncurried functions, while curried functions are used when they produce more easily readable code.

One of the syntactic features for curried functions in Scala is the ability to specify a curried argument using the curly brace syntax. Compare the two definitions of the function `summation` described in Section 1.7.5:

```
def summation1(a: Int, b: Int, g: Int => Int): Int = (a to b).map(g).sum

def summation2(a: Int, b: Int)(g: Int => Int): Int = (a to b).map(g).sum
```

To apply these functions to arguments, we need to use slightly different syntax:

```
scala> summation1(1, 10, { x => x*x*x + 2*x })
res0: Int = 3135

scala> summation2(1, 10) { x => x*x*x + 2*x }
res1: Int = 3135
```

The code that calls `summation2` is easier to read because the curried argument is syntactically separated from the rest of the code by curly braces. This is especially useful when the curried argument is itself a function with a complicated body, since Scala's curly braces syntax allows function bodies to contain their own local definitions (`val` or `def`).

Another feature of Scala is the "dotless" method syntax: for example, `xs map f` is equivalent to `xs.map(f)` and `f andThen g` is equivalent to `f.andThen(g)`. The "dotless" syntax is available only for infix methods, such as `.map`, defined on specific types such as `Seq`. In Scala 3, the "dotless" syntax will only work for methods having a special `@infix` annotation. Do not confuse Scala's "dotless" method syntax with the operator syntax used in Haskell and some other languages.

[2] The operator syntax has a long history in programming. It is used in Unix shell commands, for example `cp file1 file2`. In LISP-like languages, function applications are enclosed in parentheses but the arguments are space-separated, for example `(f 10 20)`. Operator syntax is also used in some programming languages such as Tcl, Groovy, and Coffeescript.

4.5.4 Deriving a function's code from its type signature

We have seen how the procedure of type inference derives the type signature from a function's code. A well-known algorithm for type inference is the Damas-Hindley-Milner algorithm,[3] with a Scala implementation available.[4]

It is remarkable that one can sometimes derive the function's *code* from the function's type signature. We will now look at some examples of this.

Consider a fully parametric function that performs partial applications for arbitrary other functions. A possible type signature is

```
def pa[A, B, C](x: A)(f: (A, B) => C): B => C = ???
```

The function `pa` substitutes a fixed argument value `x:A` into another given function `f`.

How can we implement `pa`? Since `pa(x)(f)` must return a function of type `B => C`, we have no choice other than to begin writing a nameless function in the code,

```
def pa[A, B, C](x: A)(f: (A, B) => C): B => C = { y: B =>
  ??? // Need to compute a value of type C in this scope.
}
```

In the inner scope, we need to compute a value of type `C`, and we have values `x:A`, `y:B`, and `f:(A, B)=>C`. How can we compute a value of type `C`? If we knew that `C = Int` when `pa(x)(f)` is applied, we could have simply selected a fixed integer value, say, `1`, as the value of type `C`. If we knew that `C = String`, we could have selected a fixed string, say, `"hello"`, as the value of type `C`. But a fully parametric function cannot use any knowledge of the types of its actual arguments.

So, a fully parametric function cannot produce a value of an arbitrary type `C` from scratch. The only way of producing a value of type `C` is by applying the function `f` to arguments of types `A` and `B`. Since the types `A` and `B` are arbitrary, we cannot obtain any values of these types other than `x:A` and `y:B`. So, the only way of getting a value of type `C` is to compute `f(x, y)`. Thus, the body of `pa` must be

```
def pa[A, B, C](x: A)(f: (A, B) => C): B => C = { y => f(x, y) }
```

In this way, we have *unambiguously* derived the body of this function from its type signature, by assuming that the function must be fully parametric.

Another example is the operation of forward composition $f \fatsemi g$ viewed as a fully parametric function with type signature

```
def before[A, B, C](f: A => B, g: B => C): A => C = ???
```

To implement `before`, we need to create a nameless function of type `A => C`,

```
def before[A, B, C](f: A => B, g: B => C): A => C = { x: A =>
  ??? // Need to compute a value of type C in this scope.
}
```

In the inner scope, we need to compute a value of type C from the values $x^{:A}$, $f^{:A \to B}$, and $g^{:B \to C}$. Since the type C is arbitrary, the only way of obtaining a value of type C is by applying g to an argument of type B. In turn, the only way of obtaining a value of type B is to apply f to an argument of type A. Finally, we have only one value of type A, namely $x^{:A}$. So, the only way of obtaining the required result is to compute $g(f(x))$.

We have unambiguously derived the body of the function from its type signature:

```
def before[A, B, C](f: A => B, g: B => C): A => C = { x => g(f(x)) }
```

In Chapter 5 and in Appendix C, we will see how a function's code can be derived from type signatures for a wide range of fully parametric functions.

[3] https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system

[4] http://dysphoria.net/2009/06/28/hindley-milner-type-inference-in-scala/

5 The logic of types. III. The Curry-Howard correspondence

Fully parametric functions (introduced in Section 4.2) perform operations so general that their code does not depend on values of any specific data types such as `Int` or `String`. An example of a fully parametric function is

```
def before[A, B, C](f: A => B, g: B => C): A => C = { x => g(f(x)) }
```

We have also seen in Section 4.5.4 that for certain functions of this kind, the code can be derived unambiguously from the type signature.

There exists a mathematical theory (called the **Curry-Howard correspondence**) that provides precise conditions for the possibility of deriving a function's code from its type and a systematic derivation algorithm. Technical details about the algorithm are found in Appendix C. This chapter describes the main results and applications of this theory to functional programming.

5.1 Values computed by fully parametric functions

5.1.1 Motivation

Consider the Scala code of a fully parametric function,

```
def f[A, B, ...]: ... = {
  val x: Either[A, B] = ... // Some expression here.
  ... }
```

If this program compiles without type errors, it means that the types match and, in particular, that the function `f` is able to compute a value `x` of type `Either[A, B]`.

It is sometimes *impossible* to compute a value of a certain type within the body of a fully parametric function. For example, the fully parametric function `fmap` shown in Section 3.2.3.1 cannot compute any values of type `A`,

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  val x: A = ??? // Cannot compute x here!
  ... }
```

The reason is that a fully parametric function cannot compute values of type `A` from scratch without using previously given values of type `A` and without applying a function that returns values of type `A`. In `fmap`, no values of type `A` are given as arguments; the given function `f: A => B` returns values of type `B` and not `A`. The code of `fmap` must perform pattern matching on a value of type `Option[A]`:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None       =>
    val x: A = ??? // Cannot compute x here!
    ...
  case Some(a)    =>
    val x: A = a   // Can compute x in this scope.
    ... }
```

Since the case `None` has no values of type `A`, we are unable to compute a value `x` in that scope (as long as `fmap` remains a fully parametric function).

Being able to compute `x:A` "within the body of a function" means that, if needed, the function should be able to *return* `x` as a result value. This requires computing `x` in all cases, not just within one part of the `match` expression.

The body of `fmap` also cannot compute any values of type `B`. Since no arguments of type `B` are given, the only way of obtaining a value of type `B` would be to apply the function `f: A => B` to *some* value of type `A`; but we just saw that the body of `fmap` cannot compute any values of type `A`.

Another example where one cannot compute a value of a certain type is in the following code:

```
def before[A, B, C](f: A => B, g: B => C): A => C = {
  //  val h: C => A = ???   // Cannot compute h here!
  a => g(f(a)) // Can compute a value of type A => C.
}
```

The body of `before` may only use the arguments `f` and `g`. We can compute a value of type `A => C` by composing `f` and `g`, but it is impossible to compute a value `h` of type `C => A`, no matter what code we try to write. The reason is that the body of `before` has no given values of type `A` and no functions that return values of type `A`, so a nameless function such as `{c:C => ???}` cannot compute its return value of type `A`. Since a fully parametric function cannot create values of an arbitrary type `A` from scratch, we see no possibility of computing `h` within the body of `before`.

Can we prove rigorously that a value of type `C => A` cannot be computed within the body of `before`? Or, perhaps, a clever trick *could* produce a value of that type? So far, we only saw informal arguments about whether values of certain types can be computed. To make the arguments rigorous, we need to translate statements such as "*a fully parametric function* `before` *can compute a value of type* `C => A`" into mathematical formulas, with rigorous rules for proving them true or false.

In Section 3.5.3, we denoted by $\mathcal{CH}(A)$ the proposition "the $\mathcal{C}$ode $\mathcal{H}$as a value of type A". By "the code" we now mean the body of a given fully parametric function. So, the notation $\mathcal{CH}(A)$ is not completely adequate because the validity of the proposition $\mathcal{CH}(A)$ depends not only on the choice of the type A but also on the place in the code fragment where the value of type A needs to be computed. What exactly is this additional dependency? In the above examples, we used the *types* of a function's arguments when reasoning about getting a value of a given type A. Thus, a precise description of the proposition $\mathcal{CH}(A)$ is

$$\begin{aligned} \mathcal{CH}\text{-proposition}: \quad & \text{a fully parametric function having arguments of types} \\ & X, Y, ..., Z \text{ can compute a value of type } A \quad . \end{aligned} \tag{5.1}$$

Here X, Y, ..., Z, A may be either type parameters or more complicated type expressions such as $B \to C$ or $(C \to D) \to E$, built from other type parameters.

If arguments of types X, Y, ..., Z are given, it means we already have values of these types. So, the propositions $\mathcal{CH}(X)$, $\mathcal{CH}(Y)$, ..., $\mathcal{CH}(Z)$ will be true. Thus, proposition (5.1) is equivalent to "$\mathcal{CH}(A)$ assuming $\mathcal{CH}(X)$, $\mathcal{CH}(Y)$, ..., $\mathcal{CH}(Z)$". In mathematical logic, a statement of this form is called a **sequent** and is denoted by

$$\mathcal{CH}(X), \mathcal{CH}(Y), ..., \mathcal{CH}(Z) \vdash \mathcal{CH}(A) \quad . \tag{5.2}$$

The assumptions $\mathcal{CH}(X)$, $\mathcal{CH}(Y)$, ..., $\mathcal{CH}(Z)$ are called **premises** and the proposition $\mathcal{CH}(A)$ is called the **goal**. Showing rigorously the possibility of computing values in functions means proving sequents of the form (5.2). Our previous examples are denoted by the following sequents:

$$\begin{aligned} \texttt{fmap for Option}: \quad & \mathcal{CH}(A \to B) \vdash \mathcal{CH}(\texttt{Option[A]} \to \texttt{Option[B]}) \\ \text{the function } \texttt{before}: \quad & \mathcal{CH}(A \to B), \mathcal{CH}(B \to C) \vdash \mathcal{CH}(A \to C) \\ \text{value of type } A \text{ within } \texttt{fmap}: \quad & \mathcal{CH}(A \to B), \mathcal{CH}(\texttt{Option[A]}) \vdash \mathcal{CH}(A) \\ \text{value of type } C \to A \text{ within } \texttt{before}: \quad & \mathcal{CH}(A \to B), \mathcal{CH}(B \to C) \vdash \mathcal{CH}(C \to A) \end{aligned}$$

Calculations in formal logic are called **proofs**. So, in this section we gave informal arguments towards proving the first two sequents and disproving the last two. We will now develop tools for rigorous reasoning about sequents.

A proposition $\mathcal{CH}(A)$ may be true for one set of premises such as $\mathcal{CH}(X)$, $\mathcal{CH}(Y)$, ..., $\mathcal{CH}(Z)$ but false for another. Here and in the following sections, we will be reasoning about $\mathcal{CH}$-propositions within the body of a *chosen* fully parametric function, i.e. with a fixed set of premises. We will then temporarily omit the premises and use the shorter notation $\mathcal{CH}(A)$.

5.1.2 Type notation and $\mathcal{CH}$-propositions for standard type constructions

In Section 3.5.3 we saw examples of reasoning about $\mathcal{CH}$-propositions for case classes and for disjunctive types. We will now extend this reasoning systematically to all type constructions that pro-

grams could use. A special type notation explained in this section will help us write type expressions more concisely. (See Appendix A for reference on the type notation.)

There are six **standard type constructions** supported by all functional languages: primitive types (including `Unit` type and the void type, called `Nothing` in Scala), product (tuple) types, co-product (disjunctive) types, function types, parameterized types, and recursive types. We will now derive the rules for writing $\mathcal{CH}$-propositions for each of these type constructions (except recursive types).

1a) Rule for `Unit` type The `Unit` type has only a single value `()`, and this value (an "empty tuple") can be *always* computed since it does not need any previous data:

```
def f[...]: ... = {
  ...
  val x: Unit = () // We can always compute a ‘Unit‘ value.
  ...
```

So, the proposition $\mathcal{CH}(\texttt{Unit})$ is always true. In the type notation, the `Unit` type is denoted by $\mathbb{1}$.

Named unit types also have a single value that is always possible to compute. For example,

```
final case class N1()
```

defines a named unit type; we can compute

```
val x: N1 = N1()
```

So, the proposition $\mathcal{CH}(\texttt{N1})$ is always true. Named unit types are denoted by $\mathbb{1}$, just as the `Unit` type itself.

1b) Rule for the void type The Scala type `Nothing` has no values, so the proposition $\mathcal{CH}(\texttt{Nothing})$ is always false. The type `Nothing` is denoted by $\mathbb{0}$ in the type notation.

1c) Rule for primitive types For a specific primitive (or library-defined) type such as `Int` or `String`, the corresponding $\mathcal{CH}$-proposition is *always true* because we could use any constant value, e.g.

```
def f[...]: ... {
   ...
   val x: String = "abc" // We can always compute a ‘String‘ value.
   ...
```

So, the rule for primitive types is the same as the rule for the `Unit` type.

2) Rule for tuple types To compute a value of a tuple type `(A, B)` requires computing a value of type `A` *and* a value of type `B`. This is expressed by the logic formula $\mathcal{CH}(\texttt{(A, B)}) = \mathcal{CH}(A) \wedge \mathcal{CH}(B)$. A similar formula holds for case classes, as Eq. (3.2) shows. In the type notation, the tuple `(A, B)` is written as $A \times B$. Tuples and case classes with more than two parts are denoted similarly as $A \times B \times ... \times C$. For example, the Scala definition

```
case class Person(firstName: String, lastName: String, age: Int)
```

is written in the type notation as $\text{String} \times \text{String} \times \text{Int}$. So, the rule for tuple types is

$$\mathcal{CH}(A \times B \times ... \times C) = \mathcal{CH}(A) \wedge \mathcal{CH}(B) \wedge ... \wedge \mathcal{CH}(C) \quad .$$

3) Rule for disjunctive types A disjunctive type may consist of several case classes. Having a value of a disjunctive type means to have a value of (at least) one of those case classes. An example of translating this relationship into a formula was shown by Eq. (3.1). For the standard disjunctive type `Either[A, B]`, we have the logical formula $\mathcal{CH}(\texttt{Either[A, B]}) = \mathcal{CH}(A) \vee \mathcal{CH}(B)$. In the type notation, the Scala type `Either[A, B]` is written as $A + B$. A longer example: the Scala definition

```
sealed trait RootsOfQ
final case class NoRoots()                    extends RootsOfQ
final case class OneRoot(x: Double)           extends RootsOfQ
final case class TwoRoots(x: Double, y: Double) extends RootsOfQ
```

is translated to the type notation as

$$\text{RootsOfQ} = 1 + \text{Double} + \text{Double} \times \text{Double} \quad .$$

The type notation is significantly shorter because it omits all case class names and part names from the type definitions. In this notation, the rule for disjunctive types is

$$\mathcal{CH}\left(A + B + ... + C\right) = \mathcal{CH}(A) \vee \mathcal{CH}(B) \vee ... \vee \mathcal{CH}(C) \quad .$$

4) Rule for function types Consider now a function type such as `A => B`. (This type is written in the type notation as $A \to B$.) To compute a value of that type, we need to write code such as

```
val f: A => B = { (a: A) =>
  ??? // Compute a value of type B in this scope.
}
```

The inner scope of the function needs to compute a value of type B, and the given value `a:A` may be used for that. So, $\mathcal{CH}(A \to B)$ is true if and only if we are able to compute a value of type B when we are given a value of type A. To translate this statement into the language of logical propositions, we need to use the logical **implication**, $\mathcal{CH}(A) \Rightarrow \mathcal{CH}(B)$, which means that $\mathcal{CH}(B)$ can be proved if $\mathcal{CH}(A)$ already holds. So the rule for function types is

$$\mathcal{CH}(A \to B) = \mathcal{CH}(A) \Rightarrow \mathcal{CH}(B) \quad .$$

5) Rule for parameterized types Consider a function with type parameters, e.g.

```
def f[A, B]: A => (A => B) => B = { x => g => g(x) }
```

Being able to define the body of such a function is equivalent to being able to compute a value of type `A => (A => B) => B` for *all* possible types `A` and `B`. In the notation of formal logic, this is written as

$$\mathcal{CH}\left(\forall(A, B).\, A \to (A \to B) \to B\right)$$

and is equivalent to

$$\forall(A, B).\, \mathcal{CH}\left(A \to (A \to B) \to B\right) \quad .$$

The code notation for the parameterized function `f` is

$$f^{A,B} : A \to (A \to B) \to B \quad ,$$

and its type can be written as

$$\forall(A, B).\, A \to (A \to B) \to B \quad .$$

The symbol $\forall$ means "for all" and is known as the **universal quantifier** in logic.

In Scala, longer type expressions can be named and their names (called **type aliases**) can be used to make code shorter. Type aliases may also contain type parameters. Defining and using a type alias for the type of the function `f` looks like this,

```
type F[A, B] = A => (A => B) => B
def f[A, B]: F[A, B] = { x => g => g(x) }
```

This is written in the type notation as

$$F^{A,B} \triangleq A \to (A \to B) \to B \quad ,$$
$$f^{A,B} : F^{A,B} \triangleq x^{:A} \to g^{:A \to B} \to g(x) \quad ,$$

or equivalently (although somewhat less readably)

$$f : \left(\forall(A, B).\, F^{A,B}\right) \triangleq \forall(A, B).\, x^{:A} \to g^{:A \to B} \to g(x) \quad .$$

In Scala 3, the function `f` can be written as a value via the syntax

```
val f: [A, B] => A => (A => B) => B = {   // Valid only in Scala 3.
  [A, B] => (x: A) => (g: A => B) => g(x)
}
```

Type construction	Scala syntax	Type notation	$\mathcal{CH}$-proposition
type parameter	`[A]`	A	$\mathcal{CH}(A)$
product type (tuple)	`(A, B)`	$A \times B$	$\mathcal{CH}(A) \wedge \mathcal{CH}(B)$
disjunctive type	`Either[A, B]`	$A + B$	$\mathcal{CH}(A) \vee \mathcal{CH}(B)$
function type	`A => B`	$A \to B$	$\mathcal{CH}(A) \Rightarrow \mathcal{CH}(B)$
unit or a "named unit" type	`Unit`	$\mathbb{1}$	$\mathcal{CH}(\mathbb{1}) = True$
primitive type	`Int`, `String`, ...	Int, String, ...	$\mathcal{CH}(\text{Int}) = True$
void type	`Nothing`	$\mathbb{0}$	$\mathcal{CH}(\mathbb{0}) = False$
value parameterized by type	`def f[A]: F[A]`	$f^A : F^A$	$\forall A.\, \mathcal{CH}(F^A)$
type with quantifier	`[A] => F[A]` (Scala 3)	$\forall A.\, F^A$	$\forall A.\, \mathcal{CH}(F^A)$

Table 5.1: The correspondence between type constructions and $\mathcal{CH}$-propositions.

This syntax corresponds more closely to the mathematical notation shown above.

So, the rule for parameterized types with the type notation F^A is

$$\mathcal{CH}(\forall A.\, F^A) = \forall A.\, \mathcal{CH}(F^A) \quad .$$

Case classes and disjunctive types use *names* for the types and their parts. However, those names only add convenience for programmers and do not affect the computational properties of types. The type notation is designed to support nameless type expressions.

Table 5.1 summarizes the type notation and also shows how to translate it into logic formulas with propositions of the form $\mathcal{CH}(...)$.

The precedence of operators in the type notation is chosen to have fewer parentheses in the type expressions that are frequently used. The rules of precedence are:

- The type product operator ($\times$) groups stronger than the disjunctive operator ($+$), so that type expressions such as $A + B \times C$ have the same operator precedence as in standard arithmetic. That is, $A + B \times C$ means $A + (B \times C)$. This convention makes type expressions easier to reason about (for people familiar with arithmetic).

- The function type arrow ($\to$) groups weaker than the operators $+$ and $\times$, so that often-used types such as $A \to \mathbb{1} + B$ (representing `A => Option[B]`) or $A \times B \to C$ (representing `((A, B)) => C`) can be written without any parentheses. Type expressions such as $(A \to B) \times C$ will require parentheses but are used less often.

- The type quantifiers group weaker than all other operators, so we can write types such as $\forall A.\, A \to A \to A$ without parentheses. Type quantifiers are most often placed outside a type expression. When this is not the case, parentheses are necessary, e.g. in the type expression $(\forall A.\, A \to A) \to \mathbb{1}$.

5.1.3 Solved examples: Type notation

From now on, we will prefer to write types in the type notation rather than in the Scala syntax. The type notation allows us to write nameless type expressions and, in particular, makes the structure of disjunctive types and their parts more transparent, compared with the Scala syntax. Names of types and parts of types are, of course, helpful for reminding the programmer of the meaning of data in a program. However, writing names for every part of every type is not helpful for reasoning about the

properties of types. Type notation makes reasoning about types easier, as we will see throughout this chapter. Once the programmer has finished deriving the necessary types and verifying their properties, the type expressions can be straightforwardly translated from the type notation into Scala code.

Let us get some experience converting between type notation and Scala code.

Example 5.1.3.1 Define a function `delta` taking an argument `x` and returning the pair `(x, x)`. Derive the most general type for this function. Write the type signature of `delta` in the type notation, and translate it into a $\mathcal{CH}$-proposition. Simplify the $\mathcal{CH}$-proposition if possible.

Solution Begin by writing the code of the function:

```
def delta(x: ...) = (x, x)
```

To derive the most general type for `delta`, first assume `x:A`, where `A` is a type parameter; then the tuple `(x, x)` has type `(A, A)`. We do not see any constraints on the type parameter `A`. So the type parameter represents an arbitrary type and needs to be added to the type signature of `delta`:

```
def delta[A](x: A): (A, A) = (x, x)
```

We find that the most general type of `delta` is `A => (A, A)`. We also note that there is only one way of implementing a fully parametric function with type signature `A => (A, A)`: the function must duplicate its given argument.

It is convenient to use the letter Δ for the function `delta`. In the type notation, the type signature of Δ is written as

$$\Delta^A : A \to A \times A \quad .$$

So the proposition $\mathcal{CH}(\Delta)$ (meaning "the function Δ can be implemented") is

$$\mathcal{CH}(\Delta) = \forall A. \mathcal{CH}\left(A \to A \times A\right) \quad .$$

In the type expression $A \to A \times A$, the product symbol ($\times$) binds stronger than the function arrow ($\to$), so the parentheses in $A \to (A \times A)$ may be omitted.

Using the rules for transforming $\mathcal{CH}$-propositions, we rewrite

$$\begin{aligned} & \mathcal{CH}(A \to A \times A) \\ \text{rule for function types :} \quad & = \mathcal{CH}(A) \Rightarrow \mathcal{CH}(A \times A) \\ \text{rule for tuple types :} \quad & = \mathcal{CH}(A) \Rightarrow (\mathcal{CH}(A) \wedge \mathcal{CH}(A)) \quad . \end{aligned}$$

Thus the proposition $\mathcal{CH}(\Delta)$ is equivalent to

$$\mathcal{CH}(\Delta) = \forall A. \mathcal{CH}(A) \Rightarrow (\mathcal{CH}(A) \wedge \mathcal{CH}(A)) \quad .$$

Example 5.1.3.2 The standard disjunctive types `Either[A, B]` and `Option[A]` are written in the type notation as

$$\text{Either}^{A,B} \triangleq A + B \quad , \qquad \text{Opt}^A \triangleq \mathbb{1} + A \quad .$$

The type `Either[A, B]` is written as $A + B$ by definition of the disjunctive type notation (+). The type `Option[A]` has two disjoint cases, `None` and `Some[A]`. The case class `None` is a "named `Unit`" and is denoted by $\mathbb{1}$. The case class `Some[A]` contains a single value of type A. So, the type notation for `Option[A]` is $\mathbb{1} + A$.

Example 5.1.3.3 The Scala definition of the disjunctive type `UserAction`,

```
sealed trait UserAction
final case class SetName(first: String, last: String) extends UserAction
final case class SetEmail(email: String)              extends UserAction
final case class SetUserId(id: Long)                  extends UserAction
```

is written in the type notation as

$$\text{UserAction} \triangleq \text{String} \times \text{String} + \text{String} + \text{Long} \quad . \tag{5.3}$$

The type operation × groups stronger than +, as in arithmetic. To derive the type notation (5.3), we first drop all names from case classes and get three nameless tuples `(String, String)`, `(String)`, and `(Long)`. Each of these tuples is then converted into a product using the operator ×, and all products are "summed" in the type notation using the operator +.

Example 5.1.3.4 The parameterized disjunctive type `Either3` is a generalization of `Either`:

```
sealed trait Either3[A, B, C]
final case class Left[A, B, C](x: A)   extends Either3[A, B, C]
final case class Middle[A, B, C](x: B) extends Either3[A, B, C]
final case class Right[A, B, C](x: C)  extends Either3[A, B, C]
```

This disjunctive type is written in the type notation as

$$\text{Either3}^{A,B,C} \triangleq A + B + C \quad .$$

Example 5.1.3.5 Define a Scala type constructor `F[A]` corresponding to the type notation

$$F^A \triangleq \mathbb{1} + \text{Int} \times A \times A + \text{Int} \times (\text{Int} \to A) \quad .$$

Solution The formula for F^A defines a disjunctive type `F[A]` with three parts. To implement `F[A]` in Scala, we need to choose names for each of the disjoint parts, which will become case classes. For the purposes of this example, let us choose names `F1`, `F2`, and `F3`. Each of these case classes needs to have the same type parameter `A`. So we begin writing the code as

```
sealed trait F[A]
final case class F1[A](...) extends F[A]
final case class F2[A](...) extends F[A]
final case class F3[A](...) extends F[A]
```

Each of these case classes represents one part of the disjunctive type: `F1` represents $\mathbb{1}$, `F2` represents $\text{Int} \times A \times A$, and `F3` represents $\text{Int} \times (\text{Int} \to A)$. To define these case classes, we need to name their parts. The final code is

```
sealed trait F[A]
final case class F1[A]()                     extends F[A]  // Named unit type.
final case class F2[A](n: Int, x1: A, x2: A) extends F[A]
final case class F3[A](n: Int, f: Int => A)  extends F[A]
```

The names `n`, `x1`, `x2`, and `f` are chosen purely for convenience.

Example 5.1.3.6 Write the type signature of the function

```
def fmap[A, B](f: A => B): Option[A] => Option[B]
```

in the type notation.

Solution This is a curried function, so we first rewrite the type signature as

```
def fmap[A, B]: (A => B) => Option[A] => Option[B]
```

The type notation for `Option[A]` is $\mathbb{1} + A$. Now we can write the type signature of `fmap` as

$$\text{fmap}^{A,B} : (A \to B) \to \mathbb{1} + A \to \mathbb{1} + B \quad ,$$

$$\text{or equivalently :} \quad \text{fmap} : \forall (A, B).\ (A \to B) \to \mathbb{1} + A \to \mathbb{1} + B \quad .$$

We do not put parentheses around $\mathbb{1} + A$ and $\mathbb{1} + B$ because the function arrow (→) groups weaker than the other type operations. Parentheses around $(A \to B)$ are required.

We will usually prefer to write type parameters in superscripts rather than under type quantifiers. So, for example, we will write $\text{id}^A \triangleq x^{:A} \to x$ rather than $\text{id} \triangleq \forall A.\ x^{:A} \to x$.

5.1.4 Exercises: Type notation

Exercise 5.1.4.1 Define a Scala disjunctive type `Q[T,A]` corresponding to the type notation

$$Q^{T,A} \triangleq \mathbb{1} + T \times A + \text{Int} \times (T \to T) + \text{String} \times A \quad .$$

Exercise 5.1.4.2 Rewrite `Either[(A, Int), Either[(A, Char), (A, Float)]]` in the type notation.

Exercise 5.1.4.3 Define a Scala type `OptE[A, B]` written in the type notation as $\text{OptE}^{A,B} \triangleq \mathbb{1} + A + B$.

Exercise 5.1.4.4 Write a Scala type signature for the fully parametric function

$$\text{flatMap}^{A,B} : \mathbb{1} + A \to (A \to \mathbb{1} + B) \to \mathbb{1} + B$$

and implement this function, preserving information as much as possible.

5.2 The logic of $\mathcal{CH}$-propositions

5.2.1 Motivation and first examples

So far, we were able to convert statements such as "*a fully parametric function can compute values of type A*" into logical propositions of the form $\mathcal{CH}(A)$ that we called $\mathcal{CH}$-propositions. The next step is to determine the proof rules suitable for reasoning about $\mathcal{CH}$-propositions.

Formal logic uses axioms and derivation rules for proving that certain formulas are true or false. A simple example of a true formula is "any proposition α is equivalent to itself",

$$\forall \alpha. \, \alpha = \alpha \quad .$$

In logic, equivalence of propositions is usually understood as **implication** ($\Rightarrow$) in both directions: $\alpha = \beta$ means $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$. So, the above formula is the same as

$$\forall \alpha. \, \alpha \Rightarrow \alpha \quad .$$

If the proposition α is a $\mathcal{CH}$-proposition, $\alpha \triangleq \mathcal{CH}(A)$ for some type A, we obtain the formula

$$\forall A. \, \mathcal{CH}(A) \Rightarrow \mathcal{CH}(A) \quad . \tag{5.4}$$

We expect true $\mathcal{CH}$-propositions to correspond to types that *can* be computed in a fully parametric function. Let us see if this example fits our expectations. We can rewrite Eq. (5.4) as

$$\begin{aligned} & \forall A. \, \underline{\mathcal{CH}(A) \Rightarrow \mathcal{CH}(A)} \\ \text{rule for function types :} \quad & = \underline{\forall A.} \, \mathcal{CH}\left(A \to A\right) \\ \text{rule for parameterized types :} \quad & = \mathcal{CH}\left(\forall A. \, A \to A\right) \quad . \end{aligned}$$

The last line shows the $\mathcal{CH}$-proposition that corresponds to the function type $\forall A. \, A \to A$. Translating the type notation into a Scala type signature, we get

```
def f[A]: A => A
```

This type signature can be easily implemented,

```
def f[A]: A => A = { x => x }
```

So, in this example we see how we converted a true formula in logic into the type of a value `f` that can be implemented.

While the formula $\forall \alpha. \, \alpha = \alpha$ may be self-evident, the point of using formal logic is to have a set of axioms and proof rules that allow us to deduce *all* correct formulas systematically, without need for intuition or guessing. What axioms and proof rules are suitable for proving $\mathcal{CH}$-propositions?

A well-known set of logical rules is called Boolean logic. In that logic, each proposition is either $True$ or $False$, and the implication operation ($\Rightarrow$) is *defined* by

$$(\alpha \Rightarrow \beta) \triangleq ((\neg\alpha) \vee \beta) \quad . \tag{5.5}$$

To verify a formula, substitute $True$ or $False$ into every variable and check if the formula has the value $True$ in all possible cases. The result can be arranged into a truth table. The basic operations (disjunction, conjunction, negation, and implication) have the following truth tables:

α	β	$\alpha \vee \beta$	$\alpha \wedge \beta$	$\neg\alpha$	$\alpha \Rightarrow \beta$
$True$	$True$	$True$	$True$	$False$	$True$
$True$	$False$	$True$	$False$	$False$	$False$
$False$	$True$	$True$	$False$	$True$	$True$
$False$	$False$	$False$	$False$	$True$	$True$

The formula $\alpha \Rightarrow \alpha$ has the value $True$ whether α itself is $True$ or $False$. This check is sufficient to show that $\forall\alpha.\, \alpha \Rightarrow \alpha$ is true in Boolean logic.

Here is the truth table for the formula $\forall(\alpha, \beta).\, (\alpha \wedge \beta) \Rightarrow \alpha$; that formula is true in Boolean logic since all values in the last column are $True$:

α	β	$\alpha \wedge \beta$	$(\alpha \wedge \beta) \Rightarrow \alpha$
$True$	$True$	$True$	$True$
$True$	$False$	$False$	$True$
$False$	$True$	$False$	$True$
$False$	$False$	$False$	$True$

The formula $\forall(\alpha, \beta).\, \alpha \Rightarrow (\alpha \wedge \beta)$ is not true in Boolean logic, which we can see from the following truth table (one value in the last column is $False$):

α	β	$\alpha \wedge \beta$	$\alpha \Rightarrow (\alpha \wedge \beta)$
$True$	$True$	$True$	$True$
$True$	$False$	$False$	$False$
$False$	$True$	$False$	$True$
$False$	$False$	$False$	$True$

Table 5.2 shows more examples of logical formulas that are true in Boolean logic. Each formula is first given in terms of $\mathcal{CH}$-propositions (we denoted $\alpha \triangleq \mathcal{CH}(A)$ and $\beta \triangleq \mathcal{CH}(B)$ for brevity) and then into a Scala type signature of a function that can be implemented.

Table 5.3 some examples of formulas that are *not true* in Boolean logic. Translated into type formulas and then into Scala, these formulas yield type signatures that *cannot* be implemented by fully parametric functions.

At first sight, it appears from these examples that whenever a logical formula is true in Boolean logic, the corresponding type signature can be implemented in code, and vice versa. However, this is *incorrect*: the rules of Boolean logic are not suitable for reasoning about types in a functional language. Below we will see some examples of formulas that are true in Boolean logic but yield unimplementable type signatures.

Logic formula	Type formula	Scala code
$\forall\alpha.\, \alpha \Rightarrow \alpha$	$\forall A.\, A \to A$	`def id[A](x: A): A = x`
$\forall\alpha.\, \alpha \Rightarrow True$	$\forall A.\, A \to \mathbb{1}$	`def toUnit[A](x: A): Unit = ()`
$\forall(\alpha,\beta).\, \alpha \Rightarrow (\alpha \vee \beta)$	$\forall(A,B).\, A \to A + B$	`def toL[A, B](x: A): Either[A, B] = Left(x)`
$\forall(\alpha,\beta).\, (\alpha \wedge \beta) \Rightarrow \alpha$	$\forall(A,B).\, A \times B \to A$	`def first[A, B](p: (A, B)): A = p._1`
$\forall(\alpha,\beta).\, \alpha \Rightarrow (\beta \Rightarrow \alpha)$	$\forall(A,B).\, A \to (B \to A)$	`def const[A, B](x: A): B => A = (_ => x)`

Table 5.2: Examples of logical formulas that are true theorems in Boolean logic.

Logic formula	Type formula	Scala type signature
$\forall\alpha.\, True \Rightarrow \alpha$	$\forall A.\, \mathbb{1} \to A$	`def f[A](x: Unit): A`
$\forall(\alpha,\beta).\, (\alpha \vee \beta) \Rightarrow \alpha$	$\forall(A,B).\, A + B \to A$	`def f[A,B](x: Either[A, B])`
$\forall(\alpha,\beta).\, \alpha \Rightarrow (\alpha \wedge \beta)$	$\forall(A,B).\, A \to A \times B$	`def f[A,B](p: A): (A, B)`
$\forall(\alpha,\beta).\, (\alpha \Rightarrow \beta) \Rightarrow \alpha$	$\forall(A,B).\, (A \to B) \to A$	`def f[A,B](x: A => B): A`

Table 5.3: Examples of logical formulas that are *not* true in Boolean logic.

5.2.2 Example: Failure of Boolean logic for type reasoning

To see an explicit example of obtaining an incorrect result when using Boolean logic to reason about values computed by fully parametric functions, consider the following type,

$$\forall(A,B,C).\, (A \to B + C) \to (A \to B) + (A \to C) \quad , \tag{5.6}$$

which corresponds to the Scala type signature

```
def bad[A, B, C](g: A => Either[B, C]): Either[A => B, A => C] = ???
```

The function `bad` cannot be implemented as a fully parametric function. To see why, consider that the only available data is a function $g^{:A \to B+C}$, which returns values of type B or C depending (in some unknown way) on the input value of type A. The function `bad` must return either a function of type $A \to B$ or a function of type $A \to C$. How can the code of `bad` make this decision? The only input data is the function g that takes an argument of type A. We could imagine applying g to various arguments of type A and to see whether g returns a B or a C. However, the type A is arbitrary, and a fully parametric function cannot produce a value of type A in order to apply g to it. So the decision about whether to return $A \to B$ or $A \to C$ must be independent of g; that decision must be hard-coded in the function `bad`.

Suppose we hard-coded the decision to return a function of type $A \to B$. How can we create a function of type $A \to B$ in the body of `bad`? Given a value $x^{:A}$ of type A, we would need to compute some value of type B. Since the type B is arbitrary (it is a type parameter), we cannot produce a value of type B from scratch. The only potential source of values of type B is the given function g. The only way of using g is to apply it to $x^{:A}$. However, for some x, the value $g(x)$ may be of the form `Right(c)`, where `c` is of type C. In that case, we will have a value of type C, not B. So, in general, we cannot guarantee that we can always obtain a value of type B from a given value $x^{:A}$. This means we cannot build a function of type $A \to B$ out of the function g. Similarly, we cannot build a function of type $A \to C$ out of g.

Whether we decide to return $A \to B$ or $A \to C$, we will not be able to return a value of the required type, as we just saw. We must conclude that we cannot implement `bad` as a fully parametric function.

We could try to switch between $A \to B$ and $A \to C$ depending on a given value of type A. This idea, however, means that we are working with a different type signature:

$$\forall(A, B, C).\ (A \to B + C) \to A \to (A \to B) + (A \to C) \quad .$$

This type signature *can* be implemented, for instance, by this Scala code:

```
def q[A, B, C](g: A => Either[B, C]): A => Either[A => B, A => C] = { a =>
  g(a) match {
    case Left(b) => Left(_ => b)
    case Right(c) => Right(_ => c)
  }
}
```

But this is not the required type signature (5.6).

Now let us convert the type signature (5.6) into a $\mathcal{CH}$-proposition:

$$\forall(\alpha, \beta, \gamma).\ (\alpha \Rightarrow (\beta \vee \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma)) \quad , \tag{5.7}$$
$$\text{where} \quad \alpha \triangleq \mathcal{CH}(A), \quad \beta \triangleq \mathcal{CH}(B), \quad \gamma \triangleq \mathcal{CH}(C) \quad .$$

It turns out that this formula is true in Boolean logic. To prove this, we need to show that Eq. (5.7) is equal to $True$ for any Boolean values of the variables α, β, γ. One way is to rewrite the expression (5.7) using the rules of Boolean logic, such as Eq. (5.5):

$$\begin{aligned}
& \underline{\alpha \Rightarrow} (\beta \vee \gamma) \\
\text{definition of } \Rightarrow \text{ via Eq. (5.5)}: \quad & = (\neg\alpha) \vee \beta \vee \gamma \quad , \\
& \underline{(\alpha \Rightarrow \beta)} \vee \underline{(\alpha \Rightarrow \gamma)} \\
\text{definition of } \Rightarrow \text{ via Eq. (5.5)}: \quad & = \underline{(\neg\alpha)} \vee \beta \vee \underline{(\neg\alpha)} \vee \gamma \\
\text{property } x \vee x = x \text{ in Boolean logic}: \quad & = (\neg\alpha) \vee \beta \vee \gamma \quad ,
\end{aligned}$$

showing that $\alpha \Rightarrow (\beta \vee \gamma)$ is in fact *equal* to $(\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma)$ in Boolean logic.

Let us also give a proof via truth-value reasoning. The only possibility for an implication $X \Rightarrow Y$ to be $False$ is when $X = True$ and $Y = False$. So, Eq. (5.7) can be $False$ only if $(\alpha \Rightarrow (\beta \vee \gamma)) = True$ and $(\alpha \Rightarrow \beta) \vee (\alpha \Rightarrow \gamma) = False$. A disjunction can be false only when both parts are false; so we must have both $(\alpha \Rightarrow \beta) = False$ and $(\alpha \Rightarrow \gamma) = False$. This is only possible if $\alpha = True$ and $\beta = \gamma = False$. But, with these value assignments, we find $(\alpha \Rightarrow (\beta \vee \gamma)) = False$ rather than $True$ as we assumed. It follows that we cannot ever make Eq. (5.7) equal to $False$. This proves Eq. (5.7) to be true in Boolean logic.

5.2.3 The rules of proof for $\mathcal{CH}$-propositions

Section 5.2.2 shows that some true formulas in Boolean logic do not correspond to types of *implementable* fully parametric functions. However, we have also seen several other examples where Boolean logic does provide correct results: some true formulas correspond to implementable type signatures, while some false formulas correspond to non-implementable type signatures.

Instead of guessing whether the rules of Boolean logic are suitable in a given case, let us derive the suitable logical axioms and proof rules systematically.

The proposition $\mathcal{CH}(A)$ is true when a value of type A can be computed by a fully parametric function with a given type signature. To describe all possible ways of computing a value of type A, we need to enumerate all possible ways of writing code within a fully parametric function. The requirement of parametricity means that we are not allowed to use any specific types such as `Int` or `String`. We are only allowed to work with values of unknown types described by the given type parameters. We cannot use any concrete values such as `123` or `"hello"`, or any library functions that work with specific (non-parametric) types; however, we are permitted to use fully parametric types, such as `Either[A, B]` or `Option[A]`. The allowed eight code constructs are illustrated in this code fragment:

```
def f[A, B, ...](a: A, b: B)... = { // (A given type signature.)
  val x1: Unit = ()            // 1) Create a value of type Unit.
  val x2: A = a                // 2) Use a given argument.
  val x3 = { x: A => ... }     // 3) Create a function.
  val x4: D = x3(x2)           // 4) Use a function.
  val x5: (A, B) = (a, b)      // 5) Create a tuple.
  val x6: B = x5._2            // 6) Use a tuple.
  val x7: Option[A] = Some(x2) // 7) Create values of a disjunctive type.
  val x8 = x7 match { ... }    // 8) Use values of a disjunctive type.
}
```

A value of type X can be computed (i.e. $\mathcal{CH}(X)$ is true) if and only if we can create a sequence of computed values such as `x1`, `x2`, ..., each being the result of one of these eight code constructs, ending with a value of type X. So, each of the eight code constructs should correspond to a logical rule for proving a $\mathcal{CH}$-proposition.

A set of axioms and proof rules defines a **formal logic**. So, we will now find the proof rules that will define *the* formal logic suitable for reasoning about $\mathcal{CH}$-propositions.

Because each proof rule will be obtained from a specific code construct, any $\mathcal{CH}$-proposition such as $\mathcal{CH}(X)$ proved by applying a sequence of these rules will automatically correspond to a code fragment that combines the relevant code constructs to compute a value of type X. Conversely, any fully parametric code computing a value of type X must be a combination of some of the eight code constructs, and that combination can be automatically translated into a sequence of applications of proof rules in the logic to produce a proof of the proposition $\mathcal{CH}(X)$.

Let us now write down the proof rules that follow from the eight code constructs. We will need to consider the full formulation (5.1) of $\mathcal{CH}$-propositions and write them as sequents such as Eq. (5.2). For brevity, we define $\alpha \triangleq \mathcal{CH}(A)$, $\beta \triangleq \mathcal{CH}(B)$, etc. It is also customary to use the letter Γ to denote a set of premises, such as $\mathcal{CH}(X)$, $\mathcal{CH}(Y)$, ..., $\mathcal{CH}(Z)$ in Eq. (5.2). So, we can write a shorter formula $\Gamma \vdash \alpha$ instead of the sequent (5.2).

With these notations, we will enumerate all the possible ways of proving that a $\mathcal{CH}$-proposition is true.

1) Create a `Unit` value At any place in the code, we may write the expression `()` of type `Unit`. This expression corresponds to a proof of the proposition $\mathcal{CH}(\mathbb{1})$ with any set Γ of premises (even with an empty set of premises). So, the sequent $\Gamma \vdash \mathcal{CH}(\mathbb{1})$ is always true. The code corresponding to the proof of this sequent is an expression that creates a value of the `Unit` type:

$$\text{Proof}\,(\Gamma \vdash \mathcal{CH}(\mathbb{1})) = 1 \quad ,$$

where we denoted by 1 the value `()`.

In formal logic, a sequent that is found to be always true, such as our $\Gamma \vdash \mathcal{CH}(\mathbb{1})$, is called an **axiom** and is written in the following notation,

$$\frac{}{\Gamma \vdash \mathcal{CH}(\mathbb{1})}\ \text{(create unit)} \qquad .$$

The "fraction with a label" represents a proof rule. The denominator of the "fraction" is the target sequent that we need to prove. The numerator of the "fraction" can have zero or more other sequents that need to be proved before the target sequent can be proved. In this case, the set of previous sequents is empty: the target sequent is an axiom and so requires no previous sequents for its proof. The label "create unit" is an arbitrary name used to refer to the rule.

2) Use a given argument At any place within the code of a fully parametric function, we may use one of the function's arguments, say $x^{:A}$. If some argument has type A, it means that $\alpha \triangleq \mathcal{CH}(A)$ belongs to the set of premises of the sequent we are trying to prove. To indicate this, we write the set of premises as "Γ, α". The code construct `x:A` computes a value of type A, i.e. show that α is true, given these premises. This is expressed by the sequent $\Gamma, \alpha \vdash \alpha$. The proof of this sequent corresponds to an expression that returns one of the given arguments (which we here called $x^{:A}$),

$$\text{Proof}(\Gamma, \alpha \vdash \alpha) = x^{:A} \quad .$$

This sequent is an axiom since its proof requires no previous sequents. The formal logic notation for this axiom is

$$\frac{}{\Gamma, \alpha \vdash \alpha} \quad \text{(use arg)} \qquad .$$

3) Create a function At any place in the code, we may compute a nameless function of type, say, $A \to B$, by writing `(x:A) => expr` as long as a value `expr` of type B can be computed in the inner scope of the function. The code for `expr` is also required to be fully parametric; it may use `x` and/or other values visible in that scope. So we now need to answer the question of whether a fully parametric function can compute a value of type B, given an argument of type A as well as all other arguments previously given to the parent function. This question is answered by a sequent whose premises contain one more proposition, $\mathcal{CH}(A)$, in addition to all previously available premises. Translating this into the language of $\mathcal{CH}$-propositions, we find that we will prove the sequent

$$\Gamma \vdash \mathcal{CH}(A \to B) \quad = \quad \Gamma \vdash \mathcal{CH}(A) \Rightarrow \mathcal{CH}(B) \quad \triangleq \quad \Gamma \vdash \alpha \Rightarrow \beta$$

if we can prove the sequent $\Gamma, \mathcal{CH}(A) \vdash \mathcal{CH}(B) = \Gamma, \alpha \vdash \beta$. In the notation of formal logic, this is a **derivation rule** (rather than an axiom) and is written as

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta} \quad \text{(create function)} \qquad .$$

The **turnstile** symbol, $\vdash$, groups weaker than other operators. So, we can write sequents such as $(\Gamma, \alpha) \vdash (\beta \Rightarrow \gamma)$ with fewer parentheses: $\Gamma, \alpha \vdash \beta \Rightarrow \gamma$.

What code corresponds to the "create function" rule? The proof of $\Gamma \vdash \alpha \Rightarrow \beta$ depends on a proof of another sequent. So, the corresponding code must be a *function* that takes a proof of the previous sequent as an argument and returns a proof of the new sequent. By the CH correspondence, a proof of a sequent corresponds to a code expression of the type given by the goal of the sequent; the expression may use arguments of types corresponding to the premises of the sequent. So, a proof of the sequent $\Gamma, \alpha \vdash \beta$ is an expression `exprB` of type B that may use a given value of type A as well as any other arguments given previously. Then we can write the proof code for the sequent $\Gamma \vdash \alpha \Rightarrow \beta$ as the nameless function `(x:A) => exprB`. This function has type $A \to B$ and requires us to already have a suitable `exprB`. This exactly corresponds to the proof rule "create function". We may write the corresponding code as

$$\text{Proof}(\Gamma \vdash \mathcal{CH}(A) \Rightarrow \mathcal{CH}(B)) = x^{:A} \to \text{Proof}(\Gamma, x^{:A} \vdash \mathcal{CH}(B)) \quad .$$

Here we wrote $x^{:A}$ instead of $\mathcal{CH}(A)$ since the value $x^{:A}$ is a proof of the proposition $\mathcal{CH}(A)$. We will see in Section 5.2.4 how premises such as $\Gamma, x^{:A}$ are implemented in code.

4) Use a function At any place in the code, we may apply an already defined function of type $A \to B$ to an already computed value of type A. The result will be a value of type B. This corresponds to assuming $\mathcal{CH}(A \to B)$ and $\mathcal{CH}(A)$, and then deriving $\mathcal{CH}(B)$. The formal logic notation for this proof rule is

$$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta} \quad \text{(use function)} \qquad .$$

The code corresponding to this proof rule takes previously computed values `x:A` and `f:A => B`, and writes the expression `f(x)`. This can be written as a function application,

$$\text{Proof}(\Gamma \vdash \beta) = \text{Proof}\,(\Gamma \vdash \alpha \Rightarrow \beta)\,(\text{Proof}(\Gamma \vdash \alpha)) \quad .$$

5) Create a tuple If we have already computed some values `a:A` and `b:B`, we may write the expression `(a, b)` and so compute a value of the tuple type `(A, B)`. The proof rule is

$$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta} \quad \text{(create tuple)} \qquad .$$

Writing $a \times b$ to mean the pair `(a, b)`, we can write the corresponding code expression as

$$\text{Proof}\,(\Gamma \vdash \alpha \wedge \beta) = \text{Proof}\,(\Gamma \vdash \alpha) \times \text{Proof}\,(\Gamma \vdash \beta) \quad .$$

This rule describes creating a tuple of 2 values. A larger tuple, such as `(w, x, y, z)`, can be expressed via nested pairs, e.g. as `(w, (x, (y, z)))`. So, it suffices to have a sequent rule for creating pairs; this rule can express the sequent rules for creating all other tuples, and we do not need to define separate rules for, say, $\Gamma \vdash \alpha \wedge \beta \wedge \gamma$.

6) Use a tuple If we already have a value `t:(A,B)` of a tuple type $A \times B$, we can extract one of the parts of the tuple and obtain a value of type `A` or a value of type `B`. The code is `t._1` and `t._2` respectively, and the corresponding sequent proof rules are

$$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha} \quad \text{(use tuple-1)} \qquad \frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta} \quad \text{(use tuple-2)} \quad .$$

The code can be written as

$$\begin{aligned} \text{Proof}\,(\Gamma \vdash \alpha) &= \nabla_1\,(\text{Proof}\,(\Gamma \vdash \alpha \wedge \beta)) \quad , \\ \text{Proof}\,(\Gamma \vdash \beta) &= \nabla_2\,(\text{Proof}\,(\Gamma \vdash \alpha \wedge \beta)) \quad , \end{aligned}$$

where we introduced the notation ∇_1 and ∇_2 to mean the Scala code `_._1` and `_._2`.

Since all tuples can be expressed through pairs, it is sufficient to have proof rules for pairs.

7) Create a disjunctive value The type `Either[A, B]` corresponding to the disjunction $\alpha \vee \beta$ can be used to define any other disjunctive type; e.g. a disjunctive type with three parts can be expressed as `Either[A, Either[B, C]]`. So it is sufficient to have proof rules for a disjunction of *two* propositions.

There are two ways of creating a value of the type `Either[A, B]`: the code expressions are `Left(x:A)` and `Right(y:B)`. The values `x:A` or `y:B` must have been computed previously (and correspond to previously proved sequents). So, the sequent proof rules are

$$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta} \quad \text{(create Left)} \qquad \frac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta} \quad \text{(create Right)} \quad .$$

The corresponding code can be written as

$$\begin{aligned} \text{Proof}\,(\Gamma \vdash \alpha \vee \beta) &= \text{Left}(\text{Proof}\,(\Gamma \vdash \alpha)) \quad , \\ \text{Proof}\,(\Gamma \vdash \alpha \vee \beta) &= \text{Right}(\text{Proof}\,(\Gamma \vdash \beta)) \quad . \end{aligned}$$

8) Use a disjunctive value The only way we may use a value of a disjunctive type `Either[A, B]` is by pattern matching on it:

```
val result: C = (e: Either[A, B]) match {
  case Left(x:A)    => expr1(x)
  case Right(y:B)   => expr2(y)
}
```

Here, `expr1(x)` must be an expression of some type `C`, computed using `x:A` and any previously available arguments (i.e. the premises Γ). Similarly, `expr2(y)` must be an expression of type `C` computed using `y:B` and previous arguments. It is clear that `expr1(x)` represents a proof of a sequent with an additional premise of type `A`, i.e. $\Gamma, \alpha \vdash \gamma$, where we denoted $\gamma \triangleq \mathcal{CH}(C)$. Similarly, `expr2(y)` is a proof of the sequent $\Gamma, \beta \vdash \gamma$. So, we can write the proof rule corresponding to the `match`/`case` expression as a rule with three previous sequents:

$$\frac{\Gamma \vdash \alpha \vee \beta \qquad \Gamma, \alpha \vdash \gamma \qquad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma} \quad \text{(use Either)} \quad .$$

The code can be written as

$$\text{Proof}\,(\Gamma \vdash \gamma) = \text{Proof}\,(\Gamma \vdash \alpha \vee \beta) \ \text{match} \begin{cases} \text{case } a^{:A} \to & \mathbb{0}^{:B} + a^{:A} \\ \text{case } b^{:B} \to & b^{:B} + \mathbb{0}^{:A} \end{cases} \quad .$$

Table 5.4 summarizes the eight proof rules derived in this section. These proof rules define a logic known as the **intuitionistic propositional logic** or **constructive propositional logic**. We will call this logic "constructive" for short.

axioms :
$$\frac{}{\Gamma \vdash \mathcal{CH}(\mathbb{1})} \text{ (create unit)} \qquad \frac{}{\Gamma, \alpha \vdash \alpha} \text{ (use arg)}$$

derivation rules :
$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta} \text{ (create function)}$$
$$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta} \text{ (use function)}$$
$$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta} \text{ (create tuple)}$$
$$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha} \text{ (use tuple-1)} \qquad \frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta} \text{ (use tuple-2)}$$
$$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta} \text{ (create Left)} \qquad \frac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta} \text{ (create Right)}$$
$$\frac{\Gamma \vdash \alpha \vee \beta \qquad \Gamma, \alpha \vdash \gamma \qquad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma} \text{ (use Either)}$$

Table 5.4: Proof rules for the constructive logic.

5.2.4 Example: Proving a $\mathcal{CH}$-proposition and deriving code

The task is to implement a fully parametric function

```
def f[A, B]: ((A => A) => B) => B = ???
```

Implementing this function is the same as being able to compute a value of type

$$F \triangleq \forall(A, B).\, ((A \to A) \to B) \to B \quad .$$

Since the type parameters A and B are arbitrary, the body of the fully parametric function `f` cannot use any previously defined values of types A or B. So, the task is formulated as computing a value of type F with no previously defined values. This is written as the sequent $\Gamma \vdash \mathcal{CH}(F)$, where the set Γ of premises is empty, $\Gamma = \emptyset$. Rewriting this sequent using the rules of Table 5.1, we get

$$\forall(\alpha, \beta).\, \emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta \quad , \tag{5.8}$$

where we denoted $\alpha \triangleq \mathcal{CH}(A)$ and $\beta \triangleq \mathcal{CH}(B)$.

The next step is to prove the sequent (5.8) using the logic proof rules of Section 5.2.3. For brevity, we will omit the quantifier $\forall(\alpha, \beta)$ since it will be present in front of every sequent.

Begin by looking for a proof rule whose "denominator" has a sequent similar to Eq. (5.8), i.e. has an implication $(p \Rightarrow q)$ in the goal. We have only one rule that can prove a sequent of the form $\Gamma \vdash (p \Rightarrow q)$; this is the rule "create function". That rule requires us to already have a proof of the sequent $(\Gamma, p) \vdash q$. So, we use this rule with $\Gamma = \emptyset$, $p = (\alpha \Rightarrow \alpha) \Rightarrow \beta$, and $q = \beta$:

$$\frac{(\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta}{\emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta} \quad .$$

We now need to prove the sequent $(\alpha \Rightarrow \alpha) \Rightarrow \beta) \vdash \beta$, which we can write as $\Gamma_1 \vdash \beta$ where $\Gamma_1 \triangleq [(\alpha \Rightarrow \alpha) \Rightarrow \beta]$ denotes the set containing the single premise $(\alpha \Rightarrow \alpha) \Rightarrow \beta$.

There are no proof rules that derive a sequent with an explicit premise of the form of an implication $p \Rightarrow q$. However, we have a rule called "use function" that derives a sequent by assuming another sequent containing an implication. We would be able to use that rule,

$$\frac{\Gamma_1 \vdash \alpha \Rightarrow \alpha \qquad \Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta}{\Gamma_1 \vdash \beta} \quad ,$$

if we could prove the two sequents $\Gamma_1 \vdash \alpha \Rightarrow \alpha$ and $\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta$. To prove these sequents, note that the rule "create function" applies to $\Gamma_1 \vdash \alpha \Rightarrow \alpha$ as follows,

$$\frac{\Gamma_1, \alpha \vdash \alpha}{\Gamma_1 \vdash \alpha \Rightarrow \alpha} \quad .$$

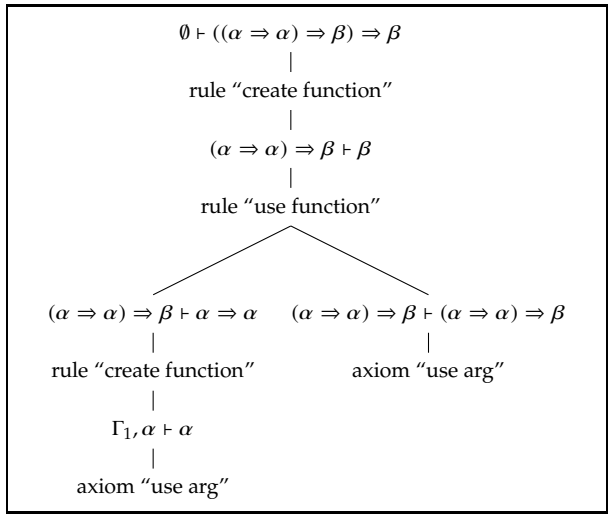


Figure 5.1: Proof tree for the sequent (5.8).

The sequent $\Gamma_1, \alpha \vdash \alpha$ is proved directly by the axiom "use arg". The sequent $\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta$ is also proved by the axiom "use arg" because Γ_1 already contains $(\alpha \Rightarrow \alpha) \Rightarrow \beta$.

The proof of the sequent (5.8) is now complete and can be visualized as a tree (Figure 5.1). The next step is to derive the code from this proof.

To do that, we combine the code expressions that correspond to each of the proof rules we used. We need to retrace the proof backwards, starting from the leaves of the tree and going towards the root, and to assemble the Proof (...) code expressions one by one.

Begin with the left-most leaf "use arg". This rule corresponds to the code $x^{:A}$,

$$\text{Proof}\,(\Gamma_1, \alpha \vdash \alpha) = x^{:A} \quad .$$

Here $x^{:A}$ must be a proof of the premise α in the sequent $\Gamma_1, \alpha \vdash \alpha$. So, we need to use the same $x^{:A}$ when we write the code for the previous rule, "create function":

$$\text{Proof}\,(\Gamma_1 \vdash \alpha \Rightarrow \alpha) = (x^{:A} \to \text{Proof}\,(\Gamma_1, \alpha \vdash \alpha)) = (x^{:A} \to x) \quad .$$

The right-most leaf "use arg" corresponds to the code $f^{:(A \to A) \to B}$, where f is the premise contained in Γ_1. So we can write

$$\text{Proof}\,(\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta) = f^{:(A \to A) \to B} \quad .$$

The previous rule, "use function", combines the two preceding proofs:

$$\begin{aligned} &\text{Proof}\,((\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta) \\ &= \text{Proof}(\Gamma_1 \vdash (\alpha \Rightarrow \alpha) \Rightarrow \beta)\ (\text{Proof}(\Gamma_1 \vdash \alpha \Rightarrow \alpha)) \\ &= f(x^{:A} \to x) \quad . \end{aligned}$$

Going further backwards, we find that the rule applied before "use function" was "create function". We need to provide the same $f^{:(A \to A) \to B}$ as in the premise above, and so we obtain the code

$$\begin{aligned} &\text{Proof}\,(\emptyset \vdash ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta) \\ &= f^{:(A \to A) \to B} \to \text{Proof}\,((\alpha \Rightarrow \alpha) \Rightarrow \beta \vdash \beta) \\ &= f^{:(A \to A) \to B} \to f(x^{:A} \to x) \quad . \end{aligned}$$

This is the final code expression that implements the type $((A \to A) \to B) \to B$. In this way, we have systematically derived the code from the type signature of a function. This function can be implemented in Scala as

```
def f[A, B]: ((A => A) => B) => B = { f => f(x => x) }
```

We found the proof tree in Figure 5.1 by guessing how to combine various proof rules. If we *somehow* find a proof tree for a sequent, we can prove the sequent and derive the corresponding code. However, it is not always obvious how to combine the proof rules to prove a given initial sequent. This is so because the rules of Table 5.7 do not provide an algorithm for finding a proof tree automatically. It turns out that such an algorithm exists (the "LJT algorithm", see Appendix C). That algorithm can find proofs and derive code from type signatures containing tuples, disjunctive types, and function types (if the given type signature can be implemented).

The library `curryhoward`[1] implements the LJT algorithm. Here are some examples of using this library. We will run the `ammonite`[2] shell to load the library more easily.

As a non-trivial (if artificial) example, consider the type signature

$$\forall(A, B).\ ((((A \to B) \to A) \to A) \to B) \to B \quad .$$

It is not immediately clear whether it is possible to implement a function with this type signature. It turns out that it *is* possible, and the code can be derived automatically by the LJT algorithm. The library does this via the method `implement`:

```
@ import $ivy.`io.chymyst::curryhoward:0.3.7`, io.chymyst.ch._

@ def f[A, B]: ((((A => B) => A) => A) => B) => B = implement
defined function f

@ println(f.lambdaTerm.prettyPrint)
a => a (b => b (c => a (d => c)))
```

The code $a \to a\,(b \to b\,(c \to a\,(d \to c)))$ was derived automatically for the function `f`. The function `f` has been compiled and is ready to be used in any subsequent code.

A compile-time error occurs when trying to use a type signature that cannot be implemented as a fully parametric function:

```
@ def g[A, B]: ((A => B) => A) => A = implement
cmd3.sc:1: type ((A => B) => A) => A cannot be implemented
def g[A, B]: ((A => B) => A) => A = implement
                                    ^
Compilation Failed
```

The logical formula corresponding to this type signature is

$$\forall(\alpha, \beta).\ ((\alpha \Rightarrow \beta) \Rightarrow \alpha) \Rightarrow \alpha \quad . \tag{5.9}$$

This formula is known as "Peirce's law".[3] It is another example showing that the logic of types in functional programming languages is not Boolean. Peirce's law is a true theorem in Boolean logic but does not hold in the constructive logic (i.e. it cannot be derived using the proof rules of Table 5.7). If we try to implement `g[A, B]` with the type signature shown above, we will fail to write fully parametric code for `g` that compiles without type errors. This is because no such code exists, — not because we are insufficiently clever. The LJT algorithm can *prove* that the given type signature cannot be implemented; the `curryhoward` library will then print an error message, and compilation will fail.

As another example, let us verify that the type signature from Section 5.2.2 cannot be implemented as a fully parametric function:

```
@ def bad[A, B, C](g: A => Either[B, C]): Either[A => B, A => C] = implement
cmd4.sc:1: type (A => Either[B, C]) => Either[A => B, A => C] cannot be implemented
def bad[A, B, C](g: A => Either[B, C]): Either[A => B, A => C] = implement
                                                                 ^
Compilation Failed
```

[1] https://github.com/Chymyst/curryhoward
[2] http://ammonite.io/#Ammonite-Shell
[3] https://en.wikipedia.org/wiki/Peirce%27s_law

5.3 Solved examples: Equivalence of types

We found a correspondence between types, code, logical propositions, and proofs, which is known as the **Curry-Howard correspondence**. An example of the CH correspondence is that a proof of the logical proposition

$$\forall(\alpha, \beta).\, \alpha \Rightarrow (\beta \Rightarrow \alpha) \tag{5.10}$$

corresponds to the code of the function

```
def f[A, B]: A => (B => A) = { x => _ => x }
```

With the CH correspondence in mind, we may say that the function `f`'s code "is" the proof of the proposition (5.10). In this sense, the *existence* of the code `x => _ => x` with the type $A \to (B \to A)$ "is" a proof of the logical formula (5.10).

The Curry-Howard correspondence maps logic formulas such as $(\alpha \vee \beta) \wedge \gamma$ into type expressions such as $(A + B) \times C$. We have seen that types behave similarly to logic formulas in one respect: A logic formula is a true theorem when the corresponding type signature can be implemented as a fully parametric function, and vice versa.

It turns out that the similarity ends here. In other respects, type expressions behave as *arithmetic* expressions and not as logic formulas. For this reason, the type notation used in this book denotes disjunctive types by $A + B$ and tuples by $A \times B$, which is designed to remind us of arithmetic expressions (such as $1 + 2$ and 2×3) rather than of logical formulas (such as $A \vee B$ and $A \wedge B$).

The most important use of the type notation is for writing equations with types. Can we write type equations using the arithmetic intuition, such as

$$(A + B) \times C = A \times C + B \times C \quad ? \tag{5.11}$$

In this section, we will learn how to check whether one type expression is equivalent to another.

5.3.1 Logical identity does not correspond to type equivalence

The CH correspondence maps Eq. (5.11) into the logic formula

$$\forall(A, B, C).\, (A \vee B) \wedge C = (A \wedge C) \vee (B \wedge C) \quad . \tag{5.12}$$

This formula is the well-known "distributive law"[4] valid in Boolean logic as well as in the constructive logic. Since a logical equation $P = Q$ means $P \Rightarrow Q$ and $Q \Rightarrow P$, the distributive law (5.12) means that the two formulas hold,

$$\forall(A, B, C).\, (A \vee B) \wedge C \Rightarrow (A \wedge C) \vee (B \wedge C) \quad , \tag{5.13}$$

$$\forall(A, B, C).\, (A \wedge C) \vee (B \wedge C) \Rightarrow (A \vee B) \wedge C \quad . \tag{5.14}$$

The CH correspondence maps these logical formulas to fully parametric functions with types

```
def f1[A, B, C]: ((Either[A, B], C))    => Either[(A, C), (B, C)] = ???
def f2[A, B, C]: Either[(A, C), (B, C)] => (Either[A, B], C)      = ???
```

In the type notation, these type signatures are written as

$$f_1^{A,B,C} : (A + B) \times C \to A \times C + B \times C \quad ,$$

$$f_2^{A,B,C} : A \times C + B \times C \to (A + B) \times C \quad .$$

Since the two logical formulas (5.13)–(5.14) are true theorems in constructive logic, we expect to be able to implement the functions `f1` and `f2`. It is not straightforward to guess how to combine the proof rules of Table 5.7 to obtain proofs of Eqs. (5.13)–(5.14). So, instead of deriving the implementations of `f1` and `f2` from the CH correspondence, we will write the Scala code directly.

To implement `f1`, we need to do pattern matching on the argument:

[4] https://en.wikipedia.org/wiki/Distributive_property#Rule_of_replacement

```
def f1[A, B, C]: ((Either[A, B], C)) => Either[(A, C), (B, C)] = {
  case (Left(a), c)   => Left((a, c))  // No other choice here.
  case (Right(b), c)  => Right((b, c)) // No other choice here.
}
```

In both cases, we have only one possible expression of the correct type.

Similarly, the implementation of `f2` leaves us no choices:

```
def f2[A, B, C]: Either[(A, C), (B, C)] => (Either[A, B], C) = {
  case Left((a, c))   => (Left(a), c)  // No other choice here.
  case Right((b, c))  => (Right(b), c) // No other choice here.
}
```

The code of `f1` and `f2` never discards any given values; in other words, these functions appear to preserve information. We can formulate this property rigorously as a requirement that an arbitrary value `x: (Either[A, B], C)` is mapped by `f1` to some value `y: Either[(A, C), (B, C)]` and then mapped by `f2` back to *the same* value `x`. Similarly, any value `y` of type `Either[(A, C), (B, C)]` should be transformed by `f2` and then by `f1` back to the same value `y`.

Let us write these conditions as equations,

$$\forall x^{:(A+B)\times C}.\, f_2(f_1(x)) = x \quad , \qquad \forall y^{:A\times C+B\times C}.\, f_1\,(f_2(y)) = y \quad .$$

If we show that these equations hold, it will follow that all the information in a value $x^{:(A+B)\times C}$ is completely preserved inside the value $y \triangleq f_1(x)$; the original value x can be recovered as $x = f_2(y)$. Conversely, all the information in a value $y^{:A\times C+B\times C}$ is preserved inside $x \triangleq f_2(y)$ and can be recovered by applying f_1. Since the values $x^{:(A+B)\times C}$ and $y^{:A\times C+B\times C}$ are arbitrary, it will follow that the *data types* themselves, $(A+B)\times C$ and $A\times C + B\times C$, carry equivalent information. Such types are called equivalent or isomorphic.

Generally, we say that types P and Q are **equivalent** or **isomorphic** (denoted $P \cong Q$) when there exist functions $f_1^{:P\to Q}$ and $f_2^{:Q\to P}$ that are inverses of each other. We can write these conditions using the notation $(f_1 \fatsemi f_2)(x) \triangleq f_2(f_1(x))$ as

$$f_1 \fatsemi f_2 = \mathrm{id} \quad , \qquad f_2 \fatsemi f_1 = \mathrm{id} \quad .$$

(In Scala, the forward composition $f_1 \fatsemi f_2$ is the function `f1 andThen f2`. We omit type annotations since we already checked that the types match.) If these conditions hold, there is a one-to-one correspondence between values of types P and Q. This is the same as to say that the data types P and Q "carry equivalent information".

To verify that the Scala functions `f1` and `f2` defined above are inverses of each other, we first check if $f_1 \fatsemi f_2 = \mathrm{id}$. Applying $f_1 \fatsemi f_2$ means to apply f_1 and then to apply f_2 to the result. Begin by applying f_1 to an arbitrary value $x^{:(A+B)\times C}$. A value x of that type can be in only one of the two disjoint cases: a tuple `(Left(a), c)` or a tuple `(Right(b), c)`, for some values `a:A`, `b:B`, and `c:C`. The Scala code of `f1` maps these tuples to `Left((a, c))` and to `Right((b, c))` respectively; we can see this directly from the code of `f1`. We then apply f_2 to those values, which maps them back to a tuple `(Left(a), c)` or a tuple `(Right(b), c)` respectively, according to the code of `f2`. These tuples are exactly the value x we started with. So, applying $f_1 \fatsemi f_2$ to an arbitrary $x^{:(A+B)\times C}$ does not change the value x; this is the same as to say that $f_1 \fatsemi f_2 = \mathrm{id}$.

To check whether $f_2 \fatsemi f_1 = \mathrm{id}$, we apply f_2 to an arbitrary value $y^{:A\times C+B\times C}$, which must be one of the two disjoint cases, `Left((a, c))` or `Right((b, c))`. The code of `f2` maps these two cases into tuples `(Left(a), c)` and `(Right(b), c)` respectively. Then we apply `f1` and map these tuples back to `Left((a, c))` and `Right((b, c))` respectively. It follows that applying f_2 and then f_1 will always recover the initial value y. In other words, $f_2 \fatsemi f_1 = \mathrm{id}$.

By looking at the code of `f1` and `f2`, we can directly observe that these functions are inverses of each other: the tuple pattern `(Left(a), c)` is mapped to `Left((a, c))`, and the pattern `(Right(b), c)` to `Right((b, c))`, or vice versa. It is visually clear that no information is lost and that the original values are restored by function compositions $f_1 \fatsemi f_2$ or $f_2 \fatsemi f_1$.

We find that the logical identity (5.12) leads to an equivalence of the corresponding types,

$$(A + B) \times C \cong A \times C + B \times C \quad . \tag{5.15}$$

To get Eq. (5.15) from Eq. (5.12), we need convert a logical formula to an arithmetic expression by mentally replacing the disjunction operations $\vee$ by $+$ and the conjunctions $\wedge$ by $\times$ everywhere.

Consider another example of a logical identity: the associativity law for conjunction,

$$(\alpha \wedge \beta) \wedge \gamma = \alpha \wedge (\beta \wedge \gamma) \quad . \tag{5.16}$$

The corresponding types are $(A \times B) \times C$ and $A \times (B \times C)$; in Scala, `((A, B), C)` and `(A, (B, C))`. We can define functions that convert between these types without information loss:

```
def f3[A, B, C]: (((A, B), C)) => (A, (B, C)) = { case ((a, b), c) => (a, (b, c)) }
def f4[A, B, C]: (A, (B, C)) => (((A, B), C)) = { case (a, (b, c)) => ((a, b), c) }
```

By applying these functions to arbitrary values of types `((A, B), C)` and `(A, (B, C))`, it is easy to see that the functions `f3` and `f4` are inverses of each other. This is also directly visible in the code: the nested tuple pattern `((a, b), c)` is mapped to the pattern `(a, (b, c))` and back. So, the types $(A \times B) \times C$ and $A \times (B \times C)$ are equivalent, and we can write $A \times B \times C$ without parentheses.

Does a logical identity always correspond to an equivalence of types? This turns out to be *not* so. A simple example of a logical identity that does not correspond to a type equivalence is

$$True \vee \alpha = True \quad . \tag{5.17}$$

Since the CH correspondence maps the logical constant $True$ into the unit type $\mathbb{1}$, the type equivalence corresponding to Eq. (5.17) is $\mathbb{1} + A \cong \mathbb{1}$. The type denoted by $\mathbb{1} + A$ means `Option[A]` in Scala, so the corresponding equivalence is `Option[A]`$\cong$`Unit`. Intuitively, this type equivalence should not hold: an `Option[A]` may carry a value of type `A`, which cannot possibly be stored in a value of type `Unit`. We can verify this intuition rigorously by proving that any fully parametric functions with type signatures $g_1 : \mathbb{1} + A \to \mathbb{1}$ and $g_2 : \mathbb{1} \to \mathbb{1} + A$ will not satisfy $g_1 \fatsemi g_2 = \text{id}$. To verify this, we note that $g_2 : \mathbb{1} \to \mathbb{1} + A$ must have type signature

```
def g2[A]: Unit => Option[A] = ???
```

Such a function must always return `None`, since a fully parametric function cannot produce values of an arbitrary type `A` from scratch. Therefore, $g_1 \fatsemi g_2$ is also a function that always returns `None`. The function $g_1 \fatsemi g_2$ has type signature $\mathbb{1} + A \to \mathbb{1} + A$ or, in Scala syntax, `Option[A] => Option[A]`, and is not equal to the identity function, because the identity function does not *always* return `None`.

Another example of a logical identity without a type equivalence is the distributive law

$$\forall(A, B, C).\ (A \wedge B) \vee C = (A \vee C) \wedge (B \vee C) \quad , \tag{5.18}$$

which is "dual" to the law (5.12), i.e. it is obtained from Eq. (5.12) by swapping all conjunctions ($\wedge$) with disjunctions ($\vee$). In logic, a dual formula to an identity is often also an identity. The CH correspondence maps Eq. (5.18) into the type equation

$$\forall(A, B, C).\ (A \times B) + C = (A + C) \times (B + C) \quad . \tag{5.19}$$

However, the types $A \times B + C$ and $(A + C) \times (B + C)$ are *not* equivalent. To see why, look at the possible code of the function $g_3 : (A + C) \times (B + C) \to A \times B + C$:

```
def g3[A,B,C]: ((Either[A, C], Either[B, C])) => Either[(A, B), C] = {
  case (Left(a), Left(b))      => Left((a, b)) // No other choice.
  case (Left(a), Right(c))     => Right(c)     // No other choice.
  case (Right(c), Left(b))     => Right(c)     // No other choice.
  case (Right(c1), Right(c2))  => Right(c1)    // Must discard c1 or c2 here!
}   // May return Right(c2) instead of Right(c1) in the last line.
```

In line 5, we have a choice of returning `Right(c1)` or `Right(c2)`. Whichever we choose, we will lose information because we will have discarded one of the given values `c1`, `c2`. After evaluating g_3, we will not be able to restore *both* `c1` and `c2` – no matter what code we write for g_4. So, the composition $g_3 \fatsemi g_4$ cannot be equal to the identity function. The type equation (5.19) is incorrect.

We conclude that a logical identity $\mathcal{CH}(P) = \mathcal{CH}(Q)$ guarantees, via the CH correspondence, that we can implement *some* fully parametric functions of types $P \to Q$ and $Q \to P$. However, it is not guaranteed that these functions are inverses of each other, i.e. that the type conversions $P \to Q$ or $Q \to P$ have no information loss. So, the type equivalence $P \cong Q$ does not automatically follow from the logical identity $\mathcal{CH}(P) = \mathcal{CH}(Q)$.

The CH correspondence means that we can compute *some* value $x^{:X}$ of a given type X when the proposition $\mathcal{CH}(X)$ holds. However, the CH correspondence does not guarantee that the computed value $x^{:X}$ will satisfy any additional properties or laws.

5.3.2 Arithmetic identity corresponds to type equivalence

Looking at the examples of equivalent types, we notice that correct type equivalences correspond to *arithmetical* identities rather than *logical* identities. For instance, the logical identity in Eq. (5.12) leads to the type equivalence (5.15), which looks like a standard identity of arithmetic, such as

$$(1 + 10) \times 20 = 1 \times 20 + 10 \times 20 \quad .$$

The logical identity in Eq. (5.18), which does *not* yield a type equivalence, leads to an incorrect arithmetic equation 5.19, e.g. $(1 \times 10) + 20 \neq (1 + 20) \times (10 + 20)$. Similarly, the associativity law (5.16) leads to a type equivalence and to the arithmetic identity

$$(a \times b) \times c = a \times (b \times c) \quad ,$$

while the logical identity in Eq. (5.17), which does not yield a type equivalence, leads to an incorrect arithmetic statement $(\forall a.\, 1 + a = 1)$.

Table 5.5 summarizes these and other examples of logical identities, with the corresponding type equivalences. In all rows, quantifiers such as $\forall \alpha$ or $\forall (A, B)$ are implied when necessary.

Because we chose the type notation to be similar to the ordinary arithmetic notation, it is easy to translate a possible type equivalence into an arithmetic equation. In all cases, valid arithmetic identities correspond to type equivalences, and failures to obtain a type equivalence correspond to incorrect arithmetic identities. With regard to type equivalence, types such as $A + B$ and $A \times B$ behave similarly to arithmetic expressions such as $10 + 20$ and 10×20 and not similarly to logical formulas such as $\alpha \vee \beta$ and $\alpha \wedge \beta$.

We already verified the first line and the last three lines of Table 5.5. Other identities are verified in a similar way. Let us begin with lines 3 and 4 of Table 5.5, which involve the proposition $False$ and the corresponding void type $\mathbb{0}$ (Scala's `Nothing`). Reasoning about the void type needs a special technique that we will now develop while verifying the type isomorphisms $\mathbb{0} \times A \cong \mathbb{0}$ and $\mathbb{0} + A \cong A$.

Example 5.3.2.1 Verify the type equivalence $\mathbb{0} \times A \cong \mathbb{0}$.

Solution Recall that the type notation $\mathbb{0} \times A$ represents the Scala tuple type `(Nothing, A)`. To demonstrate that the type `(Nothing, A)` is equivalent to the type `Nothing`, we need to show that the type `(Nothing, A)` has *no* values. Indeed, how could we create a value of type, say, `(Nothing, Int)`? We would need to fill *both* parts of the tuple. We have values of type `Int`, but we can never get a value of type `Nothing`. So, regardless of the type `A`, it is impossible to create any values of type `(Nothing, A)`. In other words, the set of values of the type `(Nothing, A)` is empty; but that is the definition of the void type `Nothing`. The types `(Nothing, A)` (denoted by $\mathbb{0} \times A$) and `Nothing` (denoted by $\mathbb{0}$) are both void and therefore equivalent.

Example 5.3.2.2 Verify the type equivalence $\mathbb{0} + A \cong A$.

Logical identity	Type equivalence (if it holds)
$True \vee \alpha = True$	$\mathbb{1} + A \not\cong \mathbb{1}$
$True \wedge \alpha = \alpha$	$\mathbb{1} \times A \cong A$
$False \vee \alpha = \alpha$	$\mathbb{0} + A \cong A$
$False \wedge \alpha = False$	$\mathbb{0} \times A \cong \mathbb{0}$
$\alpha \vee \beta = \beta \vee \alpha$	$A + B \cong B + A$
$\alpha \wedge \beta = \beta \wedge \alpha$	$A \times B \cong B \times A$
$(\alpha \vee \beta) \vee \gamma = \alpha \vee (\beta \vee \gamma)$	$(A + B) + C \cong A + (B + C)$
$(\alpha \wedge \beta) \wedge \gamma = \alpha \wedge (\beta \wedge \gamma)$	$(A \times B) \times C \cong A \times (B \times C)$
$(\alpha \vee \beta) \wedge \gamma = (\alpha \wedge \gamma) \vee (\beta \wedge \gamma)$	$(A + B) \times C \cong A \times C + B \times C$
$(\alpha \wedge \beta) \vee \gamma = (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$	$(A \times B) + C \not\cong (A + C) \times (B + C)$

Table 5.5: Logic identities with disjunction and conjunction, and the corresponding equivalences of types.

Solution Recall that the type notation $\mathbb{0} + A$ represents the Scala type `Either[Nothing, A]`. We need to show that any value of that type can be mapped without loss of information to a value of type `A`, and vice versa. This means implementing functions $f_1 : \mathbb{0} + A \to A$ and $f_2 : A \to \mathbb{0} + A$ such that $f_1 \fatsemi f_2 = \text{id}$ and $f_2 \fatsemi f_1 = \text{id}$.

The argument of f_1 is of type `Either[Nothing, A]`. How can we create a value of that type? Our only choices are to create a `Left(x)` with `x:Nothing`, or to create a `Right(y)` with `y:A`. However, we cannot create a value `x` of type `Nothing` because the type `Nothing` has *no* values; so we cannot create a `Left(x)`. The only remaining possibility is to create a `Right(y)` with some value `y` of type `A`. So, any values of type $\mathbb{0} + A$ must be of the form `Right(y)`, and we can extract that `y` to obtain a value of type `A`:

```
def f1[A]: Either[Nothing, A] => A = {
  case Right(y) => y
  // No need for ‘case Left(x) => ...‘ since no ‘x‘ can ever be given in ‘Left(x)‘.
}
```

For the same reason, there is only one implementation of the function `f2`,

```
def f2[A]: A => Either[Nothing, A] = { y => Right(y) }
```

It is clear from the code that the functions `f1` and `f2` are inverses of each other.

We have just seen that a value of type $\mathbb{0} + A$ is always a `Right(y)` with some `y:A`. Similarly, a value of type $A + \mathbb{0}$ is always a `Left(x)` with some `x:A`. So, we will use the notation $A + \mathbb{0}$ and $\mathbb{0} + A$ to *denote* the `Left` and the `Right` parts of the disjunctive type `Either`. This notation agrees with the behavior of the Scala compiler, which will infer the types `Either[A, Nothing]` or `Either[Nothing, A]` for these parts:

```
def toLeft[A, B]: A => Either[A, B] = x => Left(x)
def toRight[A, B]: B => Either[A, B] = y => Right(y)

scala> toLeft(123)
res0: Either[Int, Nothing] = Left(123)

scala> toRight("abc")
res1: Either[Nothing, String] = Right("abc")
```

We can write the functions `toLeft` and `toRight` in a code notation as

$$\text{toLeft}^{A,B} = x^{:A} \to x^{:A} + \mathbb{0}^{:B} \quad ,$$
$$\text{toRight}^{A,B} = y^{:B} \to \mathbb{0}^{:A} + y^{:B} \quad .$$

In this notation, a value of the disjunctive type is shown without using Scala class names such as `Either`, `Right`, and `Left`. This shortens the writing and speeds up code reasoning.

The type annotation $\mathbb{0}^{:A}$ is helpful to remind ourselves about the type parameter A used e.g. by the disjunctive value $\mathbb{0}^{:A} + y^{:B}$ in the body of `toRight[A, B]`. Without this type annotation, $\mathbb{0} + y^{:B}$ means

a value of type `Either[A, B]` where the parameter A is left unspecified and should be determined by matching the types of other expressions.

In the notation $\mathbb{0} + y^{:B}$, we use the symbol $\mathbb{0}$ rather than an ordinary zero (0), to avoid suggesting that 0 is a value of type $\mathbb{0}$. The void type $\mathbb{0}$ has no values, unlike the `Unit` type, $\mathbb{1}$, which has a value denoted by 1 in the code notation.

Example 5.3.2.3 Verify the type equivalence $A \times \mathbb{1} \cong A$.

Solution The corresponding Scala types are the tuple `(A, Unit)` and the type `A`. We need to implement functions $f_1 : \forall A.\, A \times \mathbb{1} \to A$ and $f_2 : \forall A.\, A \to A \times \mathbb{1}$ and to demonstrate that they are inverses of each other. The Scala code for these functions is

```
def f1[A]: ((A, Unit)) => A = { case (a, ()) => a }
def f2[A]: A => (A, Unit) = { a => (a, ()) }
```

Let us first write a proof by reasoning directly with Scala code:

```
(f1 andThen f2)((a,())) == f2(f1((a,()))) == f2(a) == (a, ())
(f2 andThen f1)(a) == f1(f2(a)) == f1((a, ())) = a
```

Now let us write a proof in the code notation. The codes of f_1 and f_2 are

$$f_1 = a^{:A} \times 1 \to a \quad ,$$
$$f_2 = a^{:A} \to a \times 1 \quad ,$$

where we denoted by 1 the value `()` of the `Unit` type. We find

$$(f_1 \fatsemi f_2)(a^{:A} \times 1) = f_2\,(f_1(a \times 1)) = f_2\,(a) = a \times 1 \quad ,$$
$$(f_2 \fatsemi f_1)(a^{:A}) = f_1(f_2(a)) = f_1(a \times 1) = a \quad .$$

This shows that both compositions are identity functions. Another way of writing the proof is by computing the function compositions symbolically, without applying to a value $a^{:A}$,

$$f_1 \fatsemi f_2 = (a \times 1 \to a) \fatsemi (a \to a \times 1) = (a \times 1 \to a \times 1) = \mathrm{id}^{A \times \mathbb{1}} \quad ,$$
$$f_2 \fatsemi f_1 = (a \to a \times 1) \fatsemi (a \times 1 \to a) = (a \to a) = \mathrm{id}^{A} \quad .$$

Example 5.3.2.4 Verify the type equivalence $A + B \cong B + A$.

Solution The corresponding Scala types are `Either[A, B]` and `Either[B, A]`. We use pattern matching to implement the functions required for the type equivalence:

```
def f1[A, B]: Either[A, B] => Either[B, A] = {
  case Left(a)    => Right(a) // No other choice here.
  case Right(b)   => Left(b)  // No other choice here.
}
def f2[A, B]: Either[B, A] => Either[A, B] = f1[B, A]
```

The functions `f1` and `f2` are implemented by code that can be derived unambiguously from the type signatures. For instance, the line `case Left(a) => ...` is required to return a value of type `Either[B, A]` by using only a given value `a:A`. The only way of doing that is by returning `Right(a)`.

It is clear from the code that the functions `f1` and `f2` are inverses of each other. To verify that rigorously, we need show that `f1 andThen f2` is equal to an identity function. The function `f1 andThen f2` applies `f2` to the result of `f1`. The code of `f1` contains two `case ...` lines, each returning a result. So, we need to apply `f2` separately in each line. Evaluate the code symbolically:

```
(f1 andThen f2) == {
  case Left(a)    => f2(Right(a))
  case Right(b)   => f2(Left(b))
} == {
  case Left(a)    => Left(a)
  case Right(b)   => Right(b)
}
```

The result is a function of type `Either[A, B] => Either[A, B]` that does not change its argument; so it is equal to the identity function.

Let us now write the function `f1` in the code notation and perform the same derivation. We will also develop a useful notation for functions operating on disjunctive types.

The pattern matching construction in the Scala code of `f1` contains a pair of functions with types `A => Either[B, A]` and `B => Either[B, A]`. One of these functions is chosen depending on whether the argument of `f1` has type $A + \mathbb{0}$ or $\mathbb{0} + B$. So, we may write the code of `f1` as

$$f_1 \triangleq x^{:A+B} \to \begin{cases} \text{if } x = a^{:A} + \mathbb{0}^{:B} & : \quad \mathbb{0}^{:B} + a^{:A} \\ \text{if } x = \mathbb{0}^{:A} + b^{:B} & : \quad b^{:B} + \mathbb{0}^{:A} \end{cases}$$

Since both the argument and the result of f_1 are disjunctive types with 2 parts each, it is convenient to write the code of f_1 as a 2×2 matrix that maps the input parts to the output parts:

```
def f1[A, B]: Either[A, B] => Either[B, A] = {
  case Left(a)    => Right(a)
  case Right(b)   => Left(b)
}
```

$$f_1 \triangleq \left| \begin{array}{c|c||c|c} & & B & A \\ \hline A & & \mathbb{0} & a^{:A} \to a \\ B & & b^{:B} \to b & \mathbb{0} \end{array} \right| .$$

The rows of the matrix correspond to the `case` rows in the Scala code; there is one row for each part of the disjunctive type of the argument. The columns of the matrix correspond to the parts of the disjunctive type of the result. The double line marks the input types of the functions.

The code of f_2 is written similarly; let us rename arguments for clarity:

```
def f2[A, B]: Either[B, A] => Either[A, B] = {
  case Left(y)    => Right(y)
  case Right(x)   => Left(x)
}
```

$$f_2 \triangleq \left| \begin{array}{c|c||c|c} & & A & B \\ \hline B & & \mathbb{0} & y^{:B} \to y \\ A & & x^{:A} \to x & \mathbb{0} \end{array} \right| .$$

The forward composition $f_1 \fatsemi f_2$ is computed by the standard rules of row-by-column matrix multiplication.[5] Any terms containing $\mathbb{0}$ are omitted, and the remaining functions are composed:

$$f_1 \fatsemi f_2 = \left| \begin{array}{c|c||c|c} & & B & A \\ \hline A & & \mathbb{0} & a^{:A} \to a \\ B & & b^{:B} \to b & \mathbb{0} \end{array} \right| \fatsemi \left| \begin{array}{c|c||c|c} & & A & B \\ \hline B & & \mathbb{0} & y^{:B} \to y \\ A & & x^{:A} \to x & \mathbb{0} \end{array} \right|$$

$$\text{matrix multiplication :} \quad = \left| \begin{array}{c|c||c|c} & & A & B \\ \hline A & & (a^{:A} \to a) \fatsemi (x^{:A} \to x) & \mathbb{0} \\ B & & \mathbb{0} & (b^{:B} \to b) \fatsemi (y^{:B} \to y) \end{array} \right|$$

$$\text{function composition :} \quad = \left| \begin{array}{c|c||c|c} & & A & B \\ \hline A & & \text{id} & \mathbb{0} \\ B & & \mathbb{0} & \text{id} \end{array} \right| = \text{id}^{:A+B \to A+B} .$$

Several features of the matrix notation are helpful in such calculations. The parts of the code of f_1 are automatically composed with the corresponding parts of the code of f_2. To check that the types match in the function composition, we just need to compare the types in the output row $\left\| \; B \quad A \; \right|$ of f_1 with the input column $\left| \begin{array}{c} B \\ A \end{array} \right\|$ of f_2. Once we verified that all types match, we may omit the type

[5] https://en.wikipedia.org/wiki/Matrix_multiplication

annotations and write the same derivation more concisely as

$$f_1 \fatsemi f_2 = \left\| \begin{array}{cc} \mathbb{0} & a^{:A} \to a \\ b^{:B} \to b & \mathbb{0} \end{array} \right| \fatsemi \left\| \begin{array}{cc} \mathbb{0} & y^{:B} \to y \\ x^{:A} \to x & \mathbb{0} \end{array} \right|$$

$$\text{matrix multiplication :} \quad = \left\| \begin{array}{cc} (a^{:A} \to a) \fatsemi (x^{:A} \to x) & \mathbb{0} \\ \mathbb{0} & (b^{:B} \to b) \fatsemi (y^{:B} \to y) \end{array} \right|$$

$$\text{function composition :} \quad = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \end{array} \right| = \text{id} \quad .$$

The identity function is represented by the diagonal matrix $\left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \end{array} \right|$, just as in standard matrix notation.

Exercise 5.3.2.5 Verify the type equivalence $A \times B \cong B \times A$.

Exercise 5.3.2.6 Verify the type equivalence $(A + B) + C \cong A + (B + C)$. Since Section 5.3.1 verified the equivalences $(A + B) + C \cong A + (B + C)$ and $(A \times B) \times C \cong A \times (B \times C)$, we may write $A + B + C$ and $A \times B \times C$ without any parentheses.

Exercise 5.3.2.7 Verify the type equivalence

$$(A + B) \times (A + B) = A \times A + \mathbb{2} \times A \times B + B \times B \quad ,$$

where $\mathbb{2}$ denotes the `Boolean` type (defined as $\mathbb{2} \triangleq \mathbb{1} + \mathbb{1}$).

5.3.3 Type cardinalities and type equivalence

To understand why type equivalences are related to arithmetic identities, consider the question of how many different values a given type can have.

Begin by counting the number of distinct values for simple types. For example, the `Unit` type has only one distinct value; the type `Nothing` has zero values; the `Boolean` type has two distinct values, `true` and `false`; and the type `Int` has 2^{32} distinct values.

It is more difficult to count the number of distinct values in a type such as `String`, which is equivalent to a list of unknown length, `List[Char]`. However, each computer's memory is limited, so there will exist a maximum length for values of type `String`, and so the total number of possible different strings will be finite (at least, for any given computer).

For a given type A, let us denote by $|A|$ the number of distinct values of type A. The number $|A|$ is called the **cardinality** of type A; this is the same as the number of elements in the set of all values of type A. Since any computer's memory is finite, and since we may assume that we are already working with the largest possible computer, then there will be *finitely* many different values of a given type A that can exist in the computer. So, we may assume that $|A|$ is always a finite integer value. This assumption will simplify our reasoning. We will not actually need to compute the precise number of, say, all the different possible strings; it is sufficient to know that the set of all strings is finite, so that we can denote its cardinality by |String|.

The next step is to consider the cardinality of types such as $A \times B$ and $A + B$. If the types A and B have cardinalities $|A|$ and $|B|$, it follows that the set of all distinct pairs `(A, B)` has $|A| \times |B|$ elements. So the cardinality of the type $A \times B$ is equal to the (arithmetic) product of the cardinalities of A and B. The set of all pairs

$$\{(a, b) : a \in A, b \in B\}$$

is also known as the **Cartesian product** of sets A and B, and is denoted by $A \times B$. For this reason, the tuple type is also called the **product type**. Accordingly, the type notation adopts the symbol $\times$ for the product type.

The set of all distinct values of the type $A + B$, i.e. of the Scala type `Either[A, B]`, is a disjoint union of the set of values of the form `Left(a)` and the set of values of the form `Right(b)`. It is clear that the cardinalities of these sets are equal to $|A|$ and $|B|$ respectively. So the cardinality of the type `Either[A, B]` is equal to $|A| + |B|$. For this reason, disjunctive types such as `Either[A, B]` are also called **sum types**, and the type notation adopts the symbol + for these types.

We can write our conclusions as

$$|A \times B| = |A| \times |B| \quad ,$$
$$|A + B| = |A| + |B| \quad .$$

The type notation, $A \times B$ for pairs and $A + B$ for `Either[A, B]`, translates directly into type cardinalities.

The last step is to notice that two types can be equivalent, $P \cong Q$, only if their cardinalities are equal, $|P| = |Q|$. When the cardinalities are not equal, $|P| \neq |Q|$, it will be impossible to have a one-to-one correspondence between the sets of values of type P and values of type Q. So it will be impossible to convert values from type P to type Q and back without loss of information.

We conclude that types are equivalent when a logical identity *and* an arithmetic identity hold.

The presence of both identities does not automatically guarantee a useful type equivalence. The fact that information in one type can be identically stored in another type does not necessarily mean that it is helpful to do so in a given application.

For example, the types `Option[Option[A]]` and `Either[Boolean, A]` are equivalent because both types contain $2 + |A|$ distinct values. The short notation for these types is $\mathbb{1} + \mathbb{1} + A$ and $\mathbb{2} + A$ respectively (the type Boolean is denoted by $\mathbb{2}$ since it has only two distinct values).

One could easily write code to convert between these types without loss of information:

```
def f1[A]: Option[Option[A]] => Either[Boolean, A] = {
  case None           => Left(false) // Or maybe Left(true)?
  case Some(None)     => Left(true)
  case Some(Some(x))  => Right(x)
}

def f2[A]: Either[Boolean, A] => Option[Option[A]] = {
  case Left(false)    => None
  case Left(true)     => Some(None)
  case Right(x)       => Some(Some(x))
}
```

A sign of trouble is the presence of an arbitrary choice in this code. In `f1`, we could map `None` to `Left(false)` or to `Left(true)`, and adjust the rest of the code accordingly; the type equivalence would still hold. So, formally speaking, these types *are* equivalent, but there is no "natural" choice of the conversion functions `f1` and `f2` that will work correctly in all applications, because the meaning of these data types is application-dependent. This type equivalence is "accidental".

Example 5.3.3.1 Are the types `Option[A]` and `Either[Unit, A]` equivalent? Check whether the corresponding logic identity and arithmetic identity hold.

Solution Begin by writing the given types in the type notation: `Option[A]` is written as $\mathbb{1} + A$, and `Either[Unit, A]` is written also as $\mathbb{1} + A$. The notation already indicates that the types are equivalent. But let us verify explicitly that the type notation is not misleading here.

To establish type equivalence, we need to implement two fully parametric functions

```
def f1[A]: Option[A] => Either[Unit, A] = ???
def f2[A]: Either[Unit, A] => Option[A] = ???
```

such that $f_1 \fatsemi f_2 = \text{id}$ and $f_2 \fatsemi f_1 = \text{id}$. It is straightforward to implement `f1` and `f2`:

```
def f1[A]: Option[A] => Either[Unit, A] = {
  case None       => Left(())
  case Some(x)    => Right(x)
}
def f2[A]: Either[Unit, A] => Option[A] = {
```

```
  case Left(())   => None
  case Right(x)   => Some(x)
}
```

The code clearly shows that `f1` and `f2` are inverses of each other; this verifies the type equivalence.

The logic identity is $True \vee A = True \vee A$ and holds trivially. It remains to check the arithmetic identity, which relates the number of distinct values of types `Option[A]` and `Either[Unit, A]`. Assume that the number of distinct values of type `A` is $|A|$. Any possible value of type `Option[A]` must be either `None` or `Some(x)`, where `x` is a value of type `A`. So the number of distinct values of type `Option[A]` is $1 + |A|$. All possible values of type `Either[Unit, A]` are of the form `Left(())` or `Right(x)`, where `x` is a value of type `A`. So the number of distinct values of type `Either[Unit, A]` is $1 + |A|$. We see that the arithmetic identity holds: the types `Option[A]` and `Either[Unit, A]` have equally many distinct values.

This example shows that the type notation is helpful for reasoning about type equivalences. The solution was found immediately when we wrote the type notation, $\mathbb{1} + A$, for the given types.

5.3.4 Type equivalence involving function types

Until now, we have looked at product types and disjunctive types. Let us now consider type constructions involving function types.

Consider two types A and B, whose cardinalities are known as $|A|$ and $|B|$. What is the cardinality of the set of all maps between given sets A and B? In other words, how many distinct values does the function type $A \to B$ have? A function `f: A => B` needs to select a value of type B for each possible value of type A. Therefore, the number of different functions `f: A => B` is $|B|^{|A|}$ (the arithmetic exponent, $|B|$ to the power $|A|$).

For the types $A = B =$ Int, we have $|A| = |B| = 2^{32}$, and so the estimate will give

$$|A \to B| = \left(2^{32}\right)^{\left(2^{32}\right)} = 2^{32 \times 2^{32}} = 2^{2^{37}} \approx 10^{4.1 \times 10^{10}} \quad .$$

In fact, most of these functions will map integers to integers in a complicated (and practically useless) way and will be impossible to implement on a realistic computer because their code will be much longer than the available memory. So, the number of practically implementable functions of type $A \to B$ is often much smaller than $|B|^{|A|}$. Nevertheless, the estimate $|B|^{|A|}$ is useful since it shows the number of distinct functions that are possible in principle.

Let us now look for logic identities and arithmetic identities involving function types. Table 5.6 lists the available identities and the corresponding type equivalences. (In the last column, we defined $a \triangleq |A|$, $b \triangleq |B|$, and $c \triangleq |C|$ for brevity.)

It is notable that no logic identity is available for the formula $\alpha \Rightarrow (\beta \vee \gamma)$, and correspondingly no type equivalence is available for the type expression $A \to B + C$ (although there is an identity for $A \to B \times C$). The presence of type expressions of the form $A \to B + C$ makes type reasoning more complicated because they cannot be transformed into equivalent formulas with simpler parts.

We will now prove some of the type identities in Table 5.6.

Example 5.3.4.1 Verify the type equivalence $\mathbb{1} \to A \cong A$.

Solution Recall that the type notation $\mathbb{1} \to A$ means the Scala function type `Unit => A`. There is only one value of type `Unit`, so the choice of a function of the type `Unit => A` is the same as the choice of a value of type `A`. Thus, the type $\mathbb{1} \to A$ has $|A|$ distinct values, so the arithmetic identity holds.

To verify the type equivalence explicitly, we need to implement two functions

```
def f1[A]: (Unit => A) => A = ???
def f2[A]: A => Unit => A = ???
```

The first function needs to produce a value of type `A`, given an argument of the function type `Unit => A`. The only possibility is to apply that function to the value of type `Unit`; we can always produce that value as `()`:

```
def f1[A]: (Unit => A) => A = (h: Unit => A) => h(())
```

Logical identity (if holds)	Type equivalence	Arithmetic identity
$(True \Rightarrow \alpha) = \alpha$	$\mathbb{1} \to A \cong A$	$a^1 = a$
$(False \Rightarrow \alpha) = True$	$\mathbb{0} \to A \cong \mathbb{1}$	$a^0 = 1$
$(\alpha \Rightarrow True) = True$	$A \to \mathbb{1} \cong \mathbb{1}$	$1^a = 1$
$(\alpha \Rightarrow False) \neq False$	$A \to \mathbb{0} \not\cong \mathbb{0}$	$0^a \neq 0$
$(\alpha \vee \beta) \Rightarrow \gamma = (\alpha \Rightarrow \gamma) \wedge (\beta \Rightarrow \gamma)$	$A + B \to C \cong (A \to C) \times (B \to C)$	$c^{a+b} = c^a \times c^b$
$(\alpha \wedge \beta) \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$	$A \times B \to C \cong A \to B \to C$	$c^{a \times b} = (c^b)^a$
$\alpha \Rightarrow (\beta \wedge \gamma) = (\alpha \Rightarrow \beta) \wedge (\alpha \Rightarrow \gamma)$	$A \to B \times C \cong (A \to B) \times (A \to C)$	$(b \times c)^a = b^a \times c^a$

Table 5.6: Logical identities with implication, and the corresponding type equivalences and arithmetic identities.

Implementing `f2` is straightforward; we can just discard the `Unit` argument:

```
def f2[A]: A => Unit => A = (x: A) => _ => x
```

It remains to show that the functions `f1` and `f2` are inverses of each other. Let us perform the proof using Scala code and then using the code notation.

Writing Scala code, compute `f1(f2(x))` for an arbitrary `x:A`. Substituting the code, we get

```
f1(f2(x)) == f1(_ => x) == (_ => x)(()) == x
```

Now compute `f2(f1(h))` for arbitrary `h: Unit => A` in Scala code:

```
f2(f1(h)) == f2(h(())) == { _ => h(()) }
```

How can we show that the function `{_ => h(())}` is equal to `h`? Whenever we apply equal functions to equal arguments, they return equal results. In our case, the argument of `h` is of type `Unit`, so we only need to verify that the result of applying `h` to the value `()` is the same as the result of applying `{_ => h(())}` to `()`. In other words, we need to apply both sides to an additional argument `()`:

```
f2(f1(h))(()) == { _ => h(()) } (()) == h(())
```

This completes the proof.

For comparison, let us show the same proof in the code notation. The functions f_1 and f_2 are

$$f_1 \triangleq h^{:\mathbb{1}\to A} \to h(1) \quad ,$$
$$f_2 \triangleq x^{:A} \to 1 \to x \quad .$$

Now write the function compositions in both directions:

$$\begin{aligned} f_1 \fatsemi f_2 &= (h^{:\mathbb{1}\to A} \to h(1)) \fatsemi (x^{:A} \to 1 \to x) \\ \text{compute composition :} \quad &= (h \to 1 \to h(1)) \\ \text{note that } 1 \to h(1) \text{ is the same as } h : \quad &= (h \to h) = \text{id} \quad . \\ f_2 \fatsemi f_1 &= (x^{:A} \to 1 \to x) \fatsemi (h^{:\mathbb{1}\to A} \to h(1)) \\ \text{compute composition :} \quad &= x \to (1 \to x)(1) \\ \text{apply function :} \quad &= (x \to x) = \text{id} \quad . \end{aligned}$$

The type $\mathbb{1} \to A$ is equivalent to the type A, but these types are not the same. The most important difference between these types is that a value of type A is available immediately, while a value of type $\mathbb{1} \to A$ is a function that still needs to be applied to an argument (of type $\mathbb{1}$) before a value

of type A is obtained. The type $\mathbb{1} \to A$ may represent an "on-call" value of type A; that is, a value computed on demand every time. (See Section 2.6.3 for more details about "on-call" values.)

The void type $\mathbb{0}$ needs special reasoning:

Example 5.3.4.2 Verify the type equivalence $\mathbb{0} \to A \cong \mathbb{1}$.

Solution What could be a function $f^{:\mathbb{0}\to A}$ from the type $\mathbb{0}$ to a type A? Since there exist no values of type $\mathbb{0}$, the function f will never be applied to any arguments and so *does not need* to compute any actual values of type A. So, f is a function whose body may be empty; or at least it does not need to contain any expressions of type A. In Scala, such a function can be written as

```
def absurd[A]: Nothing => A = { ??? }
```

This code will compile without type errors. An equivalent code is

```
def absurd[A]: Nothing => A = { x => ??? }
```

The symbol `???` is defined in the Scala library and represents code that is "not implemented". Trying to evaluate this symbol will produce an error:

```
scala> ???
scala.NotImplementedError: an implementation is missing
  scala.Predef$.$qmark$qmark$qmark(Predef.scala:288)
```

Since the function `absurd` can never be applied to an argument, this error will never happen. So, we can pretend that the result value (which will never be computed) has any required type, e.g. type A.

Let us now verify that there exists *only one* distinct function of type $\mathbb{0} \to A$. Take any two functions of that type, $f^{:\mathbb{0}\to A}$ and $g^{:\mathbb{0}\to A}$. Are they different? The only way of showing that f and g are different is by producing a value $x^{:\mathbb{0}}$ such that $f(x) \neq g(x)$. But there are *no* values of type $\mathbb{0}$, and for this reason, we will never be able to find the required value $x^{:\mathbb{0}}$. It follows that any two functions f and g of type $\mathbb{0} \to A$ are equal, $f = g$. In other words, there exists only one distinct value of type $\mathbb{0} \to A$; i.e. the cardinality of the type $\mathbb{0} \to A$ is 1. So, the type $\mathbb{0} \to A$ is equivalent to the type $\mathbb{1}$.

Example 5.3.4.3 Show that $A \to \mathbb{0} \not\cong \mathbb{0}$ and $A \to \mathbb{0} \not\cong \mathbb{1}$.

Solution To prove that two types are *not* equivalent, it is sufficient to show that their type cardinalities are different. Let us determine the cardinality of the type $A \to \mathbb{0}$, assuming that the cardinality of A is known. We note that a function of type, say, $\text{Int} \to \mathbb{0}$ is impossible to implement. (If we had such a function $f^{:\text{Int}\to\mathbb{0}}$, we could evaluate, say, $x \triangleq f(123)$ and obtain a value x of type $\mathbb{0}$, which is impossible by definition of the type $\mathbb{0}$. It follows that $|\text{Int} \to \mathbb{0}| = 0$. However, Example 5.3.4.2 shows that $\mathbb{0} \to \mathbb{0}$ has cardinality 1. So, the cardinality $|A \to \mathbb{0}| = 1$ if the type A is itself $\mathbb{0}$ but $|A \to \mathbb{0}| = 0$ for all other types A. We conclude that the type $A \to \mathbb{0}$ is not equivalent to $\mathbb{0}$ or $\mathbb{1}$ for all A. The type $A \to \mathbb{0}$ it is equivalent to $\mathbb{0}$ only for non-void types A.

Example 5.3.4.4 Verify the type equivalence $A \to \mathbb{1} \cong \mathbb{1}$.

Solution There is only one fully parametric function that returns $\mathbb{1}$:

```
def f[A]: A => Unit = { _ => () }
```

The function f cannot use its argument of type A since nothing is known about that type. So the code of f *must* discard its argument and return the fixed value `()` of type `Unit`. In the code notation, this function is written as

$$f^{:A\to\mathbb{1}} \triangleq (\_ \to 1) \quad .$$

We can show that there exist only *one* distinct function of type $A \to \mathbb{1}$ (that is, the type $A \to \mathbb{1}$ has cardinality 1). Assume that f and g are two such functions, and try to find a value $x^{:A}$ such that $f(x) \neq g(x)$. We cannot find any such x because $f(x) = 1$ and $g(x) = 1$ for all x. So, any two functions f and g of type $A \to \mathbb{1}$ must be equal to each other. The cardinality of the type $A \to \mathbb{1}$ is 1.

Any type having cardinality 1 is equivalent to the `Unit` type, $\mathbb{1}$. So $A \to \mathbb{1} \cong \mathbb{1}$.

Example 5.3.4.5 Verify the type equivalence

$$A + B \to C \cong (A \to C) \times (B \to C) \quad .$$

Solution Begin by implementing two functions with type signatures

```
def f1[A,B,C]: (Either[A, B] => C) => (A => C, B => C) = ???
def f2[A,B,C]: ((A => C, B => C)) => Either[A, B] => C = ???
```

The code can be derived unambiguously from the type signatures. For the first function, we need to produce a pair of functions of type `(A => C, B => C)`. Can we produce the first part of that pair? Computing a function of type `A => C` means that we need to produce a value of type `C` given an arbitrary value `a:A`. The available data is a function of type `Either[A, B] => C` called, say, `h`. We can apply that function to `Left(a)` and obtain a value of type `C` as required. So, a function of type `A => C` is computed as `a => h(Left(a))`. Similarly, we produce a function of type `B => C`. The code is

```
def f1[A,B,C]: (Either[A, B] => C) => (A => C, B => C) =
  (h: Either[A, B] => C) => (a => h(Left(a)), b => h(Right(b)))
```

A code notation for this function is

$$f_1 : (A + B \to C) \to (A \to C) \times (B \to C) \quad ,$$
$$f_1 \triangleq h^{:A+B\to C} \to \left(a^{:A} \to h(a + \mathbb{0}^{:B})\right) \times \left(b^{:B} \to h(\mathbb{0}^{:A} + b)\right) \quad .$$

For the function `f2`, we need to apply pattern matching to both curried arguments and then return a value of type `C`. This can be achieved in only one way:

```
def f2[A,B,C]: ((A => C, B => C)) => Either[A, B] => C = { case (f, g) =>
  {
    case Left(a)    => f(a)
    case Right(b)   => g(b)
  }
}
```

A code notation for this function can be written as

$$f_2 : (A \to C) \times (B \to C) \to A + B \to C \quad ,$$
$$f_2 \triangleq f^{:A\to C} \times g^{:B\to C} \to \left|\begin{array}{c||c} & C \\ \hline\hline A & a \to f(a) \\ B & b \to g(b) \end{array}\right| \quad .$$

The matrix in the last line has only one column because the result type, C, is not known to be a disjunctive type. We may simplify the functions, e.g. $a \to f(a)$ into f, and write

$$f_2 \triangleq f^{:A\to C} \times g^{:B\to C} \to \left|\begin{array}{c||c} & C \\ \hline\hline A & f \\ B & g \end{array}\right| \quad .$$

It remains to verify that $f_1 \fatsemi f_2 = \text{id}$ and $f_2 \fatsemi f_1 = \text{id}$. To compute the composition $f_1 \fatsemi f_2$, we write (omitting types)

$$f_1 \fatsemi f_2 = (h \to (a \to h(a + \mathbb{0})) \times (b \to h(\mathbb{0} + b))) \fatsemi \left(f \times g \to \left\| \begin{array}{c} f \\ g \end{array} \right|\right)$$
$$\text{compute composition :} \quad = h \to \left\| \begin{array}{l} a \to h(a + \mathbb{0}) \\ b \to h(\mathbb{0} + b) \end{array} \right| \quad .$$

To proceed, we need to simplify the expressions $h(a + \mathbb{0})$ and $h(\mathbb{0} + b)$. We rewrite the argument h

(an arbitrary function of type $A + B \to C$) in the matrix notation:

$$h \triangleq \begin{array}{|c||c|}\hline & C \\ \hline\hline A & a \to p(a) \\ \hline B & b \to q(b) \\ \hline\end{array} = \begin{array}{|c||c|}\hline & C \\ \hline\hline A & p \\ \hline B & q \\ \hline\end{array} \quad ,$$

where $p^{:A \to C}$ and $q^{:B \to C}$ are new arbitrary functions. Since we already checked the types, we can omit all type annotations and express h as

$$h \triangleq \begin{array}{|c||}\hline p \\ \hline\hline q \\ \hline\end{array} \quad .$$

To evaluate expressions such as $h(a + \mathbb{0})$ and $h(\mathbb{0} + b)$, we need to use one of the rows of the column matrix h. The correct row will be selected *automatically* by the rules of matrix multiplication if we place a row vector to the left of the matrix and use the convention of omitting terms containing $\mathbb{0}$:

$$\begin{array}{|cc|} a & \mathbb{0} \end{array} \triangleright \begin{array}{|c||}\hline p \\ \hline\hline q \\ \hline\end{array} = a \triangleright p \quad , \qquad \begin{array}{|cc|} \mathbb{0} & b \end{array} \triangleright \begin{array}{|c||}\hline p \\ \hline\hline q \\ \hline\end{array} = b \triangleright q \quad .$$

Here we used the symbol $\triangleright$ to separate an argument from a function when the argument is written to the *left* of the function. The symbol $\triangleright$ (pronounced "pipe") is defined by $x \triangleright f \triangleq f(x)$. In Scala, this operation is available as `x.pipe(f)` as of Scala 2.13.

We can write values of disjunctive types, such as $a + \mathbb{0}$, as row vectors $\begin{array}{|cc|} a & \mathbb{0} \end{array}$:

$$h(a + \mathbb{0}) = (a + \mathbb{0}) \triangleright h = \begin{array}{|cc|} a & \mathbb{0} \end{array} \triangleright h \quad . \tag{5.20}$$

With these notations, we can compute further, by omitting terms containing $\mathbb{0}$:

$$h(a + \mathbb{0}) = \begin{array}{|cc|} a & \mathbb{0} \end{array} \triangleright \begin{array}{|c||}\hline p \\ \hline\hline q \\ \hline\end{array} = a \triangleright p = p(a) \quad ,$$

$$h(\mathbb{0} + b) = \begin{array}{|cc|} \mathbb{0} & b \end{array} \triangleright \begin{array}{|c||}\hline p \\ \hline\hline q \\ \hline\end{array} = b \triangleright q = q(b) \quad .$$

Now we can complete the proof of $f_1 \fatsemi f_2 = \text{id}$:

$$f_1 \fatsemi f_2 = h \to \begin{array}{|c||}\hline a \to h(a + \mathbb{0}) \\ \hline\hline b \to h(\mathbb{0} + b) \\ \hline\end{array}$$

previous equations :

$$= \begin{array}{|c||}\hline p \\ \hline\hline q \\ \hline\end{array} \to \begin{array}{|c||}\hline a \to p(a) \\ \hline\hline b \to q(b) \\ \hline\end{array}$$

simplify functions :

$$= \left(\begin{array}{|c||}\hline p \\ \hline\hline q \\ \hline\end{array} \to \begin{array}{|c||}\hline p \\ \hline\hline q \\ \hline\end{array} \right) = \text{id} \quad .$$

To prove that $f_2 \fatsemi f_1 = \text{id}$, use the notation (5.20):

$$f_2 \fatsemi f_1 = \left(f \times g \to \left\| \begin{array}{c} f \\ g \end{array} \right| \right) \fatsemi (h \to (a \to h(a + \mathbb{0})) \times (b \to h(\mathbb{0} + b)))$$

$$\text{compute composition :} \quad = f \times g \to (a \to \left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| \triangleright \left\| \begin{array}{c} f \\ g \end{array} \right|) \times (b \to \left| \begin{array}{cc} \mathbb{0} & b \end{array} \right| \triangleright \left\| \begin{array}{c} f \\ g \end{array} \right|)$$

$$\text{matrix notation :} \quad = f \times g \to (a \to \underline{a \triangleright f}) \times (b \to \underline{b \triangleright g})$$

$$\text{definition of } \triangleright : \quad = f \times g \to \underline{(a \to f(a))} \times \underline{(b \to g(b))}$$

$$\text{simplify functions :} \quad = (f \times g \to f \times g) = \text{id} \quad .$$

In this way, we have proved that f_1 and f_2 are mutual inverses. The proofs appear long because we took time to motivate and introduce new notation for applying matrices to row vectors. Given this notation, the proof for $f_1 \fatsemi f_2 = \text{id}$ can be written as

$$f_1 \fatsemi f_2 = (h \to (a \to (a + \mathbb{0}) \triangleright h) \times (b \to (\mathbb{0} + b) \triangleright h)) \fatsemi \left(f \times g \to \left\| \begin{array}{c} f \\ g \end{array} \right| \right)$$

$$\text{compute composition :} \quad = h \to \left\| \begin{array}{cc} a \to \left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| \triangleright h \\ b \to \left| \begin{array}{cc} \mathbb{0} & b \end{array} \right| \triangleright h \end{array} \right| = \left\| \begin{array}{c} p \\ q \end{array} \right| \to \left\| \begin{array}{c} a \to \left| \begin{array}{cc} a & \mathbb{0} \end{array} \right| \triangleright \left\| \begin{array}{c} p \\ q \end{array} \right| \\ b \to \left| \begin{array}{cc} \mathbb{0} & b \end{array} \right| \triangleright \left\| \begin{array}{c} p \\ q \end{array} \right| \end{array} \right|$$

$$\text{matrix notation :} \quad = \left\| \begin{array}{c} p \\ q \end{array} \right| \to \left\| \begin{array}{c} a \to a \triangleright p \\ b \to b \triangleright q \end{array} \right| = \left(\left\| \begin{array}{c} p \\ q \end{array} \right| \to \left\| \begin{array}{c} p \\ q \end{array} \right| \right) = \text{id} \quad .$$

The code notation makes proofs shorter than they would be if we were manipulating code in Scala syntax. From now on, we will prefer to use the code notation in the proofs, keeping in mind that we can always unambiguously convert the code notation to Scala code and back.

Example 5.3.4.6 Verify the type equivalence

$$A \times B \to C \cong A \to B \to C \quad .$$

Solution Begin by implementing the two functions

```
def f1[A,B,C]: (((A, B)) => C) => A => B => C = ???
def f2[A,B,C]: (A => B => C) => ((A, B)) => C = ???
```

The Scala code can be derived from the type signatures unambiguously:

```
def f1[A,B,C]: (((A, B)) => C) => A => B => C = g => a => b => g((a, b))
def f2[A,B,C]: (A => B => C) => ((A, B)) => C = h => { case (a, b) => h(a)(b) }
```

Write these functions in the code notation:

$$f_1 = g^{:A \times B \to C} \to a^{:A} \to b^{:B} \to g(a \times b) \quad ,$$
$$f_2 = h^{:A \to B \to C} \to (a \times b)^{:A \times B} \to h(a)(b) \quad .$$

We denote by $(a \times b)^{:A \times B}$ the argument of type `(A, B)` with pattern matching implied. This notation allows us to write shorter code notation involving tupled arguments.

Compute the function composition $f_1 \fatsemi f_2$:

$$\begin{aligned} f_1 \fatsemi f_2 &= (g \to \underline{a \to b \to g(a \times b)}) \fatsemi (h \to a \times b \to h(a)(b)) \\ \text{substitute } h = a \to b \to g(a \times b): \quad &= g \to \underline{a \times b \to g(a \times b)} \\ \text{simplify function}: \quad &= (g \to g) = \text{id} \quad . \end{aligned}$$

Compute the function composition $f_2 \fatsemi f_1$:

$$\begin{aligned} f_2 \fatsemi f_1 &= (h \to \underline{a \times b \to h(a)(b)}) \fatsemi (g \to a \to b \to g(a \times b)) \\ \text{substitute } g = a \times b \to h(a)(b): \quad &= h \to a \to \underline{b \to h(a)(b)} \\ \text{simplify function } b \to h(a)(b): \quad &= h \to \underline{a \to h(a)} \\ \text{simplify function } a \to h(a) \text{ to } h: \quad &= (h \to h) = \text{id} \quad . \end{aligned}$$

Exercise 5.3.4.7 Verify the type equivalence $(A \to B \times C) \cong (A \to B) \times (A \to C)$.

5.4 Summary

What tasks can we perform now?

- Use the type notation (Table 5.1) for reasoning about types to:
  - Decide type equivalence using the rules in Tables 5.5–5.6.
  - Simplify type expressions before writing code.
- Use the matrix notation and the pipe notation to write code that works on disjunctive types.
- Convert a fully parametric type signature into a logical formula to:
  - Decide whether the type signature can be implemented in code.
  - If possible, derive the code using the CH correspondence.

What tasks *cannot* be performed with these tools?

- Automatically generate code for a *recursive* function. (The CH correspondence is based on propositional logic, which cannot describe recursion.)
- Automatically generate code satisfying a property (e.g. isomorphism). We may generate the code, but it is not guaranteed that properties will hold. The workaround is to verify the required properties manually, after deriving the code.
- Express complicated conditions (e.g. "array is sorted") in a type signature. This can be done using **dependent types** (i.e. types that depend on run-time values in an arbitrary way) — an advanced technique for which Scala has limited support. (Programming languages such as Coq, Agda, and Idris support full dependent types.)
- Generate code using type constructors with known properties (e.g. the `.map` method).

As an example of using type constructors with properties, consider this type signature:

```
def q[A]: Array[A] => (A => Option[B]) => Array[Option[B]]
```

Can we generate the code of this function from its type signature? We know that the Scala library defines a `.map` method on the `Array` type constructor, so the implementation of `q` is simple,

```
def q[A]: Array[A] => (A => Option[B]) => Array[Option[B]] = { arr => f => arr.map(f) }
```

However, it is hard to create an *algorithm* that can derive this implementation automatically from the type signature of `q` via the Curry-Howard correspondence. The algorithm would have to convert the type signature of `q` into the logical formula

$$\mathcal{CH}(\text{Array}^A) \Rightarrow \mathcal{CH}(A \to \text{Opt}^B) \Rightarrow \mathcal{CH}(\text{Array}^{\text{Opt}^B}) \quad . \tag{5.21}$$

To derive an implementation, the algorithm would need to use the available `.map` method for `Array`. That method has a type signature such as

$$\text{map} : \forall(A, B).\, \text{Array}^A \to (A \to B) \to \text{Array}^B \quad .$$

To derive the $\mathcal{CH}$-proposition (5.21), the algorithm will need to assume that the $\mathcal{CH}$-proposition

$$\mathcal{CH}\left(\forall(A, B).\, \text{Array}^A \to (A \to B) \to \text{Array}^B\right) \tag{5.22}$$

already holds, i.e. that Eq. (5.22) is one of the premises of a sequent to be proved. Reasoning about propositions such as Eq. (5.22) requires **first-order logic** — a logic whose proof rules can handle quantified types such as $\forall(A, B)$ *inside* premises. However, first-order logic is **undecidable**: no algorithm can guarantee finding a proof or showing the absence of a proof in all cases.

The constructive propositional logic (with the rules listed in Section 5.2.3) is **decidable**, i.e. it has an algorithm that either finds a proof or disproves any given formula. However, that logic cannot handle premises containing type quantifiers such as $\forall(A, B)$ inside, because all the available rules have the quantifiers placed *outside* the premises.

So, code for functions such as `q` can only be derived by trial and error, informed by intuition. This book will help functional programmers to acquire the necessary intuition and technique.

5.4.1 Solved examples

Example 5.4.1.1 Find the cardinality of the type `P = Option[Option[Boolean] => Boolean]`. Write `P` in the type notation and simplify to an equivalent type.

Solution Begin with the type `Option[Boolean]`, which can be either `None` or `Some(x)` with an `x:Boolean`. Since the type `Boolean` has 2 possible values, the type `Option[Boolean]` has 3 values:

$$\left|\text{Opt}^{\text{Bool}}\right| = |\mathbb{1} + \text{Bool}| = 1 + |\text{Bool}| = 3 \quad .$$

In the type notation, `Boolean` is denoted by the symbol 2, and the type `Option[Boolean]` by $\mathbb{1} + 2$. So, the type notation $\mathbb{1} + 2$ is consistent with the cardinality 3 of that type,

$$|\mathbb{1} + \text{Bool}| = |\mathbb{1} + 2| = 1 + 2 = 3 \quad .$$

The function type `Option[Boolean] => Boolean` is denoted by $\mathbb{1} + 2 \to 2$. Its cardinality is computed as the arithmetic power

$$\left|\text{Opt}^{\text{Bool}} \to \text{Bool}\right| = |\mathbb{1} + 2 \to 2| = |2|^{|\mathbb{1}+2|} = 2^3 = 8 \quad .$$

Finally, the we write `P` in the type notation as $P = \mathbb{1} + (\mathbb{1} + 2 \to 2)$ and find

$$|P| = |\mathbb{1} + (\mathbb{1} + 2 \to 2)| = 1 + |\mathbb{1} + 2 \to 2| = 1 + 8 = 9 \quad .$$

Example 5.4.1.2 Implement a Scala type `P[A]` for the type notation

$$P^A \triangleq 1 + A + \text{Int} \times A + (\text{String} \to A) \quad .$$

Solution To translate type notation into Scala code, begin by defining the disjunctive types as case classes (with names chosen for convenience). In this case, P^A is a disjunctive type with four parts, so we will need four case classes:

```
sealed trait P[A]
final case class P1[A](???) extends P[A]
final case class P2[A](???) extends P[A]
final case class P3[A](???) extends P[A]
final case class P4[A](???) extends P[A]
```

Each of the case classes represents one part of the disjunctive type. Now we write the contents for each of the case classes, in order to implement the data in each of the disjunctive parts:

```
sealed trait P[A]
final case class P1[A]()               extends P[A]
final case class P2[A](x: A)           extends P[A]
final case class P3[A](n: Int, x: A)   extends P[A]
final case class P4[A](f: String => A) extends P[A]
```

Example 5.4.1.3 Find an equivalent disjunctive type for the type `P = (Either[A, B], Either[C, D])`.

Solution Begin by writing the given type in the type notation. The tuple becomes the product type, and `Either` becomes the disjunctive (or "sum") type:

$$P \triangleq (A + B) \times (C + D) \quad .$$

We can use the usual rules of arithmetic to expand brackets in this type expression and to obtain an equivalent type:

$$P \cong A \times C + A \times D + B \times C + B \times D \quad .$$

This is a disjunctive type having 4 parts.

Example 5.4.1.4 Show that the following type equivalences do *not* hold: $A + A \not\cong A$ and $A \times A \not\cong A$, although the corresponding logical identities hold.

Solution Note that the arithmetic equalities do not hold, $A + A \neq A$ and $A \times A \neq A$. This already indicates that the types are not equivalent. To build further intuition, consider that a value of type $A + A$ (in Scala, `Either[A, A]`) is a `Left(a)` or a `Right(a)` for some `a:A`. In the code notation, it is either $a^{:A} + \mathbb{0}$ or $\mathbb{0} + a^{:A}$. So, a value of type $A + A$ contains a value of type A with the additional information about whether it is the first or the second part of the disjunctive type. We cannot represent that information in a single value of type A.

Similarly, a value of type $A \times A$ contains two (possibly different) values of type A, which cannot be represented by a single value of type A without loss of information.

However, the corresponding logical identities $\alpha \vee \alpha = \alpha$ and $\alpha \wedge \alpha = \alpha$ hold. To see that, we could derive the four formulas

$$\alpha \vee \alpha \Rightarrow \alpha \quad , \qquad \alpha \Rightarrow \alpha \vee \alpha \quad ,$$
$$\alpha \wedge \alpha \Rightarrow \alpha \quad , \qquad \alpha \Rightarrow \alpha \wedge \alpha \quad ,$$

using the proof rules of Section 5.2.3. Alternatively, we may use the CH correspondence and show that the type signatures

$$\forall A.\, A + A \to A \quad , \qquad \forall A.\, A \to A + A \quad ,$$
$$\forall A.\, A \times A \to A \quad , \qquad \forall A.\, A \to A \times A$$

can be implemented via fully parametric functions. For a programmer, it is easier to write code than to guess the correct sequence of proof rules. For the first pair of type signatures, we find

```
def f1[A]: Either[A, A] => A = {
  case Left(a)    => a   // No other choice here.
  case Right(a)   => a   // No other choice here.
}
def f2[A]: A => Either[A, A] = { a => Left(a) } // Can be also Right(a).
```

The presence of an arbitrary choice, to return `Left(a)` or `Right(a)`, is a warning sign showing that additional information is required to create a value of type `Either[A, A]`. This is precisely the information present in the type $A + A$ but missing in the type A.

The code notation for these functions is

$$f_1 \triangleq \left\| \begin{array}{c|c} & A \\ \hline A & a \to a \\ A & a \to a \end{array} \right\| = \left\| \begin{array}{c|c} & A \\ \hline A & \mathrm{id} \\ A & \mathrm{id} \end{array} \right\| \quad , \qquad f_2 \triangleq a^{:A} \to a + \mathbb{0}^{:A} = \left\| \begin{array}{c|cc} & A & A \\ \hline A & a \to a & \mathbb{0} \end{array} \right\| = \left\| \begin{array}{c|cc} & A & A \\ \hline A & \mathrm{id} & \mathbb{0} \end{array} \right\| \quad .$$

The composition of these functions is not equal to identity:

$$f_1 \fatsemi f_2 = \left\| \begin{array}{c} \mathrm{id} \\ \mathrm{id} \end{array} \right\| \fatsemi \left\| \begin{array}{cc} \mathrm{id} & \mathbb{0} \end{array} \right\| = \left\| \begin{array}{cc} \mathrm{id} & \mathbb{0} \\ \mathrm{id} & \mathbb{0} \end{array} \right\| \neq \mathrm{id} = \left\| \begin{array}{cc} \mathrm{id} & \mathbb{0} \\ \mathbb{0} & \mathrm{id} \end{array} \right\| \quad .$$

For the second pair of type signatures, the code is

```
def f1[A]: ((A, A)) => A = { case (a1, a2) => a1 }   // Can be also ‘a2‘.
cef f2[A]:   A => (A, A) = { a => (a, a) }           // No other choice here.
```

It is clear that the first function loses information when it returns `a1` and discards `a2` (or vice versa).

The code notation for the functions `f1` and `f2` is

$$f_1 \triangleq a_1^{:A} \times a_2^{:A} \to a_1 = \nabla_1^{:A \times A \to A} \quad , \qquad f_2 \triangleq a^{:A} \to a \times a = \Delta^{:A \to A \times A} \quad .$$

Computing the compositions of these functions, we find that $f_2 \fatsemi f_1 = \mathrm{id}$ while $f_1 \fatsemi f_2 \neq \mathrm{id}$:

$$\begin{aligned} f_1 \fatsemi f_2 &= (a_1 \times a_2 \to a_1) \fatsemi (a \to a \times a) \\ &= (a_1 \times a_2 \to a_1 \times a_1) \neq \mathrm{id} = (a_1 \times a_2 \to a_1 \times a_2) \quad . \end{aligned}$$

We have implemented all four type signatures as fully parametric functions, which shows that the corresponding logical formulas are all true (i.e. can be derived using the proof rules). However, the functions cannot be inverses of each other. So, the type equivalences do not hold.

Example 5.4.1.5 Show that $((A \wedge B) \Rightarrow C) \neq (A \Rightarrow C) \vee (B \Rightarrow C)$ in the constructive logic, but the equality holds in Boolean logic. (This is another example where Boolean reasoning about types fails.)

Solution Begin by rewriting the logical equality as two implications,

$$\begin{aligned} &(A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow C) \vee (B \Rightarrow C) \quad , \\ &((A \Rightarrow C) \vee (B \Rightarrow C)) \Rightarrow ((A \wedge B) \Rightarrow C) \quad . \end{aligned}$$

It is sufficient to show that one of these implications is incorrect. Rather than looking for a proof tree in the constructive logic (which would be difficult, since we would need to demonstrate that *no* proof tree exists), let us use the CH correspondence. So the task is to implement fully parametric functions with the type signatures

$$\begin{aligned} &(A \times B \to C) \to (A \to C) + (B \to C) \quad , \\ &(A \to C) + (B \to C) \to A \times B \to C \quad . \end{aligned}$$

For the first type signature, the Scala code is

```
def f1[A,B,C]: (((A, B)) => C) => Either[A => C, B => C] = { k => ??? }
```

We are required to return either a `Left(g)` with `g: A => C`, or a `Right(h)` with `h: B => C`. The only given data is a function `k` of type $A \times B \to C$, so the decision of whether to return a `Left` or a `Right` must be hard-coded in the function `f1` independently of `k`. Can we produce a function `g` of type `A => C`? Given a value of type `A`, we would need to return a value of type `C`. The only way to obtain a value of type

`C` is by applying `k` to some arguments. But to apply `k`, we need a value of type `B`, which we do not have. So we cannot produce a `g: A => C`. Similarly, we cannot produce a function `h` of type `B => C`.

To repeat the same argument in the type notation: Obtaining a value of type $(A \to C) + (B \to C)$ means to compute either $g^{:A \to C} + \mathbb{0}$ or $\mathbb{0} + h^{:B \to C}$. This decision must be hard-coded since the only data is a function $k^{:A \times B \to C}$. We can compute a $g^{:A \to C}$ only by partially applying $k^{:A \times B \to C}$ to a value of type B. However, we cannot obtain any values of type B. Similarly, we cannot get an $h^{:B \to C}$.

The inverse type signature *can* be implemented:

```
def f2[A,B,C]: Either[A=>C, B=>C] => ((A,B)) => C = {
  case Left(g)    =>   { case (a, b)    => g(a) }
  case Right(h)   =>   { case (a, b)    => h(b) }
}
```

$$f_2 \triangleq \begin{array}{|c||c|} \hline & A \times B \to C \\ \hline\hline A \to C & g^{:A \to C} \to a \times b \to g(a) \\ \hline B \to C & h^{:B \to C} \to a \times b \to h(b) \\ \hline \end{array} \quad .$$

Let us now show that the logical identity

$$((\alpha \wedge \beta) \Rightarrow \gamma) = ((\alpha \Rightarrow \gamma) \vee (\beta \Rightarrow \gamma)) \tag{5.23}$$

holds in Boolean logic. A straightforward calculation is to simplify the Boolean expression using Eq. (5.5), which only holds in Boolean logic (but not in the constructive logic). We find

$$\begin{aligned} \text{left-hand side of Eq. (5.23)}: \quad & (\alpha \wedge \beta) \underline{\Rightarrow} \gamma \\ \text{use Eq. (5.5)}: \quad & = \underline{\neg(\alpha \wedge \beta)} \vee \gamma \\ \text{use de Morgan's law}: \quad & = \neg\alpha \vee \neg\beta \vee \gamma \quad . \\ \text{right-hand side of Eq. (5.23)}: \quad & \underline{(\alpha \Rightarrow \gamma)} \vee \underline{(\beta \Rightarrow \gamma)} \\ \text{use Eq. (5.5)}: \quad & = \neg\alpha \vee \underline{\gamma} \vee \neg\beta \vee \underline{\gamma} \\ \text{use identity } \gamma \vee \gamma = \gamma: \quad & = \neg\alpha \vee \neg\beta \vee \gamma \quad . \end{aligned}$$

Both sides of Eq. (5.23) are equal to the same formula, $\neg\alpha \vee \neg\beta \vee \gamma$, so the identity holds.

This calculation does not work in the constructive logic because its proof rules can derive neither the Boolean formula (5.5) nor the **law of de Morgan**, $\neg(\alpha \wedge \beta) = (\neg\alpha \vee \neg\beta)$.

Another way of proving the Boolean identity (5.23) is to enumerate all possible truth values for the variables α, β, and γ. The left-hand side, $(\alpha \wedge \beta) \Rightarrow \gamma$, can be $False$ only if $\alpha \wedge \beta = True$ (that is, both α and β are $True$) and $\gamma = False$; for all other truth values of α, β, and γ, the formula $(\alpha \wedge \beta) \Rightarrow \gamma$ is $True$. Let us determine when the right-hand side, $(\alpha \Rightarrow \gamma) \vee (\beta \Rightarrow \gamma)$, can be $False$. This can happen only if both parts of the disjunction are $False$; that means $\alpha = True$, $\beta = True$, and $\gamma = False$. So, the two sides of the identity (5.23) are both $True$ or both $False$ with any choice of truth values of α, β, and γ. In Boolean logic, this is sufficient to prove the identity (5.23).

It is important to note that the proof rules of the constructive logic are not equivalent to checking whether some propositions are $True$ or $False$. A general form of this statement was proved by K. Gödel in 1932.[6] In this sense, constructive logic does not imply that every proposition is either $True$ or $False$. This is not intuitive and requires getting used to.

The following example shows how to use the identities from Tables 5.5–5.6 to derive type equivalence for complicated type expressions, without need for proofs.

Example 5.4.1.6 Use known rules to verify the type equivalences:

(a) $A \times (A + \mathbb{1}) \times (A + \mathbb{1} + \mathbb{1}) \cong A \times (\mathbb{1} + \mathbb{1} + A \times (\mathbb{1} + \mathbb{1} + \mathbb{1} + A))$.

(b) $\mathbb{1} + A + B \to \mathbb{1} \times B \cong (B \to B) \times (A \to B) \times B$.

Solution **(a)** We can expand brackets in the type expression as in arithmetic,

$$\begin{aligned} A \times (A + \mathbb{1}) &\cong A \times A + A \times \mathbb{1} \cong A \times A + A \quad , \\ A \times (A + \mathbb{1}) \times (A + \mathbb{1} + \mathbb{1}) &\cong (A \times A + A) \times (A + \mathbb{1} + \mathbb{1}) \\ &\cong A \times A \times A + A \times A + A \times A \times (\mathbb{1} + \mathbb{1}) + A \times (\mathbb{1} + \mathbb{1}) \\ &\cong A \times A \times A + A \times A \times (\mathbb{1} + \mathbb{1} + \mathbb{1}) + A \times (\mathbb{1} + \mathbb{1}) \quad . \end{aligned}$$

[6] See plato.stanford.edu/entries/intuitionistic-logic-development/

The result looks like a polynomial in A, which we can now rearrange into the required form:

$$A \times A \times A + A \times A \times (\mathbb{1} + \mathbb{1} + \mathbb{1}) + A \times (\mathbb{1} + \mathbb{1}) \cong A \times (\mathbb{1} + \mathbb{1} + A \times (\mathbb{1} + \mathbb{1} + \mathbb{1} + A)) \quad .$$

(b) Keep in mind that the conventions of the type notation make the function arrow ($\to$) group weaker than other type operations. So, the type expression $\mathbb{1} + A + B \to \mathbb{1} \times B$ means a function from $\mathbb{1} + A + B$ to $\mathbb{1} \times B$.

Begin by using the rule $\mathbb{1} \times B \cong B$ to obtain $\mathbb{1} + A + B \to B$. Now we use the rule

$$A + B \to C \cong (A \to C) \times (B \to C)$$

and derive the equivalence

$$\mathbb{1} + A + B \to B \cong (\mathbb{1} \to B) \times (A \to B) \times (B \to B) \quad .$$

Finally, we note that $\mathbb{1} \to B \cong B$ and that the type product is commutative, so we can rearrange the last type expression into the required form:

$$B \times (A \to B) \times (B \to B) \cong (B \to B) \times (A \to B) \times B \quad .$$

Example 5.4.1.7 Denote $\text{Read}^{E,T} \triangleq E \to T$ and implement fully parametric functions with types $A \to \text{Read}^{E,A}$ and $\text{Read}^{E,A} \to (A \to B) \to \text{Read}^{E,B}$.

Solution Begin by defining a type alias for the type constructor $\text{Read}^{E,T}$:

```
type Read[E, T] = E => T
```

The first type signature has only one implementation:

```
def p[E, A]: A => Read[E, A] = { x => _ => x }
```

We *must* discard the argument of type E; we cannot use it for computing a value of type `A` given `x:A`.

The second type signature has three type parameters. It is the curried version of the function `map`:

```
def map[E, A, B]: Read[E, A] => (A => B) => Read[E, B] = ???
```

Expanding the type alias, we see that the two curried arguments are functions of types $E \to A$ and $A \to B$. The forward composition of these functions is a function of type $E \to B$, or $\text{Read}^{E,B}$, which is exactly what we are required to return. So the code can be written as

```
def map[E, A, B]: (E => A) => (A => B) => E => B = { r => f => r andThen f }
```

If we did not notice this shortcut, we would reason differently: We are required to compute a value of type B given *three* curried arguments $r^{:E \to A}$, $f^{:A \to B}$, and $e^{:E}$. Write this requirement as

$$\text{map} \triangleq r^{:E \to A} \to f^{:A \to B} \to e^{:E} \to ???^{:B} \quad ,$$

The symbol $???^{:B}$ is called a **typed hole**; it stands for a value that we are still figuring out how to compute, but whose type is already known. Typed holes are supported in Scala by an experimental compiler plugin.[7] The plugin will print the known information about the typed hole.

To fill the typed hole $???^{:B}$, we need a value of type B. Since no arguments have type B, the only way of getting a value of type B is to apply $f^{:A \to B}$ to some value of type A. So we write

$$\text{map} \triangleq r^{:E \to A} \to f^{:A \to B} \to e^{:E} \to f(???^{:A}) \quad .$$

The only way of getting an A is to apply r to a value of type E,

$$\text{map} \triangleq r^{:E \to A} \to f^{:A \to B} \to e^{:E} \to f(r(???^{:E})) \quad .$$

We have exactly one value of type E, namely $e^{:E}$. So the code must be

$$\text{map}^{E,A,B} \triangleq r^{:E \to A} \to f^{:A \to B} \to e^{:E} \to f(r(e)) \quad .$$

Translate this to the Scala syntax:

[7] https://github.com/cb372/scala-typed-holes

```
def map[E, A, B]: (E => A) => (A => B) => E => B = { r => f => e => f(r(e)) }
```

We may now notice that the expression $e \to f(r(e))$ is a function composition $r \fatsemi f$ applied to e, and simplify the code accordingly.

Example 5.4.1.8 Show that the type signature `Read[A, T] => (A => B) => Read[B, T]` cannot be implemented as a fully parametric function.

Solution Expand the type signature and try implementing this function:

```
def m[A, B, T] : (A => T) => (A => B) => B => T = { r => f => b => ??? }
```

Given values $r^{:A \to T}$, $f^{:A \to B}$, and $b^{:B}$, we need to compute a value of type T:

$$m = r^{:A \to T} \to f^{:A \to B} \to b^{:B} \to ???^{:T} \quad .$$

The only way of getting a value of type T is to apply r to some value of type A,

$$m = r^{:A \to T} \to f^{:A \to B} \to b^{:B} \to r(???^{:A}) \quad .$$

However, we do not have any values of type A. We have a function $f^{:A \to B}$ that *consumes* values of type A, and we cannot use f to produce any values of type A. So we seem to be unable to fill the typed hole $???^{:A}$ and implement the function `m`.

In order to verify that `m` is unimplementable, we need to prove that the logical formula

$$\forall(\alpha, \beta, \tau).\, (\alpha \Rightarrow \tau) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \tau) \tag{5.24}$$

is not true in the constructive logic. We could use the `curryhoward` library for that:

```
@ def m[A, B, T] : (A => T) => (A => B) => B => T = implement
cmd1.sc:1: type (A => T) => (A => B) => B => T cannot be implemented
def m[A, B, T] : (A => T) => (A => B) => B => T = implement
                                                 ^

Compilation Failed
```

Another way is to check whether this formula is true in Boolean logic. A formula that holds in constructive logic will always hold in Boolean logic, because all rules shown in Section 5.2.3 preserve Boolean truth values (see Section 5.5.4 for a proof). It follows that any formula that fails to hold in Boolean logic will also not hold in constructive logic.

It is relatively easy to check whether a given Boolean formula is always equal to $True$. Simplifying Eq. (5.24) with the rules of Boolean logic, we find

$$\begin{aligned}
& (\alpha \Rightarrow \tau) \underline{\Rightarrow} (\alpha \Rightarrow \beta) \underline{\Rightarrow} (\beta \Rightarrow \tau) \\
\text{use Eq. (5.5)}: \quad & = \neg\underline{(\alpha \Rightarrow \tau)} \vee \neg\underline{(\alpha \Rightarrow \beta)} \vee \underline{(\beta \Rightarrow \tau)} \\
\text{use Eq. (5.5)}: \quad & = \underline{\neg(\neg\alpha \vee \tau)} \vee \underline{\neg(\neg\alpha \vee \beta)} \vee (\neg\beta \vee \tau) \\
\text{use de Morgan's law}: \quad & = (\alpha \wedge \neg\tau) \vee \underline{(\alpha \wedge \neg\beta) \vee \neg\beta} \vee \tau \\
\text{use identity } (p \wedge q) \vee q = q: \quad & = \underline{(\alpha \wedge \neg\tau)} \vee \neg\beta \vee \underline{\tau} \\
\text{use identity } (p \wedge \neg q) \vee q = p \vee q: \quad & = \alpha \vee \neg\beta \vee \tau \quad .
\end{aligned}$$

This formula is not identically $True$: it is $False$ when $\alpha = \tau = False$ and $\beta = True$. So, Eq. (5.24) is not true in Boolean logic, and thus is not true in constructive logic. By the CH correspondence, we conclude that the type signature of `m` cannot be implemented by a fully parametric function.

Example 5.4.1.9 Define the type constructor $P^A \triangleq \mathbb{1} + A + A$ and implement `map` for it,

$$\text{map}^{A,B} : P^A \to (A \to B) \to P^B \quad .$$

To check that `map` preserves information, verify the law `map(p)(x => x) == p` for all `p:P[A]`.

Solution It is implied that `map` should be fully parametric and information-preserving. Begin by defining a Scala type constructor for the notation $P^A \triangleq \mathbb{1} + A + A$:

```
sealed trait P[A]
final case class P1[A]()     extends P[A]
final case class P2[A](x: A) extends P[A]
final case class P3[A](x: A) extends P[A]
```

Now we can write code to implement the required type signature. Each time we have several choices of an implementation, we will choose to preserve information as much as possible.

```
def map[A, B]: P[A] => (A => B) => P[B] =
  p => f => p match {
    case P1() => P1() // No other choice.
    case P2(x) => ???
    case P3(x) => ???
  }
```

In the case `P2(x)`, we are required to produce a value of type P^B from a value $x^{:A}$ and a function $f^{:A\to B}$. Since P^B is a disjunctive type with three parts, we can produce a value of type P^B in three different ways: `P1()`, `P2(...)`, and `P3(...)`. If we return `P1()`, we will lose the information about the value `x`. If we return `P3(...)`, we will preserve the information about `x` but lose the information that the input value was a `P2` rather than a `P3`. By returning `P2(...)` in that scope, we preserve the entire input information.

The value under `P2(...)` must be of type B, and the only way of getting a value of type B is to apply f to x. So, we return `P2(f(x))`.

Similarly, in the case `P3(x)`, we should return `P3(f(x))`. The final code of `map` is

```
def map[A, B]: P[A] => (A => B) => P[B] = p => f => p match {
  case P1() => P1()         // No other choice here.
  case P2(x) => P2(f(x))    // Preserve information.
  case P3(x) => P3(f(x))    // Preserve information.
}
```

To verify the given law, we first write a matrix notation for `map`:

$$\text{map}^{A,B} \triangleq p^{:\mathbb{1}+A+A} \to f^{:A\to B} \to p \triangleright \left|\begin{array}{c||ccc} & \mathbb{1} & B & B \\ \hline \mathbb{1} & \text{id} & \mathbb{0} & \mathbb{0} \\ A & \mathbb{0} & f & \mathbb{0} \\ A & \mathbb{0} & \mathbb{0} & f \end{array}\right| \quad .$$

The required law is written as an equation (called the **identity law**)

$$\text{map}(p)(\text{id}) = p \quad .$$

Substituting the code notation for `map`, we verify the law:

$$\begin{aligned} \text{expect to equal } p: \quad & \text{map}(p)(\text{id}) \\ \text{apply map()() to arguments}: \quad & = p \triangleright \left|\begin{array}{ccc} \text{id} & \mathbb{0} & \mathbb{0} \\ \mathbb{0} & \text{id} & \mathbb{0} \\ \mathbb{0} & \mathbb{0} & \text{id} \end{array}\right| \\ \text{identity function in matrix notation}: \quad & = p \triangleright \text{id} \\ \triangleright\text{-notation}: \quad & = \text{id}(p) = p \quad . \end{aligned}$$

Example 5.4.1.10 Implement `map` and `flatMap` for `Either[L, R]`, applied to the type parameter `L`.

Solution For a type constructor, say, P^A, the standard type signatures for `map` and `flatMap` are

$$\begin{aligned} \text{map} &: P^A \to (A \to B) \to P^B \quad , \\ \text{flatMap} &: P^A \to (A \to P^B) \to P^B \quad . \end{aligned}$$

If a type constructor has more than one type parameter, e.g. $P^{A,S,T}$, one can define the functions `map` and `flatMap` applied to a chosen type parameter. For example, when applied to the type parameter A, the type signatures are

$$\text{map} : P^{A,S,T} \to (A \to B) \to P^{B,S,T} \quad ,$$
$$\text{flatMap} : P^{A,S,T} \to (A \to P^{B,S,T}) \to P^{B,S,T} \quad .$$

Being "applied to the type parameter A" means that the other type parameters S, T in $P^{A,S,T}$ remain fixed while the type parameter A is replaced by B in the type signatures of `map` and `flatMap`.

For the type `Either[L, R]` (i.e. $L + R$), we keep the type parameter R fixed while L is replaced by M. So we obtain the type signatures

$$\text{map} : L + R \to (L \to M) \to M + R \quad ,$$
$$\text{flatMap} : L + R \to (L \to M + R) \to M + R \quad .$$

Implementing these functions is straightforward:

```
def map[L,M,R]: Either[L, R] => (L => M) => Either[M, R] = e => f => e match {
  case Left(x)    => Left(f(x))
  case Right(y)   => Right(y)
}
def flatMap[L,M,R]: Either[L, R] => (L => Either[M, R]) => Either[M, R] = e => f => e match {
  case Left(x)    => f(x)
  case Right(y)   => Right(y)
}
```

The code notation for these functions is

$$\text{map} \triangleq e^{:L+R} \to f^{:L\to M} \to e \triangleright \begin{array}{c|cc} & M & R \\ \hline L & f & \mathbb{0} \\ R & \mathbb{0} & \text{id} \end{array} \quad ,$$

$$\text{flatMap} \triangleq e^{:L+R} \to f^{:L\to M+R} \to e \triangleright \begin{array}{c|c} & M + R \\ \hline L & f \\ R & y^{:R} \to \mathbb{0}^{:M} + y \end{array} \quad .$$

Note that we cannot split f into the M and R columns since $f(x^{:L})$ could return either part of the disjunctive type $M + R$.

Example 5.4.1.11* Define a type constructor $\text{State}^{S,A} \equiv S \to A \times S$ and implement the functions:

(a) $\text{pure}^{S,A} : A \to \text{State}^{S,A}$.

(b) $\text{map}^{S,A,B} : \text{State}^{S,A} \to (A \to B) \to \text{State}^{S,B}$.

(c) $\text{flatMap}^{S,A,B} : \text{State}^{S,A} \to (A \to \text{State}^{S,B}) \to \text{State}^{S,B}$.

Solution It is assumed that all functions must be fully parametric and preserve as much information as possible. We define the type alias

```
type State[S, A] = S => (A, S)
```

(a) The type signature is $A \to S \to A \times S$, and there is only one implementation,

```
def pure[S, A]: A => State[S, A] = a => s => (a, s)
```

In the code notation, this is written as

$$\text{pu}^{S,A} \triangleq a^{:A} \to s^{:A} \to a \times s \quad .$$

(b) The type signature is

$$\text{map}^{S,A,B} : (S \to A \times S) \to (A \to B) \to S \to B \times S \quad .$$

Begin writing a Scala implementation:

```
def map[S, A, B]: State[S, A] => (A => B) => State[S, B] = { t => f => s => ??? }
```

We need to compute a value of $B \times S$ from the curried arguments $t^{:S \to A \times S}$, $f^{:A \to B}$, and $s^{:S}$. We begin writing the code of `map` using a typed hole,

$$\text{map} \triangleq t^{:S \to A \times S} \to f^{:A \to B} \to s^{:S} \to ???^{:B} \times ???^{:S} \quad .$$

The only way of getting a value of type B is by applying f to a value of type A:

$$\text{map} \triangleq t^{:S \to A \times S} \to f^{:A \to B} \to s^{:S} \to f(???^{:A}) \times ???^{:S} \quad .$$

The only possibility of filling the typed hole $???^{:A}$ is to apply t to a value of type S; we already have such a value, $s^{:S}$. Computing $t(s)$ yields a pair of type $A \times S$, from which we may take the first part (of type A) to fill the typed hole $???^{:A}$. The second part of the pair is a value of type S that we may use to fill the second typed hole, $???^{:S}$. So the Scala code is

```
1 def map[S, A, B]: State[S, A] => (A => B) => State[S, B] = {
2   t => f => s =>
3     val (a, s2) = t(s)
4     (f(a), s2)     // We could also return ‘(f(a), s)‘ here.
5 }
```

Why not return the original value `s` in the tuple $B \times S$, instead of the new value `s2`? The reason is that we would like to preserve information as much as possible. If we return `(f(a), s)` in line 4, we will have discarded the computed value `s2`, which is a loss of information.

To write the code notation for `map`, we need to destructure the pair that $t(s)$ returns. We can write explicit destructuring code like this:

$$\text{map} \triangleq t^{:S \to A \times S} \to f^{:A \to B} \to s^{:S} \to (a^{:A} \times s_2^{:S} \to f(a) \times s_2)(t(s)) \quad .$$

If we temporarily denote by q the destructuring function

$$q \triangleq (a^{:A} \times s_2^{:S} \to f(a) \times s_2) \quad ,$$

we will notice that the expression $s \to q(t(s))$ is a function composition applied to s. So, we rewrite $s \to q(t(s))$ as the composition $t \fatsemi q$ and obtain shorter code,

$$\text{map} \triangleq t^{:S \to A \times S} \to f^{:A \to B} \to t \fatsemi (a^{:A} \times s^{:S} \to f(a) \times s) \quad .$$

Shorter formulas are often easier to reason about in derivations (although not necessarily easier to read when converted to program code).

(c) The required type signature is

$$\text{flatMap}^{S,A,B} : (S \to A \times S) \to (A \to S \to B \times S) \to S \to B \times S \quad .$$

We perform code reasoning with typed holes:

$$\text{flatMap} \triangleq t^{:S \to A \times S} \to f^{:A \to S \to B \times S} \to s^{:S} \to ???^{:B} \times ???^{:S} \quad .$$

To fill $???^{:B}$, we need to apply f to some arguments, since f is the only function that returns any values of type B. A saturated application of f will yield a value of type $B \times S$, which we can return without change:

$$\text{flatMap} \triangleq t^{:S \to A \times S} \to f^{:A \to S \to B \times S} \to s^{:S} \to f(???^{:A})(???^{:S}) \quad .$$

To fill the new typed holes, we need to apply t to an argument of type S. We have only one given value $s^{:S}$ of type S, so we must compute $t(s)$ and destructure it:

$$\text{flatMap} \triangleq t^{:S \to A \times S} \to f^{:A \to S \to B \times S} \to s^{:S} \to (a \times s_2 \to f(a)(s_2))\,(t(s)) \quad .$$

Translating this notation into Scala code, we obtain

```
def flatMap[S,A,B]: State[S, A] => (A => State[S, B]) => State[S, B] = {
  t => f => s =>
    val (a, s2) = t(s)
    f(a)(s2)            // We could also return `f(a)(s)` here.
}
```

As before, in order to preserve information, we choose not to discard the computed value `s2`.

The code notation for this `flatMap` can be simplified to

$$\text{flatMap} \triangleq t^{:S \to A \times S} \to f^{:A \to S \to B \times S} \to t \mathbin{⨟} (a \times s \to f(a)(s)) \quad .$$

5.4.2 Exercises

Exercise 5.4.2.1 Find the cardinality of the type `P = Option[Boolean => Option[Boolean]]`. Show that `P` is equivalent to `Option[Boolean] => Boolean`, and that the equivalence is accidental and not "natural".

Exercise 5.4.2.2 Verify the type equivalences $A + A \cong 2 \times A$ and $A \times A \cong 2 \to A$, where 2 denotes the `Boolean` type.

Exercise 5.4.2.3 Show that $A \Rightarrow (B \vee C) \neq (A \Rightarrow B) \wedge (A \Rightarrow C)$ in constructive and Boolean logic.

Exercise 5.4.2.4 Use known rules to verify the type equivalences without need for proofs:

(a) $(A + B) \times (A \to B) \cong A \times (A \to B) + (\mathbb{1} + A \to B)$.

(b) $(A \times (\mathbb{1} + A) \to B) \cong (A \to B) \times (A \to A \to B)$.

(c) $A \to (\mathbb{1} + B) \to C \times D \cong (A \to C) \times (A \to D) \times (A \times B \to C) \times (A \times B \to D)$.

Exercise 5.4.2.5 Write the type notation for `Either[(A, Int), Either[(A, Char), (A, Float)]]`. Transform this type into an equivalent type of the form $A \times (...)$.

Exercise 5.4.2.6 Define a type $\text{OptE}^{T,A} \triangleq \mathbb{1} + T + A$ and implement information-preserving `map` and `flatMap` for it, applied to the type parameter A. Get the same result using the equivalent type $(\mathbb{1} + A) + T$, i.e. `Either[Option[A], T]`. The required type signatures are

$$\text{map}^{A,B,T} : \text{OptE}^{T,A} \to (A \to B) \to \text{OptE}^{T,B} \quad ,$$
$$\text{flatMap}^{A,B,T} : \text{OptE}^{T,A} \to (A \to \text{OptE}^{T,B}) \to \text{OptE}^{T,B} \quad .$$

Exercise 5.4.2.7 Implement the `map` function for `P[A]` (see Example 5.4.1.2). The required type signature is $P^A \to (A \to B) \to P^B$. Preserve information as much as possible.

Exercise 5.4.2.8 For the type constructor $Q^{T,A}$ defined in Exercise 5.1.4.1, define the `map` function, preserving information as much as possible,

$$\text{map}^{T,A,B} : Q^{T,A} \to (A \to B) \to Q^{T,B} \quad .$$

Exercise 5.4.2.9 Define a recursive type constructor Tr_3 as $\text{Tr}_3{}^A \triangleq \mathbb{1} + A \times A \times A \times \text{Tr}_3{}^A$ and implement the `map` function for it, with the standard type signature: $\text{map}^{A,B} : \text{Tr}_3{}^A \to (A \to B) \to \text{Tr}_3{}^B$.

Exercise 5.4.2.10 Implement fully parametric, information-preserving functions with the types:

(a) $Z + A \times A \to (A \to B) \to Z + B \times B$.

(b) $A + Z \to B + Z \to (A \to B \to C) \to C + Z$.

(c) $\text{flatMap}^{E,A,B} : \text{Read}^{E,A} \to (A \to \text{Read}^{E,B}) \to \text{Read}^{E,B}$.

(d) $\text{State}^{S,A} \to (S \times A \to S \times B) \to \text{State}^{S,B}$.

Exercise 5.4.2.11* Denote $\text{Cont}^{R,T} \triangleq (T \to R) \to R$ and implement the functions:

(a) $\text{map}^{R,T,U} : \text{Cont}^{R,T} \to (T \to U) \to \text{Cont}^{R,U}$.

(b) $\text{flatMap}^{R,T,U} : \text{Cont}^{R,T} \to (T \to \text{Cont}^{R,U}) \to \text{Cont}^{R,U}$.

Exercise 5.4.2.12* Denote $\text{Select}^{Z,T} \triangleq (T \to Z) \to T$ and implement the functions:

(a) $\text{map}^{Z,A,B} : \text{Select}^{Z,A} \to (A \to B) \to \text{Select}^{Z,B}$.

(b) $\text{flatMap}^{Z,A,B} : \text{Select}^{Z,A} \to (A \to \text{Select}^{Z,B}) \to \text{Select}^{Z,B}$.

5.5 Discussion

5.5.1 Using the Curry-Howard correspondence for writing code

This chapter shows how the CH correspondence performs two practically important reasoning tasks: checking whether a type signature can be implemented as a fully parametric function, and determining whether two types are equivalent. The first task is accomplished by mapping type expressions into formulas in the constructive logic and by applying the proof rules of that logic. The second task is accomplished by mapping type expressions into *arithmetic* formulas and applying the ordinary rules of arithmetic.

Fully parametric functions can be often derived from their type signatures alone. It is useful for a programmer to know that certain type signatures, such as

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

have only one possible implementation, while other type signatures, such as

```
def g[A, B]: A => (B => A) => B
def h[A, B]: ((A => B) => A) => A
```

cannot be implemented as fully parametric functions.

Although tools such as the `curryhoward` library can sometimes derive code from types, it is more beneficial if a programmer is able to derive an implementation by hand, or to recognize quickly that an implementation is impossible. Exercises in this chapter help to build up the required technique and intuition. For instance, we have seen heuristics such as "values of parametric types cannot be constructed from scratch" and "one must hard-code the decision to return a chosen part of a disjunctive type". These heuristics can be justified by the rigorous rules of proof (Section 5.2.3).

Throughout this chapter, we required all functions to be fully parametric. The reason is that the CH correspondence becomes informative only with parameterized types and with fully parametric functions. For concrete types, e.g. `Int`, one can always produce *some* value even with no previous data, so the proposition $\mathcal{CH}(\text{Int})$ is always true within any code.

Consider the function `(x:Int) => x + 1`. Its type signature, `Int => Int`, is insufficient to specify the code of the function, because there are many different functions with the same type signature, such as `x => x - 1`, `x => x * 2`, etc. So, deriving code from the type signature `Int => Int` is not a meaningful task. Only a fully parametric type signature, such as $A \to (A \to B) \to B$, gives enough information for possibly deriving the function's code. If we permit functions that are not fully parametric, we will not be able to reason about implementability of type signatures or about code derivation.

Information about the implementability of type signatures is given by logical formulas involving $\mathcal{CH}$-propositions. Validity of a $\mathcal{CH}$-proposition $\mathcal{CH}(T)$ means that we can compute *some* value of the given type T but does not give any information about the properties of that value, such as whether it satisfies any additional laws. This is why type equivalence (which requires the laws of isomorphism) is not determined by an equivalence of logical formulas.

It is useful for programmers to be able to reason about types and transform type expressions to equivalent simpler types before starting to write code. The type notation introduced in this book is designed to help programmers to recognize patterns in type expressions and to reason about them more easily. We have shown that a type equivalence corresponds to *each* standard arithmetic identity such as $(a+b)+c = a+(b+c)$, $(a \times b) \times c = a \times (b \times c)$, $1 \times a = a$, $(a+b) \times c = a \times c + b \times c$, etc. So, we are allowed to transform and simplify types as if they were arithmetic expressions, e.g. to rewrite

$$\mathbb{1} \times (A + B) \times C + D \cong D + A \times C + B \times C \quad .$$

The type notation makes this reasoning more intuitive (for people familiar with arithmetic).

These results apply to all type expressions built up using product types, disjunctive types (also called "sum" types because they correspond to arithmetic sums), and function types (also called "exponential" types because they correspond to arithmetic exponentials). Type expressions that contain only products and sum types may be called **polynomial**. Type expressions that also contain function types may be called **exponential-polynomial**.[8] This book focuses on exponential-polynomial types because they are sufficient for almost all design patterns used in functional programming.

There are no type constructions corresponding to subtraction or division, so equations such as

$$(1-t)\times(1+t) = 1 - t\times t \quad \text{or} \quad \frac{t + t\times t}{t} = 1 + t$$

do not directly yield any type equivalences. However, consider this well-known formula,

$$\frac{1}{1-t} = 1 + t + t^2 + t^3 + ... + t^n + ... \quad .$$

At first sight, this formula appears to involve subtraction, division, and an infinite series, and thus cannot be directly translated into a type equivalence. However, the formula can be rewritten as

$$\frac{1}{1-t} \triangleq L(t) = 1 + t + t^2 + t^3 + ... + t^n \times L(t) \quad , \tag{5.25}$$

which is finite and only contains additions and multiplications. So, Eq. (5.25) can be translated into a type equivalence:

$$L^A \cong 1 + A + A\times A + A\times A\times A + ... + \underbrace{A\times ...\times A}_{n \text{ times}} \times L^A \quad . \tag{5.26}$$

This type formula (with $n = 1$) is equivalent to a recursive definition of the type constructor `List`,

$$\text{List}^A \triangleq 1 + A \times \text{List}^A \quad .$$

The type equivalence (5.26) suggests that we may view the recursive type `List` as an "infinite disjunction" describing lists of zero, one, etc. elements.

5.5.2 Implications for designing new programming languages

The functional programming paradigm assumes that programmers will use the six standard type constructions (Section 5.1.2) and the eight standard code constructions (Section 5.2.3). These constructions are foundational in the sense that they are used to express all design patterns of functional programming. A language that does not directly support some of these constructions cannot be considered a functional programming language.

A remarkable consequence of the CH correspondence is that the type system of any programming language (functional or not) is mapped into a *certain logic*, i.e. a system of logical operations and proof rules. A logical operation will correspond to each of the type constructions available in the programming language; a proof rule will correspond to each of the available code constructions. Functional programming languages that support all the standard type and code constructions – for instance, OCaml, Haskell, F#, Scala, Swift, etc., – will be mapped into the constructive logic with all standard logical operations available ($True$, $False$, disjunction, conjunction, and implication).

Languages such as C, C++, Java, C# are mapped into logics that do not have the disjunction operation or the constants $True$ and $False$. In other words, these languages are mapped into *incomplete* logics where some theorems will not be provable. (If $\mathcal{CH}(A)$ is true but not provable, a value of type A is not directly computable by programs, although it could have been.) Incompleteness of the logic

[8] Polynomial types are often called "algebraic data types".

of types will make a programming language unable to express certain computations, e.g. directly handle data that belongs to a disjoint domain.

Languages such as Python, JavaScript, Ruby, Clojure have no type checking and so are mapped to *inconsistent* logics where any proposition can be derived — even propositions normally considered $False$ may be derived from $True$. The CH correspondence will map such derivations to code that *appears to* compute a certain value although that value is not actually available. In practice, such code *crashes* (although all types match!) because the computed value has a wrong type, is "null", or is a pointer to an invalid memory location.

None of these errors will happen in a programming language whose logic of types is complete and consistent, provided that types are checked at compile time.

So, the CH correspondence gives a mathematically justified procedure for designing type systems in new programming languages. The procedure has the following steps:

- Choose a formal logic that is complete and free of inconsistencies.
- For each logical operation, provide a type construction in the language.
- For each proof rule and axiom, provide a code construction in the language.

Mathematicians have studied different logics: e.g. modal logic, temporal logic, or linear logic. Compared with the constructive logic, these other logics have some additional operations. (For instance, modal logic adds the operations "necessarily" and "possibly", and temporal logic adds the operation "until".) For each logic, mathematicians have determined the minimal complete sets of operations, axioms, and proof rules that do not lead to inconsistency. Programming language designers can choose a logic and translate it into a minimal programming language where the code is guaranteed *not to crash* as long as types match. This mathematical guarantee (known as **type safety**) is a powerful help for programmers since it automatically prevents a large set of programming errors. So, programmers will benefit if their programming language is designed using the CH correspondence.

Practically useful programming languages will, of course, introduce many more features than the minimal, mathematically necessary constructions derived from the chosen logic. Programmers will still benefit from type safety as long as their programs stay within the mathematically consistent subset of the language. For Scala, a "safe" subset is identified by the `scalazzi` project.[9]

At present, it is not fully understood whether a practical programming language can use e.g. modal or linear logic as its logic of types. Experience suggests that, at least, the operations of the plain constructive logic should be available. So, it appears that the six type constructions and the eight code constructions will remain available in all future languages of functional programming.

5.5.3 Uses of the void type

Scala's void type (`Nothing`) corresponds to the logical constant $False$. The practical uses of $False$ are quite limited. One use case is for a branch of a `match` / `case` expression that does not return a value because it throws an exception. Such branches are considered formally to return a value of type `Nothing`, which can then be mapped to a value of any other type (through the function `absurd[A]: Nothing => A`, see Example 5.3.4.2).

To see how this trick is used, consider this code defining a value `x`,

```
val x: Double = if (t >= 0.0) math.sqrt(t) else { throw new Exception("error") }
```

The `else` branch does not return a value, but `x` is declared to be of type `Double`. For this code to typecheck, both branches must return values of the same type. So, the compiler needs to pretend that the `else` branch also returns a value of type `Double`. The compiler first assigns the type `Nothing` to the expression `throw ...` and then implicitly uses the function `absurd: Nothing => Double` to convert that type to `Double`. In this way, types will match in the definition of the value `x`.

[9] https://github.com/scalaz/scalazzi

Constructive logic	Boolean logic
$\frac{}{\Gamma \vdash \mathcal{CH}(\mathbb{1})}$ (create unit)	$\neg\Gamma \vee True = True$
$\frac{}{\Gamma, \alpha \vdash \alpha}$ (use arg)	$\neg\Gamma \vee \neg\alpha \vee \alpha = True$
$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta}$ (create function)	$(\neg\Gamma \vee \neg\alpha \vee \beta) = (\neg\Gamma \vee (\alpha \Rightarrow \beta))$
$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \alpha \Rightarrow \beta}{\Gamma \vdash \beta}$ (use function)	$((\neg\Gamma \vee \alpha) \wedge (\neg\Gamma \vee (\alpha \Rightarrow \beta))) \Rightarrow (\neg\Gamma \vee \beta)$
$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta}$ (create tuple)	$(\neg\Gamma \vee \alpha) \wedge (\neg\Gamma \vee \beta) = (\neg\Gamma \vee (\alpha \wedge \beta))$
$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha}$ (use tuple-1)	$(\neg\Gamma \vee (\alpha \wedge \beta)) \Rightarrow (\neg\Gamma \vee \alpha)$
$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta}$ (create Left)	$(\neg\Gamma \vee \alpha) \Rightarrow (\neg\Gamma \vee (\alpha \vee \beta))$
$\frac{\Gamma \vdash \alpha \vee \beta \quad \Gamma, \alpha \vdash \gamma \quad \Gamma, \beta \vdash \gamma}{\Gamma \vdash \gamma}$ (use Either)	$((\neg\Gamma \vee \alpha \vee \beta) \wedge (\neg\Gamma \vee \neg\alpha \vee \gamma) \wedge (\neg\Gamma \vee \neg\beta \vee \gamma)) \Rightarrow (\neg\Gamma \vee \gamma)$

Table 5.7: Proof rules of constructive logic are true also in the Boolean logic.

We will not use exceptions in this book. The functional programming paradigm avoids exceptions because their presence significantly complicates reasoning about code.

As another example of using the void type, suppose an external library implements a function

```
def parallel_run[E, A, B](f: A => Either[E, B]) = ???
```

that performs some parallel computations using a given function f. In general, the library supports functions $f^{:A \to E+B}$ that may return an error of type E or a result of type B. Suppose we know that a particular function f never fails to compute its result. To express that knowledge in code, we may explicitly set the type parameter E to the void type `Nothing` when applying `parallel_run`:

```
parallel_run[Nothing, A, B](f)  // Types match only when values f(a) always are of the form Right(b).
```

Returning an error is now impossible (the type `Nothing` has no values). If the function `parallel_run` is fully parametric, it will work in the same way with all types E, including $E = \mathbb{0}$. The code implements our intention via type parameters, giving a compile-time guarantee of correct results.

So far, none of our examples involved the logical **negation** operation. It is defined as

$$\neg\alpha \triangleq \alpha \Rightarrow False \quad ,$$

and its practical use is as limited as that of $False$ and the void type. However, logical negation plays an important role in Boolean logic, which we will discuss next.

5.5.4 Relationship between Boolean logic and constructive logic

We have seen that some true theorems of Boolean logic are not true in constructive logic. For example, the Boolean identities $\neg(\neg\alpha) = \alpha$ and $(\alpha \Rightarrow \beta) = (\neg\alpha \vee \beta)$ do not hold in the constructive logic. However, any theorem of constructive logic is also a theorem of Boolean logic. The reason is that all eight rules of constructive logic (Section 5.2.3) are also true in Boolean logic.

To verify that a formula is true in Boolean logic, we only need to check that the value of the formula is $True$ for all possible truth values ($True$ or $False$) of its variables. A sequent such as $\alpha, \beta \vdash \gamma$ is true in Boolean logic if and only if $\gamma = True$ under the assumption that $\alpha = \beta = True$. So, the sequent $\alpha, \beta \vdash \gamma$ is translated into the Boolean formula

$$\alpha, \beta \vdash \gamma = ((\alpha \wedge \beta) \Rightarrow \gamma) = (\neg\alpha \vee \neg\beta \vee \gamma) \quad .$$

Table 5.7 translates all proof rules of Section 5.2.3 into Boolean formulas. The first two lines are axioms, while the subsequent lines are Boolean theorems that can be verified by calculation.

To simplify the calculations, note that all terms in the formulas contain the operation $(\neg\Gamma \vee ...)$ corresponding to the context Γ. Now, if Γ is $False$, the entire formula becomes automatically $True$,

and there is nothing else to check. So, it remains to verify the formula in case $\Gamma = True$, and then we can simply omit all instances of $\neg\Gamma$ in the formulas. Let us show the Boolean derivations for the rules "use function" and "use Either"; other formulas are checked in a similar way.

$$\begin{aligned}
\text{formula "use function"}: &\quad (\alpha \wedge (\alpha \Rightarrow \beta)) \Rightarrow \beta \\
\text{use Eq. (5.5)}: &\quad = \underline{\neg}(\alpha \underline{\wedge} (\neg\alpha \underline{\vee} \beta)) \vee \beta \\
\text{de Morgan's laws}: &\quad = \underline{\neg\alpha \vee (\alpha \wedge \neg\beta)} \vee \beta \\
\text{identity } p \vee (\neg p \wedge q) = p \vee q \text{ with } p = \neg\alpha \text{ and } q = \beta: &\quad = \neg\alpha \vee \underline{\neg\beta \vee \beta} \\
\text{axiom "use arg"}: &\quad = True \quad .
\end{aligned}$$

$$\begin{aligned}
\text{formula "use Either"}: &\quad ((\alpha \vee \beta) \wedge (\alpha \Rightarrow \gamma) \wedge (\beta \Rightarrow \gamma)) \Rightarrow \gamma \\
\text{use Eq. (5.5)}: &\quad = \neg\left((\alpha \vee \beta) \wedge (\neg\alpha \vee \gamma) \wedge (\neg\beta \vee \gamma)\right) \vee \gamma \\
\text{de Morgan's laws}: &\quad = (\neg\alpha \wedge \neg\beta) \vee \underline{(\alpha \wedge \neg\gamma)} \vee \underline{(\beta \wedge \neg\gamma)} \vee \gamma \\
\text{identity } p \vee (\neg p \wedge q) = p \vee q: &\quad = \underline{(\neg\alpha \wedge \neg\beta) \vee \alpha} \vee \beta \vee \gamma \\
\text{identity } p \vee (\neg p \wedge q) = p \vee q: &\quad = \underline{\neg\alpha \vee \alpha} \vee \beta \vee \gamma \\
\text{axiom "use arg"}: &\quad = True \quad .
\end{aligned}$$

Since each proof rule of the constructive logic is translated into a true formula in Boolean logic, it follows that a proof tree in the constructive logic will be translated into a tree of Boolean formulas that have value $True$ for each axiom or proof rule. The result is that any constructive proof for a sequent such as $\emptyset \vdash f(\alpha, \beta, \gamma)$ is translated into a chain of Boolean implications that look like this,

$$True = (...) \Rightarrow (...) \Rightarrow ... \Rightarrow f(\alpha, \beta, \gamma) \quad .$$

Since $(True \Rightarrow \alpha) = \alpha$, this chain proves the Boolean formula $f(\alpha, \beta, \gamma)$.

For example, the proof tree shown in Figure 5.1 is translated into

$$\begin{aligned}
\text{axiom "use arg"}: &\quad True = (\neg\left((\alpha \Rightarrow \alpha) \Rightarrow \beta\right) \vee \neg\alpha \vee \alpha) \\
\text{rule "create function"}: &\quad \Rightarrow \neg\left((\alpha \Rightarrow \alpha) \Rightarrow \beta\right) \vee (\alpha \Rightarrow \alpha) \quad . \\
\text{axiom "use arg"}: &\quad True = \neg\left((\alpha \Rightarrow \alpha) \Rightarrow \beta\right) \vee ((\alpha \Rightarrow \alpha) \Rightarrow \beta) \quad . \\
\text{rule "use function"}: &\quad True \Rightarrow (\neg\left((\alpha \Rightarrow \alpha) \Rightarrow \beta\right) \vee \beta) \\
\text{rule "create function"}: &\quad \Rightarrow (((\alpha \Rightarrow \alpha) \Rightarrow \beta) \Rightarrow \beta) \quad .
\end{aligned}$$

It is easier to check Boolean truth than to find a proof tree in constructive logic (or to establish that no proof tree exists). So, if we find that a formula is not true in Boolean logic, we know it is also not true in constructive logic. This gives us a quick way of proving that some type signatures are not implementable as fully parametric functions. In addition to formulas shown in Table 5.3 (Section 5.2.1), further examples of formulas that are not true in Boolean logic are

$$\begin{aligned}
&\forall\alpha.\, \alpha \quad , \\
&\forall(\alpha, \beta).\, \alpha \Rightarrow \beta \quad , \\
&\forall(\alpha, \beta).\, (\alpha \Rightarrow \beta) \Rightarrow \beta \quad .
\end{aligned}$$

Table 5.7 uses the Boolean identity $(\alpha \Rightarrow \beta) = (\neg\alpha \vee \beta)$, which does not hold in the constructive logic, to translate the constructive axiom "use arg" into the Boolean axiom $\neg\alpha \vee \alpha = True$. The formula $\neg\alpha \vee \alpha = True$ is known as the **law of excluded middle**,[10] and is equivalent to saying that any proposition α is either true or false. It is remarkable that the constructive logic *does not have* the law of excluded middle; it is neither an axiom nor a derived theorem of constructive logic.

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

To see why, consider what it would mean for $\neg\alpha \vee \alpha = True$ to hold in the constructive logic. The negation operation, $\neg\alpha$, is defined as the implication $\alpha \Rightarrow False$. So, the logical formula $\forall\alpha.\, \neg\alpha \vee \alpha$ corresponds to the type $\forall A.\, (A \to \mathbb{0}) + A$. Can we compute a value of this type in a fully parametric function? We would need to compute either a value of type $A \to \mathbb{0}$ or a value of type A; this decision needs to be made in advance independently of A, because the code of a fully parametric function must operate in the same way for all types. Should we decide to return A or $A \to \mathbb{0}$? We certainly cannot compute a value of type A from scratch, since A is a type parameter. As we have seen in Example 5.3.4.3, a value of type $A \to \mathbb{0}$ exists if the type A is itself $\mathbb{0}$; but we do not know that, and a fully parametric function needs to have the same code for all types A. Since there are no values of type $\mathbb{0}$, and the type parameter A could be, say, `Int`, we cannot compute a value of type $A \to \mathbb{0}$.

Example 5.3.4.3 showed that the type $A \to \mathbb{0}$ is equivalent to $\mathbb{0}$ if A is not itself void ($A \ncong \mathbb{0}$), and to $\mathbb{1}$ otherwise. Surely, any type A is either void or not void. So, why exactly is it impossible to implement a value of the type $(A \to \mathbb{0}) + A$? We could say that if A is void then $(A \to \mathbb{0}) \cong \mathbb{1}$ is not void, and so one of the types in the disjunction $(A \to \mathbb{0}) + A$ should be non-void (i.e. have values).

However, this reasoning is incorrect. A fully parametric function's code must work in the same general way for all types (void or not); the code cannot decide what to do depending on a specific type. So, it is insufficient to show that a value "should exist"; the real requirement is to compute a value of type $(A \to \mathbb{0}) + A$ in fully parametric code. But, as we have seen, this is impossible.

In Boolean logic, it is sufficient to prove that a value "should exist" (or that the non-existence of a value is contradictory in some way). However, any practically useful program needs to "construct" (i.e. compute) actual values and return them. The "constructive" logic got its name from this requirement. So, it is the constructive logic (not the Boolean logic) that provides correct reasoning about the types of values computable by functional programs.

6 Functors, contrafunctors, and profunctors

Type constructors such as `Seq[A]` or `Array[A]` are data structures that hold or "wrap" zero or more values of a given type `A`. These data structures are fully parametric: they work in the same way for every type `A`. Working with parametric "data wrappers" or "data containers" turns out to be a powerful design pattern of functional programming. To fully realize its benefits, we will formalize the concept of "data wrapping" through a set of mathematical laws. We will then extend that design pattern to all data types for which the laws hold.

6.1 Practical use

6.1.1 Motivation: Type constructors that wrap data

How to formalize the idea of "wrapped data"? An intuitive view is that the data is "still there", i.e. we should be able to manipulate the data held within the wrapper. In functional programming, to "manipulate" means to apply functions to data. So, if an integer value 123 is "wrapped", we should be able somehow to apply a function such as `{x => x * 2}` and obtain a "wrapped" value 246.

Let us look at some often used type constructors defined in the Scala standard library, such as `Seq[A]`, `Try[A]`, and `Future[A]`. We notice the common features:

- There are some methods for creating a data structure that wraps zero or more values of a given type. For example, the Scala code `List.fill(10)(0)` creates a list of ten zeros of type `Int`.
- There are some methods for reading the wrapped values, if they exist. For example, the `List` class has the method `.headOption` that returns a non-empty `Option` when the first element exists.
- There are some methods for manipulating the wrapped values while *keeping* them wrapped. For example, `List(10, 20, 30).map(_ + 5)` evaluates to `List(15, 25, 35)`.

The data types `Seq[A]`, `Try[A]`, and `Future[A]` express quite different kinds of "wrapping". The data structure implementing `Seq[A]` can hold a variable number of values of type `A`. The data structure `Try[A]` holds either a successfully computed value of type `A` or a failure. The data structure `Future[A]` implements a computation that has been scheduled to run but may not have finished yet, and may compute a value of type `A` (or fail) at a later time.

Since the meaning of the "wrappers" `Seq`, `Try`, and `Future` is quite different, the methods for creating and reading the wrapped values have different type signatures for each "wrapper". However, the method `.map` is similar in all three examples. We can say generally that the `.map` method will apply a given function $f^{:A \to B}$ to the data of type A held inside the wrapper, and the new data (of type B) will remain within a wrapper of the same type:

```
val a = List(x,y,z).map(f) // Result is List(f(x), f(y), f(z)).
val b = Try(x).map(f)      // Result is Try(f(x)).
val c = Future(x).map(f)   // Result is Future(f(x)).
```

This motivates us to use the `map` function as the requirement for the "wrapping" functionality: A type constructor `Wrap[A]` is a "wrapper" if there exists a function `map` with the type signature

```
def map[A, B]: Wrap[A] => (A => B) => Wrap[B]
```

We can see that `Seq`, `Try`, and `Future` are "wrappers" because they have a suitable `.map` method. This chapter focuses on the properties of `.map` that are common to *all* "wrapper" types. We will ignore

all other features — reading data out of the wrapper, inserting or deleting data, waiting until data becomes available etc., — implemented by different methods specific to each wrapper type.

6.1.2 Example: `Option` and the identity law

As another example of a "data wrapper", consider the type constructor `Option[A]`, which is written in the type notation as

$$\text{Opt}^A \triangleq \mathbb{1} + A \quad .$$

The type signature of its `map` function is

$$\text{map}^{A,B} : \mathbb{1} + A \to (A \to B) \to \mathbb{1} + B \quad .$$

This function produces a new `Option[B]` value, possibly holding transformed data. We will now use this example to develop intuition about manipulating data in a wrapper.

Two possible implementations of `map` fit the type signature:

```
def mapX[A, B](oa: Option[A])(f: A => B): Option[B] = None

def mapY[A, B](oa: Option[A])(f: A => B): Option[B] =
  oa match {
    case None       => None
    case Some(x)    => Some(f(x))
  }
```

The code of `mapX` loses information since it always returns `None` and ignores all input. The implementation `mapY` is more useful since it preserves information.

How can we formulate this property of `map` in a rigorous way? The trick is to choose the argument $f^{:A\to B}$ in the expression `map(oa)(f)` to be the identity function $\text{id}^{:A\to A}$ (setting `map`'s type parameters as $A = B$, so that the types match). Applying an identity function to a value wrapped in an `Option[A]` should not change that value. To verify that, substitute the identity function instead of `f` into `mapY` and compute:

```
mapY[A, A](oa: Option[A])(identity[A]: A => A): Option[A]
  == oa match {
       case None       => None        // No change.
       case Some(x)    => Some(x)     // No change.
    } == oa
```

The result is always equal to `oa`. We can write that fact as an equation,

$$\forall x^{:A}. \, \text{map}(x)(\text{id}) = x \quad .$$

This equation is called the **identity law** of `map`. The identity law is a formal way of expressing the information-preserving property of the `map` function. The implementation `mapX` violates the identity law since it always returns `None` and so `mapX(oa)(id) == None` and not equal to `oa` for arbitrary values of `oa`. A data wrapper should not unexpectedly lose information when we manipulate the wrapped data. So, the correct implementation of `map` is `mapY`. The code notation for `map` is

$$\text{map}^{A,B} \triangleq p^{:\mathbb{1}+A} \to f^{:A\to B} \to p \triangleright \left|\begin{array}{c||cc} & \mathbb{1} & B \\ \hline\hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & f \end{array}\right| \quad .$$

When writing code, it is convenient to use is the `.map` method defined by the Scala library. However, when reasoning about the properties of `map`, it turns out to be more convenient to flip the order of the curried arguments and to use the equivalent function, called `fmap`, with the type signature

$$\text{fmap}^{A,B} : (A \to B) \to \mathbb{1} + A \to \mathbb{1} + B \quad .$$

The Scala implementation of `fmap` and the code notation are shorter than those of `map`:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None       => None
  case Some(x)    => Some(f(x))
}
```

$$\text{fmap}\,(f^{:A\to B}) \triangleq \left|\begin{array}{c||cc} & \mathbb{1} & B \\ \hline\hline \mathbb{1} & \text{id} & \mathbb{0} \\ A & \mathbb{0} & f \end{array}\right| \quad . \tag{6.1}$$

The identity law also looks simpler if expressed in terms of `fmap`, namely fmap (id) = id. In writing fmap (id) = id, we omitted the type parameters A and B, which must be both equal.

Note that the type signature of `fmap` looks like a transformation from functions of type `A => B` to functions of type `Option[A] => Option[B]`. This transformation is called **lifting** because it "lifts" a function $f^{:A\to B}$ operating on simple values into a function operating on `Option`-wrapped values.

So, the identity law can be formulated as "a lifted identity function is also an identity function". If we lift an identity function (or a composition of functions that equals an identity), we expect the wrapped data not to change. The identity law expresses this expectation in a mathematical equation.

6.1.3 Motivation for the composition law

The main feature of a "data wrapper" or "data container" is to allow us to manipulate the data inside it by applying functions to that data. The corresponding Scala code is `p.map(f)`, where `p` is a value of a "wrapper" type. It is natural to expect that lifted functions behave in the same way as the "unlifted" ones. For example, suppose we need to increment a counter `c` of type `Option[Int]`. The `Option` type means that the counter may be empty or non-empty; if it is non-empty, we increment the integer value wrapped inside the `Option` using the incrementing function

$$\text{incr} \triangleq x^{:\text{Int}} \to x + 1 \quad .$$

In order to apply a function to the counter `c`, we need to lift that function. The Scala code is

```
def incr: Int => Int = x => x + 1
val c: Option[Int] = Some(0)

scala> c.map(incr)
res0: Option[Int] = Some(1)
```

If we apply the lifted function twice, we expect that the counter will be incremented twice:

```
scala> c.map(incr).map(incr)
res1: Option[Int] = Some(2)
```

This result is the same as when applying a lifted function $x \to x + 2$:

```
scala> c.map(x => x + 2)
res2: Option[Int] = Some(2)
```

It would be confusing and counter-intuitive if `c.map(x => x + 2)` did not give the same result as `c.map(incr).map(incr)`.

We can formulate this property more generally: liftings should preserve function composition for arbitrary functions $f^{:A\to B}$ and $g^{:B\to C}$. This is written as

```
c.fmap(f).fmap(g) == c.fmap(f andThen g) == c.fmap(x => g(f(x)))
```

$$c^{:F^A} \triartriangleright \text{fmap}\,(f^{:A\to B}) \triartriangleright \text{fmap}\,(g^{:B\to C}) = c \triartriangleright \text{fmap}\,(f) \fatsemi \text{fmap}\,(g) = c \triartriangleright \text{fmap}\,(f^{:A\to B} \fatsemi g^{:B\to C}) \quad .$$

This equation is called the **composition law**. The law has the form $c \triartriangleright p = c \triartriangleright q$ with some functions p and q, which is the same as $p(c) = q(c)$ for all c, and is equivalent to an equality between functions, $p = q$. So we may omit the argument c and rewrite the law as

$$\text{fmap}\,(f) \fatsemi \text{fmap}\,(g) = \text{fmap}\,(f \fatsemi g) \quad .$$

Let us formally verify the composition law for the `Option` type. To practice the code derivations, we will perform the calculations by using both the code notation and the Scala syntax.

The Scala code for the function `fmap` was given in Section 6.1.2. To evaluate fmap $(f \fatsemi g)$, we apply `fmap(f andThen g)`, where `f: A => B` and `g: B => C` are arbitrary functions, to an arbitrary value `oa:Option[A]`. In Scala code, it is convenient to use the infix method `.map` and write `oa.map(f)` instead of the equivalent expression `fmap(f)(oa)`:

```
fmap(f andThen g)(oa) == oa.map(f andThen g) == oa match {
  case None      => None
  case Some(x)   => (f andThen g)(x)
}
```

Since `(f andThen g)(x) == g(f(x))`, we rewrite the result as

```
oa.map(f andThen g) == oa match {
  case None      => None
  case Some(x)   => g(f(x))
}
```

Now we consider the left-hand side of the law, $\text{fmap}\,(f) \fatsemi \text{fmap}\,(g)$, and write the Scala expressions:

```
oa.map(f).map(g) == (oa match {
  case None      => None
  case Some(x)   => f(x)
}).map(g) == (oa match {
  case None      => None
  case Some(x)   => f(x)
}) match {
  case None      => None
  case Some(y)   => g(y)
} == oa match {
  case None      => None
  case Some(x)   => g(f(x))
}
```

So we find that the two sides of the law have identical code.

The derivation is much shorter in the matrix notation; we use Eq. (6.1) as the definition of `fmap` and omit the types:

$$\text{fmap}\,(f) \fatsemi \text{fmap}\,(g) = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & f \end{array} \right\| \fatsemi \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & g \end{array} \right\|$$

$$\text{matrix composition :} \quad = \left\| \begin{array}{cc} \text{id} \fatsemi \text{id} & \mathbb{0} \\ \mathbb{0} & f \fatsemi g \end{array} \right\| = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & f \fatsemi g \end{array} \right\|$$

$$\text{definition of fmap :} \quad = \text{fmap}\,(f \fatsemi g) \quad .$$

These calculations prove that the `map` method of the `Option` type satisfies the composition law. If the composition law did not hold, we would not be able to understand how `map` manipulates data within the `Option` wrapper. Looking at the Scala code example above, we expect `c.map(incr).map(incr)` to increment the data wrapped by `c` two times. If the result of `c.map(incr).map(incr)` were not `Some(2)` but, say, `Some(1)` or `None`, our ordinary intuitions about data transformations would be incorrect. Violations of the composition law prevent us from understanding the code via mathematical reasoning about transformation of data values.

The composition law is a rigorous formulation of the requirement that wrapped data should be transformed (by lifted functions) in the same way as ordinary data. For example, the following associativity property holds for lifted functions:

Statement 6.1.3.1 For arbitrary functions $f^{:A\to B}$, $g^{:B\to C}$, and $h^{:C\to D}$, we have

$$\text{fmap}(f) \fatsemi \text{fmap}(g \fatsemi h) = \text{fmap}(f \fatsemi g) \fatsemi \text{fmap}(h) \quad .$$

Proof The left-hand side is rewritten as

$$\begin{aligned} & \text{fmap}(f) \fatsemi \underline{\text{fmap}(g \fatsemi h)} \\ \text{composition law for } (g \fatsemi h) : \quad & = \text{fmap}(f) \fatsemi \left(\text{fmap}(g) \fatsemi \text{fmap}(h) \right) \\ \text{associativity law (4.3)} : \quad & = \underline{\left(\text{fmap}(f) \fatsemi \text{fmap}(g) \right)} \fatsemi \text{fmap}(h) \\ \text{composition law for } (f \fatsemi g) : \quad & = \text{fmap}(f \fatsemi g) \fatsemi \text{fmap}(h) \quad , \end{aligned}$$

which now equals the right-hand side. This proves the statement.

6.1.4 Functors: definition and examples

Separating the functionality of "data wrapper" from any other features of a data type, we obtain:

- A data type with a type parameter, e.g. `L[A]`. We will use the notation $L^\bullet$ (in Scala, `L[_]`) for the type constructor itself, and also when the name of the type parameter is not needed.
- A fully parametric function `fmap` with type signature

$$\text{fmap}_L : (A \to B) \to L^A \to L^B \quad .$$

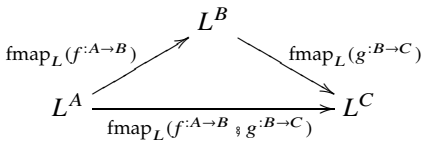
- The function `fmap` satisfies two laws:

$$\text{identity law for } L: \quad \text{fmap}_L(\text{id}^{:A\to A}) = \text{id}^{:L^A\to L^A} \quad , \tag{6.2}$$

$$\text{composition law for } L: \quad \text{fmap}_L(f^{:A\to B} \fatsemi g^{:B\to C}) = \text{fmap}_L(f^{:A\to B}) \fatsemi \text{fmap}_L(g^{:B\to C}) \quad . \tag{6.3}$$

A type constructor $L^\bullet$ with these properties is called a **functor**. The laws (6.2)–(6.3) are the functor laws of identity and composition.

When a law involves function compositions, it is helpful to draw a type diagram to clarify how the functions transform various types involved in the law. A **type diagram** is a directed graph whose vertices are types and edges are functions mapping one type to another. Function composition corresponds to following a path in the diagram. A type diagram for the composition law (6.3) is shown at left. There are two paths from L^A to L^C; by Eq. (6.3), both paths must give the same result. Mathematicians call such diagrams **commutative**.



Type diagrams are easier to read when using the *forward* composition ($f \fatsemi g$) because the order of edges is the same as the order of functions in the composition. To see this, compare Eq. (6.3) and the type diagram above with the same law written using the backward composition,

$$\text{fmap}_L(g^{:B\to C} \circ f^{:A\to B}) = \text{fmap}_L(g^{:B\to C}) \circ \text{fmap}_L(f^{:A\to B}) \quad .$$

The function `map` is computationally equivalent to `fmap` and can be defined through `fmap` by

$$\begin{aligned} &\text{map}_L : L^A \to (A \to B) \to L^B \quad , \\ &\text{map}_L(x^{:L^A})(f^{:A\to B}) = \text{fmap}_L(f^{:A\to B})(x^{:L^A}) \quad . \end{aligned}$$

Each of the type constructors `Option`, `Seq`, `Try`, and `Future` has its own definition of `map`; but the functor laws remain the same. We use the subscript L when writing map_L and fmap_L, in order to indicate clearly the type constructor those functions work with.

We will now look at some examples of type constructors that are functors.

Standard data structures Many type constructors defined in the Scala library have a `.map` method, and almost all of them are functors. The most often used functors are:

- The standard disjunctive types `Option`, `Try`, and `Either[A, B]` (where, by default, transformations apply to the type parameter `B`).
- The linear sequence `Seq` and its various derived classes such as `List`, `Range`, `Vector`, `IndexedSeq`, and `Stream`.
- The "task-like" constructors: `Future` and its alternatives: `Task` (provided by the `monix` library), `Async` and `Concurrent` (provided by the `cats-effect` library), `ZIO` (provided by the `zio` library).
- Dictionaries: `Map[K, V]` with respect to the type parameter `V`. The method is called `.mapValues` instead of `.map`: it transforms the values in the dictionary, leaving the keys unchanged.

Application-specific, custom type constructors defined by the programmer, such as case classes with type parameters, are often functors. Their structure is simple and helps build intuition for functors, so let us now consider some examples of case classes that are functors. In this book, they are called polynomial functors.

Polynomial functors Type constructors built with primitive types, type parameters, products, and disjunctions (or "sums") are often used to represent application-specific data. Consider the code

```
case class Counted[A](n: Int, a: A) {
  def map[B](f: A => B): Counted[B] = Counted(n, f(a))
}
```

The data type `Counted[A]` may be used to describe `n` repetitions of a given value `a:A`. The code already defines the infix method `.map` for the `Counted` class, which can be used like this,

```
scala> Counted(10, "abc").map(s => "prefix " + s)
res0: Counted[String] = Counted(10,prefix abc)
```

It is often more convenient to implement `.map` as an infix method rather than as a function such as

```
def map[A, B](c: Counted[A])(f: A => B): Counted[B] = c match {
  case Counted(n, a) => Counted(n, f(a))
}
```

The type notation for `Counted` is

$$\text{Counted}^A \triangleq \text{Int} \times A \quad ,$$

showing that `Counted[_]` is a polynomial type constructor. The existence of a `map` method suggests that `Counted[_]` is a functor. We still need to check that the functor laws hold for it.

Example 6.1.4.1 Verify that the above implementation of `map` for `Counted` satisfies the functor laws.

Solution The implementation of `map` is fully parametric since it does not perform any type-specific operations; it uses the value `n:Int` as if `Int` were a type parameter. It remains to check that the laws hold. We will first verify the laws using the Scala syntax and then using the code notation.

The identity law means that for all `n:Int` and `a:A` we must have

```
Counted(n, a).map(identity) == Counted(n, a)
```

To verify this, we substitute the code of `.map` and find

```
Counted(n, a).map(identity) == Counted(n, identity(a)) == Counted(n, a)
```

The composition law means that for all `n:Int`, `a:A`, `f: A => B`, and `g: B => C`, we must have

```
Counted(n, a).map(f).map(g) == Counted(n, a).map(f andThen g)
```

Substitute the Scala code of `.map` into the left-hand side:

```
Counted(n, a).map(f).map(g) == Counted(n, f(a)).map(g) == Counted(n, g(f(a)))
```

The right-hand side can be transformed to the same expression:

```
Counted(n, a).map(f andThen g) == Counted(n, (f andThen g)(a)) == Counted(n, g(f(a)))
```

Let us now write a proof in the code notation, formulating the laws via the `fmap` method:

$$\text{fmap}_{\text{Counted}}(f^{:A \to B}) \triangleq (n^{:\text{Int}} \times a^{:A} \to n \times f(a)) \quad .$$

To verify the identity law, we write

$$\begin{aligned} \text{expect to equal id}: \quad & \text{fmap}_{\text{Counted}}(\text{id}) \\ \text{definition of fmap}_{\text{Counted}}: \quad & = (n \times a \to n \times \underline{\text{id}(a)}) \\ \text{definition of id}: \quad & = (n \times a \to n \times a) = \text{id} \quad . \end{aligned}$$

To verify the composition law,

$$\begin{aligned} \text{expect to equal fmap}_{\text{Counted}}(f \fatsemi g): \quad & \text{fmap}_{\text{Counted}}(f) \fatsemi \text{fmap}_{\text{Counted}}(g) \\ \text{definition of fmap}_{\text{Counted}}: \quad & = (n \times a \to n \times f(a)) \fatsemi (n \times b \to n \times g(b)) \\ \text{compute composition}: \quad & = n \times a \to n \times \underline{g(f(a))} \\ \text{definition of } (f \fatsemi g): \quad & = (n \times a \to n \times (f \fatsemi g)(a)) = \text{fmap}_{\text{Counted}}(f \fatsemi g) \quad . \end{aligned}$$

We will prove later that all polynomial type constructors have a definition of `map` that satisfies the functor laws. It will be clear without proof that our definition of `map` for `Counted` is correct.

What would be an *incorrect* implementation of `map`? As an example, `map` could transform `Counted(n, a)` as before, except that the value `n` is now used to count the number of times `map` is applied:

```
def map_bad[A, B](c: Counted[A])(f: A => B): Counted[B] = c match {
  case Counted(n, a) => Counted(n + 1, f(a))
}
```

This implementation may appear reasonable. However, it violates both functor laws; for instance,

```
Counter(n, a) != map_bad(Counter(n, a))(identity) == Counter(n + 1, a)
```

The failure of functor laws leads to surprising behavior because a code refactoring changes the result:

```
Counter(n, a).map(incr).map(incr) != Counter(n, a).map(x => x + 2)
```

Let us look at some other simple examples of polynomial type constructors.

Example 6.1.4.2 Implement the `fmap` function for the type constructor

```
case class Vec3[A](x: A, y: A, z: A)
```

Solution Begin by implementing a fully parametric function:

```
def fmap[A, B](f: A => B): Vec3[A] => Vec3[B] = {
  case Vec3(x, y, z) => Vec3(f(x), f(y), f(z))  // Apply ‘f‘ to all data of type ‘A‘.
}
```

Since all three values `f(x)`, `f(y)`, `f(z)` have type `B`, the code of `fmap` would still satisfy the required type signature by returning, say, `Vec3(f(z), f(x), f(x))` or some other combination of these values. However, that implementation does not preserve information about the values `x`, `y`, `z` and about the ordering of these values in the original data `Vec(x, y, z)`. For this reason, we use the implementation of `fmap` shown first.

The type notation for the type constructor `Vec3[_]` is

$$\text{Vec}_3{}^A \triangleq A \times A \times A \quad ,$$

and the code notation for `fmap` is

$$\text{fmap}_{\text{Vec}_3}(f^{:A \to B}) \triangleq x^{:A} \times y^{:A} \times z^{:A} \to f(x) \times f(y) \times f(z) \quad .$$

Example 6.1.4.3 Implement the `fmap` function for the type constructor

$$\text{QueryResult}^A \triangleq \text{String} + \text{String} \times \text{Long} \times A \quad .$$

Solution Begin by implementing the type constructor in Scala,

```
sealed trait QueryResult[A]
case class Error[A](message: String)                    extends QueryResult[A]
case class Success[A](name: String, time: Long, data: A) extends QueryResult[A]
```

Now implement a fully parametric, information-preserving function with the type signature of `fmap` for this type constructor:

```
def fmap[A, B](f: A => B): QueryResult[A] => QueryResult[B] = {
  case Error(message)              => Error(message)
  case Success(name, time, data)   => Success(name, time, f(data))
}
```

As in the previous example, we treat specific types (`Long`, `String`) as if they were type parameters. In this way, we obtain a correct implementation of `fmap` that satisfies the functor laws.

Recursive polynomial functors Recursive disjunctive type constructors shown in Section 3.3, such as lists and trees, are functors. Their `fmap` methods are recursive functions; they usually *cannot* be directly implemented with tail recursion.

Example Define a list of *odd* length as a recursive type $\text{LO}^\bullet$,

$$\begin{aligned} \text{LO}^A &\triangleq A + A \times A \times \text{LO}^A \\ &\cong A + A \times A \times A + A \times A \times A \times A \times A + \ldots \end{aligned} \tag{6.4}$$

and implement `fmap` for it.

Solution The Scala definition of the type constructor `LO[_]` is

```
sealed trait LO[A]
final case class LO1[A](x: A)                    extends LO[A]
final case class LO2[A](x: A, y: A, tail: LO[A]) extends LO[A]
```

We can implement `fmap` as a recursive function:

```
def fmap[A, B](f: A => B): LO[A] => LO[B] = {
  case LO1(x)            => LO1[B](f(x))
  case LO2(x, y, tail)   => LO2[B](f(x), f(y), fmap(f)(tail))
}
```

This code for `fmap` is not tail-recursive because `fmap` is called inside the case class constructor `LO2`.

The type constructor $\text{LO}^\bullet$ is a **recursive polynomial functor** because it is defined by a recursive type equation (6.4) that uses only polynomial type operations (sums and products) in its right-hand side. For the same reason, lists and trees are recursive polynomial functors.

6.1.5 Functor block expressions

Computations with wrapped values often require a chain of `.map` methods, e.g.

```
scala> val result = Map(1 -> "one", 2 -> "two", 3 -> "three").
  map { case (i, name) => (i * i, name) }.          // Compute i * i.
  map { case (x, name) => (x, s"$name * $name") }. // Compute product message.
  map { case (x, product) => s"$product is $x" }   // Compute final message.
result: Seq[String] = List(one * one is 1, two * two is 4, three * three is 9)
```

Such code can be rewritten equivalently in the **functor block** syntax:

```
val result = for {
  (i, name) <- Map(1 -> "one", 2 -> "two", 3 -> "three") // For each (i, name)...
  x = i * i                    // define ‘x‘ by computing i * i...
  product = s"$name * $name"   // define ‘product‘...
} yield s"$product is $x"      // and put these expressions into the ‘result‘ sequence.
result: Seq[String] = List(one * one is 1, two * two is 4, three * three is 9)
```

Written in this way, the computations are easier to understand for two main reasons:

- There is less code to read and to write; no `.map` or `case` and fewer curly braces.
- Values such as `name` and `x` need to be kept in tuples and passed from one `.map` function to another, but any line in a functor block can directly reuse all values defined in previous lines.

The functor block is an important idiom in functional programming because it replaces a chain of `.map` methods (as well as `.filter` and `.flatMap` methods, as we will see in later chapters) by a visually clearer sequence of definitions and expressions. Scala defines a functor block via the keywords `for` and `yield`. We will see many examples of functor blocks throughout this book. In this chapter, we only consider functor blocks that are equivalent to a chain of `.map` operations on a functor value `p:L[A]`. These functor blocks can be recognized because they contain *only one* left arrow (in the first line). Here is how to replace a chain of `.map` operations by a functor block:

```
p.map(x => f(x)).map(y => g(y)).map(z => h(z)) == for {
   x <- p           // The first line must contain a left arrow before a functor value ‘p‘.
   y = f(x)         // Some computation involving ‘x‘.
   z = g(y)         // Another computation, uses ‘y‘.
} yield h(z)        // The ‘yield‘ may use ‘x‘, ‘y‘, ‘z‘, and any other defined variables.
```

Translating functor blocks back into a chain of `.map` operations is straightforward except for one complication: if later lines in the functor block make use of variables defined in previous lines, the `.map` operations may need to create some intermediate tuples that are not present in the functor block syntax. Consider the code

```
val result: L[B] = for {
  x <- p           // The first line must contain a left arrow before a functor value ‘p‘.
  y = f(x)         // Some computation involving ‘x‘.
  z = g(x, y)      // Another computation, uses ‘x‘ and ‘y‘.
  ...
} yield q(x, y, z) // The ‘yield‘ may use ‘x‘, ‘y‘, ‘z‘, and any other defined variables.
```

The above functor block code assumes that `q(x, y, z)` has type `B`, and is equivalent to

```
val result: L[B] = p
  .map { x => (x, f(x)) } // Create a tuple because we need to keep ‘x‘.
  .map { case (x, y) => (x, y, g(x, y)) }   // Need to keep ‘x‘ and ‘y‘.
  ...
  .map { case (x, y, z) => q(x, y, z) }   // Need to keep ‘x‘, ‘y‘, ‘z‘.
```

This code creates intermediate tuples only because the values `x`, `y`, `z` need to be used in later calculations. The functor block code is easier to read, write, and modify.

If desired, functor blocks may be written in a single line, by using semicolons to separate the individual steps:

```
scala> for { x <- List(1, 2, 3); y = x * x; z = y + 2 } yield z
res0: List[Int] = List(3, 6, 11)
```

A confusing feature of the `for` / `yield` syntax is that, at first sight, functor blocks (such as the code shown at left) appear to compute (or to “yield”) the value `expr(x)`. However, this is not so. As the above examples show, if `p` is a sequence then the functor block also computes a *sequence*. In general, the result of a functor block is a “wrapped” value, where the type of the “wrapper” is determined by the first line of the functor block. The first line must have a left arrow followed by a “source”, which must be an expression of a functor type, i.e. of type `L[A]` for some functor `L[_]`. The result's type will be `L[B]` where `B` is the type of the expression after the `yield` keyword.

```
for { x <- p; ... } yield expr(x)
```

For instance, the first line of the following functor block contains an `Option` value, `Some(123)`, as the “source”. Because of that, the value of the entire functor block expression will also be of type `Option`:

```
scala> for {
  x <- Some(123)   // "Source" is Option[Int].
  y = (x - 3) / 10
} yield { if (y > 0) s"Have $y" else "Error" }
res1: Option[String] = Some(Have 12)
```

In this code, the `yield` keyword is followed by an expression of type `String`. So, the result of the entire functor block is of type `Option[String]`. Note that the expression after the “`yield`” can be a block of arbitrary code containing new `val`s, new `def`s, and/or other `for` / `yield` functor blocks if needed.

Functor blocks can be used with any functor that has an infix `.map` method, not only with library-defined type constructors such as `Seq` or `Option`. Here are some examples of defining the `.map` methods and using functor blocks with disjunctive types.

The type constructor `QueryResult[_]` may define the `.map` method on the trait itself and split its implementation between the case classes like this:

```
sealed trait QueryResult[A] {
  def map[B](f: A => B): QueryResult[B] // No implementation here.
}
case class Error[A](message: String)                    extends QueryResult[A] {
  def map[B](f: A => B): QueryResult[B] = Error(message)
}
case class Success[A](name: String, time: Long, data: A) extends QueryResult[A] {
  def map[B](f: A => B): QueryResult[B] = Success(name, time, f(data))
}
```

After these definitions, we can use `QueryResult` in functor blocks:

```
val q: QueryResult[Int] = Success("addresses", 123456L, 10)
scala> val result = for {
  x <- q
  y = x + 2
} yield s"$y addresses instead of $x"
result: QueryResult[String] = Success(addresses,123456,12 addresses instead of 10)
```

As another example, let us define the `.map` method on the `LO` trait (a recursive disjunctive type):

```
sealed trait LO[A] {
  def map[B](f: A => B): LO[B]
}
final case class LO1[A](x: A)                    extends LO[A] {
  def map[B](f: A => B): LO[B] = LO1[B](f(x))
}
final case class LO2[A](x: A, y: A, tail: LO[A]) extends LO[A] {
  def map[B](f: A => B): LO[B] = LO2[B](f(x), f(y), tail.map(f))
}
```

After these definitions, we may use values of type `LO[_]` in functor blocks:

```
scala> val result = for {
         x <- LO2("a", "quick", LO2("brown", "fox", LO1("jumped")))
         y = x.capitalize
         z = y + "/"
       } yield (z, z.length)
result: LO[(String, Int)] = LO2((A/,2),(Quick/,6),LO2((Brown/,6),(Fox/,4),LO1((Jumped/,7))))
```

Functor blocks and functor laws There is an important connection between the functor laws and the properties of code in functor blocks. Consider the following code,

```
def f(x: Int) = x * x    // Some computations.
def g(x: Int) = x - 1    // More computations.

scala> for {
  y <- List(10, 20, 30)
  x = y
  z = f(x)   // Perform computations.
} yield g(z)
res0: List[Int] = List(99, 399, 899)
```

The code says that `x = y`, so it appears reasonable to eliminate `y` and simplify this code into

```
scala> for {
  x <- List(10, 20, 30)    // Eliminated ‘y‘ from the code.
  z = f(x)   // Perform computations.
} yield g(z)
res1: List[Int] = List(99, 399, 899)
```

Another example of refactoring that appears reasonable is to combine transformations:

```
scala> for {
  x <- List(10, 20, 30)
  y = x + 1
  z = f(y)   // Perform computations.
} yield g(z)
res2: List[Int] = List(120, 440, 960)
```

The code says that `y = x + 1`, so we may want to replace `f(y)` by `f(x + 1)`:

```
scala> for {
  x <- List(10, 20, 30)
  z = f(x + 1)   // Eliminated ‘y‘ from the code.
} yield g(z)
res3: List[Int] = List(120, 440, 960)
```

Looking at these code changes, we expect that the computed results will remain the same. Indeed,

Functor block syntax	Chains of `.map` methods

`for {  // Code fragment 1a.`
`  y <- List(10, 20, 30)`
`  x = y`
`  z = f(x)`
`} yield g(z)`
	`List(10, 20, 30)  // Code fragment 1b.`
	`  .map(y =>`
	`  y).map(x =>`
	`  f(x) ).map(z =>`
	`  g(z) )`

`for {  // Code fragment 2a.`
`  x <- List(10, 20, 30)`
`  z = f(x)`
`} yield g(z)`
	`List(10, 20, 30)  // Code fragment 2b.`
	`  .map(x =>`
	`  f(x) ).map(z =>`
	`  g(z) )`

`for {  // Code fragment 3a.`
`  x <- List(10, 20, 30)`
`  y = x + 1`
`  z = f(y)`
`} yield g(z)`
	`List(10, 20, 30)  // Code fragment 3b.`
	`  .map(x =>`
	`  x + 1).map(y =>`
	`  f(y) ).map(z =>`
	`  g(z) )`

`for {  // Code fragment 4a.`
`  x <- List(10, 20, 30)`
`  z = f(x + 1)`
`} yield g(z)`
	`List(10, 20, 30)  // Code fragment 4b.`
	`  .map(x =>`
	`  f(x + 1) ).map(z =>`
	`  g(z) )`

Table 6.1: Example translations of functor blocks into `.map` methods.

when the code directly states that `x = y`, it would be confusing and counter-intuitive if the result value changed after replacing `y` by `x`. When the code says that `y = x + 1`, ordinary mathematical reasoning suggests that `f(y)` can be replaced by `f(x + 1)` without affecting the results.

To see the connection with the functor laws, we translate the functor block syntax line by line into chains of `.map` methods. The resulting code fragments are summarized in Table 6.1.

We find that code fragments `1b` and `2b` are equal if `.map(y => y)` does not modify the list to which it applies. This holds if the `.map` method obeys the functor identity law, `p.map(identity) == p`, for all `p` of the appropriate type. We also find that code fragments `3b` and `4b` are equal if we can replace `.map(x => x + 1).map(f)` by `.map(x => f(x + 1))`. This replacement is justified as long as the `.map` method obeys the functor composition law,

```
p.map(h).map(f) == p.map(x => f(h(x)))
```

for all `p` and functions `h` and `f` of appropriate types.

Functor laws guarantee that we can correctly understand and modify code written in functor blocks, reasoning about transformations of values as we do in mathematics.

6.1.6 Examples of non-functors

What properties of a data type make it a functor? To build an intuition, it is helpful to see examples of data types that are *not* functors.

There are several possibilities for a type constructor to fail being a functor:

- A `map` function's type signature cannot be implemented at all.
- A `map` function can be implemented but cannot satisfy the functor laws.
- A given `map` function is incorrect (does not satisfy the laws), although the error could be fixed: a different implementation of `map` satisfies the laws.
- A given `map[A, B]` function satisfies the laws for most types `A` and `B`, but violates the laws for certain specially chosen types.

We will now look at examples illustrating these possibilities.

Cannot implement `map`'s type signature Consider the type constructor $C^\bullet$ defined by

$$C^A \triangleq A \to \text{Int} \quad .$$

Scala code for this type notation can be

```
final case class C[A](r: A => Int)
```

The data type `C[A]` does not wrap data of type A; instead, it is a function that *consumes* data of type A. One cannot implement a fully parametric `map` function with the required type signature

$$\text{map}^{A,B} : (A \to \text{Int}) \to (A \to B) \to (B \to \text{Int}) \quad .$$

To see this, recall that a fully parametric function needs to treat all types as type parameters, including the primitive type `Int`. So the code

```
def map[A, B]: C[A] => (A => B) => C[B] = { r => f => C(_ => 123) }
```

satisfies the type signature of `map` but is not fully parametric because it returns a specific value `123` of type `Int`, which is not allowed. Replacing the type `Int` by a new type parameter N, we obtain the type signature

$$\text{map}^{A,B,N} : (A \to N) \to (A \to B) \to B \to N \quad .$$

We have seen in Example 5.4.1.8 that this type signature is not implementable. So, the type constructor C is not a functor.

Another important example of type constructors where the `map`'s type signature cannot be implemented are certain kinds of type constructors called **generalized algebraic data types** (GADTs). In this book, they are called **unfunctors** for short. An unfunctor is a type constructor having special values when its type parameter is set to certain specific types. An example of an unfunctor is

```
sealed trait ServerAction[R]
final case class GetResult[R](r: String => R) extends ServerAction[R]
final case class StoreId(x: Long, y: String)  extends ServerAction[Boolean]
final case class StoreName(name: String)      extends ServerAction[Int]
```

We see that some parts of the disjunctive type `ServerAction[R]` do not carry the type parameter `R` but instead set `R` to specific types, `R = Boolean` and `R = Int`. As a consequence, e.g. the case class `StoreName` has no type parameters and can only represent values of type `ServerAction[Int]` but not, say, `ServerAction[String]`. As a result, `ServerAction[A]` cannot have a fully parametric `map` function,

```
def map[A, B]: ServerAction[A] => (A => B) => ServerAction[B]
```

To implement `map`, we are required to support any choice of the type parameters `A` and `B`. For example, with `A = Unit`, we must be able to transform the value `StoreName("abc")` of type `ServerAction[Int]` to a value of type `ServerAction[B]` with any given `B`. However, the only way of creating a value of type `ServerAction[B]` with an arbitrary type `B` is to use the case class `GetResult[B]`. That requires us to create a function of type `String => B`. It is impossible for us to produce such a function out of `StoreName("abc")` and a function `f: Int => B` because the type `B` is unknown and no values of type `Int` can be computed from `StoreName("abc")` by fully parametric code.

We are prevented from implementing `map` because some type parameters are already set in the definition of `ServerAction[R]`. One can say that the unfunctor `ServerAction[_]` fails to be fully parametric *in its type definition*. This behavior of unfunctors is intentional; unfunctors are only used in situations where the lack of `map` does not lead to problems (see Chapter 12).

Cannot implement a lawful `map` An example of a non-functor of the second kind is

$$Q^A \triangleq (A \to \text{Int}) \times A \quad .$$

Scala code for this type constructor is

```
final case class Q[A](q: A => Int, a: A)
```

A fully parametric `map` function with the correct type signature *can* be implemented (and there is only one such implementation):

$$\text{map}^{A,B} \triangleq q^{:A\to\text{Int}} \times a^{:A} \to f^{:A\to B} \to (\_ \to q(a))^{:B\to\text{Int}} \times f(a) \quad .$$

The corresponding Scala code is

```
def map[A, B]: Q[A] => (A => B) => Q[B] = { qa => f =>
  Q[B](_ => qa.q(qa.a), f(qa.a))
}
```

This `map` function is fully parametric (since it treats the type `Int` as a type parameter) and has the right type signature, but the functor laws do not hold. To show that the identity law fails, we consider an arbitrary value $q^{:A\to\text{Int}} \times a^{:A}$ and compute:

$$\begin{aligned}
\text{expect to equal } q \times a: \quad & \text{map}(q \times a)(\text{id}) \\
\text{definition of map}: \quad & = (_ \to q(a)) \times \underline{\text{id}(a)} \\
\text{definition of id}: \quad & = (_ \to q(a)) \times a \\
\text{expanded function, } q = (x \to q(x)): \quad & \neq q \times a = (x \to q(x)) \times a \quad .
\end{aligned}$$

The law must hold for arbitrary functions $q^{:A\to\text{Int}}$, but the function $(\_ \to q(a))$ always returns the same value $q(a)$ and thus is not equal to the original function q. So, the result of evaluating the expression $\text{map}(q \times a)(\text{id})$ is not always equal to the original value $q \times a$.

Since this `map` function is the only available implementation of the required type signature, we conclude that $Q^\bullet$ is not a functor (we cannot implement `map` that satisfies the laws).

Mistakes in implementing `map` Non-functors of the third kind are type constructors with an incorrectly implemented `map`. An example is a type constructor $P^A \triangleq A \times A$ with the `map` function

$$\text{map} \triangleq x^{:A} \times y^{:A} \to f^{:A\to B} \to f(y) \times f(x) \quad .$$

Here is the Scala code corresponding to this code notation:

```
def map[A, B](p: (A, A))(f: A => B): (B, B) = p match { case (x, y) => (f(y), f(x)) }
```

This code swaps the values in the pair `(x, y)`; we could say that it fails to preserve information about the order of those values. The functor identity law does not hold:

$$\begin{aligned}
\text{expect to equal } x \times y: \quad & \text{map}(x^{:A} \times y^{:A})(\text{id}^A) \\
\text{definition of map}: \quad & = \underline{\text{id}(y)} \times \underline{\text{id}(x)} \\
\text{definition of id}: \quad & = y \times x \neq x \times y \quad .
\end{aligned}$$

We should not have swapped the values in the pair. The correct implementation of `map`,

$$\text{map} \triangleq x^{:A} \times y^{:A} \to f^{:A\to B} \to f(x) \times f(y) \quad ,$$

preserves information and satisfies the functor laws.

Example 6.1.4.2 shows the type constructor $\text{Vec}_3^\bullet$ with an incorrect implementation of `map` that reorders some parts of a tuple and duplicates other parts. The correct implementation preserves the order of parts in a tuple and does not duplicate or omit any parts.

Another case of an an incorrect implementation is the following `map` function for `Option[_]`:

```
def map_bad[A, B]: Option[A] => (A => B) => Option[B] = { _ => _ => None }
```

This function always returns `None`, losing information and violating the identity law. However, we have already seen that `Option[_]` has a different implementation of `map` that satisfies the functor laws.

Similarly, one could define `map` for the `List[_]` type constructor to always return an empty list:

```
def map_bad[A, B]: List[A] => (A => B) => List[B] = { _ => _ => List() }
```

This implementation loses information and violates the functor laws. Of course, the Scala library provides a correct implementation of `map` for `List[_]`.

Example 6.1.4.1 is another situation where an incorrectly implemented `map` violates functor laws.

Functor laws will also be violated when `map` is not fully parametric. For instance, consider an implementation of `fmap[A, B](f)` that checks whether the two type parameters A and B are equal to each other *as types*, and if so, applies the function argument `f` twice. We need to use special features of Scala (run-time type reflection and `TypeTag`) for comparing two type parameters as types:

```
import scala.reflect.runtime.universe._
def getType[T: TypeTag]: Type = weakTypeOf[T]
def equalTypes[A: TypeTag, B: TypeTag]: Boolean = getType[A] =:= getType[B]

def fmap_bad[A: TypeTag, B: TypeTag](f: A => B)(oa: Option[A]): Option[B] = oa match {
  case None      =>   None
  case Some(x)   =>                    // If A = B, compute f(f(x)), else compute f(x).
    val z: B = if (equalTypes[A, B]) f(f(x).asInstanceOf[A]) else f(x)
    Some(z)
}
```

Testing shows that this function works as designed:

```
scala> fmap_bad[Int, String](_ + " a")(Some(123))       // Appends " a" once.
res0: Option[String] = Some(123 a)

scala> fmap_bad[String, String](_ + " a")(Some("123"))  // Appends " a" twice.
res1: Option[String] = Some(123 a a)
```

The function `fmap_bad[A, B]` satisfies the identity law but violates the composition law when `A = B`:

```
scala> fmap_bad[String, String](_ + " b")(Some("123 a a"))
res2: Option[String] = Some(123 a a b b)

scala> fmap_bad[String, String](_ + " a b")(Some("123"))
res3: Option[String] = Some(123 a b a b)
```

In all these examples, we *could* implement a different `map` function that satisfies the functor laws. It is not precise to say that e.g. the type constructor `Vec3[_]` is *by itself* a functor: being a functor depends on having a lawful `map` function. Keeping that in mind, we will say that e.g. the type constructors `Option[_]` and `List[_]` "are" functors, meaning that a suitable lawful implementation of `map` is known.

Laws hold for some types but not for others The Scala standard library contains `map` methods for the type constructors `Set` (transforming the values in a set) and `Map` (transforming both the keys and values in a dictionary). However, `Set[K]` and `Map[K, V]` fail to be lawful functors with respect to the type parameter `K`. The reason for this failure is complicated. A value of type `Set[K]` represents a set of zero or more values of type `K`, and it is enforced that all values in the set are distinct. So, the correct functionality of `Set` implies that we are able to check whether two values of type `K` are equal. A standard way of comparing values for equality is the `.equals` method defined in the Scala library:

```
scala>  List(1, 2, 3).equals(List(1, 2, 3))
res0: Boolean = true

scala>  List(1, 2, 3).equals(List(1, 2, 3, 4))
res1: Boolean = false
```

However, code using the `.equals` operation will work as expected only if that operation obeys the laws of **identity** (if $x = y$ then $f(x) = f(y)$ for any f), **reflexivity** ($x = x$ for any x), and **transitivity** (if $x = y$ and $y = z$ then $x = z$). In most practical applications, the required type `K` (such as `String` or `Int`) has a mathematically correct `.equals` method. In some cases, however, data types will redefine their `.equals` method for application-specific purposes and violate some of the required laws.

We will give two non-trivial examples of types that violate the laws of equality. The first example[1] is the type $A + B$ whose `.equals` method always returns `false` for any values of type $\mathbb{0} + B$. The code is

[1] This example is based on a comment by Paweł Szulc at https://gist.github.com/tpolecat/7401433

```
final case class Weird[A, B](x: Either[A, B]) {
  override def equals(y: Any): Boolean = (x, y) match {
    case (Left(a1), Weird(Left(a2)))   => a1 == a2
    case _                             => false    // Only equal if both ‘Left‘.
  }
}
```

This implementation of `.equals` is mathematically invalid: it violates the reflexivity law ($\forall x.\, x = x$) because values of the form `Weird(Right(...))` are never equal to each other:

```
scala> Weird(Right(0)) equals Weird(Right(0))
res2: Boolean = false
```

As a result, the standard code of `Set[Weird]` will fail to detect that e.g. several values `Weird(Right(0))` are equal. The composition law of functors will fail when intermediate values of that type are used:

```
val f: Weird[Int, Int] => Int = { case Weird(Left(a)) => a; case Weird(Right(a)) => a }
val g: Int => Weird[Int, Int] = { a => Weird(Right(a)) }
val xs = Seq(0, 0, 0).map(g).toSet

scala> xs.map(f andThen g)  // ‘Set‘ fails to detect identical values.
res3: Set[Weird[Int,Int]] = Set(Weird(Right(0)), Weird(Right(0)), Weird(Right(0)))

scala> xs.map(f).map(g)     // ‘Set‘ detects identical values.
res4: Set[Weird[Int,Int]] = Set(Weird(Right(0)))
```

The second example is the pair type $A \times B$ whose `.equals` method ignores the part of type B:

```
final case class Ignore2[A, B](a: A, b: B) {
  override def equals(y: Any): Boolean = y match {
    case Ignore2(p, q)   => a == p
    case _                 => false
  }
}

scala> Ignore2(123, "abc") == IgnoreB(123, "def")
res5: Boolean = true
```

As a result, the code of `Set[Ignore2]` will fail to detect that some values are different. This also violates the functor composition law:

```
val f: Ignore2[Int, Int] => Ignore2[Int, Int] = { case Ignore2(x, y) => Ignore2(y, x) }  //  f ⨾ f = id
val xs = Set(Ignore2(0, 0), Ignore2(1, 0))

scala> xs.map(f andThen f)    // This is equal to ‘xs‘.
res6: Set[Ignore2[Int,Int]] = Set(Ignore2(0,0), Ignore2(1,0))

scala> xs.map(f).map(f)       // This is not equal to ‘xs‘.
res7: Set[Ignore2[Int,Int]] = Set(Ignore2(0,0))
```

The functor laws for a type constructor $L^\bullet$ do not assume that the types A, B used in the function

$$\text{fmap}_L : (A \to B) \to L^A \to L^B$$

should have a mathematically valid definition of the `.equals` method (or of any other operation). The `map` operation of a functor $L^\bullet$ must be **lawful**, i.e. must satisfy the functor laws (6.2)–(6.3) for all types A, B. The functor laws must hold even if a type A's implementations of certain operations violate some other laws. For this reason, `Set[_]` cannot be considered a functor in a full sense.

The `map` method for dictionaries is similar: the "keys" of a dictionary must be distinct and will be compared using the `.equals` method. In other words, a dictionary `Map[K, V]` behaves as a set with respect to the type `K` of the dictionary's "keys" and thus is not a functor with respect to `K`.

The Scala library still provides the `.map` and `.flatMap` methods for sets `Set[K]` and dictionaries `Map[K, V]` because in most applications the type `K` will have a correctly defined `.equals` operation, and so it is likely that we will not see any violations of the functor laws.

6.1.7 Contrafunctors

As we have seen in Section 6.1.6, the type constructor $C^\bullet$ defined by $C^A \triangleq A \to \text{Int}$ is not a functor because it is impossible to implement the type signature of `map` for it,

$$\text{map}^{A,B} : (A \to \text{Int}) \to (A \to B) \to B \to \text{Int} \quad .$$

To see why, begin writing the code with a typed hole,

$$\text{map}(c^{:A\to\text{Int}})(f^{:A\to B})(b^{:B}) = ???^{:\text{Int}} \quad .$$

Since $c^{:A\to\text{Int}}$ consumes (rather than wraps) values of type A, we have no values of type A and cannot apply the function $c^{:A\to\text{Int}}$. However, it would be possible to apply a function of type $B \to A$ since a value of type B is given as one of the curried arguments, $b^{:B}$. So, we can implement a function called `contramap` with a different type signature where the function type is $B \to A$ instead of $A \to B$:

$$\text{contramap}^{A,B} : (A \to \text{Int}) \to (B \to A) \to B \to \text{Int} \quad .$$

The implementation of this function is written in the code notation as

$$\text{contramap} \triangleq c^{:A\to\text{Int}} \to f^{:B\to A} \to (f \fatsemi c)^{:B\to\text{Int}} \quad ,$$

and the corresponding Scala code is

```
def contramap[A, B](ca: C[A])(f: B => A): C[B] = { f andThen ca }
```

Flipping the order of the curried arguments in `contramap`, we define `cmap` as

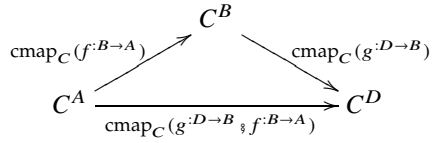
$$\begin{aligned} \text{cmap}^{A,B} &: (B \to A) \to C^A \to C^B \quad , \\ \text{cmap} &\triangleq f^{:B\to A} \to c^{:A\to\text{Int}} \to (f \fatsemi c)^{:B\to\text{Int}} \quad . \end{aligned} \tag{6.5}$$

The type signature of `cmap` has the form of a "reverse lifting": functions of type `B => A` are lifted into the type `C[A] => C[B]`. The Scala code for `cmap` is

```
def cmap[A, B](f: B => A): C[A] => C[B] = { ca => f andThen ca }
```

We can check that this `cmap` satisfies two laws analogous to the functor laws:

$$\begin{aligned} \text{identity law :} \quad & \text{cmap}^{A,A}(\text{id}^{:A\to A}) = \text{id}^{:C^A\to C^A} \quad , \\ \text{composition law :} \quad & \text{cmap}^{A,B}(f^{:B\to A}) \fatsemi \text{cmap}^{B,D}(g^{:D\to B}) = \text{cmap}(g \fatsemi f) \quad . \end{aligned}$$



Since the function argument $f^{:B\to A}$ has the reverse order of types, the composition law reverses the order of composition $(g \fatsemi f)$ on one side; in this way, all types match. To verify the identity law:

$$\begin{aligned} \text{expect to equal id :} \quad & \text{cmap}\,(\text{id}) \\ \text{use Eq. (6.5) :} \quad & = c \to \underline{(\text{id} \fatsemi c)} \\ \text{definition of id :} \quad & = (c \to c) = \text{id} \quad . \end{aligned}$$

To verify the composition law:

$$\begin{aligned} \text{expect to equal cmap } (g \fatsemi f) \text{ :} \quad & \text{cmap}\,(f) \fatsemi \text{cmap}\,(g) \\ \text{use Eq. (6.5) :} \quad & = (c \to (f \fatsemi c)) \fatsemi (\underline{c} \to (g \fatsemi \underline{c})) \\ \text{rename } c \text{ to } d \text{ for clarity :} \quad & = (c \to (f \fatsemi c)) \fatsemi (d \to (g \fatsemi d)) \\ \text{compute composition :} \quad & = (c \to g \fatsemi f \fatsemi c) \\ \text{use Eq. (6.5) :} \quad & = \text{cmap}\,(g \fatsemi f) \quad . \end{aligned}$$

A type constructor with a fully parametric `cmap` is called a **contrafunctor** if the identity and the composition laws are satisfied.

Example 6.1.7.1 Show that the type constructor $D^A \triangleq A \to A \to \text{Int}$ is a contrafunctor.

Solution The required type signature for `contramap` is

```
def contramap[A, B](d: A => A => Int)(f: B => A): B => B => Int = ???
```

We begin implementing `contramap` by writing code with a typed hole:

$$\text{contramap}^{A,B} \triangleq d^{:A\to A\to\text{Int}} \to f^{:B\to A} \to b_1^{:B} \to b_2^{:B} \to ???^{:\text{Int}} \quad .$$

To fill the typed hole, we need to compute a value of type `Int`. The only possibility is to apply d to two curried arguments of type A. We have two curried arguments of type B. So we apply $f^{:B\to A}$ to those arguments, obtaining two values of type A. To avoid information loss, we need to preserve the order of the curried arguments. So the resulting expression is

$$\text{contramap}^{A,B} \triangleq d^{:A\to A\to\text{Int}} \to f^{:B\to A} \to b_1^{:B} \to b_2^{:B} \to d\,(f(b_1))\,(f(b_2)) \quad .$$

The corresponding Scala code is

```
def contramap[A, B](d: A => A => Int)(f: B => A): B => B => Int = { b1 => b2 => d(f(b1))(f(b2)) }
```

To verify the laws, it is easier to use the equivalent `cmap` defined by

$$\text{cmap}^{A,B}(f^{:B\to A}) \triangleq d^{:A\to A\to\text{Int}} \to b_1^{:B} \to b_2^{:B} \to d\,(f(b_1))\,(f(b_2)) \quad . \tag{6.6}$$

To verify the identity law:

$$\begin{aligned}
&\text{expect to equal id}: && \text{cmap}\,(\text{id})\\
&\text{use Eq. (6.6)}: && = d \to b_1 \to b_2 \to d\,\underline{(\text{id}(b_1))\,(\text{id}(b_2))}\\
&\text{definition of id}: && = d \to \underline{b_1 \to b_2 \to d(b_1)(b_2)}\\
&\text{simplify curried function}: && = (d \to d) = \text{id} \quad .
\end{aligned}$$

To verify the composition law, we rewrite its left-hand side into the right-hand side:

$$\begin{aligned}
& && \text{cmap}\,(f)\,\text{⨟}\,\text{cmap}\,(g)\\
&\text{use Eq. (6.6)}: && = (d \to b_1 \to b_2 \to d\,(f(b_1))\,(f(b_2)))\,\text{⨟}\,(\underline{d} \to b_1 \to b_2 \to \underline{d}\,(g(b_1))\,(g(b_2)))\\
&\text{rename } d \text{ to } e: && = (d \to b_1 \to b_2 \to d\,(f(b_1))\,(f(b_2)))\,\text{⨟}\,(e \to b_1 \to b_2 \to e\,(g(b_1))\,(g(b_2)))\\
&\text{compute composition}: && = d \to b_1 \to b_2 \to d\,(f(g(b_1)))\,(f(g(b_2)))\\
&\text{use Eq. (6.6)}: && = \text{cmap}(b \to f(g(b)))\\
&\text{definition of } (g\,\text{⨟}\,f): && = \text{cmap}(g\,\text{⨟}\,f) \quad .
\end{aligned}$$

The type C^A represents a function that consumes a value of type A to produce an integer; the type D^A represents a curried function consuming *two* values of type A. These examples suggest the heuristic view that contrafunctors "consume" data while functors "wrap" data. By looking at the position of a given type parameter in a type expression such as $A \times \text{Int}$ or $A \to A \to \text{Int}$, we can see whether the type parameter is "consumed" or "wrapped": A type parameter to the left of a function arrow is being "consumed"; a type parameter to the right of a function arrow (or used without a function arrow) is being "wrapped". We will make this intuition precise in Section 6.2.

Type constructors that are not contrafunctors A type constructor that both consumes *and* wraps data is neither a functor nor a contrafunctor. An example of such a type constructor is

$$N^A \triangleq (A \to \text{Int}) \times (\mathbb{1} + A) \quad .$$

We can implement neither `map` nor `contramap` for $N^\bullet$. Intuitively, the type parameter A is used both to the left of a function arrow (being "consumed") and outside of a function (being "wrapped").

Unfunctors (type constructors that lack full parametricity) also cannot be contrafunctors because the required type signature for `contramap` cannot be implemented by a fully parametric function. To show that `ServerAction[_]` cannot be a contrafunctor, we can straightforwardly adapt the reasoning used in Section 6.1.6 when we showed that `ServerAction[_]` cannot be a functor.

6.1.8 Subtyping, covariance, and contravariance

A type P is called a **subtype** of a type Q if there exists a designated **type conversion** function of type $P \to Q$ that the compiler will automatically use whenever necessary to match types. For instance, applying a function of type $Q \to Z$ to a value of type P is ordinarily a type error,

```
val h: Q => Z = ???
val p: P = ???
h(p) // Type error: the argument of h must be of type Q, not P.
```

However, this code will work when P is a subtype of Q because the compiler will automatically use the type conversion $P \to Q$ before applying the function `h`.

Different programming languages define subtyping differently because they make different choices of the type conversion functions and of types P, Q to which type conversions apply. Most often, the language designers choose the type conversion functions to be *identity* functions that merely reassign the types. Let us look at some examples of type conversion functions of that kind.

Within the focus of this book, the main example of subtyping is with disjunctive types. Consider this definition,

```
sealed trait AtMostTwo
final case class Zero()              extends AtMostTwo
final case class One(x: Int)         extends AtMostTwo
final case class Two(x: Int, y: Int) extends AtMostTwo
```

The corresponding type notation can be written as

$$\text{AtMostTwo} \triangleq \mathbb{1} + \text{Int} + \text{Int} \times \text{Int} \quad .$$

Each of the case classes (`Zero`, `One`, and `Two`) defines a type that is a subtype of `AtMostTwo`. To see that, we need to implement type conversion functions from each of the three case classes to `AtMostTwo`. The required functions reassign the types but perform no transformations on the data:

```
def f0: Zero => AtMostTwo  = { case Zero()    => Zero()    }
def f1: One  => AtMostTwo  = { case One(x)    => One(x)    }
def f2: Two  => AtMostTwo  = { case Two(x, y) => Two(x, y) }
```

The implementation of these type conversion functions looks like the code of *identity* functions. In the matrix notation, we can write

$$f_0 \triangleq \begin{array}{c|c||ccc} & & \text{Zero} & \text{One} & \text{Two} \\ \hline & \text{Zero} & \text{id} & \mathbb{0} & \mathbb{0} \end{array} \quad , \qquad f_0(\mathbb{1}^{:\text{Zero}}) \triangleq \mathbb{1} + \mathbb{0}^{:\text{One}} + \mathbb{0}^{:\text{Two}} \quad ,$$

$$f_1 \triangleq \begin{array}{c|c||ccc} & & \text{Zero} & \text{One} & \text{Two} \\ \hline & \text{One} & \mathbb{0} & \text{id} & \mathbb{0} \end{array} \quad , \qquad f_1(x^{:\text{Int}}) \triangleq \mathbb{0}^{:\text{Zero}} + x^{:\text{One}} + \mathbb{0}^{:\text{Two}} \quad ,$$

$$f_2 \triangleq \begin{array}{c|c||ccc} & & \text{Zero} & \text{One} & \text{Two} \\ \hline & \text{Two} & \mathbb{0} & \mathbb{0} & \text{id} \end{array} \quad , \qquad f_2(x^{:\text{Int}} \times y^{:\text{Int}}) \triangleq \mathbb{0}^{:\text{Zero}} + \mathbb{0}^{:\text{One}} + (x \times y)^{:\text{Two}} \quad .$$

This notation emphasizes that the code consists of identity functions with reassigned types.

Another example is a subtyping relation between function types. Consider the types

```
type P = (AtMostTwo => Int)
type Q = (Two => Int)
```

We can convert a function f of type P into a function g of type Q because f includes all the information necessary to define g. The Scala code for that type conversion is

```
def p2q(f: P): Q = { t: Two => f(t) }
```

This is written in the code notation as

$$\text{p2q}(f^{:\text{AtMostTwo}\to\text{Int}}) \triangleq t^{:\text{Two}} \to f(t) \quad .$$

Note that $t^{:\text{Two}} \to f(t)$ is the same function as f, except applied to a subtype `Two` of `AtMostTwo`. So, the implementation of `p2q(f)` is just `f` composed with an identity function with reassigned types.

In these cases, it is useful if the compiler could insert the appropriate conversion functions automatically whenever necessary. Any function that consumes an argument of type Q could be then automatically applicable to arguments of type P. The compiler could also remove the identity functions from the code, since they do not perform any data transformations. In this way, code involving subtypes becomes more concise with no decrease in performance.

To achieve this, we need to declare to the Scala compiler that certain types have a subtyping relation involving an identity function as the type conversion. This can be done in one of three ways depending on the situation at hand:

1. Declaring a class that `extends` another class (as we have just seen).
2. Declaring type parameters with a "variance annotation" such as `L[+A]` or `L[-B]`.
3. Declaring type parameters with a "subtyping annotation" (`A <: B`).

Subtyping for disjunctive types A function with argument of type `AtMostTwo` can be applied to a value of type `Two` with no additional code written by the programmer:

```
def head: AtMostTwo => Option[Int] = {
  case Zero()      => None
  case One(x)      => Some(x)
  case Two(x, y)   => Some(x)
}

scala> head(Two(10, 20))
res0: Option[Int] = Some(10)
```

We may imagine that the compiler automatically used the type conversion function `f2` shown above to convert a value of the type `Two` into a value of the type `AtMostTwo`. Since the code of `f2` is equivalent to an identity function, the type conversion does not change any data and only reassigns the types of the given values. So the compiler does not need to insert any additional code, and the type conversion does not lead to any decrease in performance.

Subtyping for type constructors If a type constructor L^A is a functor, we can use its fmap_L method to lift a type conversion function $f : P \to Q$ into

$$\text{fmap}_L(f) : L^P \to L^Q \quad ,$$

which gives a type conversion function from L^P to L^Q. This gives a subtyping relation between the types L^P and L^Q because the code of the lifted function $\text{fmap}_L(f)$ is an identity function, due to functor L's identity law, $\text{fmap}_L(\text{id}) = \text{id}$.

If a type constructor C^A is a contrafunctor, a type conversion function $f^{:P\to Q}$ is lifted to

$$\text{cmap}_C(f) : C^Q \to C^P \quad ,$$

showing that C^Q is a subtype of C^P. The identity law of the contrafunctor C,

$$\text{cmap}_C(\text{id}) = \text{id} \quad ,$$

shows that the lifted conversion function is an identity function with reassigned types.

A type constructor F is called **covariant** if F^A is a subtype of F^B whenever A is a subtype of B. A **contravariant** type constructor C has the subtype relation in the opposite direction: C^B is a subtype of C^A. So, we have established that all functors are covariant type constructors, and all contrafunctors are contravariant type constructors.[2]

The Scala compiler does not automatically determine whether a given type constructor `F[A]` is covariant with respect to a given type parameter `A`. To indicate the covariance property, the programmer needs to use a **variance annotation**, which looks like `F[+A]`, on the relevant type parameters. For

[2] The name "contrafunctor" was chosen in this book as a shortened form of "contravariant functor".

example, the type constructor `Counted[A]` defined in Section 6.1.4 is a functor and so is covariant in its type parameter `A`. If we use the variance annotation `Counted[+A]` in the definition, Scala will automatically consider the type `Counted[Two]` as a subtype of `Counted[AtMostTwo]`. So we may now apply a function to a value of type `Counted[Two]` as if it had type `Counted[AtMostTwo]`:

```
final case class Counted[+A](n: Int, a: A)

def total(c: Counted[AtMostTwo]): Int = c match {
  case Counted(n, Zero())      => 0
  case Counted(n, One(_))      => n
  case Counted(n, Two(_, _))   => n * 2
}

scala> total(Counted(2, Two(10, 20)))
res1: Int = 4
```

The contravariance property for contrafunctors can be annotated using the syntax `F[-A]`.

A given type constructor may have several type parameters and may be covariant with respect to some of them and contravariant with respect to others. As we have seen, the position of a type parameter in a type expression indicates whether the value is "wrapped" (used in a **covariant position**) or "consumed" (used in a **contravariant position**). Covariant positions are to the right of function arrows, or outside function arrows; contravariant positions are to the left of a function arrow. The next examples confirm this intuition, which will be made rigorous in Section 6.2.

6.1.9 Solved examples: functors and contrafunctors

Example 6.1.9.1 Consider this implementation of `map` for the type constructor `Option[_]`:

```
def map[A, B](oa: Option[A])(f: A => B): Option[B] = oa match {
  case None           => None
  case Some(x: Int)   => Some(f((x+1).asInstanceOf[A]))
  case Some(x)        => Some(f(x))
}
```

This code performs a non-standard computation if the type parameter `A` is set to `Int`. Show that this implementation of `map` violates the functor laws.

Solution If the type parameter `A` is not `Int`, or if the argument `oa` is `None`, the given code is the same as the standard (correct) implementation of `map` for `Option`. The function does something non-standard when e.g. `oa == Some(123)`. Substitute this value of `oa` into the identity law, `map(oa)(identity) == oa`, and compute symbolically (using Scala syntax)

```
map(oa)(identity) == Some(identity((123+1).asInstanceOf[Int])) == Some(124) != oa
```

This shows a violation of the functor identity law.

Example 6.1.9.2 Define case classes and implement `fmap` for the given type constructors:

(a) $\text{Data}^A \triangleq \text{String} + A \times \text{Int} + A \times A \times A$.

(b) $\text{Data}^A \triangleq \mathbb{1} + A \times (\text{Int} \times \text{String} + A)$.

(c) $\text{Data}^A \triangleq (\text{String} \to \text{Int} \to A) \times A + (\text{Bool} \to \text{Double} \to A) \times A$.

Solution **(a)** Begin by defining a case class for each part of the disjunctive type:

```
sealed trait Data[A]
final case class Message[A](message: String)  extends Data[A]
final case class Have1[A](x: A, n: Int)       extends Data[A]
final case class Have3[A](x: A, y: A, z: A)   extends Data[A]
```

The names `Message`, `Have1`, `Have3`, `n`, `x`, `y`, `z` are chosen arbitrarily.

The function `fmap` must have the type signature

$$\text{fmap}^{A,B} : f^{:A \to B} \to \text{Data}^A \to \text{Data}^B \quad .$$

To implement `fmap` correctly, we need to transform each part of the disjunctive type `Data[A]` into the corresponding part of `Data[B]` without loss of information. To clarify where the transformation $f^{:A \to B}$

need to be applied, let us write the type notation for Data^A and Data^B side by side:

$$\text{Data}^A \triangleq \text{String} + A \times \text{Int} + A \times A \times A \quad ,$$
$$\text{Data}^B \triangleq \text{String} + B \times \text{Int} + B \times B \times B \quad .$$

Now it is clear that we need to apply f to each value of type A present in Data^A, preserving the order of values. The Scala code is

```
def fmap[A, B](f: A => B): Data[A] => Data[B] = {
  case Message(message)   => Message(message)
  case Have1(x, n)        => Have1(f(x), n)
  case Have3(x, y, z)     => Have3(f(x), f(y), f(z))
}
```

(b) It is convenient to define the disjunctive type $\text{Int} \times \text{String} + A$ separately as P^A:

```
sealed trait P[A]
final case class Message[A](code: Int, message: String)  extends P[A]
final case class Value[A](x: A)                          extends P[A]
```

Now we notice that the type expression $(\mathbb{1} + ...)$ can be encoded via the standard `Option` type. So, the Scala code for Data^A is

```
final case class Data[A](d: Option[(A, P[A])])
```

To help us implement `fmap` correctly, we write the type expressions

$$\text{Data}^A \triangleq \mathbb{1} + A \times (\text{Int} \times \text{String} + A) \quad ,$$
$$\text{Data}^B \triangleq \mathbb{1} + B \times (\text{Int} \times \text{String} + B) \quad ,$$

and transform Data^A into Data^B by applying $f^{:A \to B}$ at the correct places:

```
def fmap[A, B](f: A => B): Data[A] => Data[B] = {
  case Data(None)                                 => Data(None)
  case Data(Some((x, Message(code, message))))    => Data(Some((f(x), Message(code, message))))
  case Data(Some((x, Value(y))))                  => Data(Some((f(x), Value(f(y)))))
}
```

When deeply nested patterns become hard to read, we may handle the nested structure separately:

```
def fmap[A, B](f: A => B): Data[A] => Data[B] = {
  case Data(None)           => Data(None)
  case Data(Some((x, p)))   =>
      val newP: P[B] = p match {
        case Message(code, message)   => Message(code, message)
        case Value(x)                 => Value(f(x))
      }
      Data(Some((f(x), newP)))
}
```

(c) Since the type structures $(\text{String} \to \text{Int} \to A) \times A$ and $(\text{Bool} \to \text{Double} \to A) \times A$ have a similar pattern, let us define a parameterized type

$$Q^{X,Y,A} \triangleq (X \to Y \to A) \times A \quad ,$$

and express the given type expression as

$$\text{Data}^A \triangleq Q^{\text{String},\text{Int},A} + Q^{\text{Bool},\text{Double},A} \quad .$$

It is then convenient to define `Data[A]` using the standard disjunctive type `Either`:

```
type Q[X, Y, A] = (X => Y => A, A)
type Data[A] = Either[Q[String, Int, A], Q[Boolean, Double, A]]
```

To make the code clearer, we will implement `fmap` separately for $Q^\bullet$ and $\text{Data}^\bullet$.

To derive the code of `fmap` for $Q^\bullet$, we begin with the type signature

$$\text{fmap}_Q^{A,B} : (A \to B) \to (X \to Y \to A) \times A \to (X \to Y \to B) \times B$$

and start writing the code using typed holes,

$$\text{fmap}_Q(f^{:A\to B}) \triangleq g^{:X\to Y\to A} \times a^{:A} \to ???^{:X\to Y\to B} \times ???^{:B} \quad .$$

The typed hole $???^{:B}$ is filled by $f(a)$. To fill the remaining type hole, we write

$$\begin{aligned} & ???^{:X\to Y\to B} \\ & = x^{:X} \to y^{:Y} \to \underline{???^{:B}} \\ & = x^{:X} \to y^{:Y} \to f(???^{:A}) \quad . \end{aligned}$$

It would be wrong to fill the typed hole $???^{:A}$ by $a^{:A}$ because a value of type $X \to Y \to B$ should be computed using the given data $g^{:X\to Y\to A}$ of type $X \to Y \to A$. So we write

$$???^{:X\to Y\to B} = x^{:X} \to y^{:Y} \to f(g(x)(y)) \quad .$$

The corresponding Scala code is

```
def fmap_Q[A, B, X, Y](f: A => B): Q[X, Y, A] => Q[X, Y, B] = {
  case (g, a) => (x => y => f(g(x)(y)), f(a))
  // Could also write the code as
  //   case (g, a) => (x => g(x) andThen f, f(a))
}
```

Finally, we can write the code for $\text{fmap}_{\text{Data}}$:

```
def fmap_Data[A, B](f: A => B): Data[A] => Data[B] = {
  case Left(q)    => Left(fmap_Q(f)(q))
  case Right(q)   => Right(fmap_Q(f)(q))
}
```

The Scala compiler will automatically infer the type parameters required by `fmap_Q` and check that all types match. With all inferred types written out, the code above would be

```
def fmap_Data[A, B](f: A => B): Data[A] => Data[B] = {
  case Left(q: Q[String, Int, A])            =>
         Left[Q[String, Int, B]](fmap_Q[A, B, String, Int](f)(q))
  case Right(q: Q[Boolean, Double, A])       =>
         Right[Q[Boolean, Double, B]](fmap_Q[A, B, Boolean, Double](f)(q))
}
```

When types become complicated, it may help to write out some of the type parameters in the code.

Example 6.1.9.3 Decide which of these types are functors or contrafunctors, and implement `fmap` or `cmap` as appropriate:

(a) $\text{Data}^A \triangleq (A \to \text{Int}) + (A \to A \to \text{String})$.

(b) $\text{Data}^{A,B} \triangleq (A + B) \times ((A \to \text{Int}) \to B)$.

Solution **(a)** The type constructor Data^A uses its type parameter A always as an argument of some functions, i.e. to the left of function arrows:

```
type Data[A] = Either[A => Int, A => A => String]
```

So, Data^A *consumes* values of type A, and we expect that Data^A is a contrafunctor. Indeed, we can implement `cmap`:

```
def cmap[A, B](f: B => A): Data[A] => Data[B] = {
  case Left(a2Int)          => Left(b => a2Int(f(b)))
  case Right(a2a2String)    => Right(b1 => b2 => a2a2String(f(b1))(f(b2)))
}
```

(b) The type constructor $\text{Data}^{A,B}$ has *two* type parameters, and so we need to answer the question separately for each of them. Write the Scala type definition as

```
type Data[A, B] = (Either[A, B], (A => Int) => B)
```

Begin with the type parameter A and notice that a value of type $\text{Data}^{A,B}$ possibly contains a value of type A within `Either[A, B]`. In other words, A is "wrapped", i.e. it is in a covariant position within the first part of the tuple. It remains to check the second part of the tuple, which is a higher-order function of type $(A \to \text{Int}) \to B$. That function consumes a function of type $A \to \text{Int}$, which in turn consumes a value of type A. Consumers of A are contravariant in A, but it turns out that a "consumer of a consumer of A" is *covariant* in A. So we expect to be able to implement `fmap` that applies to the type parameter A of $\text{Data}^{A,B}$. Renaming the type parameter B to Z for clarity, we write the type signature for `fmap` like this,

$$\text{fmap}^{A,C,Z} : (A \to C) \to (A + Z) \times ((A \to \text{Int}) \to Z) \to (C + Z) \times ((C \to \text{Int}) \to Z) \quad .$$

We need to transform each part of the tuple separately. Transforming $A + Z$ into $C + Z$ is straightforward via the function

	C	Z
A	f	$\mathbb{0}$
Z	$\mathbb{0}$	id

.

The corresponding Scala code is

```
{
  case Left(x)    => Left(f(x))
  case Right(z)   => Right(z)
}
```

To derive code transforming $(A \to \text{Int}) \to Z$ into $(C \to \text{Int}) \to Z$, we use typed holes:

$$\begin{aligned}
& f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to \underline{???^{:(C\to\text{Int})\to Z}} \\
\text{nameless function :} \quad & = f^{:A\to C} \to g^{:(A\to\text{Int})\to Z} \to p^{:C\to\text{Int}} \to \underline{???^{:Z}} \\
\text{get a } Z \text{ by applying } g : \quad & = f \to g \to p \to g(\underline{???^{:A\to\text{Int}}}) \\
\text{nameless function :} \quad & = f \to g \to p \to g(a^{:A} \to \underline{???^{:\text{Int}}}) \\
\text{get an Int by applying } p : \quad & = f \to g \to p \to g(a \to p(\underline{???^{:C}})) \\
\text{get a } C \text{ by applying } f : \quad & = f \to g \to p \to g(a \to p(f(\underline{???^{:A}}))) \\
\text{use argument } a^{:A} : \quad & = f \to g \to p \to g(a \to p(f(a))) \quad .
\end{aligned}$$

In the resulting Scala code for `fmap`, we write out some types for clarity:

```
def fmapA[A, Z, C](f: A => C): Data[A, Z] => Data[C, Z] = {
  case (e: Either[A, Z], g: ((A => Int) => Z)) =>
    val newE: Either[C, Z] = e match {
      case Left(x)    => Left(f(x))
      case Right(z)   => Right(z)
    }
    val newG: (C => Int) => Z = { p => g(a => p(f(a))) }
    (newE, newG) // This has type Data[C, Z].
}
```

This suggests that $\text{Data}^{A,Z}$ is covariant with respect to the type parameter A. The results of Section 6.2 will show rigorously that the functor laws hold for this implementation of `fmap`.

The analysis is simpler for the type parameter B because it is never to the left of a function arrow and so is only used in covariant positions. So we expect $\text{Data}^{A,B}$ to be a functor with respect to B. Implementing the corresponding `fmap` is straightforward:

```
def fmapB[Z, B, C](f: B => C): Data[Z, A] => Data[Z, B] = {
  case (e: Either[Z, B], g: ((Z => Int) => B)) =>
    val newE: Either[Z, B] = e match {
      case Left(x)    => Left(f(x))
      case Right(z)   => Right(z)
    }
    val newG: (C => Int) => Z = { p => g(a => p(f(a))) }
    (newE, newG) // This has type Data[C, Z].
}
```

The code indicates that $\text{Data}^{A,B}$ is a functor with respect to both A and B.

Example 6.1.9.4 Rewrite the following code in the type notation; identify covariant and contravariant type usages; verify with the Scala compiler that the variance annotations are correct:

```
sealed trait Coi[A, B]
final case class Pa[A, B](b: (A, B), c: B => Int)     extends Coi[A, B]
final case class Re[A, B](d: A, e: B, c: Int)         extends Coi[A, B]
final case class Ci[A, B](f: String => A, g: B => A)  extends Coi[A, B]
```

Solution The type notation puts all parts of the disjunctive type into a single type expression:

$$\text{Coi}^{A,B} \triangleq A \times B \times (B \to \text{Int}) + A \times B \times \text{Int} + (\text{String} \to A) \times (B \to A) \quad .$$

Now find which types are wrapped and which are consumed in this type expression. The type parameter A is wrapped and never consumed, but B is both wrapped and consumed (in $B \to A$). So, the type constructor `Coi` is covariant in A but neither covariant nor contravariant in B. We can check this by compiling the corresponding Scala code with variance annotations:

```
sealed trait Coi[+A, B]
case class Pa[+A, B](b: (A, B), c: B => Int)     extends Coi[A, B]
case class Re[+A, B](d: A, e: B, c: Int)         extends Coi[A, B]
case class Ci[+A, B](f: String => A, g: B => A)  extends Coi[A, B]
```

We could also replace the fixed types `Int` and `String` by type parameters `N` and `S`. A similar analysis shows that `N` is in covariant positions while `S` is in a contravariant position. We can then check that the Scala compiler accepts the following type definition with variance annotations:

```
sealed trait Coi2[+A, B, +N, -S]
case class Pa2[+A, B, +N, -S](b: (A, B), c: B => N)  extends Coi[A, B, N, S]
case class Re2[+A, B, +N, -S](d: A, e: B, c: N)      extends Coi[A, B, N, S]
case class Ci2[+A, B, +N, -S](f: S => A, g: B => A)  extends Coi[A, B, N, S]
```

6.1.10 Exercises: functors and contrafunctors

Exercise 6.1.10.1 An implementation of `fmap` for the type constructor `Either[A, A]` is given as

```
def fmap[A, B](f: A => B): Either[A, A] => Either[B, B] = {
  case Left(a: Int)   => Left(f(a + 1))
  case Left(a)        => Right(f(a))
  case Right(a)       => Left(f(a))
}
```

Show that this implementation of `fmap` violates the functor laws. Implement `fmap` correctly for this type constructor.

Exercise 6.1.10.2 Define these type constructors in Scala, decide whether they are covariant or contravariant, and implement `fmap` or `cmap` as appropriate:

(a) $\text{Data}^A \triangleq (\mathbb{1} + A) \times (\mathbb{1} + A) \times \text{String}$.

(b) $\text{Data}^A \triangleq (A \to \text{Bool}) \to (A \times (\text{Int} + A))$.

(c) $\text{Data}^{A,B} \triangleq (A \to \text{Bool}) \times ((A + B) \to \text{Int})$.

(d) $\text{Data}^A \triangleq (\mathbb{1} + (A \to \text{Bool})) \to (\mathbb{1} + (A \to \text{Int})) \to \text{Int}$.

(e) $\text{Data}^B \triangleq (B + (\text{Int} \to B)) \times (B + (\text{String} \to B))$.

Exercise 6.1.10.3 Rewrite the following code in the type notation; identify covariant and contravariant type usages; add variance annotations and verify that the resulting code compiles:

```
sealed trait Result[A,B]
final case class P[A,B](a: A, b: B, c: Int)        extends Result[A,B]
final case class Q[A,B](d: Int => A, e: Int => B)  extends Result[A,B]
final case class R[A,B](f: A => A, g: A => B)      extends Result[A,B]
```

6.2 Laws and structure

A type constructor is a functor if it admits a lawful `map` function. How can we recognize quickly that a given type constructor is a functor or perhaps a contrafunctor? For example, consider the type constructor $Z^{A,R}$ defined by

$$Z^{A,R} \triangleq ((A \to A \to R) \to R) \times A + (\mathbb{1} + R \to A + \text{Int}) + A \times A \times \text{Int} \times \text{Int} \quad . \tag{6.7}$$

Is $Z^{A,R}$ a functor with respect to A, or perhaps with respect to R? To answer these questions, we will systematically build up various type expressions for which the functor or contrafunctor laws hold.

6.2.1 Reformulations of laws

We begin by introducing a more convenient notation for the functor laws. The laws (6.2)–(6.3) were defined in terms of the function `fmap`. When written in terms of the curried function `map`, the structure of the laws becomes less clear:

$$\text{map}_L(x^{:L^A})(\text{id}^{:A\to A}) = x \quad ,$$
$$\text{map}_L(x^{:L^A})(f^{:A\to B} \fatsemi g^{:B\to C}) = \text{map}_L(\text{map}_L(x)(f))(g) \quad .$$

However, the laws again look clearer when using the infix method `.map`:

```
x.map(identity) == x
x.map(f).map(g) == x.map(f andThen g)
```

To take advantage of this syntax, we can use the pipe notation where $x \triangleright \text{fmap}(f)$ means `x.map(f)`, and write the functor laws as

$$x \triangleright \text{fmap}_L(\text{id}) = x \quad ,$$
$$x \triangleright \text{fmap}_L(f) \triangleright \text{fmap}_L(g) = x \triangleright \text{fmap}_L(f \fatsemi g) \quad .$$

In later chapters of this book, we will find that the `.map` methods (equivalently, the `fmap` function) are used so often in different contexts that the notation $\text{fmap}_L(f)$ becomes too verbose. To make code expressions visually easy to manipulate, we need a shorter notation. At the same time, it is important to mark clearly the relevant type constructor L. Dropping the symbol L can lead to errors, since it will be sometimes unclear what type constructors are involved in an expression such as `x.map(f).map(g)` and whether we are justified in replacing that expression with `x.map(f andThen g)`.

For these reasons, we introduce the superscript notation $^{\uparrow L}$ (pronounced "lifted to L") defined, for any function f, by

$$(f^{:A\to B})^{\uparrow L} : L^A \to L^B \quad , \qquad f^{\uparrow L} \triangleq \text{fmap}_L(f) \quad .$$

Now we can write

$$x \triangleright \text{fmap}_L(f) = x \triangleright f^{\uparrow L} = f^{\uparrow L}(x) \quad ,$$
$$\text{map}_L(x)(f) = f^{\uparrow L}(x) \quad .$$

In this notation, the identity and composition laws for a functor L are especially easy to use:

$$\text{id}^{\uparrow L} = \text{id} \quad , \qquad (f \fatsemi g)^{\uparrow L} = f^{\uparrow L} \fatsemi g^{\uparrow L} \quad .$$

Applying a composition of lifted functions to a value looks like this,

$$x \triangleright (f \fatsemi g)^{\uparrow L} = x \triangleright f^{\uparrow L} \fatsemi g^{\uparrow L} = x \triangleright f^{\uparrow L} \triangleright g^{\uparrow L} \quad .$$

This equation directly represents the Scala code syntax

```
x.map(f andThen g) == x.map(f).map(g)
```

since the piping symbol ($\triangleright$) groups weaker than the composition symbol ($\fatsemi$).

Written in the *backward* notation ($f \circ g$), the functor composition law is

$$(g \circ f)^{\uparrow L} = g^{\uparrow L} \circ f^{\uparrow L} \quad .$$

The analogous notation for a contrafunctor $C^\bullet$ is

$$f^{\downarrow C} \triangleq \text{cmap}_C(f) \quad .$$

The contrafunctor laws are then written as

$$\text{id}^{\downarrow C} = \text{id} \quad , \qquad (f \fatsemi g)^{\downarrow C} = g^{\downarrow C} \fatsemi f^{\downarrow C} \quad , \qquad (g \circ f)^{\downarrow C} = f^{\downarrow C} \circ g^{\downarrow C} \quad .$$

We will mostly use the forward composition $f \fatsemi g$ in this book, keeping in mind that one can straightforwardly and mechanically translate between forward and backward notations via

$$f \fatsemi g = g \circ f \quad , \qquad x \triangleright f = f(x) \quad .$$

6.2.2 Bifunctors

A type constructor can be a functor with respect to several type parameters. A **bifunctor** is a type constructor with *two* type parameters that satisfies the functor laws with respect to both type parameters at once.

As an example, consider the type constructor F defined by

$$F^{A,B} \triangleq A \times B \times B \quad .$$

If we fix the type parameter B but let the parameter A vary, we get a type constructor that we can denote as $F^{\bullet,B}$. We see that the type constructor $F^{\bullet,B}$ is a functor, with the corresponding `fmap` function

$$\text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \triangleq a^{:A} \times b_1^{:B} \times b_2^{:B} \to f(a) \times b_1 \times b_2 \quad .$$

Instead of saying that $F^{\bullet,B}$ is a functor, we can also say more verbosely that $F^{A,B}$ is a functor with respect to A.

If we now fix the type parameter A, we find that the type constructor $F^{A,\bullet}$ is a functor, with the `fmap` function

$$\text{fmap}_{F^{A,\bullet}}(g^{:B\to D}) \triangleq a^{:A} \times b_1^{:B} \times b_2^{:B} \to a \times g(b_1) \times g(b_2) \quad .$$

Since the bifunctor $F^{\bullet,\bullet}$ is a functor with respect to each type parameter separately, we can transform a value of type $F^{A,B}$ to a value of type $F^{C,D}$ by applying the two `fmap` functions one after another. It is convenient to denote this transformation by a single operation called `bimap` that uses two functions $f^{:A\to C}$ and $g^{:B\to D}$ as arguments:

$$\begin{aligned} &\text{bimap}_F(f^{:A\to C})(g^{:B\to D}) : F^{A,B} \to F^{C,D} \quad , \\ &\text{bimap}_F(f^{:A\to C})(g^{:B\to D}) \triangleq \text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \fatsemi \text{fmap}_{F^{C,\bullet}}(g^{:B\to D}) \quad . \end{aligned} \tag{6.8}$$

In the condensed notation, this is written as

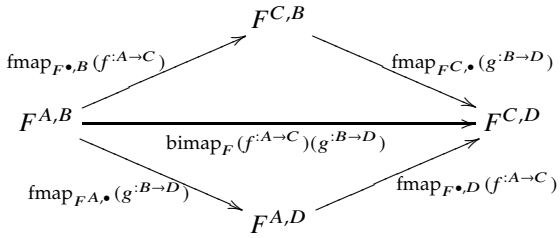
$$\text{bimap}_F(f^{:A\to C})(g^{:B\to D}) \triangleq f^{\uparrow F^{\bullet,B}} \fatsemi g^{\uparrow F^{C,\bullet}} \quad ,$$

but in this case the longer notation in Eq. (6.8) is easier to reason about.

What if we apply the two `fmap` functions in the opposite order? Since these functions work with different type parameters, it is reasonable to expect that the transformation $F^{A,B} \to F^{C,D}$ should be independent of the order of application:

$$\text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \fatsemi \text{fmap}_{F^{C,\bullet}}(g^{:B\to D}) = \text{fmap}_{F^{A,\bullet}}(g^{:B\to D}) \fatsemi \text{fmap}_{F^{\bullet,D}}(f^{:A\to C}) \quad . \tag{6.9}$$

This equation is illustrated by the type diagram below.



Different paths in this diagram give the same results if they arrive at the same vertex (as mathematicians say, "the diagram commutes"). In this way, the diagram illustrates at once the commutativity law (6.9) and the definition (6.8) of bimap_F.

Let us verify the commutativity law for the bifunctor $F^{A,B} \triangleq A \times B \times B$:

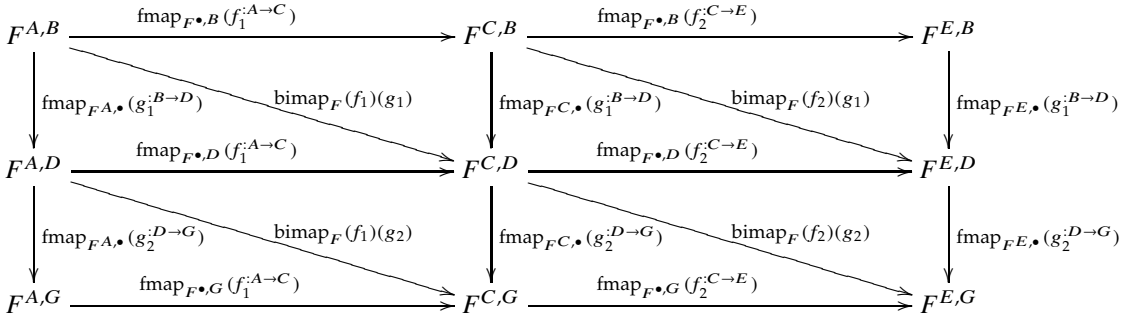
$$\begin{aligned}
&\text{left-hand side :} && \text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \fatsemi \text{fmap}_{F^{C,\bullet}}(g^{:B\to D}) \\
&\text{definitions of } \text{fmap}_{F^{\bullet,\bullet}} \text{ :} && = (a^{:A} \times b_1^{:B} \times b_2^{:B} \to f(a) \times b_1 \times b_2) \fatsemi (c^{:C} \times b_1^{:B} \times b_2^{:B} \to c \times g(b_1) \times g(b_2)) \\
&\text{compute composition :} && = a^{:A} \times b_1^{:B} \times b_2^{:B} \to f(a) \times g(b_1) \times g(b_2) \quad , \\
&\text{right-hand side :} && \text{fmap}_{F^{A,\bullet}}(g^{:B\to D}) \fatsemi \text{fmap}_{F^{\bullet,D}}(f^{:A\to C}) \\
&\text{definitions of } \text{fmap}_{F^{\bullet,\bullet}} \text{ :} && = (a^{:A} \times b_1^{:B} \times b_2^{:B} \to a \times g(b_1) \times g(b_2)) \fatsemi (a^{:A} \times d_1^{:D} \times d_2^{:D} \to f(a) \times d_1 \times d_2) \\
&\text{compute composition :} && = a^{:A} \times b_1^{:B} \times b_2^{:B} \to f(a) \times g(b_1) \times g(b_2) \quad .
\end{aligned}$$

Both sides of the law are equal.

The commutativity law (6.9) leads to the composition law for `bimap`,

$$\text{bimap}_F(f_1^{:A\to C})(g_1^{:B\to D}) \fatsemi \text{bimap}_F(f_2^{:C\to E})(g_2^{:D\to G}) = \text{bimap}_F(f_1 \fatsemi f_2)(g_1 \fatsemi g_2) \quad . \tag{6.10}$$

The following type diagram shows the relationships between various `bimap` and `fmap` functions:



To derive the composition law from Eq. (6.9), write

$$\begin{aligned}
&&& \text{bimap}_F(f_1)(g_1) \fatsemi \text{bimap}_F(f_2)(g_2) \\
&\text{use Eq. (6.8) :} && = \text{fmap}_{F^{\bullet,B}}(f_1) \fatsemi \underline{\text{fmap}_{F^{C,\bullet}}(g_1) \fatsemi \text{fmap}_{F^{\bullet,D}}(f_2)} \fatsemi \text{fmap}_{F^{E,\bullet}}(g_2) \\
&\text{commutativity law (6.9) :} && = \text{fmap}_{F^{\bullet,B}}(f_1) \underline{\fatsemi} \text{fmap}_{F^{\bullet,B}}(f_2) \fatsemi \text{fmap}_{F^{E,\bullet}}(g_1) \underline{\fatsemi} \text{fmap}_{F^{E,\bullet}}(g_2) \\
&\text{composition laws :} && = \text{fmap}_{F^{\bullet,B}}(f_1 \fatsemi f_2) \fatsemi \text{fmap}_{F^{E,\bullet}}(g_1 \fatsemi g_2) \\
&\text{use Eq. (6.8) :} && = \text{bimap}_F(f_1 \fatsemi f_2)(g_1 \fatsemi g_2) \quad .
\end{aligned}$$

Conversely, we can derive Eq. (6.9) from the composition law (6.10). We write the composition law with specially chosen functions:

$$\text{bimap}_F(f^{:A\to C})(g^{:B\to D}) = \text{bimap}_F(\text{id}^{:A\to A})(g^{:B\to D}) \fatsemi \text{bimap}_F(f^{:A\to C})(\text{id}^{:D\to D}) \quad . \tag{6.11}$$

Using Eq. (6.8), we find

$$\begin{aligned}
\text{expect } \text{fmap}_{F^{A,\bullet}}(g) \fatsemi \text{fmap}_{F^{\bullet,D}}(f) : \quad & \text{fmap}_{F^{\bullet,B}}(f^{:A\to C}) \fatsemi \text{fmap}_{F^{C,\bullet}}(g^{:B\to D}) \\
\text{use Eq. (6.8)} : \quad & = \text{bimap}_F(f^{:A\to C})(g^{:B\to D}) \\
\text{use Eq. (6.11)} : \quad & = \text{bimap}_F(\text{id}^{:A\to A})(g^{:B\to D}) \fatsemi \text{bimap}_F(f^{:A\to C})(\text{id}^{:D\to D}) \\
\text{use Eq. (6.8)} : \quad & = \underline{\text{fmap}_{F^{\bullet,B}}(\text{id})} \fatsemi \text{fmap}_{F^{A,\bullet}}(g) \fatsemi \text{fmap}_{F^{\bullet,D}}(f) \fatsemi \underline{\text{fmap}_{F^{C,\bullet}}(\text{id})} \\
\text{identity laws for } F : \quad & = \text{fmap}_{F^{A,\bullet}}(g) \fatsemi \text{fmap}_{F^{\bullet,D}}(f) \quad .
\end{aligned}$$

The identity law for `bimap` holds as well,

$$\begin{aligned}
\text{expect to equal id} : \quad & \text{bimap}_F(\text{id}^{:A\to A})(\text{id}^{:B\to B}) \\
\text{use Eq. (6.8)} : \quad & = \underline{\text{fmap}_{F^{\bullet,B}}(\text{id})} \fatsemi \underline{\text{fmap}_{F^{C,\bullet}}(\text{id})} \\
\text{identity laws for } F : \quad & = \text{id} \fatsemi \text{id} = \text{id} \quad .
\end{aligned}$$

If $F^{A,B}$ is known to be a functor separately with respect to A and B, will the commutativity law (6.9) always hold? The calculation for the example $F^{A,B} \triangleq A \times B \times B$ shows that the two `fmap` functions commute because they work on different parts of the data structure $F^{A,B}$. This turns out[3] to be true in general: the commutativity law follows from the parametricity of the `fmap` functions. Because of this, we do not need to verify the `bimap` laws as long as $F^{\bullet,B}$ and $F^{A,\bullet}$ are lawful functors.

Type constructors with more than two type parameters have similar properties. It is sufficient to check the functor laws with respect to each type parameter separately.

In general, a type constructor may be a functor with respect to some type parameters and a contrafunctor with respect to others. For brevity, we will talk about "covariant" and "contravariant" type parameters in those cases.

6.2.3 Type constructions for functors

What type expressions will produce a functor? Functional programming languages support the six standard type constructions (see Section 5.1.2). We will check whether each construction produces a new type that obeys the functor laws. The results are summarized in Table 6.2.

In each of these constructions, the `fmap` function for a new functor is defined either from scratch or by using the known `fmap` functions for previously defined type constructors. We will now derive the code for these constructions and prove their validity. We will use the code notation for brevity, occasionally showing the translation into the Scala syntax.

Statement 6.2.3.1 The type constructor $\text{Id}^A \triangleq A$ is a lawful functor (the **identity functor**).

Proof The `fmap` function is defined by

$$\begin{aligned}
&\text{fmap}_{\text{Id}} : (A \to B) \to \text{Id}^A \to \text{Id}^B \cong (A \to B) \to A \to B \quad , \\
&\text{fmap}_{\text{Id}} \triangleq (f^{:A\to B} \to f) = \text{id}^{:(A\to B)\to A\to B} \quad .
\end{aligned}$$

The identity function is the only fully parametric implementation of the type signature $(A \to B) \to A \to B$. Since the code of `fmap` is the identity function, the laws are satisfied automatically:

$$\begin{aligned}
\text{identity law} : \quad & \text{fmap}_{\text{Id}}(\text{id}) = \text{id}(\text{id}) = \text{id} \quad , \\
\text{composition law} : \quad & \text{fmap}_{\text{Id}}(f \fatsemi g) = f \fatsemi g = \text{fmap}_{\text{Id}}(f) \fatsemi \text{fmap}_{\text{Id}}(g) \quad .
\end{aligned}$$

[3] The proof is beyond the scope of this chapter; see https://byorgey.wordpress.com/2018/03/30/

Construction	Type notation	Comment
type parameter	$L^A \triangleq A$	the identity functor
product type	$L^A \triangleq P^A \times Q^A$	the functor product; P and Q must be functors
disjunctive type	$L^A \triangleq P^A + Q^A$	the functor co-product; P and Q must be functors
function type	$L^A \triangleq C^A \to P^A$	the functor exponential; P is a functor and C a contrafunctor
primitive type	$L^A \triangleq Z$	the constant functor; Z is a fixed type
type constructor	$L^A \triangleq P^{Q^A}$	functor composition; P and Q are both functors or both contrafunctors
recursive type	$L^A \triangleq S^{A,L^A}$	recursive functor; $S^{A,B}$ must be a functor w.r.t. both A and B

Table 6.2: Type constructions defining a functor L^A.

Statement 6.2.3.2 The type constructor $\text{Const}^{Z,A} \triangleq Z$ is a lawful functor (a **constant functor**) with respect to the type parameter A.

Proof The `fmap` function is defined by

$$\text{fmap}_{\text{Const}} : (A \to B) \to \text{Const}^{Z,A} \to \text{Const}^{Z,B} \cong (A \to B) \to Z \to Z \quad ,$$
$$\text{fmap}_{\text{Const}}(f^{:A \to B}) \triangleq (z^{:Z} \to z) = \text{id}^{:Z \to Z} \quad .$$

It is a constant function that ignores f and returns the identity $\text{id}^{:Z \to Z}$. The laws are satisfied:

$$\text{identity law :} \quad \text{fmap}_{\text{Const}}(\text{id}) = \text{id} \quad ,$$
$$\text{composition law :} \quad \text{fmap}_{\text{Const}}(f \fatsemi g) = \text{id} = \text{fmap}_{\text{Const}}(f) \fatsemi \text{fmap}_{\text{Const}}(g) = \text{id} \fatsemi \text{id} \quad .$$

The corresponding Scala code is

```
type Const[Z, A] = Z
def fmap[A, B](f: A => B): Const[Z, A] => Const[Z, B] = identity[Z]
```

The identity functor $\text{Id}^\bullet$ and the constant functor $\text{Const}^{Z,\bullet}$ are not often used: their `fmap` implementations are identity functions, and so they rarely provide useful functionality.

We have seen that type constructors with product types, such as $L^A \triangleq A \times A \times A$, are functors. The next construction (the **functor product**) explains why.

Statement 6.2.3.3 If $L^\bullet$ and $M^\bullet$ are two functors then the product $P^A \triangleq L^A \times M^A$ is also a functor.

Proof The `fmap` function for P is defined by

```
def fmap[A, B](f: A => B): (L[A], M[A]) => (L[B], M[B]) = {
  case (la, ma) => (la.map(f), ma.map(f))
}
```

The corresponding code notation is

$$f^{\uparrow P} \triangleq l^{:L^A} \times m^{:M^A} \to f^{\uparrow L}(l) \times f^{\uparrow M}(m) \quad .$$

Writing this code using the pipe ($\triangleright$) operation makes it somewhat closer to the Scala syntax:

$$(l^{:L^A} \times m^{:M^A}) \triangleright f^{\uparrow P} \triangleq (l \triangleright f^{\uparrow L}) \times (m \triangleright f^{\uparrow M}) \quad . \tag{6.12}$$

An alternative notation uses the **function product** symbol $\boxtimes$ defined by

$$p^{:A \to B} \boxtimes q^{:C \to D} : A \times C \to B \times D \quad ,$$
$$p \boxtimes q \triangleq a \times c \to p(a) \times q(c) \quad ,$$
$$(a \times c) \triangleright (p \boxtimes q) = (a \triangleright p) \times (b \triangleright q) \quad .$$

In this notation, the lifting for P is defined more concisely:

$$f^{\uparrow P} = f^{\uparrow L \times M} \triangleq f^{\uparrow L} \boxtimes f^{\uparrow M} \quad . \tag{6.13}$$

We need to verify the identity law and the composition law.
To verify the identity law for P, pipe an arbitrary value of type $L^A \times M^A$ into both sides of the law:

$$\begin{aligned}
\text{expect to equal } l \times m: \quad & (l^{:L^A} \times m^{:M^A}) \triangleright \mathrm{id}^{\uparrow P} \\
\text{definition of } f^{\uparrow P}: \quad & = (l \triangleright \underline{\mathrm{id}^{\uparrow L}}) \times (m \triangleright \underline{\mathrm{id}^{\uparrow M}}) \\
\text{identity laws for } L, M: \quad & = (\underline{l \triangleright \mathrm{id}}) \times (\underline{m \triangleright \mathrm{id}}) \\
\text{definition of id}: \quad & = l \times m \quad .
\end{aligned}$$

To verify the composition law for P, we need to show that

$$f^{\uparrow P} \fatsemi g^{\uparrow P} = (f \fatsemi g)^{\uparrow P} \quad .$$

Apply both sides of this equation to an arbitrary value of type $L^A \times M^A$:

$$\begin{aligned}
\text{expect to equal } (l \times m) \triangleright (f \fatsemi g)^{\uparrow P}: \quad & (l^{:L^A} \times m^{:M^A}) \triangleright \underline{f^{\uparrow P} \fatsemi g^{\uparrow P}} \\
\triangleright \text{ notation}: \quad & = \underline{(l^{:L^A} \times m^{:M^A}) \triangleright f^{\uparrow P}} \triangleright g^{\uparrow P} \\
\text{use Eq. (6.12)}: \quad & = \underline{((l \triangleright f^{\uparrow L}) \times (m \triangleright f^{\uparrow M}))} \triangleright g^{\uparrow P} \\
\text{use Eq. (6.12)}: \quad & = (l \triangleright \underline{f^{\uparrow L} \triangleright g^{\uparrow L}}) \times (m \triangleright \underline{f^{\uparrow M} \triangleright g^{\uparrow M}}) \\
\triangleright \text{ notation}: \quad & = (l \triangleright \underline{f^{\uparrow L} \fatsemi g^{\uparrow L}}) \times (m \triangleright \underline{f^{\uparrow M} \fatsemi g^{\uparrow M}}) \\
\text{composition laws for } L, M: \quad & = (l \triangleright (f \fatsemi g)^{\uparrow L}) \times (m \triangleright (f \fatsemi g)^{\uparrow M}) \\
\text{use Eq. (6.12)}: \quad & = (l \times m) \triangleright (f \fatsemi g)^{\uparrow P} \quad .
\end{aligned}$$

The calculations are shorter if we use the function product notation:

$$\begin{aligned}
\text{expect to equal } (f \fatsemi g)^{\uparrow P}: \quad & f^{\uparrow P} \fatsemi g^{\uparrow P} = (f^{\uparrow L} \boxtimes f^{\uparrow M}) \fatsemi (g^{\uparrow L} \boxtimes g^{\uparrow M}) \\
\text{composition of functions under } \boxtimes: \quad & = \underline{(f^{\uparrow L} \fatsemi g^{\uparrow L})} \boxtimes \underline{(f^{\uparrow M} \fatsemi g^{\uparrow M})} \\
\text{composition laws for } L \text{ and } M: \quad & = (f \fatsemi g)^{\uparrow L} \boxtimes (f \fatsemi g)^{\uparrow M} = (f \fatsemi g)^{\uparrow P} \quad .
\end{aligned}$$

For comparison, the same derivation using the Scala code syntax looks like this,

```
(( l, m )).map(f).map(g) == (( l.map(f), m.map(f) )).map(g)
  == (( l.map(f).map(g), m.map(f).map(g) ))
  == (( l.map(f andThen g), m.map(f andThen g) ))
```

assuming that the `.map` method is defined on pairs by Eq. (6.12),

```
(( l, m )).map(f) == (( l.map(f), m.map(f) ))
```

The proof written in the Scala syntax does not show the type constructors whose `.map` methods are used in each expression. For instance, it is not indicated that the two `.map` methods used in the expression `m.map(f).map(g)` belong to the *same* type constructor M and thus obey M's composition law. The code notation shows this more concisely and more clearly, helping us in reasoning:

$$m \triangleright f^{\uparrow M} \triangleright g^{\uparrow M} = m \triangleright f^{\uparrow M} \fatsemi g^{\uparrow M} = m \triangleright (f \fatsemi g)^{\uparrow M} \quad .$$

By the conventions of the pipe notation, we have

$$(x \triangleright f) \triangleright g = x \triangleright f \triangleright g = x \triangleright f \fatsemi g = x \triangleright (f \fatsemi g) = (f \fatsemi g)(x) = g(f(x)) \quad .$$

We will often use this notation in code derivations.

Statement 6.2.3.4 If P^A and Q^A are functors then $L^A \triangleq P^A + Q^A$ is a functor, with `fmap` defined by

```
def fmap[A, B](f: A => B): Either[P[A], Q[A]] => Either[P[B], Q[B]] = {
  case Left(pa)    => Left(fmap_P(f)(pa))   // Use fmap for P.
  case Right(qa)   => Right(fmap_Q(f)(qa))  // Use fmap for Q.
}
```

The functor $L^\bullet$ is the **functor co-product** of $P^\bullet$ and $Q^\bullet$. The code notation for the `fmap` function is

$$\text{fmap}_L(f^{:A\to B}) = f^{\uparrow L} \triangleq \left\| \begin{array}{c|cc} & P^B & Q^B \\ \hline P^A & f^{\uparrow P} & \mathbb{0} \\ Q^A & \mathbb{0} & f^{\uparrow Q} \end{array} \right\| .$$

Here we assume that lawful `fmap` functions are given for the functors P and Q.

Proof Omitting the type annotations, we write the code of $\text{fmap}_L(f)$ as

$$\text{fmap}_L(f) = f^{\uparrow L} = \left\| \begin{array}{cc} f^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & f^{\uparrow Q} \end{array} \right\| . \tag{6.14}$$

To verify the identity law, use Eq. (6.14) and the identity laws for P and Q:

$$\begin{aligned} \text{expect to equal id}: \quad \text{id}^{\uparrow L} &= \left\| \begin{array}{cc} \text{id}^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & \text{id}^{\uparrow Q} \end{array} \right\| = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & \text{id} \end{array} \right\| \\ \text{identity function in matrix notation}: \quad &= \text{id} \quad . \end{aligned}$$

To verify the composition law:

$$\begin{aligned} \text{expect to equal } (f \fatsemi g)^{\uparrow L}: \quad f^{\uparrow L} \fatsemi g^{\uparrow L} &= \left\| \begin{array}{cc} f^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & f^{\uparrow Q} \end{array} \right\| \fatsemi \left\| \begin{array}{cc} g^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & g^{\uparrow Q} \end{array} \right\| \\ \text{matrix composition}: \quad &= \left\| \begin{array}{cc} \underline{f^{\uparrow P} \fatsemi g^{\uparrow P}} & \mathbb{0} \\ \mathbb{0} & \underline{f^{\uparrow Q} \fatsemi g^{\uparrow Q}} \end{array} \right\| \\ \text{composition laws for } P, Q: \quad &= \left\| \begin{array}{cc} (f \fatsemi g)^{\uparrow P} & \mathbb{0} \\ \mathbb{0} & (f \fatsemi g)^{\uparrow Q} \end{array} \right\| = (f \fatsemi g)^{\uparrow L} \quad . \end{aligned}$$

The last two statements show that any type constructor built up only using primitive types, type parameters, products and co-products, such as $L^A \triangleq \mathbb{1} + (\text{String} + A) \times A \times \text{Int} + A$, is a functor. Functors of this kind are called **polynomial functors** because they are analogous to ordinary arithmetic polynomial functions of a variable A. The type notation with its symbols $(+, \times)$ makes this analogy visually clear.

Implementing `fmap` for a polynomial functor is straightforward: `fmap` replaces each occurrence of the a value of type A by the corresponding value of type B, leaving constant types unchanged and keeping the order of parts in all products and disjunctive types. Previously, our implementations of `fmap` for various type constructors (such as shown in Example 6.1.9.2) were guided by the idea of preserving information. Statements 6.2.3.3–6.2.3.4 explain why those implementations of the `fmap` are correct (i.e. obey the functor laws).

The next construction shows when a function type is a functor: the argument of the function must be a contrafunctor.

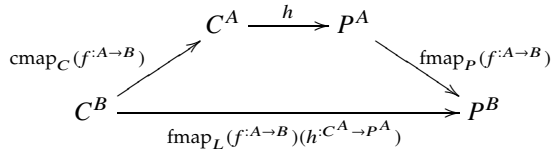
Statement 6.2.3.5 If C is a contrafunctor and P is a functor then $L^A \triangleq C^A \to P^A$ is a functor, called an **functor exponential**, with `fmap` defined by

$$\text{fmap}_L^{A,B}(f^{:A\to B}) : (C^A \to P^A) \to C^B \to P^B \quad ,$$
$$\text{fmap}_L(f^{:A\to B}) = f^{\uparrow L} \triangleq h^{:C^A \to P^A} \to f^{\downarrow C} \fatsemi h \fatsemi f^{\uparrow P} \quad . \tag{6.15}$$

The corresponding Scala code is

```
def fmap_L[A, B](f: A => B)(h: C[A] => P[A]): C[B] => P[B] = {
  cmap_C(f) andThen h andThen fmap_P(f)
}
```

A type diagram for fmap_L can be drawn as



Proof Since the types are already checked, we can use Eq. (6.15) without type annotations,

$$h \triangleright f^{\uparrow L} = f^{\downarrow C} \fatsemi h \fatsemi f^{\uparrow P} \quad . \tag{6.16}$$

To verify the identity law for L, show that $\text{id}^{\uparrow L}(h) = h$:

$$\begin{aligned}
\text{expect to equal } h: \quad & h \triangleright \text{id}^{\uparrow L} \\
\text{definition (6.16) of } {}^{\uparrow L}: \quad & = \underline{\text{id}^{\downarrow C}} \fatsemi h \fatsemi \underline{\text{id}^{\uparrow P}} \\
\text{identity laws for } C \text{ and } P: \quad & = \underline{\text{id}} \fatsemi h \fatsemi \underline{\text{id}} \\
\text{definition of id}: \quad & = h \quad .
\end{aligned}$$

To verify the composition law for L:

$$\begin{aligned}
\text{expect to equal } h \triangleright f^{\uparrow L} \fatsemi g^{\uparrow L}: \quad & h \triangleright (f \fatsemi g)^{\uparrow L} \\
\text{definition (6.16) of } {}^{\uparrow L}: \quad & = \underline{(f \fatsemi g)^{\downarrow C}} \fatsemi h \fatsemi \underline{(f \fatsemi g)^{\uparrow P}} \\
\text{composition laws for } C \text{ and } P: \quad & = g^{\downarrow C} \fatsemi \underline{f^{\downarrow C}} \fatsemi h \fatsemi \underline{f^{\uparrow P}} \fatsemi g^{\uparrow P} \\
\text{definition (6.16) of } {}^{\uparrow L}: \quad & = \underline{g^{\downarrow C}} \fatsemi (h \triangleright f^{\uparrow L}) \fatsemi \underline{g^{\uparrow P}} \\
\text{definition (6.16) of } {}^{\uparrow L}: \quad & = \left(h \triangleright f^{\uparrow L}\right) \triangleright g^{\uparrow L} = h \triangleright f^{\uparrow L} \fatsemi g^{\uparrow L} \quad .
\end{aligned}$$

It is important for this proof that C is a contrafunctor and so the order of lifted function compositions is reversed, $(f \fatsemi g)^{\downarrow C} = g^{\downarrow C} \fatsemi f^{\downarrow C}$. If C were a functor, the proof would not work: we would have obtained $f^{\uparrow C} \fatsemi g^{\uparrow C}$ instead of $g^{\downarrow C} \fatsemi f^{\downarrow C}$, and we would not be able to group $f^{\downarrow C} \fatsemi h \fatsemi f^{\uparrow P}$ together (the order of composition cannot be permuted for arbitrary functions f, g).

Examples of functors obtained via the exponential construction are $L^A \triangleq Z \to A$ (with the contrafunctor C^A chosen as the constant contrafunctor Z, where Z is a fixed type) and $L^A \triangleq (A \to Z) \to A$ (with the contrafunctor $C^A \triangleq A \to Z$). Statement 6.2.3.5 generalizes those examples to arbitrary contrafunctors C^A used as arguments of function types.

In a similar way, one can prove that $P^A \to C^A$ is a contrafunctor (Exercise 6.3.1.2). Together with the results of Statements 6.2.3.3–6.2.3.5, this establishes the rules of reasoning about covariance and contravariance of type parameters in arbitrary type expressions. Every function arrow flips the variance from covariant to contravariant and back. For instance, the identity functor $L^A \triangleq A$ is covariant in A, while $A \to Z$ is contravariant in A, and $(A \to Z) \to Z$ is again covariant in A. As we

have seen, $A \to A \to Z$ is contravariant in A, so any number of curried arrows count as one in this consideration (and, in any case, $A \to A \to Z \cong A \times A \to Z$). Products and disjunctions do not change variance, so $(A \to Z_1) \times (A \to Z_2) + (A \to Z_3)$ is still contravariant in A. We will see more examples of such reasoning below (Section 6.2.5).

The remaining constructions set a type parameter to another type constructor. The **functor composition** P^{Q^A}, written in Scala as `P[Q[A]]`, is analogous to a function composition such as $f(g(x))$ except for type constructors. Viewed in this way, type constructors are **type-level functions** (i.e. maps on the set of types). For this reason, functor composition can be also denoted by $P \circ Q$, similar to the function composition $f \circ g$.

An example of functor composition in Scala is `List[Option[A]]`. Since both `List` and `Option` have a `.map` method, we may write code such as

```
val p: List[Option[Int]] = List(Some(1), None, Some(2), None, Some(3))

scala> p.map(_.map(x => x + 10))
res0: List[Option[Int]] = List(Some(11), None, Some(12), None, Some(13))
```

The code `p.map(_.map(f))` lifts an $f^{:A \to B}$ into a function of type `List[Option[A]] => List[Option[B]]`. In this way, we may perform the `.map` operation on the composite data type `List[Option[_]]`.

The next statement shows that this code always produces a lawful `.map` function. In other words, the composition of functors is always a functor.

Statement 6.2.3.6 If P^A and Q^A are functors then $L^A \triangleq P^{Q^A}$ is also a functor, with `fmap` defined by

```
def fmap_L[A, B](f: A => B): P[Q[A]] => P[Q[B]] = fmap_P(fmap_Q(f))
```

Here we assumed that the functions fmap_P and fmap_Q are known and satisfy the functor laws.

In the code notation, fmap_L is written equivalently as

$$\begin{aligned}
\text{type signature :} \quad & \text{fmap}_L : f^{:A \to B} \to P^{Q^A} \to P^{Q^B} \quad , \\
\text{implementation :} \quad & \text{fmap}_L(f) \triangleq \text{fmap}_P(\text{fmap}_Q(f)) \quad , \\
\text{equivalent code :} \quad & \text{fmap}_L \triangleq \text{fmap}_Q \fatsemi \text{fmap}_P \quad , \qquad (6.17) \\
\text{in a shorter notation :} \quad & f^{\uparrow L} \triangleq (f^{\uparrow Q})^{\uparrow P} \triangleq f^{\uparrow Q \uparrow P} \quad . \qquad (6.18)
\end{aligned}$$

Proof To verify the identity law for L, use the identity laws for P and Q:

$$\text{id}^{\uparrow L} = (\underline{\text{id}^{\uparrow Q}})^{\uparrow P} = \underline{\text{id}^{\uparrow P}} = \text{id} \quad .$$

To verify the composition law for L, use the composition laws for P and Q:

$$(f \fatsemi g)^{\uparrow L} = ((\underline{f \fatsemi g})^{\uparrow Q})^{\uparrow P} = (\underline{f^{\uparrow Q} \fatsemi g^{\uparrow Q}})^{\uparrow P} = f^{\uparrow Q \uparrow P} \fatsemi g^{\uparrow Q \uparrow P} \quad .$$

Finally, we consider recursive data types such as lists and trees (Section 3.3). It is helpful to use the type notation for reasoning about those types. The list type,

```
sealed trait List[A]
final case class Empty()                        extends List[A]
final case class Head[A](head: A, tail: List[A]) extends List[A]
```

is written in type notation as

$$\text{List}^A \triangleq \mathbb{1} + A \times \text{List}^A \quad .$$

The binary tree type,

```
sealed trait Tree2[A]
final case class Leaf[A](a: A)                       extends Tree2[A]
final case class Branch[A](x: Tree2[A], y: Tree2[A]) extends Tree2[A]
```

Description	Type definition	Bifunctor $S^{A,R}$
list	$L^A \triangleq \mathbb{1} + A \times L^A$	$S^{A,R} \triangleq \mathbb{1} + A \times R$
non-empty list	$\text{NEL}^A \triangleq A + A \times \text{NEL}^A$	$S^{A,R} \triangleq A + A \times R$
list of odd length	$L^A \triangleq A + A \times A \times L^A$	$S^{A,R} \triangleq A + A \times A \times R$
binary tree	$L^A \triangleq A + L^A \times L^A$	$S^{A,R} \triangleq A + R \times R$
rose tree	$L^A \triangleq A + \text{NEL}^{L^A}$	$S^{A,R} \triangleq A + \text{NEL}^R$
regular-shaped tree	$L^A \triangleq A + L^{A \times A}$	not possible
abstract syntax tree	$L^A \triangleq P^A + Q^{L^A}$	$S^{A,R} = P^A + Q^R$

Table 6.3: Recursive disjunctive types defined using type equations.

is defined by

$$\text{Tree}_2^A \triangleq A + \text{Tree}_2^A \times \text{Tree}_2^A \quad .$$

These definitions of recursive types look like "type equations". We can generalize these examples to a recursive definition

$$L^A \triangleq S^{A,L^A} \quad , \tag{6.19}$$

where $S^{A,R}$ is a suitably chosen type constructor with two type parameters A, R. If the type constructor $S^{\bullet,\bullet}$ is given, the Scala code defining $L^\bullet$ can be written as

```
type S[A, R] = ... // Must be defined previously as type alias, class, or trait.
final case class L[A](x: S[A, L[A]])
```

We must use a case class to define `L` because Scala does not support recursive type aliases:

```
scala> type L[A] = Either[A, L[A]]
<console>:14: error: illegal cyclic reference involving type L
       type L[A] = Either[A, L[A]]
                           ^

scala> final case class L[A](x: Either[A, L[A]])
defined class L
```

Table 6.3 summarizes our previous examples of recursive disjunctive types and shows the relevant choices of $S^{A,R}$, which turns out to be always a bifunctor. For abstract syntax trees, the functors $P^\bullet$ and $Q^\bullet$ must be given; they specify the available shapes of leaves and branches respectively.

We will now prove that Eq. (6.19) always defines a functor when $S^{\bullet,\bullet}$ is a bifunctor.

Statement 6.2.3.7 If $S^{A,B}$ is a bifunctor (a functor with respect to both type parameters A and B) then the recursively defined type constructor L^A is a functor,

$$L^A \triangleq S^{A,L^A} \quad .$$

The `fmap` method for L is a recursive function implemented as

$$\text{fmap}_L(f^{:A\to B}) \triangleq \text{bimap}_S(f)(\text{fmap}_L(f)) \quad . \tag{6.20}$$

The corresponding Scala code is

```
final case class L[A](x: S[A, L[A]]) // The type constructor S[_, _] must be defined previously.

def bimap_S[A, B, C, D](f: A => C)(g: B => D): S[A, B] => S[C, D] = ???      // Must be defined.

def fmap_L[A, B](f: A => B): L[A] => L[B] = { case L(x) =>
    val newX: S[B, L[B]] = bimap_S(f)(fmap_L(f))(x)          // Recursive call to fmap_L.
```

```
    L(newX)           // Need to wrap the value of type S[B, L[B]] into the type constructor L.
}
```

Proof This is the first time we prove a property of a recursive function (fmap_L). Since the implementation of fmap_L is recursive, its code contains some calls to itself. When we are proving some property of fmap_L, we may *assume* in the proof that the property already holds for all recursive calls of fmap_L. As long as recursion is not infinite, this kind of proof will be equivalent to a proof by induction: The base case corresponds to the final non-recursive evaluation within the code of fmap_L, and the inductive assumption states that the recursive calls to fmap_L already satisfy the property we are trying to prove.

For clarity, we add an overline to recursive calls in the code:

$$\text{fmap}_L(f) \triangleq \text{bimap}_S(f)(\overline{\text{fmap}_L}(f)) \quad .$$

To prove the identity law:

$$\begin{aligned}
\text{expect to equal id}: \quad & \text{fmap}_L(\text{id}) \\
\text{definition of } \text{fmap}_L: \quad & = \text{bimap}_S(\text{id})(\overline{\text{fmap}_L}(\text{id})) \\
\text{(inductive assumption) the law holds for } \overline{\text{fmap}_L}: \quad & = \text{bimap}_S(\text{id})(\text{id}) \\
\text{identity law for } S: \quad & = \text{id} \quad .
\end{aligned}$$

To prove the composition law:

$$\begin{aligned}
\text{expect to equal fmap}_L(f \fatsemi g): \quad & \text{fmap}_L(f) \fatsemi \text{fmap}_L(g) \\
\text{definition of } \text{fmap}_L: \quad & = \text{bimap}_S(f)(\overline{\text{fmap}_L}(f)) \fatsemi \text{bimap}_S(g)(\overline{\text{fmap}_L}(g)) \\
\text{composition law for } S: \quad & = \text{bimap}_S(f \fatsemi g)(\overline{\text{fmap}_L}(f) \fatsemi \overline{\text{fmap}_L}(g)) \\
\text{inductive assumption}: \quad & = \text{bimap}_S(f \fatsemi g)(\overline{\text{fmap}_L}(f \fatsemi g)) \\
\text{definition of } \text{fmap}_L: \quad & = \text{fmap}_L(f \fatsemi g) \quad .
\end{aligned}$$

For the regular-shaped binary tree, the construction (6.19) is insufficient: no bifunctor S^{A,L^A} can replace the type argument A in L^A to obtain $L^{A\times A}$. To see that, consider that S^{A,L^A} is an application of a type-level function $S^{\bullet,\bullet}$ to its two type parameters, which are set to A and L^A. In Scala syntax, S^{A,L^A} is written as `S[A,L[A]]`. No matter how we define the type constructor S, the resulting type expression `S[A,L[A]]` will always use the type constructor `L` as `L[A]` and not as `L[(A,A)]`.

To describe regular-shaped trees, we need to generalize the construction by adding another arbitrary functor, $P^\bullet$, in the type argument of $L^\bullet$:

$$L^A \triangleq S^{A,L^{P^A}} \quad . \tag{6.21}$$

Regular-shaped trees are defined by Eq. (6.21) with $S^{A,R} \triangleq A + R$ and $P^A \triangleq A \times A$. The Scala code for these definitions is

```
type S[A, R] = Either[A, R]
type P[A] = (A, A)
final case class L[A](s: S[A, L[P[A]]])  // Equivalently, case class L[A](s: Either[A, L[(A, A)]])
```

Different choices of P will define regular-shaped trees with different kinds of branching.

6.2.4 Type constructions for contrafunctors

The previous section performed **structural analysis** for functors: a systematic search for type constructions that create new functors. *Mutatis mutandis*, similar constructions work for contrafunctors,

Construction	Type notation	Comment
tuple	$C^A \triangleq P^A \times Q^A$	the product contrafunctor; P and Q must be contrafunctors
disjunctive type	$C^A \triangleq P^A + Q^A$	the co-product contrafunctor; P and Q must be contrafunctors
function type	$C^A \triangleq L^A \to H^A$	the exponential contrafunctor; L is a functor and H a contrafunctor
primitive type	$C^A \triangleq Z$	the constant contrafunctor; Z is a fixed type
type constructor	$C^A \triangleq P^{Q^A}$	the composition; P is a functor and Q a contrafunctor (or vice versa)
type recursion	$C^A \triangleq S^{A,C^A}$	$S^{A,B}$ must be a contrafunctor w.r.t. A and functor w.r.t. B

Table 6.4: Type constructions defining a contrafunctor C^A.

as shown in Table 6.4. One difference with respect to Table 6.2 is that the identity type constructor, $L^A \triangleq A$, is missing because it is a functor and not a contrafunctor. However, the constant type constructor, $L^A \triangleq Z$, is a functor and a contrafunctor at the same time.

Let us now prove the validity of some of these constructions.

Statement 6.2.4.1 If Z is any fixed type, the constant type constructor $C^A \triangleq Z$ is a contrafunctor (the **constant contrafunctor**) whose `cmap` returns an identity function of type $Z \to Z$:

```
type Const[Z, A] = Z
def cmap[Z, A, B](f: B => A): Const[Z, A] => Const[Z, B] = identity[Z]
```

Proof All laws hold because `cmap` returns an identity function:

$$\text{identity law}: \quad \text{cmap}\,(\text{id}) = \text{id} \quad ,$$
$$\text{composition law}: \quad \text{cmap}\,(f) \fatsemi \text{cmap}\,(g) = \text{id} \fatsemi \text{id} = \text{id} = \text{cmap}\,(g \fatsemi f) \quad .$$

Statement 6.2.4.2 If P^A is a functor and Q^A is a contrafunctor then $L^A \triangleq P^{Q^A}$ is a contrafunctor with `cmap` defined by

```
def cmap[A, B](f: B => A): P[Q[A]] => P[Q[B]] = fmap_P(cmap_Q(f))
```

where lawful implementations of fmap_P and cmap_Q are assumed to be given.

Proof Convert the Scala implementation of cmap_L into the code notation:

$$\text{cmap}_L(f^{:B\to A}) \triangleq \text{fmap}_P(\text{cmap}_Q(f)) \quad .$$

It is easier to reason about this function if we rewrite it as

$$f^{\downarrow L} \triangleq (f^{\downarrow Q})^{\uparrow P} \quad .$$

The contrafunctor laws for L are then proved like this:

$$\text{identity law}: \quad \text{id}^{\downarrow L} = (\text{id}^{\downarrow Q})^{\uparrow P} = \text{id}^{\uparrow P} = \text{id} \quad .$$
$$\text{composition law}: \quad f^{\downarrow L} \fatsemi g^{\downarrow L} = (f^{\downarrow Q})^{\uparrow P} \underline{\fatsemi} (g^{\downarrow Q})^{\uparrow P}$$
$$\text{use } P\text{'s composition law}: \quad = \underline{(f^{\downarrow Q} \fatsemi g^{\downarrow Q})^{\uparrow P}} = \underline{((g \fatsemi f)^{\downarrow Q})^{\uparrow P}} = (g \fatsemi f)^{\downarrow L} \quad .$$

Finally, the recursive construction works for contrafunctors, except that the type constructor $S^{A,R}$ must be a contrafunctor in A (but still a functor in R). An example of such a type constructor is

$$S^{A,R} \triangleq (A \to \text{Int}) + R \times R \quad . \tag{6.22}$$

The type constructor $S^{\bullet,\bullet}$ is not a bifunctor because it is contravariant in its first type parameter; so we cannot define a `bimap` function for it. However, we can define an analogous function called `xmap`, with the type signature

```
def xmap[A, B, Q, R](f: B => A)(g: Q => R): S[A, Q] => S[B, R]
```

$$\text{xmap}_S : (B \to A) \to (Q \to R) \to S^{A,Q} \to S^{B,R} \quad ,$$
$$\text{xmap}_S(f^{:B\to A})(g^{:Q\to R}) \triangleq \text{fmap}_{S^A,\bullet}(g) \fatsemi \text{cmap}_{S^\bullet,R}(f) \quad .$$

The function `xmap` should obey the laws of identity and composition:

$$\text{identity law :} \quad \text{xmap}_S(\text{id})(\text{id}) = \text{id} \quad , \tag{6.23}$$
$$\text{composition law :} \quad \text{xmap}_S(f_1)(g_1) \fatsemi \text{xmap}_S(f_2)(g_2) = \text{xmap}_S(f_2 \fatsemi f_1)(g_1 \fatsemi g_2) \quad . \tag{6.24}$$

These laws are similar to the identity and composition laws for bifunctors (Section 6.2.2), except for inverting the order of the composition $(f_2 \fatsemi f_1)$. The laws hold automatically whenever the functor and contrafunctor methods for S ($\text{fmap}_{S^A,\bullet}$ and $\text{cmap}_{S^\bullet,R}$) are fully parametric. We omit the details since they are quite similar to what we saw in Section 6.2.2 for bifunctors.

If we define a type constructor $L^\bullet$ using the recursive equation

$$L^A \triangleq S^{A,L^A} \triangleq (A \to \text{Int}) + L^A \times L^A \quad ,$$

we obtain a contrafunctor in the shape of a binary tree whose leaves are functions of type $A \to \text{Int}$. The next statement shows that recursive type equations of this kind always define contrafunctors.

Statement 6.2.4.3 If $S^{A,R}$ is a contrafunctor with respect to A and a functor with respect to R then the recursively defined type constructor C^A is a contrafunctor,

$$C^A \triangleq S^{A,C^A} \quad .$$

Given the functions $\text{cmap}_{S^\bullet,R}$ and $\text{fmap}_{S^A,\bullet}$ for S, we implement cmap_C as

$$\text{cmap}_C(f^{:B\to A}) : C^A \to C^B \cong S^{A,C^A} \to S^{B,C^B} \quad ,$$
$$\text{cmap}_C(f^{:B\to A}) \triangleq \text{xmap}_S(f)(\overline{\text{cmap}_C}(f)) \quad .$$

The corresponding Scala code can be written as

```
final case class C[A](x: S[A, C[A]]) // The type constructor S[_, _] must be defined previously.

def xmap_S[A,B,Q,R](f: B => A)(g: Q => R): S[A, Q] => S[B, R] = ???          // Must be defined.

def cmap_C[A, B](f: B => A): C[A] => C[B] = { case C(x) =>
  val sbcb: S[B, C[B]] = xmap_S(f)(cmap_C(f))(x)        // Recursive call to cmap_C.
  C(sbcb)              // Need to wrap the value of type S[B, C[B]] into the type constructor C.
}
```

Proof The code of `cmap` is recursive, and the recursive call is marked by an overline:

$$\text{cmap}_C(f) \triangleq f^{\downarrow C} \triangleq \text{xmap}_S(f)(\overline{\text{cmap}_C}(f)) \quad .$$

To verify the identity law:

$$\begin{aligned} \text{expect to equal id :} \quad & \text{cmap}_C(\text{id}) = \text{xmap}_S(\text{id})(\underline{\overline{\text{cmap}_C}(\text{id})}) \\ \text{inductive assumption :} \quad & = \text{xmap}_S(\text{id})(\text{id}) \\ \text{identity law for } \text{xmap}_S : \quad & = \text{id} \quad . \end{aligned}$$

To verify the composition law:

$$\begin{aligned} \text{expect to equal } (g^{\downarrow C} \fatsemi f^{\downarrow C}) : \quad & (f^{:D\to B} \fatsemi g^{:B\to A})^{\downarrow C} = \text{xmap}_S(f \fatsemi g)(\underline{\overline{\text{cmap}_C}(f \fatsemi g)}) \\ \text{inductive assumption :} \quad & = \text{xmap}_S(f \fatsemi g)(\overline{\text{cmap}_C}(g) \fatsemi \overline{\text{cmap}_C}(f))) \\ \text{composition laws for } \text{xmap}_S : \quad & = \text{xmap}_S(g)(\overline{\text{cmap}_C}(g)) \fatsemi \text{xmap}_S(f)(\overline{\text{cmap}_C}(f)) \\ \text{definition of } {}^{\downarrow C} : \quad & = g^{\downarrow C} \fatsemi f^{\downarrow C} \quad . \end{aligned}$$

6.2.5 Solved examples: How to recognize functors and contrafunctors

Sections 6.2.3 and 6.2.4 describe how functors and contrafunctors are built from other type expressions. We can see from Tables 6.2 and 6.4 that *every* one of the six type constructions (Section 5.1.2) gives either a new functor or a new contrafunctor. So, we expect to be able to decide, for *every* type expression built from primitive types, type parameters, product types, sum types, function (exponential) types, and type parameter substitutions, whether it is a functor or a contrafunctor. The decision algorithm is based on the results shown in Tables 6.2 and 6.4:

- Primitive types $\mathbb{1}$, `Int`, `String`, etc. can be viewed both as constant functors and as constant contrafunctors (since they do not contain type parameters).
- Polynomial type expressions (not containing any function arrows) are always functors with respect to every type parameter. Equivalently, we may say that all polynomial type constructors are covariant in all their type parameters. For example, the type expression $A \times B + (A + \mathbb{1} + B) \times A \times C$ is covariant in each of the type parameters A, B, C.
- Type parameters to the right of a function arrow are in a covariant position. For example, $\text{Int} \to A$ is covariant in A.
- Each time a type parameter is placed to the left of an *uncurried* function arrow $\to$, the variance is reversed: covariant becomes contravariant and vice versa. For example,

$$\begin{aligned} \text{this is covariant in } A: &\quad \mathbb{1} + A \times A \quad , \\ \text{this is contravariant in } A: &\quad (\mathbb{1} + A \times A) \to \text{Int} \quad , \\ \text{this is covariant in } A: &\quad ((\mathbb{1} + A \times A) \to \text{Int}) \to \text{Int} \quad , \\ \text{this is contravariant in } A: &\quad (((\mathbb{1} + A \times A) \to \text{Int}) \to \text{Int}) \to \text{Int} \quad . \end{aligned}$$

- Repeated curried function arrows work as one arrow: $A \to \text{Int}$ is contravariant in A, and $A \to A \to A \to \text{Int}$ is still contravariant in A. This is so because the type $A \to A \to A \to \text{Int}$ is equivalent to $A \times A \times A \to \text{Int}$, which is of the form $F^A \to \text{Int}$ with a type constructor $F^A \triangleq A \times A \times A$. Exercise 6.3.1.1 will show that $F^A \to \text{Int}$ is contravariant in A.
- Nested type constructors combine their variances: e.g. if we know that F^A is contravariant in A then $F^{A \to \text{Int}}$ is covariant in A, while $F^{A \times A \times A}$ is contravariant in A.

For any exponential-polynomial type expression, such as Eq. (6.7),

$$Z^{A,R} \triangleq ((A \to A \to R) \to R) \times A + (\mathbb{1} + R \to A + \text{Int}) + A \times A \times \text{Int} \times \text{Int} \quad ,$$

we can mark the position of each type parameter as either covariant (+) or contravariant (−), according to the number of nested function arrows:

$$((\underset{+}{A} \to \underset{+}{A} \to \underset{-}{R}) \to \underset{+}{R}) \times \underset{+}{A} + (\mathbb{1} + \underset{-}{R} \to \underset{+}{A} + \text{Int}) + \underset{+}{A} \times \underset{+}{A} \times \text{Int} \times \text{Int} \quad .$$

We find that A is always in covariant positions, while R is sometimes in covariant and sometimes in contravariant positions. So, we expect that $Z^{A,R}$ is a functor with respect to A, but not a functor (nor a contrafunctor) with respect to R.

To show that $Z^{A,R}$ is indeed a functor in the parameter A, we need to implement a suitable `map` method and verify that the functor laws hold. To do that from scratch, we could use the techniques explained in this and the previous chapters: starting from the type signature

$$\text{map}_Z : Z^{A,R} \to (A \to B) \to Z^{B,R} \quad ,$$

we could derive a fully parametric, information-preserving implementation of `map`. We could then look for equational proofs of the identity and composition laws for that `map` function. This would require a lot of work for a complicated type constructor such as $Z^{A,R}$.

However, that work can be avoided if we find a way of building up $Z^{A,R}$ step by step via the known functor and contrafunctor constructions. Each step automatically provides both a fragment of the code of `map` and a proof that the functor laws hold up to that step. In this way, we will avoid the need to look for an implementation of `map` and proofs of laws for each new functor and contrafunctor. The next examples illustrate the procedure for a simpler type constructor.

Example 6.2.5.1 Rewrite this Scala definition in the type notation and decide whether it is covariant or contravariant with respect to each type parameter:

```
final case class G[A, Z](p: Either[Int, A], q: Option[Z => Int => Z => (Int, A)])
```

Solution The type notation is

$$G^{A,Z} \triangleq (\text{Int} + A) \times (\mathbb{1} + (Z \to \text{Int} \to Z \to \text{Int} \times A)) \quad .$$

Mark the covariant and the contravariant positions in this type expression:

$$(\text{Int} + \underset{+}{A}) \times (\mathbb{1} + (\underset{-}{Z} \to \text{Int} \to \underset{-}{Z} \to \text{Int} \times \underset{+}{A})) \quad .$$

All Z positions in the sub-expression $Z \to \text{Int} \to Z \to \text{Int} \times A$ are contravariant since the function arrows are curried rather than nested. We see that A is always in covariant positions (+) while Z is always in contravariant positions (−). It follows that $G^{A,Z}$ is covariant in A and contravariant in Z.

Example 6.2.5.2 Use known functor constructions to implement the `map` method with respect to `A` for the type `G[A, Z]` from Example 6.2.5.1.

Solution We need to build $G^{A,Z}$ via step-by-step constructions that start from primitive types and type parameters. At the top level of its type expression, $G^{A,Z}$ is a product type, so we begin by using the "functor product" construction (Statement 6.2.3.3),

$$G^{A,Z} \cong G_1^A \times G_2^{A,Z} \quad ,$$
$$\text{where } G_1^A \triangleq \text{Int} + A \quad \text{and} \quad G_2^{A,Z} \triangleq \mathbb{1} + (Z \to \text{Int} \to Z \to \text{Int} \times A) \quad .$$

We continue with G_1^A, which is a co-product of Int (a constant functor) and A (the identity functor). The constant functor and the identity functor have lawful `map` implementations that are already known (Statements 6.2.3.1–6.2.3.2). Now, the "functor co-product" construction (Statement 6.2.3.3) produces a `map` implementation for G_1^A together with a proof that it satisfies the functor laws:

$$\text{fmap}_{G_1}(f^{:A\to B}) = f^{\uparrow G_1} \triangleq \begin{array}{c|c||cc} & & \text{Int} & B \\ \hline \text{Int} & & \text{id} & \mathbb{0} \\ A & & \mathbb{0} & f \end{array} \quad .$$

Turning our attention to $G_2^{A,Z}$, we find that it is a disjunctive type containing a curried function type that ultimately returns the product type $\text{Int} \times A$. This tells us to use the functor constructions "co-product", the "exponential", and the "product". Write down the functor constructions needed at each step as we decompose $G_2^{A,Z}$:

$$G_2^{A,Z} \triangleq \mathbb{1} + (Z \to \text{Int} \to Z \to \text{Int} \times A) \quad .$$
$$\text{co-product}: \quad G_2^{A,Z} \cong \mathbb{1} + G_3^{A,Z} \quad \text{where } G_3^{A,Z} \triangleq Z \to \text{Int} \to Z \to \text{Int} \times A \quad .$$
$$\text{exponential}: \quad G_3^{A,Z} \cong Z \to G_4^{A,Z} \quad \text{where } G_4^{A,Z} \triangleq \text{Int} \to Z \to \text{Int} \times A \quad .$$
$$\text{exponential}: \quad G_4^{A,Z} \cong \text{Int} \to G_5^{A,Z} \quad \text{where } G_5^{A,Z} \triangleq Z \to \text{Int} \times A \quad .$$
$$\text{exponential}: \quad G_5^{A,Z} \cong Z \to G_6^A \quad \text{where } G_6^A \triangleq \text{Int} \times A \quad .$$
$$\text{product}: \quad G_6^A \cong \text{Int} \times A \cong \text{Const}^{\text{Int},A} \times \text{Id}^A \quad .$$

Each of the type constructors G_1, ..., G_6 is a functor in A because all of the functor constructions preserve functor laws. Therefore, $G^{A,Z}$ is a functor in A.

It remains to derive the code for the `fmap` method of G. Each of the functor constructions combines the `fmap` implementations from previously defined functors into a new `map` implementation, so we just need to combine the code fragments in the order of constructions. For brevity, we will use the notations $f^{\uparrow L} \triangleq \text{fmap}_L(f)$ and $x \triangleright f^{\uparrow L}$ instead of the Scala code `x.map(f)` throughout the derivations:

$$\text{product :} \quad G^{A,Z} \cong G_1^A \times G_2^{A,Z} \quad , \quad (g_1 \times g_2) \triangleright f^{\uparrow G} = (g_1 \triangleright f^{\uparrow G_1}) \times (g_2 \triangleright f^{\uparrow G_2}) \quad .$$

$$\text{co-product :} \quad G_1^A \triangleq \text{Int} + A \quad , \quad f^{\uparrow G_1} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & f \end{array} \right| \quad .$$

$$\text{co-product :} \quad G_2^{A,Z} \triangleq \mathbb{1} + G_3^{A,Z} \quad , \quad f^{\uparrow G_2} = \left\| \begin{array}{cc} \text{id} & \mathbb{0} \\ \mathbb{0} & f^{\uparrow G_3} \end{array} \right| \quad .$$

$$\text{exponential :} \quad G_3^{A,Z} \triangleq Z \to G_4^{A,Z} \quad , \quad g_3 \triangleright f^{\uparrow G_3} = g_3 \mathbin{⨟} f^{\uparrow G_4} = (z^{:Z} \to g_3(z) \triangleright f^{\uparrow G_4}) \quad .$$

Applying the exponential construction three times, we finally obtain

$$G_3^{A,Z} \triangleq Z \to \text{Int} \to Z \to G_6^A \quad , \quad g_3 \triangleright f^{\uparrow G_3} = z_1^{:Z} \to n^{:\text{Int}} \to z_2^{:Z} \to g_3(z_1)(n)(z_2) \triangleright f^{\uparrow G_6} \quad .$$

$$G_6^A \triangleq \text{Int} \times A \quad , \quad (i \times a) \triangleright f^{\uparrow G_6} = i \times f(a) \quad .$$

We can now write the corresponding Scala code for fmap_G:

```
def fmap_G[A, B, Z](f: A => B): G[A, Z] => G[B, Z] = { case G(p, q) =>
  val newP: Either[Int, B] = p.map(f)      // Use the standard .map method for Either[Int, A].
  val newQ: Option[Z => Int => Z => (Int, B)]  = q.map { // Use the .map method for Option[_].
    (g3: Z => Int => Z => (Int, A)) =>
      z1 => n => z2 =>                     // The code of .map for G_3.
        val (i, a) = g3(z1)(n)(z2)
        (i, f(a))                          // The code of .map for G_6.
    }
  G(newP, newQ)                            // The code of .map for G_1.
}
```

6.3 Summary

What tasks can we perform with the techniques of this chapter?

- Quickly decide if a given type constructor is a functor, a contrafunctor, or neither.
- Implement a `fmap` or a `cmap` function that satisfies the appropriate laws.
- Use constructions to derive the correct code of `fmap` or `cmap` without trial and error.
- Use functor blocks to manipulate data wrapped in functors with more readable code.

6.3.1 Exercises: Functor and contrafunctor constructions

Exercise 6.3.1.1 If H^A is a contrafunctor and L^A is a functor, show that $C \triangleq L^A \to H^A$ is a contrafunctor with the `cmap` method defined by the following code:

```
def cmap[A, B](f: B => A)(c: L[A] => H[A]): L[B] => H[B] = {
  lb: L[B] => cmap_H(f)(c(fmap_L(f)(lb)))
}
```

Here, `cmap_H` and `fmap_L` are the methods already defined for H and L.

Exercise 6.3.1.2 In each case, implement the required `fmap` or `cmap` function for the new type constructor L and prove that the appropriate laws hold. Write the implementations both in Scala and in the code notation. Assume that the given type constructors F and G already satisfy the appropriate laws.

(a) $L^A \triangleq F^A \times G^A$ is a contrafunctor if F^A and G^A are contrafunctors.
(b) $L^A \triangleq F^A + G^A$ is a contrafunctor if F^A and G^A are contrafunctor.
(c) $L^A \triangleq F^{G^A}$ is a functor when both F and G are contrafunctors.
(d) $L^A \triangleq F^{G^A}$ is a contrafunctor when F is a contrafunctor and G is a functor.

Exercise 6.3.1.3 Show that the type constructor L defined by Eq. (6.21) is a functor for any given bifunctor S and functor P.

Exercise 6.3.1.4 Show that $L^A \triangleq F^A \to G^A$ is, in general, neither a functor nor a contrafunctor when both F^A and G^A are functors or both are contrafunctors (an example of suitable F^A and G^A is sufficient).

Exercise 6.3.1.5 For each of the Scala type constructors defined below, formulate the definition in the type notation and decide whether the type constructors are functors, contrafunctors, or neither.

```
type F[A] = Int => (Option[A], Either[A, Int], Either[A, A])
type G[A] = ((Int, A)) => Either[Int, A]
type H[A] = Either[A, (A, Option[A])] => Int => Int
```

Exercise 6.3.1.6 Using the known constructions, determine which of the following are functors or contrafunctors (or neither) and implement `fmap` or `cmap` if appropriate. Answer this question with respect to each type parameter separately.

(a) $F^A \triangleq \text{Int} \times A \times A + (\text{String} \to A) \times A$.
(b) $G^{A,B} \triangleq (A \to \text{Int} \to \mathbb{1} + B) + (A \to \mathbb{1} + A \to \text{Int})$.
(c) $H^{A,B,C} \triangleq (A \to A \to B \to C) \times C + (B \to A)$.
(d) $P^{A,B} \triangleq (((A \to B) \to A) \to B) \to A$.

Exercise 6.3.1.7 Show that the recursive type constructor $L^\bullet$ defined by

$$L^A \triangleq \mathbb{1} + A + L^A$$

is a functor, and implement a `map` or `fmap` function for L in Scala.

6.4 Discussion

6.4.1 Profunctors

We have seen that some type constructors are neither functors nor contrafunctors because their type parameters appear both in covariant and contravariant positions. An example of such a type constructor is

$$P^A \triangleq A + (A \to \text{Int}) \quad .$$

It is not possible to define either a `map` or a `cmap` function for P: the required type signatures cannot be implemented. However, we *can* implement a function called `xmap`, with the type signature

$$\text{xmap}_P : (B \to A) \to (A \to B) \to P^A \to P^B \quad .$$

To see why, let us temporarily rename the contravariant occurrence of A to Z and define a new type constructor $\tilde{P}$ by

$$\tilde{P}^{Z,A} \triangleq A + (Z \to \text{Int}) \quad .$$

The original type constructor P^A is expressed as $P^A = \tilde{P}^{A,A}$. Now, $\tilde{P}^{Z,A}$ is covariant in A and contravariant in Z. We can implement $\text{xmap}_{\tilde{P}}$ as a composition of `fmap` with respect to A and `cmap` with

respect to Z, similarly to what we saw in the proof of Statement 6.2.4.3. The function $\texttt{xmap}_{\tilde{P}}$ will satisfy the identity and composition laws (6.23)–(6.24). Setting the type parameter $Z = A$, we will obtain the $\texttt{xmap}_P$ function for P. The identity and composition laws for $\texttt{xmap}_P$ will hold, since the laws of $\tilde{P}^{Z,A}$ hold for all type parameters:

$$\text{P's identity law :} \quad \text{xmap}_P(\text{id}^{:A\to A})(\text{id}^{:A\to A}) = \text{id} \quad ,$$

$$\text{P's composition law :} \quad \text{xmap}_P(f_1^{:B\to A})(g_1^{:A\to B}) \fatsemi \text{xmap}_P(f_2^{:C\to B})(g_2^{:B\to C}) = \text{xmap}_P(f_2 \fatsemi f_1)(g_1 \fatsemi g_2) \quad .$$

A type constructor P^A with these properties ($P^A \cong \tilde{P}^{A,A}$ where $\tilde{P}^{Z,A}$ has a lawful $\texttt{xmap}_{\tilde{P}}$) is called a **profunctor**. Sometimes the type constructor $\tilde{P}^{Z,A}$ is also called a profunctor.

Consider an exponential-polynomial type constructor P^A, no matter how complicated, such as

$$P^A \triangleq (\mathbb{1} + A \times A \to A) \times A \to \mathbb{1} + (A \to A + \text{Int}) \quad .$$

Each copy of the type parameter A will occur either in covariant or in a contravariant position because no other possibility is available in exponential-polynomial types. So, we can always rename all contravariant occurrences of the type parameter A to "Z" and so obtain a new type constructor $\tilde{P}^{Z,A}$, which will be covariant in A and contravariant in Z. Since $\tilde{P}^{A,Z}$ is a functor in A and a contrafunctor in Z, we will be able to define a function $\texttt{xmap}_{\tilde{P}}$ satisfying the identity and composition laws. Setting $Z = A$, we will obtain a lawful $\texttt{xmap}_P$, which makes P a profunctor. Thus, *every* exponential-polynomial type constructor is a profunctor.

An unfunctor, such as the disjunctive type `ServerAction[R]` shown in Section 6.1.6, cannot be made into a profunctor. The type signature of `xmap` cannot be implemented for `ServerAction[R]` because it is not a fully parametric type constructor (and so is not exponential-polynomial).

Profunctors are not often used in practical coding. We will see profunctors occasionally in later chapters.

6.4.2 Subtyping with injective or surjective conversion functions

In some cases, P is a subtype of Q when the set of values of P is a *subset* of values of Q. In other words, the conversion function $P \to Q$ is injective and embeds all information from a value of type P into a value of type Q. This kind of subtyping works for parts of disjunctive types, such as `Some[A] <: Option[A]` (in the type notation, $\mathbb{0} + A <: \mathbb{1} + A$). The set of all values of type `Some[A]` is a subset of the set of values of type `Option[A]`, and the conversion function is injective because it is an identity function, $\mathbb{0} + x^{:A} \to \mathbb{0} + x$, that merely reassigns types.

However, subtyping does not necessarily imply that the conversion function is injective. An example of a subtyping relation with a *surjective* conversion function is between the function types $P \triangleq \mathbb{1} + A \to \text{Int}$ and $Q \triangleq \mathbb{0} + A \to \text{Int}$ (in Scala, `P = Option[A] => Int` and `Q = Some[A] => Int`). We have `P <: Q` because $P \cong C^{\mathbb{1}+A}$ and $Q \cong C^{\mathbb{0}+A}$, where $C^X \triangleq X \to \text{Int}$ is a contrafunctor. The conversion function $P \to Q$ is an identity function that reassigns types,

```
def p2q[A](p: Option[A] => Int): Some[A] => Int = { x: Some[A] => p(x) }
```

In the code notation, $p \to x \to p(x)$ is easily seen to be the same as $p \to p$.

Nevertheless, it is not true that all information from a value of type P is preserved in a value of type Q: the type P describes functions that also accept `None` as an argument, while functions of type Q do not. So, there is strictly more information in the type P than in Q. The conversion function $\text{p2q} : P \to Q$ is surjective.

We have now seen examples of either injective or surjective type conversions. Suppose $P_1 <: Q_1$ and $P_2 <: Q_2$, and consider the product types $P_1 \times P_2$ and $Q_1 \times Q_2$. Since the product type is one of the functor constructions, the product $A \times B$ is covariant in both type parameters. It follows that $P_1 \times P_2 <: Q_1 \times Q_2$. If $r_1 : P_1 \to Q_1$ is injective but $r_2 : P_2 \to Q_2$ is surjective, the function product $r_1 \boxtimes r_2 : P_1 \times Q_1 \to P_2 \times Q_2$ is neither injective nor surjective. So, type conversion functions are not necessarily injective or surjective; they can also be anything "in between".

An important property of functor liftings is that they preserve injectivity and surjectivity: if a function $f^{:A\to B}$ is injective, it is lifted to an injective function $f^{\uparrow L} : L^A \to L^B$; and similarly for surjective functions f. Let us prove this property for injective functions; the proof for surjective functions is quite similar.

Statement 6.4.2.1 If L^A is a lawful functor and $f^{:A\to B}$ is an injective function then $\text{fmap}_L(f)$ is also an injective function of type $L^A \to L^B$.

Proof We begin by noting that an injective function $f^{:A\to B}$ must somehow embed all information from a value of type A into a value of type B. The **image** of f (the subset of all values of type B that can be obtained as $f(a)$ for some $a^{:A}$) thus contains a distinct value of type B for each distinct value of type A. So, there exists a function that maps any b from the image of f back to a value $a^{:A}$ it came from; call that function $g^{:B\to A}$. The function g must satisfy

$$\forall a^{:A}.\, g(f(a)) = a \quad ;$$

equivalently written as

$$g \circ f = \text{id} \quad .$$

It is important that g is a partial function. The function g is partial because it is defined only for a subset of values of type B, which are in the image of f. Despite the equation $g \circ f = \text{id}$, the function g is not an inverse for f. An inverse function for f must be a *total* (not a partial) function h satisfying $h \circ f = \text{id}$ and $f \circ h = \text{id}$. The function g is called a **left inverse** for f because $f \circ g \neq \text{id}$, since $f \circ g$ is only a partial function.

The fact that f has a left inverse is *equivalent* to the assumption that f is injective. Indeed, if any function f has a left inverse g, we can show that f is injective. Assume some x and y such that $f(x) = f(y)$; we will prove that f is injective if we show that $x = y$. Applying g to both sides of $f(x) = f(y)$, we get

$$x = g(f(x)) = g(f(y)) = y \quad .$$

Now we apply this trick to functions lifted into the functor L. To prove that $\text{fmap}_L(f)$ is injective, we need to show that it has a left inverse. We can lift both sides of the equation $g \circ f = \text{id}$ to get

$$\begin{aligned} & \text{fmap}_L(g) \circ \text{fmap}_L(f) \\ \text{composition law for } L: \quad & = \text{fmap}_L(g \circ f) \\ \text{use } g \circ f = \text{id}: \quad & = \text{fmap}_L(\text{id}) \\ \text{identity law for } L: \quad & = \text{id} \quad . \end{aligned}$$

It follows that $\text{fmap}_L(g) \circ \text{fmap}_L(f) = \text{id}$, i.e. $\text{fmap}_L(g)$ is a left inverse for $\text{fmap}_L(f)$. Since $\text{fmap}_L(f)$ has a left inverse, it is injective.

7 Typeclasses and functions of types

7.1 Motivation and first examples

7.1.1 Constraining type parameters

The summation method, `.sum`, works for any collection of numeric values:

```
scala> Seq(1, 2, 3).sum
res0: Int = 6

scala> Seq(1.0, 2.0, 3.0).sum
res1: Double = 6.0
```

We can use `.sum` to compute the average of a sequence of numbers,

```
def avg(s: Seq[Double]): Double = s.sum / s.length
```

Can we generalize the averaging function `avg` from `Double` to other numeric types, e.g.

```
def avg[T](s: Seq[T]): T = ???
```

This code is impossible because averaging works only for certain types `T`, not for arbitrary types `T` as implied by the type signature above. We will be able to define `avg[T]` only if we constrain the type parameter `T` to be a type representing a suitable numeric value (e.g. `Float`, `Double`, or `BigDecimal`).

Another example of a similar situation is a function with type signature $A \times F^B \to F^{A \times B}$,

```
def inject[F[_], A, B](a: A, f: F[B]): F[(A, B)] = f.map(b => (a, b))     // Must have ‘f.map‘.
```

This function requires the type constructor `F[_]` to have a `.map` method, i.e. to be a functor. We can implement `inject` only if we constrain the parameter `F` to be a functor.

What would that constraint be like? Consider an ordinary function with no type parameters, e.g.:

```
def f(x: Int): Int = x + 1
```

In this code, the syntax `x:Int` constrains the value of the argument `x` to be integer. It is a type error to apply `f` to a non-integer argument.

Using a similar syntax for *type parameters*, we would write the type signatures for `avg` and `inject` as

```
def avg[T: Fractional](s: Seq[T]): T
def inject[F[_]: Functor, A, B](a: A, f: F[B]): F[(A, B)]
```

Scala uses the syntax `[T: Fractional]` to constrain the type parameter `T` to "fractional numeric" types. Similarly, `[F[_]: Functor]` requires the type constructor `F[_]` to be a functor. Applying `avg` or `inject` to types that do not obey those constraints will be a type error detected at compile time.

In these examples, we are restricting a type parameter to a subset of possible types, because only types from that subset have certain properties that we need. A subset of types, together with the required properties that those types must satisfy, is called a **typeclass**. The syntax `[T: Fractional]` is a **typeclass constraint** that forces the type `T` to belong to the typeclass `Fractional`.

This chapter focuses on defining and using typeclasses and on understanding their properties. We will see in detail how the syntax such as `[T: Fractional]` is implemented and used.

7.1.2 Functions of types and values

The similarity between the type parameter `T` and the argument `s` is clear in this type signature,

```
def avg[T: Fractional](s: Seq[T]): T
```

We can view `avg` as a function that takes *two* parameters (a type `T` and a value `s`) and returns a value. We can also view `avg` as a function from a *type* `T` to a *value* of type `Seq[T] => T`. We may call functions of this kind **type-to-value** functions (TVF). The syntax for TVFs supported in a future version of Scala 3 will show this more clearly,

```
val avg: [T] => Seq[T] => T = ...   // Scala 3 only.
```

To emphasize that `avg` is a TVF, we may write the type signature of `avg` as

```
def avg[T: Fractional]: Seq[T] => T // Inconvenient in practice! Use avg[T: Fractional](s: Seq[T]): T
```

A type constructor such as `Seq[_]` can be seen as a **type-to-type** function (TTF) because it takes a type `T` and returns a new type `Seq[T]`.

Functions can map from values or from types and to values or to types, as this table shows:

functions...	from value	from type
to value	(VVF) `def f(x:Int):Int`	(TVF) `def pure[A]: A => List[A]`
to type	(VTF) dependent type	(TTF) `type MyData[A] = Either[Int, A]`

We have already seen examples of VVFs, TVFs, and TTFs. Value-to-type functions (VTFs) are known as **dependent types** (or, more verbosely, as "value-dependent types"). An example in Scala:

```
val x = new { type T = Int }
val y: x.T = 123
```

In this example, `x.T` is a dependent type because it is a type that depends on the *value* `x`. For the value `x` defined in this code, the expression `x.T` evaluates to the type `Int`.

We will not consider dependent types (VTFs) in this chapter because typeclasses only require a combination of a TTF and a TVF.

7.1.3 Partial functions of types and values

We would like to define the function `avg[T: Fractional]` as a TVF that can be applied only to a subset of possible types `T`. This is similar to a **partial function**, i.e. a function defined only for a subset of possible values of its argument's type. We may call such functions partial type-to-value functions (PTVFs), to distinguish them from partial value-to-value functions (PVVFs) we saw before.

In some situations, partial functions are safe to use. For instance, the following partial function `p`,

```
def p: Either[Int, String] => Int = { case Left(x) => x - 1 }
```

can be applied only to values of the form `Left(...)`. Applying `p` to a value `Right(...)` will cause a run-time error. However, consider this code:

```
val x = Seq(Left(1), Right("abc"), Left(2))

scala> x.filter(_.isLeft).map(p)
res0: Seq[Int] = List(0, 1)
```

Although `x.filter(_.isLeft)` has type `Seq[Either[Int, String]]`, all values in that sequence are guaranteed to be of type `Left`. So we know it is safe to apply the partial function `p` in `.map(p)`.

Although safe, this code is brittle: if the `.filter` operation were moved to another place, we might by mistake write code equivalent to `x.map(p)`, causing a run-time exception. It is better to refactor the code so that the compile-time type-checking guarantees the safety of all operations at run time. For the example shown above, the `.collect` method would make a partial function, such as `p`, safe to use:

```
scala> x.collect { case Left(y) => y - 1 }
res1: Seq[Int] = List(0, 1)
```

The `.collect` method is a **total** function because it is defined for all values of its arguments and does not throw exceptions.

Total functions are safer to use than partial functions. The partial function `p` can be converted into a total function by changing its type to `Left[Int, String] => Int`. Another example: applying `.head` to a `List` is unsafe, but the non-empty list type guarantees at compile time that the first element exists:

```
val xs: NonEmptyList[Int] = ...
val h = xs.head  // .head is a total function for a NonEmptyList.
```

In these cases, we achieve safety by making types more strictly constrained. Similarly, partial type-to-value functions (PTVFs) become safe to use if we impose suitable typeclass constraints on the type parameters. Typeclasses can be viewed as a systematic way of safely managing PTVFs.

7.2 Implementing typeclasses

A typeclass constraint `[T: Fractional]` will generate a compile-time error when a function such as `avg[T]` is applied to an incorrectly chosen type parameter `T`. If the Scala library did not already implement the `Fractional` typeclass, how could we reproduce that functionality?

7.2.1 Creating a partial function on types

The code needs to specify that the type parameter must belong to a certain subset of allowed types. To simplify the task, assume that the allowed types are `BigDecimal` and `Double`. One example of a type constraint is shown by unfunctors (see Section 6.1.6), which are type constructors whose type parameters are restricted to specific types. In this code,

```
sealed trait Frac[A] // Unfunctor.
final case class FracBD() extends Frac[BigDecimal]
final case class FracD()  extends Frac[Double]
```

values of type `Frac[A]` can be created only if `A = BigDecimal` or `A = Double`. The keywords `sealed` and `final` guarantee that no further code could extend this definition and allow us to create a value of type, say, `Frac[String]` or `Frac[Boolean]`. Although the Scala compiler will not detect any errors in this code,

```
type T = Frac[String]
type U = Frac[Boolean]
```

we will never be able to create and use any values of types `T` or `U`. In other words, the types `Frac[String]` and `Frac[Boolean]` are *void* types. Trying to create and use values of these types will result in type errors, as the following code shows:

```
1 def f[A]: Frac[A] = FracD()    // Type error.
2 val x: U = FracD()             // Type error.
3 val y: U = FracD().asInstanceOf[U]
4 y match { case FracD() => }    // Type error.
```

In line 3, we disabled the type checker and forced the Scala compiler to ignore the type error in the definition of `y`. However, line 4 shows that we are unable to use that value `y` in further computations.

The type `Frac[A]` is non-void (i.e. has values) only for A belonging to the set {`BigDecimal`, `Double`} of types. This set is called the **type domain** of the type function `Frac`. We now need to define the function `avg[T]` with a type parameter T constrained to that type domain.

The type constraint $T \in$ {`BigDecimal`, `Double`} is equivalent to the requirement that a value of type Frac^T should exist. So, we will implement the type constraint if we include an *additional argument* of type Frac^T into the type signature of `avg`:

```
def avg[T](s: Seq[T], frac: Frac[T]): T
```

The value `frac:Frac[T]` is called a **typeclass instance** value. Because that value needs to be passed to every call of `avg[T]`, we will be unable to use types `T` for which `Frac[T]` is void (i.e. has no values).

In this way, we implemented the typeclass constraint for the PTVF `avg[T]`. The main steps were:

1. Define a type constructor `Frac[_]`.
2. Make sure values of type `Frac[A]` exist only when `A = BigDecimal` or `A = Double`.
3. Pass a value of type `Frac[T]` to the function `avg[T]` as an additional argument.

It is not necessary to define the type constructor `Frac` via an unfunctor. The type constructor `Frac` is only needed to define the type domain {`BigDecimal`, `Double`}. We can use a simple case class instead:

```
final case class Frac[T]()
val fracBD: Frac[BigDecimal] = Frac()
val fracD: Frac[Double] = Frac()
```

This code creates a type constructor `Frac` and makes values of type `Frac[T]` available for chosen type parameters `T`. In this way, we implemented the required type domain.

To write the code for `avg[T]`, we need to be able to add numeric values and to divide by an integer value. More precisely, the body of `avg[T]` needs access to two PTVFs that we may call `add` and `intdiv`,

```
def add[T](x: T, y: T): T
def intdiv[T](x: T, n: Int): T
```

Since `avg[T]` now has an additional argument `frac`, we may use that argument to hold the required functions. So, we redefine `Frac` as a named tuple (case class) containing the functions `add` and `intdiv`:

```
final case class Frac[T](add: (T, T) => T, intdiv: (T, Int) => T)
```

Typeclass instances for `BigDecimal` and `Double` are then created by the following code:

```
val fracBD = Frac[BigDecimal]( (x, y) => x + y,  (x, n) => x / n )
val fracD = Frac[Double]( (x, y) => x + y,  (x, n) => x / n )
```

With these definitions, implementing `avg[T]` becomes straightforward:

```
def avg[T](s: Seq[T], frac: Frac[T]): T = {      // Assuming ‘s‘ is a non-empty sequence.
  val sum = s.reduce(frac.add)     // Here, ‘reduce‘ would fail on an empty sequence ‘s‘.
  frac.intdiv(sum, s.length)       // Compute ‘sum/length‘.
}
```

To use this function, we need to pass a typeclass instance corresponding to the type `T`:

```
scala> avg(Seq(1.0, 2.0, 3.0), fracD) // It will be a type error to use fracBD instead of fracD here.
res0: Double = 2.0

scala> avg(Seq(BigDecimal(1.0), BigDecimal(2.0)), fracBD)
res1: BigDecimal = 1.5
```

This is a fully working implementation of the `avg` function with a `Frac` typeclass constraint. We have achieved compile-time safety since `avg[T]` cannot be applied to values of unsupported types `T`. We have also achieved easy extensibility: To implement another function as a PTVF with the same type domain, we need to add an extra argument of type `Frac[T]` to the function. To add another supported type `T` to the type domain, we just write one more line of code similar to `val fracD = ...`

An equivalent implementation of the `Frac` typeclass via a `trait` with methods requires this code:

```
trait Frac[T] {                  // Trait is not ‘sealed‘.
  def add(x: T, y: T): T
  def intdiv(x: T, n: Int): T
}
val fracBD = new Frac[BigDecimal] {
  def add(x: BigDecimal, y: BigDecimal): BigDecimal = x + y
  def intdiv(x: BigDecimal, n: Int): BigDecimal = x / n
}
val fracD = new Frac[Double] {
  def add(x: Double, y: Double): Double = x + y
  def intdiv(x: Double, n: Int): Double = x / n
}
```

The function `avg[T]` will work unchanged with this implementation of the `Frac` typeclass.

The implementation via a `trait` is significantly longer than the code using a case class as shown previously. One advantage of the longer code is the ability to combine different typeclasses by `trait` mixing. We will look at that in more detail below. For now, we note that both implementations will require the programmer to add a significant amount of new code:

- Calls to `func[T](args)` need to be changed to `func[T](args, ti)` with typeclass instances `ti`.
- For each supported type `T`, a corresponding typeclass instance value needs to be created.
- All those values need to be passed to all places in the code where PTVFs are used.

The extra work can be reduced (and sometimes avoided) by using Scala's "implicit value" feature.

7.2.2 Scala's `implicit` values

An **implicit value** is made automatically available to any function that declares an "implicit argument" of the same type. Scala's syntax for implicit values is

```
implicit val x: Int = 123
```

This declaration introduces an implicit value of type `Int` into the current scope. That value will be automatically passed as an argument to any function declaring an "`implicit`" argument of type `Int`:

```
def f(a: String)(implicit n: Int) = s"$a with $n"

scala> f("xyz")
res0: String = xyz with 123
```

We need to declare the arguments as "`implicit`" in the function's type signature, and the implicit arguments must be in a separate argument group.

The simplest useful function with an implicit argument is the identity function. In the Scala library, this function is called `implicitly`. Compare its code with the code of the ordinary identity function:

```
def implicitly[T](implicit t: T): T = t
def identity[T](t: T): T = t
```

What does `implicitly[T]` do? Since its only argument is declared as `implicit`, we can simply write `implicitly[T]` with no arguments to apply that function. (The type parameter usually needs to be specified.) If no implicit value of type `T` is available, a compile-time error will occur. If an implicit value of type `T` is available in the current scope, `implicitly[T]` will return that value:

```
implicit val s: String = "qqq"

scala> implicitly[String]
res1: String = qqq
```

It is an error to declare more than one implicit value of the same type in the same scope, because implicit arguments are specified by type alone. The Scala compiler will not be able to set implicit arguments of functions automatically when the function's outer scope contains more than one implicit value of a required type, as in this code:

```
implicit val x: Int = 1
implicit val y: Int = 2

scala> implicitly[Int]
<console>:14: error: ambiguous implicit values:
 both value x of type => Int
 and value y of type => Int
 match expected type Int
       implicitly[Int]
                  ^
```

But it is not an error to declare several implicit arguments of the *same* type, e.g.

```
def f(a: String)(implicit x: MyType, y: MyType)
implicit val z: MyType = ???

f("abc") // Same as f("abc")(z, z) since z is the unique implicit value of type MyType.
```

In the example above, the arguments `x` and `y` will be set to the same value, `z`. A compile-time error will occur if no `implicit` value of type `MyType` is visible in the current scope:

```
scala> implicitly[MyType]
<console>:12: error: could not find implicit value for parameter e: MyType
       implicitly[MyType]
                  ^
```

7.2.3 Implementing typeclasses by making instances `implicit`

The main idea is to declare typeclass instance values as `implicit`. Typeclass instance arguments of functions are also declared as `implicit`. As a result, typeclass instances will be passed to all PTVFs automatically. This makes typeclasses easier to use because instance values need to be written out much less often.

The example with the `Frac` typeclass is implemented using implicit values like this:

```
final case class Frac[T](add: (T, T) => T, intdiv: (T, Int) => T)
implicit val fracBD = Frac[BigDecimal]( (x, y) => x + y,  (x, n) => x / n )
implicit val fracD = Frac[Double]( (x, y) => x + y,  (x, n) => x / n )
```

To define the function `avg[T]`, we declare an implicit argument as a `Frac` typeclass instance for `T`:

```
def avg[T](s: Seq[T])(implicit frac: Frac[T]): T = { // Assuming ‘s‘ is a non-empty sequence.
  val sum = s.reduce(frac.add)     // Here, ‘reduce‘ would fail on an empty sequence ‘s‘.
  frac.intdiv(sum, s.length)       // Compute ‘sum/length‘.
}
```

It is now easier to use the function `avg` because the typeclass instances are inserted automatically:

```
scala> avg(Seq(1.0, 2.0, 3.0))
res0: Double = 2.0

scala> avg(Seq(BigDecimal(1.0), BigDecimal(2.0)))
res1: BigDecimal = 1.5
```

Scala supports the “type constraint” syntax for implicit typeclass instance arguments: the code

```
def f[A, B](args...)(implicit t1: Typeclass1[A], t2: Typeclass2[B])
```

is equivalent to the shorter code

```
def f[A: Typeclass1, B: Typeclass2](args...)
```

The shorter code omits the names (`t1`, `t2`) of the typeclass instances. These values can be extracted via the standard function `implicitly` because all implicit arguments are automatically made available as implicit values in the scope of a function’s body. The code of `avg[T]` can then be written as

```
def avg[T: Frac](s: Seq[T]): T = {
  val frac = implicitly[Frac[T]]
  val sum = s.reduce(frac.add)
  frac.intdiv(sum, s.length)
}
```

When an implicit argument is required, the Scala compiler will search for implicit values of the required type in different places of the code. If implicit values are declared in another module, they can be made available by using an `import` declaration. In many cases, explicit `import` declarations can be avoided. One way to avoid them is to declare the required implicit values within the **companion object** of the typeclass (i.e. the Scala `object` with the same name as the type constructor):

```
final case class Frac[T](add: (T, T) => T, intdiv: (T, Int) => T)

object Frac {   // The companion object of ‘Frac[T]‘ creates some typeclass instances as ‘implicit‘.
  implicit val fracBD = Frac[BigDecimal]( (x, y) => x + y,  (x, n) => x / n )
  implicit val fracD = Frac[Double]( (x, y) => x + y,  (x, n) => x / n )
}
```

Whenever a function needs an implicit value of type `Frac[T]` for a specific type `T`, the Scala compiler will automatically look within the companion object of `Frac` (as well as within the companion object of the type `T`) for any instances declared there. So, the programmer’s code will not need to `import` those typeclass instances explicitly even if the companion object is in a different module:

```
scala> avg(Seq(1.0, 2.0, 3.0))
res0: Double = 2.0
```

7.2.4 Extension methods

In Scala, function applications can use three kinds of syntax:

1. The "function" syntax: arguments are to the right of the function as in `plus(x, y)` or `plus(x)(y)`.
2. The "method" syntax: the first argument is to the left of the function, and all other arguments (if any) are placed to the right of the function, as in `x.plus(y)` or `xs.foldLeft(0)(updater)`.
3. The "infix method" syntax (only applies to functions with two *explicit* arguments): no dot character is used, as in `x plus y`, or in `xs map {x => x + 1}`, or in `Set(1,2,3) contains 1`.

The last two syntax features are often used when writing chains of function applications, such as `xs.map(f).filter(g)`, because that code is easier to read than `filter(map(xs, f), g)`.

The method syntax is available only for methods defined in a class. A special feature of Scala allows us to add new functions with method syntax to previously defined types. New functions added in this way are called **extension methods**.

Suppose we would like to convert a previously defined function, say

```
def func(x: X, y: Y): Z = { ... }
```

into an extension method on the type `X` with the syntax `x.func(y)`. To do that, we define a new helper `class` that has a method named `func`. The class's constructor must be declared as an `implicit` function having a *single* argument of type `X`:

```
implicit class FuncSyntax(x: X) {  def func(y: Y): Z = ... }
```

After this code, we can write the method syntax `x.func(y)` as well as the infix method syntax `x func y`. The new syntax will work because the compiler automatically rewrites `x.func(y)` into `new FuncSyntax(x).func(y)`, creating a new temporary value `FuncSyntax(x)`. The method `.func` will be available since it is defined in the class `FuncSyntax`.

As an example, let us define an extension method `.average` for the type `Seq[T]`. Both the type parameter `T` and its typeclass constraint can be written in the constructor of the helper class:

```
implicit class AvgSyntax[T: Frac](xs: Seq[T]) {
  def average: T = avg(xs) // Use a different name, ‘average‘, to avoid name clash with ‘avg‘.
}
```

We can now use the method `.average` on numeric sequences:

```
scala> Seq(1.0, 2.0, 3.0).average
res0: Double = 2.0
```

The Scala compiler automatically rewrites the syntax `Seq(1.0, 2.0, 3.0).average` as the expression

```
new AvgSyntax(Seq(1.0, 2.0, 3.0))(implicitly[Frac[Double]]).average
```

In this way, the method `.average` is actually invoked on a temporarily created value of type `AvgSyntax`. These values will be created automatically because the class constructor of `AvgSyntax` is declared as `implicit`. Since the constructor of `AvgSyntax` includes the typeclass constraint `[T:Frac]`, we will not be able to create values of type `AvgSyntax[T]` for types `T` not in the type domain of `Frac`.

This example illustrates the convenience of implementing PTVFs as extension methods. An extension method is defined only once but automatically becomes available for all types in the domain of the typeclass. Because of the typeclass constraint, the new method will be available *only* on values of supported types.

This convenience comes at a cost: helper classes such as `AvgSyntax` need to be explicitly imported into every scope where extension methods are used, with a declaration like this:

```
import some.library.having.a.longwinded.packagename.AvgSyntax
```

If the helper class is defined in some library, the programmer will have to look at the library's source code to determine the full name of the helper class that needs to be imported.

7.2.5 Solved examples: Implementing typeclasses in practice

We will now look at some practical examples of programming tasks implemented via typeclasses.

Example 7.2.5.1 (metadata extractors) An application needs to work with data structures implemented in various external libraries. All those data structures are case classes containing certain metadata: "name" (a `String`) and "count" (a `Long` integer). However, the specific data structures define the metadata differently, each in its own way:

```
final case class Data1(p: String, q: String, r: Long) // "name" = p, "count" = z
final case class Data2(s: String, c: Long, x: Int)    // "name" = s, "count" = c
final case class Data3(x: Int, y: Long, z: String)    // "name" = z, "count" = x * y
```

The task is to implement two functions, `getName[T]` and `getCount[T]`, for extracting the metadata out of the data structures of type `T`, where `T` is one of `Data1`, `Data2`, `Data3`. Type signatures and sample tests:

```
def getName[T: HasMetadata](t: T): String = ???
def getCount[T: HasMetadata](t: T): Long = ???

scala> getName(Data2("abc", 123, 0))
res0: String = abc

scala> getCount(Data3(10, 20, "x"))
res1: Long = 200
```

Solution We will implement a typeclass `HasMetadata` and declare instances only for `Data1`, `Data2`, and `Data3`. The code for extracting the metadata will be contained within the typeclass instances. Since the metadata extractors have types `T => String` and `T => Long`, a simple solution is to define the typeclass as a `case class` containing these two functions:

```
final case class HasMetadata[T](getName: T => String, getCount: T => Long)
```

The required typeclass instances are declared as implicit values within the companion object:

```
object HasMetadata {    // Extract metadata from each type as appropriate.
  implicit val data1 = HasMetadata[Data1](_.p, _.r)
  implicit val data2 = HasMetadata[Data2](_.s, _.c)
  implicit val data3 = HasMetadata[Data3](_.z, data3 => data3.x * data3.y)
}
```

Now we can define `getName` and `getCount` as PTVFs with typeclass constraints. First, let us write the code using an implicit argument to pass the typeclass instance:

```
def getName[T](t: T)(implicit ti: HasMetadata[T]): String = ti.getName(t)
def getCount[T](t: T)(implicit ti: HasMetadata[T]): Long = ti.getCount(t)
```

Or we could use the typeclass constraint syntax, replacing `ti` by calls to `implicitly[...]`:

```
def getName[T: HasMetadata](t: T): String = implicitly[HasMetadata[T]].getName(t)
def getCount[T: HasMetadata](t: T): Long = implicitly[HasMetadata[T]].getCount(t)
```

This code defines PTVFs `getName` and `getCount` with the type domain that contains the three types `Data1`, `Data2`, `Data3`. In order to add a new type, say `Data4`, to the type domain, we will need to declare a new typeclass instance as an implicit value of type `HasMetadata[Data4]`. New implicit values can be defined anywhere in the code, not necessarily within the companion object `HasMetadata`. To avoid extra `import` statements, the implicit value may be defined within the companion object of `Data4`:

```
final case class Data4(x: Int, message: String)
object Data4 {   // Add Data4 to the type domain of HasMetadata:
  implicit val data4 = HasMetadata[Data4](_.message, _.x.toLong)
}

scala> getName(Data4(1, "abc"))
res2: String = abc
```

For convenience, let us declare the metadata extractors as extension methods:

```
implicit class ExtractorsSyntax[T: HasMetadata](t: T) {
  def name: String = getName(t)
  def count: Long = getCount(t)
}
```

With this definition, we can write:

```
scala> Data2("abc", 123, 0).name
res3: String = "abc"

scala> Data3(10, 20, "x").count
res4: Long = 200
```

The code looks as if the new methods `.name` and `.count` became available for each of the supported data types. It is important that the new methods were added *without* any changes to the implementations of `Data1`, `Data2`, or `Data3`. Those implementations may reside in external libraries whose code we do not need to modify. The typeclass pattern enables us to to add externally defined types to a type domain whenever necessary, and to implement new PTVFs for them.

Example 7.2.5.2 (counters) A certain application needs to count the number of files processed and to send this number to external observability services. The functionality of a counter is provided by an external library as a special class `Counter` with an `.inc()` method that increments the counter. To test the code, we want to be able to pass a test-only counter provided by another library as a type `TestCounter` with an `.incr()` method. The task is to implement a function `bump[C]()`, where `C` is a type constrained to be one of the supported types of counters. The type signature and sample code:

```
def bump[C]()(...) = ???

val counter = Counter(...)
val testCounter = TestCounter(...)

bump(counter)       // Should call counter.inc()
bump(testCounter)   // Should call testCounter.incr()
```

Solution We will implement a typeclass `Bumpable` whose type domain contains the types `Counter` and `TestCounter`. Typeclass instances should allow us to increment a counter of any supported type. So, a typeclass instance value of type `Bumpable[C]` needs to contain a function of type `C => Unit` that will increment a counter of type `C` appropriately:

```
final case class Bumpable[C](bump: C => Unit)
```

We can now create the typeclass instances and implement the function `bump[C]`:

```
object Bumpable {
  implicit val b1 = Bumpable[Counter](c => c.inc())
  implicit val b2 = Bumpable[TestCounter](c => c.incr())

  def bump[C](counter: C)(implicit ti: Bumpable[C]): Unit = ti.bump(counter)
}
```

An equivalent implementation with the type constraint syntax looks like this:

```
def bump[C: Bumpable](counter: C): Unit = implicitly[Bumpable[C]].bump(counter)
```

Example 7.2.5.3 (default values) Certain types have naturally chosen "default" values (e.g. integer zero, empty string, empty array, etc.). The task is to implement a function `default[T]` restricted to types `T` for which default values are available. The required type signature and sample tests:

```
def default[T: HasDefault]: T = ???

scala> default[Int]
res0: Int = 0

scala> default[Double]
res1: Double = 0.0
```

Solution We need to define a PTVF `default[T]` with a type domain that contains (at least) the types `Int` and `Double`. For every supported type `T`, we need to store the known default value of that type. So, the typeclass instance can be defined as a simple wrapper for values of type `T`:

```
final case class HasDefault[T](value: T)
```

Typeclass instances are declared straightforwardly:

```
object HasDefault {
  implicit val defaultInt = HasDefault[Int](0)
  implicit val defaultDouble = HasDefault[Double](0.0)
  implicit val defaultString = HasDefault[String]("")
  implicit val defaultUnit = HasDefault[Unit](())
}
```

The implementation of `default[T]` is written as

```
def default[T](implicit ti: HasDefault[T]) = ti.value
```

When using the typeclass constraint syntax, the code is

```
def default[T: HasDefault]: T = implicitly[HasDefault[T]].value
```

How to define an empty list as a default value for lists of *any* chosen type? We cannot write

```
implicit val defaultList = HasDefault[List[A]](List())     // Error: ‘A‘ is undefined.
```

The type parameter `A` needs to be defined in the left-hand side. Since Scala 2 does not support `val` declarations with type parameters (Scala 3 will), we need to write the typeclass instance as a `def`:

```
implicit def defaultList[A] = HasDefault[List[A]](List())
```

Another example of a `HasDefault` instance with a type parameter is for functions of type $A \to A$:

```
implicit def defaultFunc[A]: HasDefault[A => A] = HasDefault[A => A](identity)
```

Types that have default values are also called **pointed** types. This book defines the typeclass `Pointed` for pointed *functors* (Section 7.3.5) because they are more widely used than pointed types.

Example 7.2.5.4 (semigroups) In many cases, data items can be combined or merged into a larger data item of the same type. For instance, two numbers can be added, two sets combined into one set, two strings concatenated into one string, and two lists into one list. The "merge" operation can be defined as a function `combine[T]` taking two arguments of type `T` and returning a new value of type `T`. We will denote that operation by $\oplus$, e.g. we write $x \oplus y$. In all the examples just mentioned (integers, strings, lists, etc.), that operation is associative:

$$\forall (x, y, z).\ (x \oplus y) \oplus z = x \oplus (y \oplus z) \quad . \tag{7.1}$$

This associativity law makes parentheses in the expression $x \oplus y \oplus z$ unnecessary.

A type `T` with an associative binary operation is called a **semigroup**. The task in this example is to define the semigroup operation for the types `Int`, `String`, and `List[A]`.

Solution For every supported type T, the required data is a function of type $T \times T \to T$. So, we define the typeclass as a wrapper over that type:

```
final case class Semigroup[T](combine: (T, T) => T)
```

The typeclass instances for the supported types are defined using a short syntax as

```
object Semigroup {
  implicit val semigroupInt = Semigroup[Int](_ + _)
  implicit val semigroupString = Semigroup[String](_ + _)
  implicit def semigroupList[A] = Semigroup[List[A]](_ ++ _)
}
```

The function `combine[T]` is implemented as

```
def combine[T](x: T, y: T)(implicit ti: Semigroup[T]): T = ti.combine(x, y)
```

Since `combine` is a binary operation, it is convenient to define infix method syntax for it:

```
implicit class SemigroupSyntax[T: Semigroup](x: T) { def |+|(y: T): T = combine(x, y) }
```

After this definition, we may use the infix operation `|+|` like this,

```
scala> List(1, 2, 3) |+| List(4)
res0: List[Int] = List(1, 2, 3, 4)
```

Due to the associativity law (7.1), the result of `x |+| y |+| z` does not depend on the choice of parentheses: `(x |+| y) |+| z == x |+| (y |+| z)`. This makes programs written using the semigroup operation `|+|` easier to understand and reason about.

Semigroup types represent data that can be pairwise "merged" in a certain well-defined way. Using the `Semigroup` typeclass, we can write code that is parameterized by the type of "mergeable" data. As an example, given a `Seq[T]` where the type `T` is a semigroup, we can "merge" all elements to compute a result of type `T`. This computation can be implemented as a function parameterized by `T`,

```
def merge[T: Semigroup](ts: Seq[T]): T = ts.reduce(combine[T])
```

This function assumes a non-empty input sequence `ts` whose elements are of a semigroup type `T`. We can also implement the same function as an extension method,

```
implicit class SumSyntax[T: Semigroup](ts: Seq[T]) { def merge: T = ts.reduce(combine[T]) }
```

With the previous definitions, we can now evaluate expressions such as

```
scala> Seq(1, 2, 3).merge
res1: Int = 6

scala> Seq(List(), List(true), List(), List(true, false)).merge
res2: List[Boolean] = List(true, true, false)
```

It is important that the associativity law (7.1) should hold for each of the supported types. If that is not so, programs written with `merge` will not work as expected. (For instance, programmers would certainly expect that `xs.merge |+| ys.merge == (xs ++ ys).merge` for any sequences `xs` and `ys`.) However, the code of the typeclass *does not* check the associativity law. It is the responsibility of the programmer to verify that the implementation of each typeclass instance is lawful.

The associativity law for integers is a standard arithmetic identity

$$(x + y) + z = x + (y + z) \quad .$$

Verifying associativity for lists and strings (which are lists of characters) is intuitively simple because concatenation preserves the order of elements. If `x`, `y`, and `z` are lists, the concatenation `(x ++ y) ++ z` is a list containing all elements from `x`, `y`, and `z` in their original order. It is evident that the concatenation `x ++ (y ++ z)` is a list with the same elements in the same order. However, a rigorous proof of the associativity law for lists, starting from the code of the `concat` function, requires significant work (see Section 7.5.2).

Example 7.2.5.5 (alternative semigroup implementations) The definitions of the semigroup operation $\oplus$ as concatenation for strings and as addition for integers may appear to be "natural". However, alternative implementations are useful in certain applications. As long as the associativity law holds, *any* function of type $T \times T \to T$ may be used as the semigroup operation. The task of this example is to show that the following implementations of the semigroup operation are lawful and to implement the corresponding typeclass instances in Scala.

(a) For any type T, define $x \oplus y \triangleq x$ (ignoring the value of y).

(b) For pair types $T \triangleq A \times B$, define the operation $\oplus$ by $(a_1 \times b_1) \oplus (a_2 \times b_2) \triangleq a_1 \times b_2$.

(c) For $T \triangleq$ String, define $x \oplus y$ as the longer of the strings x and y.

(d) For $T \triangleq S \to S$ (the type S is fixed), define $x \oplus y \triangleq x \fatsemi y$ (the forward function composition) or $x \oplus y \triangleq x \circ y$ (the backward function composition).

Solution **(a)** To verify the associativity law, use the definition $x \oplus y \triangleq x$ to compute

$$(x \oplus y) \oplus z = x \oplus z = x \quad , \quad x \oplus (y \oplus z) = x \quad .$$

So the associativity law holds: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for any x, y, z.

It is clear that $x \oplus y \oplus z \oplus ... = x$ for any number of values; the binary operation keeps the first value and ignores all further values. We can implement this semigroup instance at once for all types T:

```
implicit def semigroup1[T] = Semigroup[T]{ (x, y) => x }
```

Similarly, the definition $x \oplus y \triangleq y$ gives an associative binary operation for a (different) semigroup.

(b) To verify the associativity law:

$$((a_1 \times b_1) \oplus (a_2 \times b_2)) \oplus (a_3 \times b_3) = (a_1 \times b_2) \oplus (a_3 \times b_3) = a_1 \times b_3 \quad ,$$
$$(a_1 \times b_1) \oplus ((a_2 \times b_2) \oplus (a_3 \times b_3)) = (a_1 \times b_1) \oplus (a_2 \times b_3) = a_1 \times b_3 \quad .$$

The implementation is possible for any types A, B:

```
implicit def semigroup2[A, B] = Semigroup[(A, B)]{ case ((a1, b1), (a2, b2)) => (a1, b2) }
```

One use case for this semigroup is to maintain a pair of timestamps for the first and the last events in a temporally ordered series. Merging two such pairs for consecutive events means to keep the first value from the first pair and the second value from the second pair.

(c) It is clear that $x \oplus y \oplus z$ is the longest of the strings x, y, and z. Since the definition of "longest" does not depend on the order in which we select pairs for comparison, the operation is associative. (For the same reason, any "maximum" or "minimum" operation is associative.) Implementation:

```
implicit val semigroup3 = Semigroup[String]{ (x, y) => if (x.length > y.length) x else y }
```

(d) The composition of functions is associative (see proofs in Section 4.2.2). Whether we choose to define $x \oplus y \oplus z = x \fatsemi y \fatsemi z$ or $x \oplus y \oplus z = x \circ y \circ z = z \fatsemi y \fatsemi x$, the results do not depend on inserting parentheses. The code for these two typeclass instances is

```
implicit def semigroup4[S] = Semigroup[S => S]{ (x, y) => x andThen y }
implicit def semigroup5[S] = Semigroup[S => S]{ (x, y) => x compose y }
```

Example 7.2.5.6 (monoids) When a data type is a semigroup, it is often possible to find a special value that acts as an "default" value with respect to the semigroup operation. Merging with the default value will not change any other value. For instance, concatenating with an empty list does not change any other list; so the empty list plays the role of the default value for lists. Merging an empty set with any other set does not change the other set; so the empty set is the default for sets.

A semigroup with such a default value is called a **monoid**. Formally, a type T is a monoid when it has an associative binary operation $\oplus_T$ and a chosen default or "empty" value $e^{:T}$ such that

$$\text{for all } x^{:T} : \quad e \oplus_T x = x \quad , \quad x \oplus_T e = x \quad . \tag{7.2}$$

The laws (7.2) are called the **identity laws** of monoid.

The task in this example is to define a typeclass describing monoids.

Solution The typeclass instances should contain the same information as semigroups and additionally the default value of type T. So, we write

```
final case class Monoid[T](combine: (T, T) => T, empty: T)
```

Let us define some typeclass instances for illustration:

```
object Monoid {
  implicit val monoidInt = Monoid[Int](_ + _, 0)
  implicit val monoidString = Monoid[String](_ + _, "")
  implicit def monoidList[A] = Monoid[List[A]](_ ++ _, List())
  implicit def monoidFunc[A] = Monoid[A => A](_ andThen _, identity)
}
```

Monoids formalize the general properties of data aggregation. Section 7.3.4 will study monoids in more detail and show further examples of their use. At this point, we look at one more example that defines a `Monoid` instance using two previously defined typeclasses, `Semigroup` and `HasDefault`.

Example 7.2.5.7 (monoids as semigroups with default) A monoid combines the properties of a semigroup and a type with a default value. If a type `T` is a semigroup and has a default value, it is likely that `T` is a monoid. The task is to define a `Monoid` typeclass instance given `Semigroup` and `HasDefault` instances for a type `T`.

Solution We need to define a function `monoidOf[T]` that returns the required monoid typeclass instance for `T`. The typeclass constraints for this function are `Semigroup` and `HasDefault`. So the type signature must be

```
def monoidOf[T](implicit ti1: Semigroup[T], ti2: HasDefault[T]): Monoid[T] = ???
```

To implement a value of type `Monoid[T]`, we need to provide a function of type $T \times T \to T$ and a value of type T. Precisely that data is available in the typeclass instances of `Semigroup` and `HasDefault`, and so it is natural to use them,

```
def monoidOf[T](implicit ti1: Semigroup[T], ti2: HasDefault[T]): Monoid[T] =
  Monoid(ti1.combine, ti2.value)
```

Using the type constraint syntax, the equivalent code is

```
def monoidOf[T: Semigroup : HasDefault]: Monoid[T] =
  Monoid(implicitly[Semigroup[T]].combine, implicitly[HasDefault[T]].value)
```

We can also define this function as an `implicit`, so that every type `T` with a `Semigroup` and `HasDefault` instances will automatically receive a `Monoid` typeclass instance as well.

Writing the types of the `Semigroup`, `HasDefault`, and `Monoid` instances in the type notation, we get

$$\text{Semigroup}^T \triangleq T \times T \to T \quad , \qquad \text{HasDefault}^T \triangleq T \quad , \qquad \text{Monoid}^T \triangleq (T \times T \to T) \times T \quad .$$

It is clear that

$$\text{Monoid}^T \cong \text{Semigroup}^T \times \text{HasDefault}^T \quad .$$

Indeed, the code for `monoidOf` computes a pair of values from the `Semigroup` and `HasDefault` instances.

Is this implementation lawful with respect to the monoid laws? The associativity law will hold for the monoid if the `Semigroup` typeclass instance was already lawful. However, the value stored in the `HasDefault` instance is not guaranteed to satisfy the identity laws (7.2) with respect to the `combine` operation stored in the `Semigroup` instance. The programmer must verify that the identity laws hold. It can happen that, for some type T, typeclass instances of `Semigroup` and `HasDefault` already exist but are not "compatible", so that the monoid identity laws do not hold. In that case, a simple combination of `Semigroup` and `HasDefault` instances will not work, and a different `Monoid` instance must be defined.

Are there alternative implementations of the `Monoid` typeclass instance given `Semigroup` and `HasDefault` instances? The function `monoidOf` needs to produce a value of type $(T \times T \to T) \times T$ given values of type $T \times T \to T$ and a value of type T:

$$\text{monoidOf} : (T \times T \to T) \times T \to (T \times T \to T) \times T \quad .$$

When the type signature of `monoidOf` is written in this notation, it is clear that `monoidOf` should be the identity function; indeed, that is what our code translates to. Although there are many other implementations of the same type signature, only the code shown above will satisfy the monoid laws. An example of an unlawful implementation is

```
def badMonoidOf[T](implicit ti1: Semigroup[T], ti2: HasDefault[T]): Monoid[T] =
  Monoid((x, y) => ti1.combine(x, ti1.combine(x, y)), ti2.value)
```

This implementation defines the monoid operation as $x \oplus x \oplus y$ instead of the correct definition $x \oplus y$. If we set $y = e_T$, we will get $x \oplus x$ instead of x, violating one of the identity laws.

7.2.6 Typeclasses for type constructors

An example of a function parameterized by a type constructor is

```
def inject[F[_]: Functor, A, B](a: A, f: F[B]): F[(A, B)] = ???
```

The intention is to require the type parameter `F` to be a functor. We can use a suitable typeclass to implement this constraint. Since the type parameter `F` is itself a type constructor, the typeclass constructor will use the syntax `Functor[F[_]]`.

What information needs to be wrapped by a typeclass instance? A functor `F` must have a `map` function with the standard type signature

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

In the type notation, this type signature is written as

$$\text{map} : \forall(A, B).\, F^A \to (A \to B) \to F^B \quad .$$

So, a typeclass instance of the `Functor` typeclass must contain this function as a value. But defining the typeclass as before via a `case class` does not work with Scala 2,

```
final case class Functor[F[_]](map: ∀(A, B).F[A] => (A => B) => F[B])   // Not possible in Scala 2.
```

Scala 3 will directly support an argument type that *itself* contains type quantifiers such as $\forall(A, B)$. In Scala 2, we need to represent such "nested" type quantifiers by writing a `trait` with a `def` method:

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

The type constructor `Functor` has the type parameter `F[_]`, which must be itself a type constructor. For any type constructor `F`, a value of type `Functor[F]` is a wrapper for a value of type $\forall(A, B).\, F^A \to (A \to B) \to F^B$. Values of type `Functor` (i.e. typeclass instances) are implemented with the "`new { ... }`" syntax:

```
implicit val functorSeq = new Functor[Seq] {
  def map[A, B](fa: Seq[A])(f: A => B): Seq[B] = fa.map(f)
}
```

This is currently the most common way of defining typeclasses in Scala.

It is convenient to declare `.map` as an extension method on the `Functor` type constructors,

```
implicit class FunctorSyntax[F[_]: Functor, A](fa: F[A]) {    // Syntax helper class.
  def map[B](f: A => B): F[B] = implicitly[Functor[F]].map(fa)(f)
}
```

If this class definition is in scope, the `.map` method becomes available for values of functor types.

Using the `Functor` typeclass and the syntax helper, we can now implement the function `inject`:

```
def inject[F[_]: Functor, A, B](a: A, f: F[B]): F[(A, B)] = f.map { b => (a, b) }

scala> inject("abc", Seq(1, 2, 3))   // Assuming that implicit Functor[Seq] is in scope.
res0: Seq[(String, Int)] = List(("abc", 1), ("abc", 2), ("abc", 3))
```

Just like the `Monoid` typeclass, the code of the `Functor` typeclass does not enforce the functor laws on the implementation. It is the programmer responsibility to verify that the laws hold.

One way of checking the laws is to use the `scalacheck` library[1] that automatically runs random tests for a given assertion. Using the `Functor` typeclass constraint, we can implement a function (in our terminology, a PTVF) that checks the functor laws for *any* given type constructor. The code looks like this (the required `import` declarations are omitted for brevity):

```
def checkFunctorLaws[F[_], A, B, C]()(implicit ff: Functor[F],
                    // The ‘Arbitrary‘ typeclass is provided by scalacheck.
      fa: Arbitrary[F[A]], ab: Arbitrary[A => B], bc: Arbitrary[B => C]) = {
  // Identity law. The assertion must hold for all ‘fa‘.
  forAll { (fa: F[A]) =>     fa.map(identity[A]) shouldEqual fa }
  // Composition law. The assertion must hold for all ‘f‘, ‘g‘, ‘fa‘.
  forAll { (f: A => B, g: B => C, fa: F[A]) =>
    fa.map(f).map(g) shouldEqual fa.map(f andThen g)
  }
```

[1] https://www.scalacheck.org

```
}
// Check the laws for F = Seq using specific types instead of type parameters A, B, C.
checkFunctorLaws[Seq, Int, String, Double]()
```

The `scalacheck` library will substitute a large number of random values into the given assertions, aiming to discover a set of values for which some condition fails. While a test of laws with `scalacheck` (it might find a bug), it does not provide a rigorous proof that the laws hold: the laws are tested only with a finite number of values and with type parameters set to specific types.

7.3 Deriving typeclass instances via structural analysis of types

In Chapter 6 we analyzed the structure of functors by checking whether the six standard type constructions can make new functors out of previous ones. We will now apply the same **structural analysis** to various typeclasses. Is a product of two monoids a monoid? Is a co-product of two semigroups a semigroup? Answers to such questions will enable us to:

- Quickly decide whether a given type can have a typeclass instance of `Monoid`, `Semigroup`, etc.
- If so, derive the code for the new typeclass instance without guessing.
- Have assurance that the required typeclass laws will hold for newly constructed instances.

In the following sections, we will show how to use this approach for some simple typeclasses. Later chapters of this book will systematically apply structural analysis to each considered typeclass.

7.3.1 Extractors

Example 7.2.5.1 shows a typeclass that extracts metadata of fixed types from a value of type `T`. The typeclass instance wraps a pair of functions,

```
final case class HasMetadata[T](getName: T => String, getCount: T => Long)
```

In the type notation, this type constructor is written as

$$\text{HasMetadata}^T \triangleq (T \to \text{String}) \times (T \to \text{Long}) \quad .$$

Using the standard type equivalences (see Table 5.6), we find that this type is equivalent to $T \to \text{String} \times \text{Long}$. So, let us denote $\text{String} \times \text{Long}$ by Z and consider a more general typeclass whose instances are values of type $T \to Z$. We may call this typeclass a "Z-extractor" since types T from its type domain permit us to extract values of type Z. With a fixed type Z, we denote the typeclass by

$$\text{Extractor}^T \triangleq T \to Z \quad .$$

```
final case class Extractor[Z, T](extract: T => Z)
```

To apply structural analysis, we check whether any of the standard type constructions produce new typeclass instances. The results are summarized in Table 7.1. Let us show the required calculations.

Fixed types We check whether an `Extractor` typeclass instance can be computed for the `Unit` type or for another fixed type C. To compute $\text{Extractor}^{\mathbb{1}} = \mathbb{1} \to Z$ requires creating a value of type Z from scratch, which we cannot do in a fully parametric function. For a fixed type C, a value of type $\text{Extractor}^C = C \to Z$ can be computed only if we can compute a value of type Z given a value of type C. This is possible only if we choose C as $C = Z$. The typeclass instance for Extractor^Z is implemented as an identity function of type $Z \to Z$:

```
implicit def extractorZ[Z] = Extractor[Z, Z](identity)
```

Type parameters Creating a typeclass instance Extractor^A for an arbitrary type A means to compute $\forall A.\, A \to Z$; this is not possible since we cannot create values of type Z using only a value of an unknown type A. So, there is no `Extractor` for arbitrary types `A`.

Products If the types A and B are known to be within the type domain of `Extractor`, can we add the pair $A \times B$ to that domain? If we can extract a value of type Z from each of two values $a^{:A}$ and $b^{:B}$, we can certainly extract a value of type Z from the product $a \times b$ by choosing to extract only from a or only from b. So, it appears that we have two possibilities for implementing the typeclass for $A \times B$. Reasoning more rigorously, we see that computing a new typeclass instance from two previous ones requires implementing a conversion function with type signature

$$\forall (A, B, Z).\, \text{Extractor}^A \times \text{Extractor}^B \to \text{Extractor}^{A \times B} = (A \to Z) \times (B \to Z) \to A \times B \to Z \quad .$$

We can derive only two fully parametric implementations of this type signature:

$$f^{:A \to Z} \times g^{:B \to Z} \to a \times b \to f(a) \qquad \text{and} \qquad f^{:A \to Z} \times g^{:B \to Z} \to a \times b \to g(b) \quad .$$

Both implementations give a valid `Extractor` instance (since there are no laws to check). However, every choice will use one of the two `Extractor` instances and ignore the other. So, we can simplify this construction by keeping the typeclass constraint only for A and allowing *any* type B,

$$\text{extractorPair} : \forall (A, B, Z).\, \text{Extractor}^A \to \text{Extractor}^{A \times B} \quad .$$

```
def extractorPair[Z, A, B](implicit ti: Extractor[Z, A]) =
  Extractor[Z, (A, B)] { case (a, b) => ti.extract(a) }
```

If A has an `Extractor` instance, the product of A with any type B also has an `Extractor` instance. Examples of this construction are $Z \times B$ and $P \times Q \times Z$ (since the type Z itself has an `Extractor` instance).

Co-products Given typeclass instances Extractor^A and Extractor^B, can we compute a value of type Extractor^{A+B}? Writing out the types, we get

$$\forall (A, B, Z).\, \text{Extractor}^A \times \text{Extractor}^B \to \text{Extractor}^{A+B} = (A \to Z) \times (B \to Z) \to A + B \to Z \quad .$$

Due to a known type equivalence (Table 5.6), we have a unique implementation of this function:

$$\text{extractorEither} \triangleq f^{:A \to Z} \times g^{:B \to Z} \to \begin{array}{c|c} & Z \\ \hline A & a \to f(a) \\ B & b \to g(b) \end{array} \quad .$$

```
def extractorEither[Z, A, B](implicit ti1: Extractor[Z, A], ti2: Extractor[Z, B]) =
  Extractor[Z, Either[A, B]] {
    case Left(a)    => ti1.extract(a)
    case Right(b)   => ti2.extract(b)
}
```

So, the co-product of A and B can be given a unique `Extractor` instance.

Since the product and the co-product constructions preserve `Extractor` instances, we conclude that any polynomial type expression has an `Extractor` instance as long as every product type contains at least one Z or another `Extractor` type. For example, the type expression

$$A \times Z + Z \times (P + Z \times Q) + B \times C \times Z$$

is of that form and therefore has an `Extractor` instance. Generally, a polynomial in Z will have an `Extractor` instance only if the polynomial is of the form $Z \times P^Z$ for some functor P.

Function types We need to investigate whether $C \to A$ or $A \to C$ can have an `Extractor` instance for some choice of C, assuming that we have an instance for A. The required conversion functions must have type signatures

$$\text{Extractor}^A \to \text{Extractor}^{A \to C} \quad \text{or} \quad \text{Extractor}^A \to \text{Extractor}^{C \to A} \quad .$$

Writing out the types, we find

$$(A \to Z) \to (A \to C) \to Z \quad \text{or} \quad (A \to Z) \to (C \to A) \to Z \quad .$$

None of these type signatures can be implemented. The first one is hopeless since we do not have values of type A; the second one is missing values of type C. However, since the type C is fixed, we may store a value of type C as part of the newly constructed type. So, we consider the pair type $C \times (C \to A)$ and find that its `Extractor` instance, i.e. a value of type $C \times (C \to A) \to Z$, can be derived from a value of type $A \to Z$ as

$$f^{:A \to Z} \to c^{:C} \times g^{:C \to A} \to f(g(c)) \quad .$$

```
def extractorFunc[Z, A, C](implicit ti: Extractor[Z, A]) =
  Extractor[Z, (C, C => A)] { case (c, g) => ti.extract(g(c)) }
```

Examples of this construction are the type expressions $C \times (C \to Z)$ and $D \times (D \to Z \times P)$.

Another situation where an `Extractor` instance exists for the type $C \to A$ is when the type C has a known "default value" e_C (as in the `HasDefault` typeclass). In that case, we may omit the first C in $C \times (C \to A)$ and instead substitute the default value when necessary.

Recursive types We can use type recursion with any of the non-recursive constructions that create a new type with an `Extractor` instance out of a previous such type. For clarity, let us use type constructors F_1, F_2, etc., for describing the new types produced out of previous extractor types. For the product construction, we define $F_1^{B,A} \triangleq A \times B$. For the co-product construction, $F_2^{B,A} \triangleq B + A$ (where B must be also an extractor type). For the function construction, $F_3^{C,A} \triangleq C \times (C \to A)$.

Any recursive type equation that uses F_1, F_2, and/or F_3 will define a new recursive type with an `Extractor` instance. An example of such a recursive type is a composition of F_2 and F_1 defined by

$$T \triangleq F_2^{Z \times P, F_1^{T,Q}} = Z \times P + Q \times T \quad . \tag{7.3}$$

We can visualize this recursive type as an "infinite disjunction"

$$\begin{aligned} T &\cong Z \times P + Q \times T \\ &= Z \times P + Q \times (Z \times P + Q \times (Z \times P + ...)) \\ &= Z \times P \times (\mathbb{1} + Q + Q \times Q + ...) \cong Z \times P \times \text{List}^Q \quad . \end{aligned}$$

Since the resulting type is equivalent to $Z \times C$ where $C \triangleq P \times \text{List}^Q$, we find that the recursive equation (7.3) is equivalent to the product construction with a different type.

This will happen with any recursive equation containing F_1 and F_2 (but no F_3): since F_1 and F_2 are polynomial functors, the resulting type T will be a recursive polynomial in Z. A polynomial in Z will have an `Extractor` instance only if the polynomial is of the form $Z \times P^Z$ for some functor P.

Recursive equations involving F_3 will produce new examples of `Extractor` types, such as

$$T \triangleq F_3^{C, F_2^{Z, F_1^{T,P}}} = C \times (C \to Z + P \times T) \quad . \tag{7.4}$$

Heuristically, this type can be seen as an "infinite" exponential-polynomial type expression

$$T = C \times (C \to Z + P \times C \times (C \to Z + P \times C \times (C \to Z + ...))) \quad .$$

Types of this form are useful in some applications involving lazy streams.

We will now show how to define an `Extractor` instance for any recursive type defined using F_1, F_2, and/or F_3. A recursive type equation defining a type T can be written generally as $T \triangleq S^T$, where S is a type constructor built up by composing F_1, F_2, and/or F_3 in some way. (The type constructor S may use Z or other fixed types.) For each of F_1, F_2, and/or F_3, we implemented a function with type

$$\text{extractorF} : \text{Extractor}^A \to \text{Extractor}^{F^A} \quad .$$

Since S is a composition of F_1, F_2, and/or F_3, we are able to implement a function

$$\text{extractorS} : \text{Extractor}^A \to \text{Extractor}^{S^A} \quad .$$

The `Extractor` instance for the recursive type T is then defined recursively as

$$x^{:T \to Z} \triangleq \text{extractorS}\,(x) \quad .$$

The types match because the type T is equivalent to the type S^T. As long as the definition of the recursive type T is valid (i.e. the type recursion terminates), the extractor function will also terminate.

To illustrate this construction, let us derive an `Extractor` instance for the type T defined by Eq. (7.4).

The type constructor S is defined by

$$S^A \triangleq C \times (C \to Z + P \times A) \quad .$$

```
type S[A] = (C, C => Either[Z, (P, A)])
final case class TypeT(t: S[TypeT])   // Define the recursive type ‘TypeT‘.
```

To implement the function of type `Extractor[T] => Extractor[S[T]]`, which is $(T \to Z) \to S^T \to Z$, we begin with a typed hole

$$f^{:T \to Z} \to s^{:C \times (C \to Z + P \times T)} \to ???^{:Z} \quad .$$

To fill $???^{:Z}$, we could apply $f^{:T \to Z}$ to some value of type T; but the only value of type T can be obtained if we apply the function of type $C \to Z + P \times T$ to the given value of type C. So we write

$$f^{:T \to Z} \to c^{:C} \times g^{:C \to Z + P \times T} \to g(c) \triangleright ???^{:Z + P \times T \to Z} \quad .$$

The new typed hole has a function type. We can write the code in matrix notation as

$$\text{extractorS} \triangleq f^{:T \to Z} \to c^{:C} \times g^{:C \to Z + P \times T} \to g(c) \triangleright \begin{array}{|c||c|} \hline & Z \\ \hline\hline Z & \text{id} \\ \hline P \times T & p^{:P} \times t^{:T} \to f(t) \\ \hline \end{array} \quad .$$

```
def extractorS[A](f: Extractor[A]): Extractor[S[A]] = Extractor[S[A]] { case (c, g) =>
  g(c) match {
    case Left(z)          => z
    case Right((p, t))    => f.extract(t)
  }
}
```

The recursive construction defines an `Extractor` instance for T by a recursive equation,

$$\text{extractorT} \triangleq \text{extractorS}\,(\text{extractorT}) \quad . \tag{7.5}$$

```
def extractorT: Extractor[TypeT] = Extractor[TypeT] { case TypeT(t) =>
  extractorS(extractorT).extract(t)
}
```

To test this code, we define a value of type T while setting `C = Int`, `P = Boolean`, and `Z = String`:

```
val t = TypeT((10, x => Right((true, TypeT((x * 2, y => Left("abc")))))))

scala> extractorT.extract(t) // The recursive definition of ‘extractorT‘ terminates.
res0: String = abc
```

Why the recursion terminates The above code shows, perhaps surprisingly, that the recursive definition (7.5) terminates. How does that work? One would expect that a recursive definition of the form $x \triangleq f(x)$ generates an infinite loop, as it does in this example:

```
def f(x: Int): Int = x + 1
def x: Int = f(x)

scala> x
java.lang.StackOverflowError
```

The code for `extractorT` works because `extractorT` is a value of a *function* type, and because the presence of the case class `Extractor` forces us to rewrite Eq. (7.5) in the form of an "expanded function",

$$\text{extractorT} \triangleq t \to \text{extractorS}\,(\text{extractorT})\,(t) \quad .$$

```
def extractorT: Extractor[TypeT] = Extractor { case TypeT(t) => extractorS(extractorT).extract(t) }
```

Although the "expanded function" $t \to f(t)$ is equivalent to just f, there is an important difference: using "expanded functions" makes recursive definitions terminate.

To see an example of that, consider a recursive function f defined via an equation $f \triangleq k(f)$, where k is a suitable function. If we write Scala code as $f = k(f)$ using functions as values, we will create an infinite loop:

```
def k(f: Int => Int): Int => Int = { x => if (x <= 0) 1 else 2 * f(x - 1) }
def f_bad: Int => Int = k(f_bad)    // This definition is invalid!

scala> f_bad(4)               // Infinite loop: k(k(k(k(....)))(4)
java.lang.StackOverflowError
```

This code is clearly invalid. But if we expand the right-hand side of the recursive equation to

$$f \triangleq t \to k(f)(t)$$

instead of $f \triangleq k(f)$, the code will become valid:

```
def f: Int => Int = { x => k(f)(x) }   // This defines f(n) = 2^n for n ≥ 0.

scala> f(4)
res1: Int = 16
```

The recursive use of f now occurs *within* a function body, and so $k(f)$ is evaluated only when f is applied to an argument. This allows the recursive definition of f to terminate.

Summary We derived the constructions that create new types with `Extractor` typeclass instances from previous ones. Any number of these constructions can be combined to create a new type expression that will always have an `Extractor` instance. An example is the type expression

$$K^{Z,P,Q,R,S} \triangleq Z \times P + Q \times (Q \to Z + R \times (R \to Z \times S)) \quad .$$

Since the type K is built up step by step from fixed types via the product, co-product, and function constructions, an `Extractor` instance for K can be derived systematically with no guessing:

```
type K[Z,P,Q,R,S] = Either[(Z, P), (Q, Q => Either[Z, (R, R => (Z, S))])]
implicit def extractorK[Z,P,Q,R,S]: Extractor[Z, K[Z,P,Q,R,S]] = {    // Create Extractor for types:
  implicit val e1 = extractorPair[Z, Z, S]                            // Z × S.
  implicit val e2 = extractorFunc[Z, (Z, S), R]                       // R × (R ⇒ Z × S).
  implicit val e3 = extractorEither[Z, Z, (R, R => (Z, S))]           // Z + R × (R ⇒ Z × S).
  implicit val e4 = extractorFunc[Z, Either[Z, (R, R => (Z, S))], Q]  // Q × (Q ⇒ Z + R × (R ⇒ Z × S)).
  implicit val e5 = extractorPair[Z, Z, P]                            // Z × P.
  extractorEither[Z, (Z,P), (Q, Q => Either[Z, (R, R => (Z, S))])]    // Extractor for type K.
}
```

The `Extractor` typeclass is often used with `Z = String` as a way to "print" values of different types, and it is then called `Show`. When `Z = Array[Byte]`, the typeclass is often called a "serializer".

Construction	Type signature to implement	Results
The `Unit` type, or other fixed type C	Extractor^1 or Extractor^C	Extractor^Z
Product of extractor type A and any B	$\text{Extractor}^A \to \text{Extractor}^{A \times B}$	one possibility
Co-product of extractor types A and B	$\text{Extractor}^A \times \text{Extractor}^B \to \text{Extractor}^{A+B}$	one possibility
Function from or to another type C	$\text{Extractor}^A \to \text{Extractor}^{A \to C}$ or $\text{Extractor}^{C \to A}$	$\text{Extractor}^{C \times (C \to A)}$
Recursive types	$\text{Extractor}^A \to \text{Extractor}^{S^A}$ where $T \triangleq S^T$	Extractor^T

Table 7.1: Type constructions for the `Extractor` typeclass.

7.3.2 The `Eq` typeclass

In Scala, the built-in operation `==` is not type-safe because the code `x == y` will compile regardless of the types of x and y. W can replace `==` by a new operation `===` constrained to a typeclass called `Eq`, ensuring that types can be meaningfully compared for equality. The equality comparison for values of type A is a function of type $A \times A \to \mathbb{2}$ (where $\mathbb{2}$ denotes the `Boolean` type). A function of that type is wrapped by typeclass instances of `Eq`. We also define `===` as an extension method:

```
final case class Eq[A](equal: (A, A) => Boolean)
object Eq {
  implicit class EqOps[A: Eq](a: A) {
    def ===(b: A): Boolean = implicitly[Eq[A]].equal(a, b)
  }             // Define some typeclass instances here:
  implicit val eqInt: Eq[Int] = Eq[Int](_ == _)
  implicit val eqString: Eq[String] = Eq[String](_ == _)
}
```

We may, if desired, define an `Eq` typeclass instance for all types at once through the code

```
implicit def eqTypeA[A] = Eq[A](_ == _)
```

However, this will not work as expected for function types:

```
import Eq._

scala> ((n: Int) => n) === ((n: Int) => n)
res1: Boolean = false
```

The operations `==` and `===` do not seem to compare function values correctly. The equality comparison must satisfy the laws of identity, reflexivity and transitivity. The law of reflexivity states that $x = x$ for every x; so the comparison `x === x` should always return `true`. The example shown above violates that law when we choose $x \triangleq n^{:\text{Int}} \to n$.

Let us perform structural analysis for the `Eq` typeclass, defining $\text{Eq}^A \triangleq A \times A \to \mathbb{2}$. The results (see Table 7.2 below) will show which types can be reasonably compared for equality.

Fixed types All primitive types have `Eq` instances that use type-specific equality comparisons.

Products If A and B have equality comparisons, we can compare pairs of type $A \times B$ by comparing each part of the pair separately:

```
implicit def eqPair[A: Eq, B: Eq] = Eq[(A,B)]{ case ((a1, b1), (a2, b2)) => a1 === a2 && b1 === b2 }
```

It is easy to check that the identity, reflexivity, and transitivity laws hold for the new comparison operation if they hold for comparisons of A and B separately.

In this construction, it is important that both types A and B should have an `Eq` instance. If the above code contained only, say, the comparison `a1 === a2`, the resulting comparison operation would have violated the identity law of the equality operation (if $x = y$ then $f(x) = f(y)$ for any function f): we would have pairs such as $a \times b_1$ and $a \times b_2$ that would be "equal" according to this definition,

and yet many functions $f^{:A\times B\to C}$ exist such that $f(a\times b_1)\neq f(a\times b_2)$.

Co-products If A and B have equality comparisons, we can compare values of type $A+B$ while ensuring that a value of type $A+\mathbb{0}$ is never equal to a value of type $\mathbb{0}+B$:

```
implicit def eqEither[A: Eq, B: Eq] = Eq[Either[A, B]] {
  case (Left(a1), Left(a2))     => a1 === a2    // Compare a1 + 0 and a2 + 0.
  case (Right(b1), Right(b2))   => b1 === b2    // Compare 0 + b1 and 0 + b2.
  case _                        => false   // a + 0 is never equal to 0 + b.
}
```

The laws hold for the new operation because the code defines values of type `Either[A, B]` as equal only when the types are both `Left` or both `Right`. If the comparisons of types A and B satisfy the laws separately, the laws for $A+B$ will be satisfied separately for values of type `Left` and of type `Right`.

Defining the comparison operation in any other way (e.g., setting $a_1+\mathbb{0}\neq a_2+\mathbb{0}$ for all $a_1^{:A}$ and $a_2^{:A}$, or $\mathbb{0}+b_1=\mathbb{0}+b_2$ for all $b_1^{:B}$ and $b_2^{:B}$) would have violated the reflexivity law or the identity law.

Since the `Eq` typeclass has both the product and the co-product constructions, any polynomial type expression made of primitive types will also have an `Eq` typeclass instance.

Functions If A has an `Eq` instance, can we create an `Eq` instance for $R\to A$ where R is some other type? This would be possible with a function of type

$$\forall A.\,(A\times A\to\mathbb{2})\to(R\to A)\times(R\to A)\to\mathbb{2}\quad.\tag{7.6}$$

Here we assume that the type R is an arbitrary chosen type, so no values of type R can be computed from scratch. (This would not be the case when $R=\mathbb{1}$ or $R=\mathbb{2}$, say. But in those cases the type $R\to A$ can be simplified to a polynomial type, e.g. $\mathbb{1}\to A\cong A$ and $\mathbb{2}\to A\cong A\times A$, etc.) Without values of type R, we cannot compute any values of type A and cannot apply comparisons to them. Then the only possible implementations of the type signature (7.6) are constant functions returning `true` or `false`. However, the implementation that always returns `false` will violate the reflexivity law $x=x$. The implementation that always returns `true` is not useful.

We will be able to evaluate functions of type $R\to A$ if some chosen values of type R are available. Examples of that situation are types R of the form $R\cong\mathbb{1}+S$ for some type S; we then always have a chosen value $r_1\triangleq 1+\mathbb{0}^{:S}$ of type R. What if we compare functions only via their values at r_1?

```
def equals[R, A: Eq](f: R => A, g: R => A): Boolean = f(r1) === g(r1)  // Violates the identity law.
```

The above code defines a comparison operation that violates the identity law: there are many functions `f` and `g` that will give different results for arguments not equal to r_1.

Another way to see the problem is to write the type equivalence

$$R\to A\cong\mathbb{1}+S\to A\cong A\times(S\to A)\quad,$$

which reduces this case to the product construction we saw above. It follows that we need to have an `Eq` instance for $S\to A$ to define the equality operation for $R\to A$. If we have a chosen value of type S, e.g. if $S\cong\mathbb{1}+T$, we will again reduce the situation to the product construction with the function type $T\to A$. This process will end only if the type R has the form

$$R\cong\mathbb{1}+\mathbb{1}+\ldots+\mathbb{1}\quad,$$

i.e. if R has a *known* and finite number of distinct values. In that case, we can write code that applies functions of type $R\to A$ to every possible value of the argument of type R and compares all the resulting values of type A.

So, we can compare functions $f_1^{:R\to A}$ and $f_2^{:R\to A}$ only if we are able to check whether $f_1(r)=f_2(r)$ for *every* possible value of type R. We cannot implement such a comparison for a general type R.

We conclude that functions of type $R\to A$ cannot have an `Eq` instance for a general type R. A similar argument shows that functions of type $A\to R$ also cannot have a useful `Eq` instance. The only exceptions are types $R\to A$ or $A\to R$ that are equivalent to some polynomial types.

Recursive types Since all polynomial type expressions preserve `Eq` instances, the same logic can be applied to recursive polynomial types. For instance, lists and trees with `Eq`-comparable values are also `Eq`-comparable.

Consider a recursive polynomial type T defined using a polynomial functor $S^\bullet$,

$$T \triangleq S^T \quad .$$

The functor $S^\bullet$ may use other fixed types that have `Eq` instances. To construct the typeclass instance for T, we first implement a function eqS of type

$$\text{eqS} : \text{Eq}^A \to \text{Eq}^{S^A} \quad .$$

This function produces an `Eq` instance for S^A using `Eq` instances of A and of all other types that S^A depends on. The product and co-product constructions guarantee that it is always possible to implement this function for a polynomial functor $S^\bullet$. Then we define an `Eq` instance for T recursively:

$$\text{eqT} : \text{Eq}^T \quad , \qquad \text{eqT} \triangleq \text{eqS}\,(\text{eqT}) \quad .$$

The recursive equation for `eqT` needs to be implemented as an expanded function,

$$\text{eqT} \triangleq t^{:T} \times t^{:T} \to \text{eqS}\,(\text{eqT})\,(t \times t) \quad ,$$

and then, as we have seen in the previous section, the recursion will terminate.

As an example, let us define an `Eq` instance for the type T defined by $T \triangleq \text{Int} + T + \text{Int} \times T \times T$.

```
type S[A] = Either[Either[Int, A], (Int, (A, A))]
final case class T(s: S[T])                    // Recursive type equation T ≜ Int + T + Int × T × T.
def eqS[A](implicit ti: Eq[A]): Eq[S[A]] = {   // Function of type Eq[A] => Eq[S[A]].
  implicit val e1 = eqEither[Int, A]           // Instance for Int + A.
  implicit val e2 = eqPair[A, A]               // Instance for A × A.
  implicit val e3 = eqPair[Int, (A, A)]        // Instance for Int × A × A.
  eqEither[Either[Int, A], (Int, (A, A))]      // Instance for Int + A + Int × A × A.
}
implicit def eqT: Eq[T] = Eq { case (T(s1), T(s2)) => eqS(eqT).equal(s1, s2) }
```

To test that the recursion terminates, define a value of type T and run a comparison:

```
val t = T(Left(Right(T(Left(Left(10))))))   // t : 𝕆 + (10:Int + 𝕆 + 𝕆):T + 𝕆

scala> t === t
res0: Boolean = true
```

Checking laws for recursive instances To verify that the laws of identity, reflexivity, and transitivity will hold for the `Eq` instance we just defined, we need to use induction. The `Eq` instance for T was defined as a recursive function `eqT = eqS(eqT)`. We know that `eqS(x)` satisfies all the required laws if `x` satisfies them ("`eqS` preserves the laws"). Let us visualize what happens when the function `eqT` is applied to some values of type T. Are the laws satisfied in that case (e.g. is `eqT(t, t) == true` for all `t:T`)? The function `eqT` will call itself recursively whenever the body of `eqS` calls `eqT`. *Assuming* that all the recursive calls to `eqT` satisfy the laws, we find that the outer call to `eqT` also satisfies the laws because `eqS` preserves them. This corresponds to the inductive step.

Eventually the recursion will terminate; so, *some* calls of `eqT(t1, t2)` with certain values `t1` and `t2` will not cause any more recursive self-calls to `eqT`. For *those* values `t1` and `t2`, the function `eqT` will satisfy the required laws with no additional assumptions, because `eqS(eqT)(t1, t2)`, which satisfies the laws, does not call `eqT`. This corresponds to the base case of an inductive proof.

So, the recursively defined `Eq` instance is lawful. This argument is general and works in all cases when a typeclass instance t is defined by recursion, $t \triangleq s(t)$, via a function s that preserves the laws.

Construction	Type signature to implement	Results
The `Unit` type, or other primitive type	$\text{Eq}^T \triangleq T \times T \to 2$	the method `==`
Product of `Eq` types A and B	$\text{Eq}^A \times \text{Eq}^B \to \text{Eq}^{A \times B}$	one possibility
Co-product of `Eq` types A and B	$\text{Eq}^A \times \text{Eq}^B \to \text{Eq}^{A+B}$	one possibility
Recursive types	$\text{Eq}^A \to \text{Eq}^{S^A}$ where $T \triangleq S^T$	Eq^T

Table 7.2: Type constructions for the `Eq` typeclass.

Summary Instances of the `Eq` typeclass can be derived for any polynomial or recursive polynomial type expressions containing primitive types or type parameters constrained to be `Eq`-comparable. The derivation of the `Eq` instance is unambiguous and can be automated with libraries such as the `kittens`[2], `magnolia`[3] , or `scalaz-deriving`[4].

7.3.3 Semigroups

A type T has an instance of `Semigroup` when an associative binary operation of type $T \times T \to T$ is available. We will now apply structural analysis to this typeclass. The results are shown in Table 7.3.

Fixed types Each of the primitive types (`Boolean`, `Int`, `Double`, `String`, etc.) has at least one well-known associative binary operation that can be used to define a semigroup instance. Booleans have the conjunction and the disjunction operations; numbers can be added or multiplied, or the maximum or the minimum number chosen; strings can be concatenated or chosen in the alphabetical order. Examples 7.2.5.4 and 7.2.5.5 show several implementations of such binary operations. The `Unit` type has a trivially defined binary operation, which is also associative (since it always returns the same value). The same is true for any fixed type that has a chosen "default" value: the binary operation that always returns the default value is associative (although not likely to be useful).

Type parameters A semigroup instance parametric in type T means a value of type $\forall T. T \times T \to T$. There are two implementations of this type signature: $a^{:T} \times b^{:T} \to a$ and $a^{:T} \times b^{:T} \to b$. Both provide an associative binary operation, as Example 7.2.5.5(a) shows. So, any type T can be made into a "trivial" semigroup in one of these two ways. ("Trivial" semigroups are occasionally useful.)

Products If types A and B are semigroups, the product $A \times B$ can be also given a `Semigroup` instance. To compute that instance means, in the general case, to implement a function with type

$$\text{semigroupPair} : \forall (A, B). \text{Semigroup}^A \times \text{Semigroup}^B \to \text{Semigroup}^{A \times B} \quad .$$

Writing out the type expressions, we get the type signature

$$\text{semigroupPair} : \forall (A, B).\ (A \times A \to A) \times (B \times B \to B) \to (A \times B \times A \times B \to A \times B) \quad .$$

While this type signature can be implemented in a number of ways, we look for code that preserves information, in hopes of satisfying the associativity law. The code should be a function of the form

$$\text{semigroupPair} \triangleq f^{:A \times A \to A} \times g^{:B \times B \to B} \to a_1^{:A} \times b_1^{:B} \times a_2^{:A} \times b_2^{:B} \to ???^{:A} \times ???^{:B} \quad .$$

Since we are trying to define the new semigroup operation through the previously given operations f and g, it is natural to apply f and g to the given data a_1, a_2, b_1, b_2 and write

$$f^{:A \times A \to A} \times g^{:B \times B \to B} \to a_1^{:A} \times b_1^{:B} \times a_2^{:A} \times b_2^{:B} \to f(a_1, a_2) \times g(b_1, b_2) \quad .$$

[2] https://github.com/typelevel/kittens
[3] https://github.com/propensive/magnolia
[4] https://github.com/scalaz/scalaz-deriving

This code defines a new binary operation $\oplus_{A\times B}$ via the previously given $\oplus_A$ and $\oplus_B$ as

$$(a_1^{:A} \times b_1^{:B}) \oplus_{A\times B} (a_2^{:A} \times b_2^{:B}) = (a_1 \oplus_A a_2) \times (b_1 \oplus_B b_2) \quad . \tag{7.7}$$

```
def semigroupPair[A: Semigroup, B: Semigroup] =
  Semigroup[(A, B)]{ case ((a1, b1),(a2, b2)) => (a1 |+| a2, b1 |+| b2) }
```

This implementation satisfies the associativity law if the operations $\oplus_A$, $\oplus_B$ already do, i.e. if the results of computing $a_1 \oplus_A a_2 \oplus_A a_3$ and $b_1 \oplus_B b_2 \oplus_B b_3$ do not depend on the order of parentheses:

$$\begin{aligned}
&((a_1 \times b_1) \oplus_{A\times B} (a_2 \times b_2)) \oplus_{A\times B} (a_3 \times b_3) = ((a_1 \oplus_A a_2) \times (b_1 \oplus_B b_2)) \oplus_{A\times B} (a_3 \times b_3) \\
&\quad = (a_1 \oplus_A a_2 \oplus_A a_3) \times (b_1 \oplus_B b_2 \oplus_B b_3) \quad , \\
&(a_1 \times b_1) \oplus_{A\times B} ((a_2 \times b_2) \oplus_{A\times B} (a_3 \times b_3)) = (a_1 \times b_1) \oplus_{A\times B} ((a_2 \oplus_A a_3) \times (b_2 \oplus_B b_3)) \\
&\quad = (a_1 \oplus_A a_2 \oplus_A a_3) \times (b_1 \oplus_B b_2 \oplus_B b_3) \quad .
\end{aligned}$$

Co-products To compute a `Semigroup` instance for the co-product $A + B$ of two semigroups, we need

$$\text{semigroupEither} : \forall(A, B).\, \text{Semigroup}^A \times \text{Semigroup}^B \to \text{Semigroup}^{A+B} \quad .$$

Writing out the type expressions, we get the type signature

$$\text{semigroupEither} : \forall(A, B).\ (A \times A \to A) \times (B \times B \to B) \to (A + B) \times (A + B) \to A + B \quad .$$

Begin by writing a function with a typed hole:

$$\text{semigroupEither} \triangleq f^{:A\times A\to A} \times g^{:B\times B\to B} \to c^{:(A+B)\times(A+B)} \to ???^{:A+B} \quad .$$

Transforming the type expression $(A + B) \times (A + B)$ into an equivalent disjunctive type,

$$(A + B) \times (A + B) \cong A \times A + A \times B + B \times A + B \times B \quad ,$$

we can continue to write the function's code in matrix notation,

$$f^{:A\times A\to A} \times g^{:B\times B\to B} \to \left\| \begin{array}{c|cc} & A & B \\ \hline A \times A & ???^{:A\times A\to A} & ???^{:A\times A\to B} \\ A \times B & ???^{:A\times B\to A} & ???^{:A\times B\to B} \\ B \times A & ???^{:B\times A\to A} & ???^{:B\times A\to B} \\ B \times B & ???^{:B\times B\to A} & ???^{:B\times B\to B} \end{array} \right| \quad .$$

The matrix is 4×2 because the input type, $A \times A + A \times B + B \times A + B \times B$, is a disjunction with 4 parts, while the result type $A + B$ is a disjunction with 2 parts. In each row, we need to fill only one of the two typed holes because only one part of the disjunction $A + B$ can have a value.

To save space, we will omit the types in the matrices. The first and the last rows of the matrix must contain functions of types $A \times A \to A$ and $B \times B \to B$, and so it is natural to fill them with f and g:

$$f^{:A\times A\to A} \times g^{:B\times B\to B} \to \left\| \begin{array}{cc} f^{:A\times A\to A} & \mathbb{0} \\ ???^{:A\times B\to A} & ???^{:A\times B\to B} \\ ???^{:B\times A\to A} & ???^{:B\times A\to B} \\ \mathbb{0} & g^{:B\times B\to B} \end{array} \right| \quad .$$

The remaining two rows can be filled in four different ways:

$$f \times g \to \left\| \begin{array}{cc} f^{:A\times A\to A} & \mathbb{0} \\ a^{:A} \times b^{:B} \to a & \mathbb{0} \\ b^{:B} \times a^{:A} \to a & \mathbb{0} \\ \mathbb{0} & g^{:B\times B\to B} \end{array} \right| , \quad f \times g \to \left\| \begin{array}{cc} f & \mathbb{0} \\ \mathbb{0} & a^{:A} \times b^{:B} \to b \\ \mathbb{0} & b^{:B} \times a^{:A} \to b \\ \mathbb{0} & g \end{array} \right| ,$$

$$f \times g \to \left\| \begin{array}{cc} f^{:A\times A\to A} & \mathbb{0} \\ \mathbb{0} & a^{:A} \times b^{:B} \to b \\ b^{:B} \times a^{:A} \to a & \mathbb{0} \\ \mathbb{0} & g^{:B\times B\to B} \end{array} \right| , \quad f \times g \to \left\| \begin{array}{cc} f & \mathbb{0} \\ a^{:A} \times b^{:B} \to a & \mathbb{0} \\ \mathbb{0} & b^{:B} \times a^{:A} \to b \\ \mathbb{0} & g \end{array} \right| .$$

The Scala code corresponding to the four possible definitions of $\oplus$ is

```
def semigroupEither1[A: Semigroup, B: Semigroup] = Semigroup[Either[A, B]] {
  case (Left(a1), Left(a2))     =>  a1 |+| a2 // Here a1 |+| a2 is a_1 ⊕_A a_2.
  case (Right(b1), Right(b2))   =>  b1 |+| b2 // Here b1 |+| b2 is b_1 ⊕_B b_2.
  case (Left(a), Right(b))      =>  a         // "Take A" - discard all data of type B.
  case (Right(b), Left(a))      =>  a
}
def semigroupEither2[A: Semigroup, B: Semigroup] = Semigroup[Either[A, B]] {
  case (Left(a1), Left(a2))     =>  a1 |+| a2
  case (Right(b1), Right(b2))   =>  b1 |+| b2
  case (Left(a), Right(b))      =>  b         // "Take B" - discard all data of type A.
  case (Right(b), Left(a))      =>  b
}
def semigroupEither3[A: Semigroup, B: Semigroup] = Semigroup[Either[A, B]] {
  case (Left(a1), Left(a2))     =>  a1 |+| a2
  case (Right(b1), Right(b2))   =>  b1 |+| b2
  case (Left(a), Right(b))      =>  a         // "Take first" - discard y in x ⊕ y.
  case (Right(b), Left(a))      =>  b
}
def semigroupEither4[A: Semigroup, B: Semigroup] = Semigroup[Either[A, B]] {
  case (Left(a1), Left(a2))     =>  a1 |+| a2
  case (Right(b1), Right(b2))   =>  b1 |+| b2
  case (Left(a), Right(b))      =>  b         // "Take last" - discard x in x ⊕ y.
  case (Right(b), Left(a))      =>  a
}
```

The four different choices of the binary operation can be described as:

1. ("Take A") Discard all data of type B: $(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = a$ and $(\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = a$.
2. ("Take B") Discard all data of type A: $(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = b$ and $(\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = b$.
3. ("Take first") $x \oplus y$ discards y: $(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = a$ and $(\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = b$.
4. ("Take last") $x \oplus y$ discards x: $(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = b$ and $(\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = a$.

Does the semigroup law (7.1) hold for the new typeclass instance with any of these implementations?

It turns out that *all four* implementations are lawful. To verify the associativity law, we need to show that values such as `Left(x) |+| Left(y) |+| Right(z)` or `Left(x) |+| Right(y) |+| Right(z)` do not depend on the order of inserted parentheses. Expressions of the form $x \oplus y \oplus z$ can have 8 possible combinations of `Left` and `Right` types. Each of them needs to be checked against each of the 4 implementations of $\oplus$. Instead of doing 32 separate derivations, we will reason case by case.

First consider the case when all three values are of type `Left(x)`. In all four implementations, the binary operation reduces to the binary operation of the semigroup A, which is associative since we assume that A is a lawful semigroup:

```
Left(x) |+| Left(y) |+| Left(z) == Left(x |+| y |+| z) == Left(x) |+| ( Left(y) |+| Left(z) )
```

The same argument applies to three values of type `Right`. It remains to consider the "mixed" cases.

In the first implementation ("take A"), we discard all data of type `Right` from the expression $x \oplus y \oplus z$, keeping only data of type `Left`. It is clear that discarding data of type `Right` will yield the same result regardless of the order of parentheses. If more than one item of type `Left` remains, the data is aggregated with the operation $\oplus_A$. So, the results do not depend on the order of parentheses:

```
Right(x) |+| Right(y) |+| Left(z) == Left(z) == Right(x) |+| ( Right(y) |+| Left(z) )
Left(x) |+| Right(y) |+| Left(z) == Left(x |+| z) == Left(x) |+| ( Right(y) |+| Left(z) )
```

A similar argument shows that the second implementation ("take B") is also associative.

The implementation "take first" will select the first value whenever the types are mixed. Therefore

```
Left(x) |+| Left(y) |+| Right(z) == Left(x |+| y) == Left(x) |+| ( Left(y) |+| Right(z) )
Left(x) |+| Right(y) |+| z == Left(x) == Left(x) |+| ( Right(y) |+| z )    // Regardless of z.
Right(x) |+| Left(y) |+| z == Right(x) == Right(x) |+| ( Left(y) |+| z )   // Regardless of z.
Right(x) |+| Right(y) |+| Left(z) == Right(x |+| y) == Right(x) |+| ( Right(y) |+| Left(z) )
```

The results are independent of the parentheses. The same applies to the "take last" implementation.

Functions If A is a semigroup and E is any fixed type, are the types $A \to E$ and/or $E \to A$ semigroups? To create a `Semigroup` instance for $E \to A$ means to implement the type signature

$$\text{Semigroup}^A \to \text{Semigroup}^{E \to A} = (A \times A \to A) \to (E \to A) \times (E \to A) \to E \to A \quad .$$

An implementation that preserves information is

$$\text{semigroupFunc} \triangleq f^{:A \times A \to A} \to g_1^{:E \to A} \times g_2^{:E \to A} \to e^{:E} \to f(g_1(e), g_2(e)) \quad .$$

This defines the new $\oplus$ operation by $g_1 \oplus g_2 \triangleq e \to g_1(e) \oplus_A g_2(e)$.

```
def semigroupFunc[E, A: Semigroup] = Semigroup[E => A] { case (g1, g2) => e => g1(e) |+| g2(e) }
```

In the pipe notation, $e \triangleright (f \oplus g) = f(e) \oplus_A g(e)$. The associativity law holds for this operation:

$$\begin{aligned} &\text{left-hand side :} \quad e \triangleright ((f \oplus g) \oplus h) = (e \triangleright (f \oplus g)) \oplus_A h(e) = f(e) \oplus_A g(e) \oplus_A h(e) \quad . \\ &\text{right-hand side :} \quad e \triangleright (f \oplus (g \oplus h)) = f(e) \oplus_A (e \triangleright (g \oplus h)) = f(e) \oplus_A g(e) \oplus_A h(e) \quad . \end{aligned}$$

The type $A \to E$ only allows semigroup operations that discard the left or the right element: the type signature $f^{:A \times A \to A} \to h_1^{:A \to E} \times h_2^{:A \to E} \to ???^{:A \to E}$ can be implemented only by discarding f and one of h_1 or h_2. Either choice makes $A \to E$ into a trivial semigroup.

We have seen constructions that create new semigroups via products, co-products, and functions. Thus, any exponential-polynomial type expression built up from primitive types and/or existing semigroups is again a semigroup.

Recursive types A type T defined by a recursive type equation $T \triangleq S^T$ can have a semigroup instance when S^A is any exponential-polynomial type expression built up from primitive types, products, co-products, and the type parameter A. The known semigroup constructions guarantee that a typeclass instance `Semigroup[S[A]]` can be created out of `Semigroup[A]`. This gives us a function

$$\text{semigroupS} : \text{Semigroup}^A \to \text{Semigroup}^{S^A} \quad .$$

Then a semigroup instance for T is defined recursively as

$$\text{semigroupT} \triangleq \text{semigroupS}\,(\text{semigroupT}) \quad .$$

The recursive definition will terminate as long as we implement it in code as an expanded function. The associativity law holds for the semigroup T by induction, as shown at the end of Section 7.3.2.

Summary Any type expression whatsoever can have a `Semigroup` instance. Since the instances have several inequivalent implementations, automatic derivation of `Semigroup` instances is not often useful.

Construction	Type signature to implement	Results
The `Unit` type, or other fixed type C	Semigroup^C	two trivial semigroups
Product of semigroups A and B	$\text{Semigroup}^A \times \text{Semigroup}^B \to \text{Semigroup}^{A \times B}$	one possibility
Co-product of semigroups A and B	$\text{Semigroup}^A \times \text{Semigroup}^B \to \text{Semigroup}^{A+B}$	four possibilities
Function from another type E	$\text{Semigroup}^A \to \text{Semigroup}^{E \to A}$	one possibility
Recursive types	$\text{Semigroup}^A \to \text{Semigroup}^{S^A}$ where $T \triangleq S^T$	Semigroup^T

Table 7.3: Type constructions for the `Semigroup` typeclass.

7.3.4 Monoids

Since a monoid is a semigroup with a default value, a `Monoid` instance is a value of type

$$\text{Monoid}^A \triangleq (A \times A \to A) \times A \quad .$$

For the binary operation $A \times A \to A$, we can re-use the results of structural analysis for semigroups. Additionally, we will need to verify that the default value satisfies monoid's identity laws. The results are shown in Table 7.4.

Fixed types Each of the primitive types (`Boolean`, `Int`, `Double`, `String`, etc.) has a well-defined monoidal operation (addition or multiplication for numbers, concatenation for strings, and so on). The unit type, $\mathbb{1}$, also has a monoid instance where all methods return the unit value 1.

Type parameters This construction works for semigroups but *not* for monoids: the "trivial" semigroup operations $x \oplus y = x$ and $x \oplus y = y$ are not compatible with monoid's identity laws. (E.g. with the definition $x \oplus y = x$, no default value e could possibly satisfy the left identity law $e \oplus y = y$ because $e \oplus y = e$ for all y).

Products For two monoids A and B, a monoid instance for the product $A \times B$ is computed by

$$\text{monoidPair} : \forall (A, B).\, \text{Monoid}^A \times \text{Monoid}^B \to \text{Monoid}^{A \times B} \quad .$$

The empty value for the monoid $A \times B$ is $e_{A \times B} \triangleq e_A \times e_B$, the pair of empty values from the monoids A and B. The new binary operation is defined by Eq. (7.7) as in the pair semigroup construction. We can now verify the new monoid's identity laws, assuming that they hold for the monoids A and B:

$$(a_1 \times b_1) \oplus (e_A \times e_B) = (a_1 \oplus_A e_A) \times (b_1 \oplus_B e_B) = a_1 \times b_1 \quad ,$$
$$(e_A \times e_B) \oplus (a_2 \times b_2) = (e_A \oplus_A a_2) \times (e_B \oplus_B b_2) = a_2 \times b_2 \quad .$$

An implementation in Scala is

```
def monoidPair[A: Monoid, B: Monoid]: Monoid[(A, B)] = Monoid[(A, B)](
  { case ((a1, b1),(a2, b2)) => (a1 |+| a2, b1 |+| b2) },
  ( implicitly[Monoid[A]].empty, implicitly[Monoid[B]].empty )
)
```

Co-products For two monoids A and B, how can we implement a `Monoid` instance for $A + B$? We have seen four versions of the semigroup operation $\oplus$ for the type $A + B$. Independently of those, we need to define the empty element e_{A+B}, which must have type $A + B$. There are two possibilities:

$$e_{A+B} \triangleq e_A + \mathbb{0}^{:B} \qquad \text{or} \qquad e_{A+B} \triangleq \mathbb{0}^{:A} + e_B \quad .$$

It remains to see which of the eight combinations will satisfy the monoid identity laws,

$$(a + \mathbb{0}) \oplus e_{A+B} = a + \mathbb{0} \quad , \qquad e_{A+B} \oplus (a + \mathbb{0}) = a + \mathbb{0} \quad ,$$
$$(\mathbb{0} + b) \oplus e_{A+B} = \mathbb{0} + b \quad , \qquad e_{A+B} \oplus (\mathbb{0} + b) = \mathbb{0} + b \quad .$$

First choose $e_{A+B} = e_A + \mathbb{0}$; the reasoning for the other case will be quite similar. The first line above,

$$(a + \mathbb{0}) \oplus (e_A + \mathbb{0}) = a + \mathbb{0} \quad , \qquad (e_A + \mathbb{0}) \oplus (a + \mathbb{0}) = a + \mathbb{0} \quad ,$$

will hold because all four versions of the operation $\oplus$ will reduce to $\oplus_A$ on values of type $A + \mathbb{0}$. The second line, however, is compatible only with one version of the $\oplus$ operation, namely with "take B":

$$(a^{:A} + \mathbb{0}) \oplus (\mathbb{0} + b^{:B}) = b \qquad \text{and} \qquad (\mathbb{0} + b^{:B}) \oplus (a^{:A} + \mathbb{0}) = b \quad .$$

So, the co-product construction must choose one of the monoids, say B, as "preferred". The code is

```
def monoidEitherPreferB[A: Monoid, B: Monoid] = Monoid[Either[A, B]]( {
  case (Left(a1), Left(a2))     => Left(a1 |+| a2)
  case (Left(a), Right(b))      => Right(b) // "Take B".
  case (Right(b), Left(a))      => Right(b)
  case (Right(b1), Right(b2))   => Right(b1 |+| b2)
}, Left(implicitly[Monoid[A]].empty) )
```

Similarly, the choice $e_{A+B} \triangleq \mathbb{0} + e_B$ forces us to choose the version "take A" of the $\oplus$ operation.

Functions The semigroup construction for function types works for monoids. Exercise 7.4.2.3 will show that the function type $R \to A$ is a lawful monoid for any type R and any monoid A.

Additionally, the function type $R \to R$ is a monoid for any type R (even if R is not a monoid). The operation $\oplus$ and the empty value are defined as $f^{:R\to R} \oplus g^{:R\to R} \triangleq f \fatsemi g$ and $e_{R\to R} \triangleq \text{id}^R$. The code is

```
def monoidFunc1[R]: Monoid[R => R] = Monoid( (f, g) => f andThen g, identity )
```

The monoidal operation $\oplus$ is the forward function composition $f \fatsemi g$, so the monoid laws for this operation are the identity and the associativity laws of function composition (see Section 4.2.2).

We could also define the monoidal operation as the backward function composition, $f \circ g$.

Recursive types Can we define a `Monoid` instance for a type T defined by $T \triangleq S^T$, where $S^\bullet$ is some type constructor? As we have seen, products, co-products, and function type constructions preserve monoids. For any type built up via these constructions from monoids, a `Monoid` instance can be derived. These constructions cover all exponential-polynomial types. So, let us consider an exponential-polynomial type constructor S^A that contains a type parameter A, primitive types, and other known monoid types. For such type constructors $S^\bullet$, we will always be able to implement a function `monoidS` that derives a `Monoid` instance for S^A from a monoid instance for A:

$$\text{monoidS} : \text{Monoid}^A \to \text{Monoid}^{S^A} \quad .$$

A monoid instance for T is then defined recursively by

$$\text{monoidT} \triangleq \text{monoidS}\ (\text{monoidT}) \quad .$$

As we saw before, the code for this definition will terminate only if we implement it as a recursive function. However, the type Monoid^A is not a function type: it is a pair $(A \times A \to A) \times A$. To obtain a working implementation of `monoidT`, we need to rewrite that type into an equivalent function type,

$$\text{Monoid}^A = (A \times A \to A) \times A \cong (A \times A \to A) \times (\mathbb{1} \to A) \cong (\mathbb{1} + A \times A \to A) \quad ,$$

where we used the known type equivalences $A \cong \mathbb{1} \to A$ and $(A \to C) \times (B \to C) \cong A + B \to C$.

```
final case class Monoid[A](methods: Option[(A, A)] => A)
```

With this new definition of the `Monoid` typeclass (and with the appropriate changes to the code of `monoidPair`, `monoidEitherPreferB`, and `monoidFunc`), we can now implement the recursive construction.

To illustrate how that works, consider the exponential-polynomial type constructor $S^\bullet$ defined as

$$S^A \triangleq (\text{Int} + A) \times \text{Int} + \text{String} \times (A \to (A \to \text{Int}) \to A) \quad .$$

Construction	Type signature to implement	Results
The `Unit` type, or primitive types	$\text{Monoid}^{\mathbb{1}}$, $\text{Monoid}^{\text{Int}}$, etc.	custom code
Product of monoids A and B	$\text{Monoid}^A \times \text{Monoid}^B \to \text{Monoid}^{A \times B}$	one possibility
Co-product of monoid A and semigroup B	$\text{Monoid}^A \times \text{Semigroup}^B \to \text{Monoid}^{A+B}$	one possibility
Function from another type E	$\text{Monoid}^A \to \text{Monoid}^{E \to A}$	$\text{Monoid}^{E \to A}$
Recursive types	$\text{Monoid}^A \to \text{Monoid}^{S^A}$ where $T \triangleq S^T$	Monoid^T

Table 7.4: Type constructions for the `Monoid` typeclass.

```
type S[A] = Either[(Either[Int, A], Int), (String, A => (A => Int) => A)]
```

It is clear that S^A is built up from type constructions that preserve monoids at each step. So, we expect that the recursive type $T \triangleq S^T$ is a monoid. We first implement the function `monoidS`,

```
def monoidS[A](implicit ti: Monoid[A]): Monoid[S[A]] = {
  implicit val m0 = monoidEitherPreferB[Int, A]
  implicit val m1 = monoidPair[Either[Int, A], Int]
  implicit val m2 = monoidFunc[A, A => Int]
  implicit val m3 = monoidFunc[(A => Int) => A, A]
  implicit val m4 = monoidPair[String, A => (A => Int) => A]
  monoidEitherPreferB[(Either[Int, A], Int), (String, A => (A => Int) => A)]
}
```

We can now use this function to define the recursive type T and a `Monoid` instance for it,

```
final case class T(s: S[T])
def monoidT: Monoid[T] = Monoid[T] {
  case None             => T(monoidS[T](monoidT).methods(None))
  case Some((t1, t2))   => T(monoidS[T](monoidT).methods(Some(t1.s, t2.s)))
}
```

To test this code, create a value of type T and perform a computation:

```
val t = T(Right(("a", t => f => T(Left((Left(f(t)), 10))))))

scala> t |+| t
res0: T = T(Right((aa,<function1>)))
```

Another way of implementing the recursive construction is to write the `Monoid` typeclass using a `trait`. Although the code is longer, it is easier to read. The recursive instance is implemented by

```
def monoidT: Monoid[T] = new Monoid[T] {
  def empty: T = T(monoidS[T](monoidT).empty)   // This must be a ‘def empty‘, not a ‘val empty‘.
  def combine: (T, T) => T = (x, y) => T(monoidS[T](monoidT).combine(x.s, y.s))
}
```

The recursive definition of `monoidT` terminates because the methods of the `trait` are declared as `def` (not as `val`). The code of `monoidS` remains the same; we need to rewrite `monoidPair`, `monoidEitherPreferB`, and `monoidFunc` to accommodate the new definition of the `Monoid` typeclass. The full code is shown in Figure 7.1.

The monoid laws hold for the recursive instances by induction (see Section 7.3.2).

Summary A `Monoid` instance can be implemented, in at least one way, for *any* exponential-polynomial type expression (including recursive types) built from primitive types and other monoids.

```
trait Monoid[T] {
  def empty: T
  def combine: (T, T) => T
}
implicit val monoidInt: Monoid[Int] = new Monoid[Int] {
  def empty: Int = 0
  def combine: (Int, Int) => Int = _ + _
}
implicit val monoidString: Monoid[String] = new Monoid[String] {
  def empty: String = ""
  def combine: (String, String) => String = _ + _
}
implicit class MonoidOps[T: Monoid](t: T) {
  def |+|(a: T): T = implicitly[Monoid[T]].combine(t, a)
}
def monoidPair[A: Monoid, B: Monoid]: Monoid[(A, B)] = new Monoid[(A, B)] {
  def empty: (A, B) = (implicitly[Monoid[A]].empty, implicitly[Monoid[B]].empty)
  def combine: ((A, B), (A, B)) => (A, B) = {
    case ((a1, b1), (a2, b2)) => (a1 |+| a2, b1 |+| b2)
  }
}
def monoidEitherPreferB[A: Monoid, B: Monoid] = new Monoid[Either[A, B]] {
  def empty: Either[A, B] = Left(implicitly[Monoid[A]].empty)
  def combine: (Either[A, B], Either[A, B]) => Either[A, B] = {
    case (Left(a1), Left(a2))     => Left(a1 |+| a2)
    case (Left(a), Right(b))      => Right(b) // "Take B".
    case (Right(b), Left(a))      => Right(b)
    case (Right(b1), Right(b2))   => Right(b1 |+| b2)
  }
}
def monoidFunc[A: Monoid, E] = new Monoid[E => A] {
  def empty: E => A = _ => implicitly[Monoid[A]].empty
  def combine: (E => A, E => A) => E => A = {
    case (f, g) => e => f(e) |+| g(e)
  }
}
// This type constructor will be used below to define a recursive type T.
type S[A] = Either[(Either[Int, A], Int), (String, A => (A => Int) => A)]

// If we have a Monoid instance for A, we can get a Monoid instance for S[A].
def monoidS[A](implicit ti: Monoid[A]): Monoid[S[A]] = {
  implicit val m0 = monoidEitherPreferB[Int, A]
  implicit val m1 = monoidPair[Either[Int, A], Int]
  implicit val m2 = monoidFunc[A, A => Int]
  implicit val m3 = monoidFunc[(A => Int) => A, A]
  implicit val m4 = monoidPair[String, A => (A => Int) => A]
  monoidEitherPreferB[(Either[Int, A], Int), (String, A => (A => Int) => A)]
}
// Define a recursive type T and a Monoid instance for it.
final case class T(s: S[T])
implicit def monoidT: Monoid[T] = new Monoid[T] {
  def empty: T = T(monoidS[T](monoidT).empty) // Here, ‘val empty‘ will cause a StackOverflowError.
  def combine: (T, T) => T = (x, y) => T(monoidS[T](monoidT).combine(x.s, y.s))
}
val t = T(Right(("abc", t => f => T(Left((Left(f(t)), 10))))))
val e = implicitly[Monoid[T]].empty

scala> t |+| t |+| e   // Expect to see the string "abcabc".
res0: T = T(Right((abcabc,<function1>)))
```

Figure 7.1: Full code using a `trait` for implementing the `Monoid` typeclass and a recursive instance.

7.3.5 Pointed functors: motivation and laws

Section 7.2.6 showed how to implement typeclasses for type *constructors*, e.g. the `Functor` typeclass. Typeclass instances in such cases often contain a nested type quantifier such as $\forall A.\ (...)$, so the implementation needs to use Scala's `trait` with `def` methods inside. We will now look at some examples of typeclasses that add further methods to `Functor`. Chapter 6 performed a structural analysis of functors, which we will extend to the new typeclasses.

The first typeclass is a "pointed" functor. A functor type F^T represents, in a generalized sense, "wrapped" values of type T. A frequently used operation is to create a "wrapped" value of type F^T out of a single given value of type T. This operation, usually called `pure` in Scala libraries, is implemented as a function with a type signature

```
def pure[A]: A => F[A]
```

The code notation for this function is pu_F, and the type signature is written as $\text{pu}_F : \forall A.\ A \to F^A$.

Some examples of pointed functors in Scala are `Option`, `List`, `Try`, and `Future`. Each of these type constructors has a method that "wraps" a given single value:

```
val x: Option[Int] = Some(10)                    // A non-empty option that holds a value.
val y: List[String] = List("abc")                // A list that holds a single value.
val z: Try[Int] = Success(200)                   // A value computed without errors.
val f: Future[String] = Future.successful("OK")  // A 'Future' value that is already available.
```

As we can see, "wrapping a single value" means a different thing for each of the type constructors. Although the relevant methods of these type constructors are not called "`pure`", we can create a PTVF `pure[F]` that would be defined only for `F` that can "wrap" a single value. Such type constructors `F` are called "**pointed**". We may define the typeclass `Pointed` via this code:

```
trait Pointed[F[_]] { def pure[A]: A => F[A] }
```

Now we can implement instances of the `Pointed` typeclass for some functors:

```
implicit val pointedOption = new Pointed[Option] { def pure[A]: A => Option[A] = x => Some(x)    }
implicit val pointedList   = new Pointed[List]   { def pure[A]: A => List[A]   = x => List(x)    }
implicit val pointedTry    = new Pointed[Try]    { def pure[A]: A => Try[A]    = x => Success(x) }
```

The PTVF `pure` can be defined and used like this,

```
def pure[F[_]: Pointed, A](x: A): F[A] =
  implicitly[Pointed[F]].pure(x)

scala> pure[Option, Int](123)
res0: Option[Int] = Some(123)
```

When a pointed type constructor F is a functor, we may use both the functor's `map` method and the `pure` method. Do these two methods need to be compatible in some way? If we "wrap" a value 123 in a `List` and then apply `.map(x => x + 1)`, we expect to obtain a list containing 124; any other result would break our intuition about "wrapping". We can generalize this situation to an arbitrary value $x^{:A}$ wrapped using `pure` and a function $f^{:A\to B}$ applied to the wrapped value via `map`:

```
pure(x).map(f)    //   pu_F(x) ▷ f^↑F
```

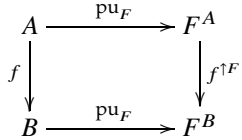
We expect that the result should be the same as a wrapped $f(x)$. This expectation can be formulated as a law, called the **naturality law** of `pure`; it must hold for any $f^{:A\to B}$:

```
pure(x).map(f) == pure(f(x))
```

$$\text{pu}_F(x) \triangleright f^{\uparrow F} = \text{pu}_F(f(x)) \quad .$$

In the $\triangleright$-notation, this law is $x \triangleright \text{pu}_F \triangleright f^{\uparrow F} = x \triangleright f \triangleright \text{pu}_F$ or equivalently $x \triangleright \text{pu}_F \fatsemi f^{\uparrow F} = x \triangleright f \fatsemi \text{pu}_F$. Since both sides of the law are functions applied to an arbitrary value $x^{:A}$, we can omit x and write

$$\text{pu}_F \fatsemi f^{\uparrow F} = f \fatsemi \text{pu}_F \quad . \tag{7.8}$$



This motivates the following definition: A functor $F^\bullet$ is **pointed** if there exists a fully parametric function $\text{pu}_T : \forall A.\ A \to F^A$ satisfying the naturality law (7.8) for any function $f^{:A\to B}$.

It turns out that we can avoid checking the naturality law for pointed functors if we use a trick: reduce `pure` to a simpler but equivalent form for which the law is satisfied automatically.

Both sides of the naturality law (7.8) are functions of type $A \to F^B$. The trick is to set $A = \mathbb{1}$ and $f^{:\mathbb{1}\to B} \triangleq (\_ \to b)$, where $b^{:B}$ is any fixed value. Both sides of the naturality law can then be applied to the unit value 1 and must evaluate to the same result,

$$1 \triangleright \text{pu}_F \triangleright (\_ \to b)^{\uparrow F} = 1 \triangleright f \triangleright \text{pu}_F \quad .$$

Since $1 \triangleright f = f(1) = b$, we find

$$\text{pu}_F(1) \triangleright (\_ \to b)^{\uparrow F} = \text{pu}_F(b) \quad . \tag{7.9}$$

The naturality law (7.8) applies to all types A, B and to any function $f^{:A\to B}$. Thus, Eq. (7.9) must apply to an arbitrary value $b^{:B}$ for any type B. That formula expresses the function pu_F through one value $\text{pu}_F(1)$ of type $F^{\mathbb{1}}$. This value can be viewed as a "wrapped unit" value.

To perform the same derivation in Scala syntax, we may write

```
val one: Unit = ()
val f: Unit => B = { _ => b }
pure(one).map(f) == pure(f(one)) == pure(b)    // Because f(one) == b.
```

It follows that if pu_F satisfies the naturality law then a single "wrapped unit" value of that function, $\text{pu}_F(1)$, is sufficient to recover the entire function $\text{pu}_F(b)$ by using the code

$$\text{pu}_F(b) \triangleq \text{pu}_F(1) \triangleright (\_ \to b)^{\uparrow F}$$

So, given just a "wrapped unit" value (denoted wu_F) of type $F^{\mathbb{1}}$, we can define a new function pu_F:

```
def pure[A](x: A): F[A] = wu.map { _ => x }
```

$$\text{pu}_F^{:A\to F^A} \triangleq x^{:A} \to \text{wu}_F \triangleright (\_ \to x)^{\uparrow F} \quad . \tag{7.10}$$

Does this function satisfy the naturality law with respect to an arbitrary $f^{:A\to B}$? It does:

$$\begin{aligned}
\text{expect to equal } x \triangleright f \fatsemi \text{pu}_F : \quad & x \triangleright \text{pu}_F \fatsemi f^{\uparrow F} \\
\text{definition of } \text{pu}_F : \quad & = \text{wu}_F \triangleright (_ \to x)^{\uparrow F} \fatsemi f^{\uparrow F} \\
\text{functor composition law of } F : \quad & = \text{wu}_F \triangleright ((_ \to x) \fatsemi f)^{\uparrow F} \\
\text{compute function composition} : \quad & = \text{wu}_F \triangleright (_ \to f(x))^{\uparrow F} \\
\text{definition of } \text{pu}_F : \quad & = \text{pu}_F(f(x)) \\
\triangleright\text{-notation} : \quad & = x \triangleright f \triangleright \text{pu}_F = x \triangleright f \fatsemi \text{pu}_F \quad .
\end{aligned}$$

Applied to the unit value, this new function gives $\text{pu}_F(1) = \text{wu}_F$ because

$$\begin{aligned}
\triangleright\text{-notation} : \quad & \text{pu}_F(1) = 1 \triangleright \text{pu}_F \\
\text{definition of } \text{pu}_F \text{ via } \text{wu}_F : \quad & = \text{wu}_F \triangleright (_ \to 1)^{\uparrow F} \\
\text{the function } (_ \to 1) \text{ is the identity function id}^{:\mathbb{1}\to\mathbb{1}} : \quad & = \text{wu}_F \triangleright \text{id}^{\uparrow F} \\
\text{functor identity law of } F : \quad & = \text{wu}_F \triangleright \text{id} = \text{wu}_F \quad .
\end{aligned}$$

To summarize our results: for any functor F,

- If F is a lawful pointed functor, its `pure` method satisfies Eq. (7.10) where $\text{wu}_F \triangleq \text{pu}_F(1)$ is a fixed "wrapped unit" value of type $F^{\mathbb{1}}$.
- If any "wrapped unit" value $\text{wu}_F : F^{\mathbb{1}}$ is given, we may define a `pure` method by Eq. (7.10) and make the functor F into a lawful pointed functor; the naturality law will be satisfied automatically. The value $\text{pu}_F(1)$ will be equal to the originally given value wu_F.

So, the function pu_F and the value wu_F are **computationally equivalent**: each one can be converted into the other and back, with no loss of information. We may define a pointed functor equivalently as a functor with a chosen value wu_F of type $F^{\mathbb{1}}$. When reasoning about pointed functors, it is simpler to use the definition via the "wrapped unit" `wu` because it has a simpler type and no laws. When writing code, the `pure` method is more convenient.

7.3.6 Pointed functors: structural analysis

To perform structural analysis, we begin with the known functor-building constructions from Chapter 6 and impose an additional requirement that a "wrapped unit" value $\text{wu}_F : F^{\mathbb{1}}$ should exist. As we have seen in the previous section, no additional laws need to be checked. The results are shown in Table 7.5. The typeclass can be defined by the simpler code,

```
final case class Pointed[F[_]](wu: F[Unit])
def pure[F[_]: Pointed : Functor, A](a: A): F[A] = implicitly[Pointed[F]].wu.map(_ => a)
```

When the type $F^{\mathbb{1}}$ has several distinct values, the choice of wu_F is application-dependent. E.g. if $F^A \triangleq \text{List}^A$, the type $\text{List}^{\mathbb{1}}$ has values such as an empty list `List[Unit]()`, the list of length 1, i.e. `List(())`, the list of length 2, i.e. `List((), ())`, etc. Each of these choices gives a valid `Pointed` instance for the `List` functor. It is up to the programmer to choose the `Pointed` instance that will be useful for the application at hand. In the case of `List`, the standard choice `wu == List(())` and correspondingly `pure(x) = List(x)` is motivated by the usage of the `List` type constructor to represent several possibilities, e.g. in a search problem; then the "pure" list represents the situation with only one possibility.

Nameless type-to-type functions To implement the `Pointed` typeclass instances for the following constructions, we need to use some advanced syntax features of Scala. In the previous sections, we wrote PTVFs parameterized by a type built out of other types, for example

```
def monoidPair[A: Monoid, B: Monoid]: Monoid[(A, B)] = ???
```

The function `monoidPair` creates a typeclass instance for the type $A \times B$, which is built out of the types A and B. In Scala, the typeclass instance is a value of type `Monoid[(A, B)]`. The analog for type constructors is a value of type `Pointed[L]` where the type constructor `L` is defined by the type alias

```
type L[A] = (F[A], G[A])
```

However, the following declaration of the analogous function `pointedPair` is invalid in Scala:

```
def pointedPair[F[_]: Pointed, G[_]: Pointed]: Pointed[L]     // Does not work in Scala.
```

It is not possible to use the type alias `L` within this function declaration, because the type alias needs to use the type parameters `F` and `G` that are defined only within the type signature of the function. To achieve that, we would need somehow to insert a new type alias declaration within the type signature of `pointedPair`, but the syntax of Scala does not support that:

```
def pointedPair[F[_]: Pointed, G[_]: Pointed]: {            // Not a valid Scala syntax.
    type L[A] = (F[A], G[A]) // Temporarily define a type constructor L, and now use it:
    Pointed[L] }
```

The return type is required to be `Pointed[L]`, where `L` needs to be a type expression that defines a type constructor, i.e. a type-to-type function with a *single* type parameter. Writing a type alias with parameters `F` and `G`,

```
type L[F[_], G[_], A] = (F[A], G[A])
def pointedPair[F[_]: Pointed, G[_]: Pointed]: Pointed[L[F, G]]    // Still incorrect.
```

will not help because the type expression `L[F, G]` in `Pointed[L[F, G]]` is not a valid type constructor with one type parameter. We cannot define a new type name (such as `L`) within the type signature; what we need is a *nameless* type-to-type function (i.e. a nameless type constructor).

The special Scala plugin called the "kind projector"[5] adds syntax for nameless type constructors. The syntax is similar to defining a nameless function: for instance, the pair functor $F^\bullet \times G^\bullet$ is defined as `Lambda[X => (F[X], G[X])]`. Such type expressions can be understood as nameless type-to-type functions. When using the "kind projector" plugin, the syntax for defining `pointedPair` is

```
def pointedPair[F[_]: Pointed, G[_]: Pointed]: Pointed[Lambda[X => (F[X], G[X])]] = ???
```

[5] https://github.com/typelevel/kind-projector

Scala 3 will support a shorter syntax for nameless type-to-type functions, `[X] => (F[X], G[X])`. We will use the "kind projector" syntax in this book's code examples.

Fixed types A constant functor $\text{Const}^{Z,\bullet}$ is defined as $\text{Const}^{Z,A} \triangleq Z$, where Z is a fixed type. A "wrapped unit" value is thus also a value of type Z. Since we cannot produce values of an arbitrary type Z from scratch, the constant functor is not pointed in general. The constant functor *will* be pointed when there exists a known value of type Z. Examples are $Z = \mathbb{1}$ or $Z = \mathbb{1} + U$ (where U is an arbitrary type). If we know that Z is equivalent to $\mathbb{1} + U$, we will be able to produce a value of type Z as $1 + \mathbb{0}^{:U}$. In that case, we set $\text{wu}_{\text{Const}^{Z,\bullet}} = 1 + \mathbb{0}^{:U}$.

```
type Const[Z, A] = Z
def pointedOpt[U]: Pointed[Lambda[X => Const[Option[U], X]]] = Pointed(None: Const[Option[U], Unit])
```

Other cases are types such as $Z = \text{Int}$, $Z = \text{String}$, etc., that have well-known "default" values.

Type parameters The identity functor $\text{Id}^A \triangleq A$ is pointed since $\text{Id}^{\mathbb{1}} = \mathbb{1}$, and we can set $\text{wu}_{\text{Id}} = 1$.

```
type Id[A] = A
def pointedId: Pointed[Id] = Pointed[Id](())
```

The other functor constructions that work by setting type parameters are functor compositions. If F and G are two functors or two contrafunctors then $F \circ G$ is a functor. The functor $F \circ G$ is pointed when we can create a value of type $F^{G^{\mathbb{1}}}$. If both F and G are pointed, we can apply F's `pure` method to $\text{wu}_G : G^{\mathbb{1}}$ and obtain a value of type $F^{G^{\mathbb{1}}}$.

```
def pointedFoG[F[_]: Pointed : Functor, G[_]: Pointed]: Pointed[Lambda[X => F[G[X]]]]
  = Pointed[Lambda[X => F[G[X]]]](pure[F, G[Unit]](implicitly[Pointed[G]].wu))
```

The case when F and G are contrafunctors requires us to assume that F belongs to the "pointed contrafunctor" typeclass (see Section 7.3.8 below). A pointed contrafunctor has a "wrapped unit" value of type $F^{\mathbb{1}}$, which can be transformed into F^A for any type A by using `contramap` with a constant function $A \to \mathbb{1}$:

```
def cpure[F[_]: Pointed : Contrafunctor, A]: F[A] = implicitly[Pointed[F]].wu.cmap(_ => ())
```

In this way, we can create a value of type $F^{G^{\mathbb{1}}}$. The contrafunctor G does not need to be pointed.

```
def pointedCFoG[F[_]: Pointed : Contrafunctor, G[_]]: Pointed[Lambda[X => F[G[X]]]] =
  Pointed[Lambda[X => F[G[X]]]](cpure[F, G[Unit]])
```

Products If F and G are two pointed functors, is the functor product $L^A \triangleq F^A \times G^A$ a pointed functor? We need to produce a value $\text{wu}_L : F^{\mathbb{1}} \times G^{\mathbb{1}}$, and we have values $\text{wu}_F : F^{\mathbb{1}}$ and $\text{wu}_G : G^{\mathbb{1}}$. It is clear that we must set $\text{wu}_L = \text{wu}_F \times \text{wu}_G$.

```
def pointedFxG[F[_]: Pointed, G[_]: Pointed]: Pointed[Lambda[X => (F[X],G[X])]] =
  Pointed[Lambda[X => (F[X],G[X])]]((implicitly[Pointed[F]].wu, implicitly[Pointed[G]].wu))
```

Co-products If F and G are two pointed functors, is the functor co-product $L^A \triangleq F^A + G^A$ a pointed functor? We need to produce a value $\text{wu}_L : F^{\mathbb{1}} + G^{\mathbb{1}}$, and we have values $\text{wu}_F : F^{\mathbb{1}}$ and $\text{wu}_G : G^{\mathbb{1}}$. There are two choices, $\text{wu}_L = \text{wu}_F + \mathbb{0}^{:G^{\mathbb{1}}}$ and $\text{wu}_L = \mathbb{0}^{:F^{\mathbb{1}}} + \text{wu}_G$, both making $L^\bullet$ a pointed functor.

It is sufficient if just $F^\bullet$ is a pointed functor: $\text{wu}_L \triangleq \text{wu}_F + \mathbb{0}^{:G^{\mathbb{1}}}$ is a `Pointed` typeclass instance for $F^\bullet + G^\bullet$, even if $G^\bullet$ is not pointed.

```
def pointedEitherFG[F[_]: Pointed, G[_]]: Pointed[Lambda[X => Either[F[X],G[X]]]] =
  Pointed[Lambda[X => Either[F[X],G[X]]]](Left(implicitly[Pointed[F]].wu))
```

Functions If C is any contrafunctor and F is a pointed functor, the exponential functor $L^A \triangleq C^A \to F^A$ will be pointed if we are able to produce a value $\text{wu}_L : C^{\mathbb{1}} \to F^{\mathbb{1}}$. We already have a value $\text{wu}_F : F^{\mathbb{1}}$, and we cannot use a value of type $C^{\mathbb{1}}$ with a general contrafunctor C. So, we have to set $\text{wu}_L \triangleq (\_ \to \text{wu}_F)$. This makes L into a pointed functor.

```
def pointedFuncFG[F[_]: Pointed, C[_]]: Pointed[Lambda[X => C[X] => F[X]]] =
```

```
Pointed[Lambda[X => C[X] => F[X]]](_ => implicitly[Pointed[F]].wu)
```

Recursive types The recursive construction for functors (see Statement 6.2.3.7) assumes a bifunctor $S^{\bullet,\bullet}$ and defines a recursive functor $F^\bullet$ via the type equation $F^A \triangleq S^{A,F^A}$. The functor $F^\bullet$ will be pointed if we can compute a value wu_F of type $F^{\mathbb{1}}$. The type $F^{\mathbb{1}}$ is a recursive type defined via the type equation $F^{\mathbb{1}} \triangleq S^{\mathbb{1},F^{\mathbb{1}}}$. If that type is not void, i.e. if there exists some value of that type, we will be able to define wu_F as that value.

How can we construct wu_F for a given bifunctor S? The procedure can be derived by structural analysis of S (see Section 7.5.1 below). For *polynomial* bifunctors S (which is the most often used kind of bifunctors), the necessary and sufficient condition is that the type $S^{\mathbb{1},\mathbb{0}}$ should be non-void. If we can create a value of type $S^{\mathbb{1},\mathbb{0}}$, arguments shown in Section 7.5.1 will guarantee that we can also create a value of type $F^{\mathbb{1}}$, and so the recursive functor F will be pointed.

As an example, consider the polynomial bifunctor $S^{A,R} \triangleq A + A \times R$. The corresponding recursive functor $F^A \triangleq S^{A,F^A} = A + A \times F^A$ is the non-empty list (see Example 3.3.2.1, Table 6.3, and Statement 6.2.3.7). The type F^A can be (non-rigorously) viewed as an "infinite disjunction"

$$F^A = A + A \times (A + A \times (A + ...)) = A + A \times A + A \times A \times A + ...$$

Since the type $S^{\mathbb{1},\mathbb{0}} = \mathbb{1} + \mathbb{1} \times \mathbb{0} \cong \mathbb{1}$ is non-void, the necessary and sufficient condition holds, so we expect that the recursive construction will work. The type $F^{\mathbb{1}}$ is defined by

$$F^{\mathbb{1}} \triangleq S^{\mathbb{1},F^{\mathbb{1}}} = \mathbb{1} + \mathbb{1} \times F^{\mathbb{1}} \cong \mathbb{1} + F^{\mathbb{1}} \quad .$$

This type can be (non-rigorously) viewed as an "infinite disjunction"

$$F^{\mathbb{1}} = \mathbb{1} + \mathbb{1} + \mathbb{1} + ...$$

It is clear that a value of that type can be computed, for example, as

$$\text{wu}_F = 1 + \mathbb{0} + \mathbb{0} + ... \cong 1 + \mathbb{0}^{:F^{\mathbb{1}}} \quad .$$

In Scala, this is `Left(())`. So, a `Pointed` typeclass instance for F is implemented by

```
type S[A, R] = Either[A, (A, R)]
final case class F[A](s: S[A, F[A]])
implicit val pointedF: Pointed[F] =
  Pointed( F(Left(())) )
```

The corresponding `pure` method will use F's `map` to transform $\text{wu}_F = 1 + \mathbb{0} + \mathbb{0} + ...$ into

$$a + \mathbb{0} + \mathbb{0} + ... : A + A \times A + A \times A \times A + ... \quad .$$

The `pure` method of F creates a non-empty list with a single element $a^{:A}$.

Summary Can we recognize a pointed functor F by looking at its type expression, e.g.

$$F^{A,B} \triangleq ((\mathbb{1} + A \to \text{Int}) \to A \times B) + \text{String} \times A \times A \quad ?$$

This type constructor is a functor in both A and B, and we ask whether F is pointed with respect to A and/or with respect to B.

To answer this question with respect to A, we set $A = \mathbb{1}$ in $F^{A,B}$ and obtain the type expression

$$F^{\mathbb{1},B} = ((\mathbb{1} + \mathbb{1} \to \text{Int}) \to \mathbb{1} \times B) + \text{String} \times \mathbb{1} \times \mathbb{1} \cong ((\mathbb{2} \to \text{Int}) \to B) + \text{String} \quad .$$

The functor F will be pointed with respect to A if we can compute a value of this type from scratch. At the outer level, this type expression is a disjunctive type with two parts; it is sufficient to compute one of the parts. Can we compute a value of type $(\mathbb{2} \to \text{Int}) \to B$? Since the parameter B is an arbitrary, unknown type, we cannot construct values of type B using a given value of type $\mathbb{2} \to \text{Int}$. The remaining possibility is to compute the second part of the co-product, which is $\text{String} \times \mathbb{1} \times \mathbb{1}$. We are able to compute a value of this type because `String` is a fixed type with a known default value (an empty string). So, we conclude that $F^{A,B}$ is pointed with respect to A.

Construction	Type signature to implement	Results
Constant functor returning a fixed type Z	value of type Z	Z has a default
Identity functor	$\mathbb{1}$	one possibility
Composition of pointed functors/contrafunctors	$\text{Pointed}^{F^\bullet} \times \text{Pointed}^{G^\bullet} \to \text{Pointed}^{F^{G^\bullet}}$	one possibility
Product of pointed functors F and G	$\text{Pointed}^{F^\bullet} \times \text{Pointed}^{G^\bullet} \to \text{Pointed}^{F^\bullet \times G^\bullet}$	one possibility
Co-product of a pointed functor F and any G	$\text{Pointed}^{F^\bullet} \times \text{Functor}^{G^\bullet} \to \text{Pointed}^{F^\bullet + G^\bullet}$	one possibility
Function from any C to a pointed F	$\text{Pointed}^{F^\bullet} \times \text{Contrafunctor}^{C^\bullet} \to \text{Pointed}^{C^\bullet \to F^\bullet}$	one possibility
Recursive types	$\text{Pointed}^{F^\bullet} \to \text{Pointed}^{S^{\bullet,F^\bullet}}$ where $F^A \triangleq S^{A,F^A}$	$\text{Pointed}^{F^\bullet}$

Table 7.5: Type constructions for the `Pointed` functor typeclass.

Considering now the type parameter B, we set $B = \mathbb{1}$ and obtain

$$F^{A,\mathbb{1}} = ((\mathbb{1} + A \to \text{Int}) \to A \times \mathbb{1}) + \text{String} \times A \times A \quad .$$

The type A is now an arbitrary and unknown type, so we cannot compute any values of A or $\text{String} \times A \times A$ from scratch. The function type $(\mathbb{1} + A \to \text{Int}) \to A \times \mathbb{1}$ cannot be implemented because a value of type A cannot be computed from a function $\mathbb{1} + A \to \text{Int}$ that *consumes* values of type A. So, $F^{A,B}$ is not pointed with respect to B.

7.3.7 Co-pointed functors

Pointed functors provide the functionality of wrapping a given value in a "pure wrapper". Another useful operation is *extracting* a value from a given "wrapper":

```
def extract[F[_], A]: F[A] => A
```

$$\text{ex} : \forall A.\, F^A \to A \quad .$$

Functors having this operation are called **co-pointed**. We may define the `Copointed` typeclass as

```
trait Copointed[F[_]] { def ex[A]: F[A] => A }
def extract[F[_]: Copointed, A](f: F[A]): A = implicitly[Copointed[F]].ex(f)
```

The `extract` function must be fully parametric and obey the **naturality law** (compare to Eq. (7.8)),

$$\begin{array}{ccc} F^A & \xrightarrow{\text{ex}_F} & A \\ {\scriptstyle f^{\uparrow F}}\downarrow & & \downarrow{\scriptstyle f} \\ F^B & \xrightarrow{\text{ex}_F} & B \end{array}$$

$$\text{ex}_F \fatsemi f = f^{\uparrow F} \fatsemi \text{ex}_F \quad . \tag{7.11}$$

The naturality law formulates our expectation that the extractor function somehow "selects" a value of type A among all the values wrapped by F^A, and the "selection" works independently of the values. If all wrapped values are transformed by a function f into wrapped values of type B, the extractor function will still select a value of type B in the same way as it did for values of type A. So, the result will be the same as if we first extracted a value of type A and then transformed that value with f.

Both sides of the law (7.11) are functions of type $F^A \to B$. We saw in the previous section that the `pure` method of the `Pointed` typeclass is computationally equivalent to a single chosen value of type $F^\mathbb{1}$ when F is a functor. That provides a simpler form of the `Pointed` typeclass. For co-pointed functors, there is no simpler form of the `extract` method. If we set $A = \mathbb{1}$ and $f^{:\mathbb{1}\to B} \triangleq (1 \to b)$ in the naturality law, both sides will become functions of type $F^\mathbb{1} \to B$. But the type $F^\mathbb{1}$ might be void, or a value of type $F^\mathbb{1}$ may not be computable via fully parametric code. So, we cannot deduce any further information from the naturality law of co-pointed functors.

However, if F is a pointed functor, we *will* have a chosen value $\text{wu}_F : F^\mathbb{1}$ to which we may then apply the naturality law (7.11) and obtain

$$\text{wu}_F \triangleright \text{ex}_F \fatsemi f = \text{wu}_F \triangleright f^{\uparrow F} \fatsemi \text{ex}_F \quad .$$

Calculating both sides separately, we find

$$\mathrm{wu}_F \triangleright \mathrm{ex}_F \fatsemi f = \mathrm{wu}_F \triangleright \mathrm{ex}_F \triangleright f = 1 \triangleright f = b \quad .$$
$$\mathrm{wu}_F \triangleright f^{\uparrow F} \fatsemi \mathrm{ex}_F = b \triangleright \mathrm{pu}_F \fatsemi \mathrm{ex}_F \quad .$$

It remains to show that $b \triangleright \mathrm{pu}_F \fatsemi \mathrm{ex}_F = b$. This can hold for all $b^{:B}$ only if

$$\mathrm{pu}_F \fatsemi \mathrm{ex}_F = \mathrm{id} \quad .$$

This additional **compatibility law** is a consequence of naturality laws if the functor F is pointed and co-pointed at the same time.

Let us perform structural analysis for co-pointed functors; the results are shown in Table 7.6.

Fixed types A constant functor $\mathrm{Const}^{Z,A} \triangleq Z$ is *not* co-pointed because we cannot implement $\forall A.\, Z \to A$ (a value of an arbitrary type A cannot be computed from a value of a fixed type Z).

Type parameters The identity functor $\mathrm{Id}^A \triangleq A$ is co-pointed with $\mathrm{ex} \triangleq \mathrm{id}^{:A \to A}$. An identity function will always satisfy any naturality law.

```
type Id[A] = A
def copointedId: Copointed[Id] = new Copointed[Id] { def ex[A]: Id[A] => A = identity }
```

Composition of two co-pointed functors F, G is co-pointed:

$$\mathrm{ex}_{F \circ G} \triangleq h^{:F^{G^A}} \to \mathrm{ex}_G(\mathrm{ex}_F(h)) \quad \text{or equivalently} \quad \mathrm{ex}_{F \circ G} = \mathrm{ex}_F \fatsemi \mathrm{ex}_G \quad .$$

```
def copointedFoG[F[_]: Copointed, G[_]: Copointed]: Copointed[Lambda[X => F[G[X]]]] =
  new Copointed[Lambda[X => F[G[X]]]] {
    def ex[A]: F[G[A]] => A = extract[F,G[A]] _ andThen extract[G, A]
  }
```

The naturality law holds for $\mathrm{ex}_{F \circ G}$ because

$$\begin{aligned}
\text{expect to equal } \mathrm{ex}_{F\circ G} \fatsemi f: \quad & \underline{f^{\uparrow F \circ G}} \fatsemi \underline{\mathrm{ex}_{F\circ G}} \\
\text{definition of } f^{\uparrow F\circ G}\text{, see Eq. (6.18)}: \quad & = \underline{(f^{\uparrow G})^{\uparrow F} \fatsemi \mathrm{ex}_F} \fatsemi \mathrm{ex}_G \\
\text{naturality law for } \mathrm{ex}_F: \quad & = \mathrm{ex}_F \fatsemi \underline{f^{\uparrow G} \fatsemi \mathrm{ex}_G} \\
\text{naturality law for } \mathrm{ex}_G: \quad & = \underline{\mathrm{ex}_F \fatsemi \mathrm{ex}_G} \fatsemi f = \mathrm{ex}_{F\circ G} \fatsemi f \quad .
\end{aligned}$$

Products If functors F and G are co-pointed, we can implement a function of type $F^A \times G^A \to A$ in two different ways: by discarding F^A or by discarding G^A. With either choice, the functor product $F^\bullet \times G^\bullet$ is made into a co-pointed functor. For instance, if we choose to discard G^A then the functor G will not need to be co-pointed, and the code for the `extract` method will be

$$\mathrm{ex}_{F\times G} \triangleq f^{:F^A} \times g^{:G^A} \to \mathrm{ex}_F(f) = \nabla_1 \fatsemi \mathrm{ex}_F \quad ,$$

where we used the pair projection function $\nabla_1 \triangleq (a \times b \to a)$.

```
def copointedFxG[F[_]: Copointed, G[_]]: Copointed[Lambda[X => (F[X],G[X])]] =
  new Copointed[Lambda[X => (F[X], G[X])]] {
    def ex[A]: ((F[A], G[A])) => A = { case (f, g) => extract(f) }
  }
```

The following calculation verifies the naturality law (7.11) for this definition of $\text{ex}_{F\times G}$:

$$\begin{aligned}
\text{expect to equal } \text{ex}_{F\times G} \fatsemi f: \quad & f^{\uparrow F\times G} \fatsemi \text{ex}_{F\times G} \\
\text{definition of } f^{\uparrow F\times G}\text{, see Eq. (6.13)}: \quad & = (f^{\uparrow F} \boxtimes f^{\uparrow G}) \fatsemi \text{ex}_{F\times G} \\
\text{definition of } \text{ex}_{F\times G}: \quad & = (f^{\uparrow F} \boxtimes f^{\uparrow G}) \fatsemi \nabla_1 \fatsemi \text{ex}_F \\
\text{use the property } (p \boxtimes q) \fatsemi \nabla_1 = \nabla_1 \fatsemi p: \quad & = \nabla_1 \fatsemi f^{\uparrow F} \fatsemi \text{ex}_F \\
\text{naturality law of } \text{ex}_F: \quad & = \nabla_1 \fatsemi \text{ex}_F \fatsemi f \\
\text{definition of } \text{ex}_{F\times G}: \quad & = \text{ex}_{F\times G} \fatsemi f \quad .
\end{aligned}$$

To demonstrate the property $(p \boxtimes q) \fatsemi \nabla_1 = \nabla_1 \fatsemi p$ used in this proof, apply both sides to $a \times b$:

$$\begin{aligned}
&(a \times b) \triangleright (p \boxtimes q) \fatsemi \nabla_1 = (a \times b) \triangleright (p \boxtimes q) \triangleright \nabla_1 = (p(a) \times q(b)) \triangleright \nabla_1 = p(a) \quad , \\
&(a \times b) \triangleright \nabla_1 \fatsemi p = (a \times b) \triangleright \nabla_1 \triangleright p = a \triangleright p = p(a) \quad .
\end{aligned}$$

Co-products For co-pointed functors F and G, there is only one possible implementation of the type signature $\text{ex}_{F+G} : F^A + G^A \to A$, given that we have functions ex_F and ex_G:

```
def copointedEitherFG[F[_]: Copointed, G[_]: Copointed]:
                Copointed[Lambda[X => Either[F[X],G[X]]]] =
  new Copointed[Lambda[X => Either[F[X],G[X]]]] {
    def ex[A]: Either[F[A], G[A]] => A = {
      case Left(f)    => extract(f)
      case Right(g)   => extract(g)
    }
  }
```

$$\text{ex}_{F+G} \triangleq \left\| \begin{array}{c|c} & A \\ \hline F^A & \text{ex}_F \\ G^A & \text{ex}_G \end{array} \right\| \quad .$$

To verify that ex_{F+G} satisfies the naturality law, we compute:

$$\begin{aligned}
\text{expect to equal } \text{ex}_{F+G} \fatsemi f: \quad & f^{\uparrow F+G} \fatsemi \text{ex}_{F+G} \\
\text{definition of } f^{\uparrow F+G}\text{, see Eq. (6.14)}: \quad & = \left\| \begin{array}{cc} f^{\uparrow F} & \mathbb{0} \\ \mathbb{0} & f^{\uparrow G} \end{array} \right\| \fatsemi \left\| \begin{array}{c} \text{ex}_F \\ \text{ex}_G \end{array} \right\| \\
\text{matrix function composition}: \quad & = \left\| \begin{array}{c} f^{\uparrow F} \fatsemi \text{ex}_F \\ f^{\uparrow G} \fatsemi \text{ex}_G \end{array} \right\| \\
\text{naturality laws for } \text{ex}_F \text{ and } \text{ex}_G: \quad & = \left\| \begin{array}{c} \text{ex}_F \fatsemi f \\ \text{ex}_G \fatsemi f \end{array} \right\| = \left\| \begin{array}{c} \text{ex}_F \\ \text{ex}_G \end{array} \right\| \fatsemi f = \text{ex}_{F+G} \fatsemi f \quad .
\end{aligned}$$

Functions An exponential functor of the form $L^A \triangleq C^A \to P^A$ (where C is a contrafunctor and P is a functor) will be co-pointed if we can implement a function of type $\forall A.\, (C^A \to P^A) \to A$,

$$\text{ex}_L \triangleq h^{:C^A \to P^A} \to ???^{:A} \quad .$$

Since the type A is arbitrary, the only way of computing a value of type A is somehow to use the function h. The only way of using h is to apply it to a value of type C^A, which will yield a value of type P^A. So, we need to assume that we can somehow create a value of type C^A, for any type A. We may call a contrafunctor C with a method `cpure` of type $\forall A.\, C^A$ a **pointed contrafunctor**. Assuming that C is pointed and denoting its `cpure` method by pu_C, we can thus compute a value of type P^A as $h(\text{pu}_C)$. To extract A from P^A, we need to assume additionally that P is co-pointed and use its method $\text{ex}_P : P^A \to A$. Finally we have

$$\text{ex}_L \triangleq h^{:C^A \to P^A} \to \text{ex}_P(h(\text{pu}_C)) \qquad \text{or equivalently} \qquad h^{:C^A \to P^A} \triangleright \text{ex}_L = \text{pu}_C \triangleright h \triangleright \text{ex}_P \quad . \tag{7.12}$$

To verify the naturality law, we apply both sides to an arbitrary $h^{:C^A \to P^A}$ and compute

$$\begin{aligned}
\text{expect to equal } h \triangleright \text{ex}_L \mathbin{;} f : \quad & \triangleright \text{ex}_L h \triangleright f^{\uparrow L} \mathbin{;} \text{ex}_L = (h \triangleright f^{\uparrow L}) \\
\text{use Eq. (7.12)} : \quad & = \text{pu}_C \triangleright (h \triangleright f^{\uparrow L}) \triangleright \text{ex}_P = \text{pu}_C \triangleright \underline{(h \triangleright f^{\uparrow L}) \mathbin{;} \text{ex}_P} \\
\text{definition of } f^{\uparrow L}\text{, see Eq. (6.16)} : \quad & = \text{pu}_C \triangleright f^{\downarrow C} \mathbin{;} h \mathbin{;} \underline{f^{\uparrow P} \mathbin{;} \text{ex}_P} \\
\text{naturality law for } \text{ex}_P : \quad & = \text{pu}_C \triangleright f^{\downarrow C} \mathbin{;} h \mathbin{;} \text{ex}_P \mathbin{;} f \quad .
\end{aligned}$$

We expect the last expression to equal

$$h \triangleright \text{ex}_L \mathbin{;} f = \text{pu}_C \triangleright h \triangleright \text{ex}_P \triangleright f = \text{pu}_C \triangleright h \mathbin{;} \text{ex}_P \mathbin{;} f \quad .$$

This is possible only if $\text{pu}_C \triangleright f^{\downarrow C} = \text{pu}_C$ for all f. This motivates us to assume that law as the **naturality law** of pu_C for pointed contrafunctors. With that last assumption, we have finished proving the naturality law for ex_L. The code for ex_L is

```
def copointedFunc[C[_]: Pointed, P[_]: Copointed]: Copointed[Lambda[X => C[X] => P[X]]] =
  new Copointed[Lambda[X => C[X] => P[X]]] {
    def ex[A]: (C[A] => P[A]) => A = h => extract[P, A](h(cpure[C, A]))
  }      // In Scala 2.13: h => cpure[C, A] pipe h pipe extract[P, A] as in h ⇒ pu_C ▷ h ▷ ex_P
```

We will analyze pointed contrafunctors and their naturality law in Section 7.3.8.

Recursive types Consider a functor F defined by a recursive equation $F^A \triangleq S^{A,F^A}$ where S is a bifunctor (see Section 6.2.2). The functor F is co-pointed if a method $\text{ex}_F : (F^A \to A) \cong (S^{A,F^A} \to A)$ can be defined. Since the recursive definition of F uses F^A as a type argument in S^{A,F^A}, we may assume (by induction) that an extractor function $F^A \to A$ is already available when applied to the recursively used F^A. Then we can use the `bimap` method of S to map $S^{A,F^A} \to S^{A,A}$. It remains to extract a value of type A out of a bifunctor value $S^{A,A}$. We call a bifunctor S **co-pointed** if a fully parametric function $\text{ex}_S : S^{A,A} \to A$ exists satisfying the corresponding naturality law

$$\text{ex}_S \mathbin{;} f = \text{bimap}_S(f)(f) \mathbin{;} \text{ex}_S \quad . \tag{7.13}$$

Assuming that S is co-pointed, we can finally define ex_F by recursion,

$$\text{ex}_F \triangleq s^{:S^{A,F^A}} \to s \triangleright (\text{bimap}_S(\text{id})(\text{ex}_F)) \triangleright \text{ex}_S \quad \text{or equivalently} \quad \text{ex}_F \triangleq \text{bimap}_S(\text{id})(\text{ex}_F) \mathbin{;} \text{ex}_S \quad .$$

To verify the naturality law for ex_F, we denote recursive uses by an overline and compute:

$$\begin{aligned}
\text{expect to equal } \text{ex}_F \mathbin{;} f : \quad & \underline{f^{\uparrow F}} \mathbin{;} \text{ex}_F \\
\text{definition of } f^{\uparrow F}\text{, see Eq. (6.20)} : \quad & = \text{bimap}_S(f)(\overline{f^{\uparrow F}}) \mathbin{;} \underline{\text{ex}_F} \\
\text{definition of } \text{ex}_F : \quad & = \underline{\text{bimap}_S(f)(\overline{f^{\uparrow F}}) \mathbin{;} \text{bimap}_S(\text{id})(\overline{\text{ex}_F})} \mathbin{;} \text{ex}_S \\
\text{bifunctor composition law (6.10)} : \quad & = \text{bimap}_S(f \mathbin{;} \text{id})(\underline{\overline{f^{\uparrow F} \mathbin{;} \text{ex}_F}}) \mathbin{;} \text{ex}_S \\
\text{naturality law of } \text{ex}_F : \quad & = \underline{\text{bimap}_S(f)(\overline{\text{ex}_F \mathbin{;} f})} \mathbin{;} \text{ex}_S \\
\text{bifunctor composition law in reverse} : \quad & = \text{bimap}_S(\text{id})(\overline{\text{ex}_F}) \mathbin{;} \text{bimap}_S(f)(\overline{f}) \mathbin{;} \text{ex}_S \\
\text{naturality law (7.13) of } \text{ex}_S : \quad & = \underline{\text{bimap}_S(\text{id})(\overline{\text{ex}_F}) \mathbin{;} \text{ex}_S} \mathbin{;} f \\
\text{definition of } \text{ex}_F : \quad & = \text{ex}_F \mathbin{;} f \quad .
\end{aligned}$$

An example illustrating the recursive construction is the bifunctor $S^{A,R} \triangleq A + R \times R$ that defines F^A to be the binary tree functor,

$$F^A \triangleq S^{A,F^A} = A + F^A \times F^A \quad .$$

The bifunctor S is co-pointed because there exists a suitable function $\text{ex}_S : S^{A,A} \to A$.

Construction	Type signature to implement	Results
Identity functor	$\text{id} : A \to A$	one possibility
Composition of co-pointed functors	$\text{Copointed}^{F^\bullet} \times \text{Copointed}^{G^\bullet} \to \text{Copointed}^{F^{G^\bullet}}$	one possibility
Product of co-pointed functor F and any G	$\text{Copointed}^{F^\bullet} \times \text{Functor}^{G^\bullet} \to \text{Copointed}^{F^\bullet \times G^\bullet}$	one possibility
Co-product of co-pointed functors F and G	$\text{Copointed}^{F^\bullet} \times \text{Copointed}^{G^\bullet} \to \text{Copointed}^{F^\bullet + G^\bullet}$	one possibility
Function from pointed C to co-pointed F	$\text{Pointed}^{C^\bullet} \times \text{Copointed}^{F^\bullet} \to \text{Copointed}^{C^\bullet \to F^\bullet}$	one possibility
Recursive types	$\text{Copointed}^{F^\bullet} \to \text{Copointed}^{S^{\bullet, F^\bullet}}$ where $F^A \triangleq S^{A, F^A}$	$\text{Copointed}^{F^\bullet}$

Table 7.6: Type constructions for the `Copointed` functor typeclass.

```
type S[A, R] = Either[A, (R, R)]
def exS[A]: S[A, A] => A = {
  case Left(a)           => a
  case Right((a1, a2))   => a1  // Could be a2.
}
```

In the code notation, the function ex_S is written as

$$\text{ex}_S \triangleq \begin{array}{c||c} & A \\ \hline\hline A & \text{id} \\ \hline A \times A & \nabla_1 \end{array} .$$

This function extracts the left-most leaf of a binary tree.

To implement the co-pointed instance ex_F for the binary tree functor, we need to use the implementation of bimap_S,

```
def bimap_S[A,B,P,Q](f: A => B)(g: P => Q): S[A, P] => S[B, Q] = {
  case Left(a)         => Left(f(a))
  case Right((x, y))   => Right((g(x), g(y)))
}
```

Now we can define the recursive type constructor $F^\bullet$ and a co-pointed instance for it,

```
final case class F[A](s: S[A, F[A]])
val copointedF: Copointed[F] = new Copointed[F1] {
  def ex[A]: F1[A] => A = { case F1(s) => exS1(bimap_S(identity[A], ex[A])(s)) }
}
```

The naturality law holds for the recursive instance by induction (see Section 7.3.2).

Summary Can we recognize a co-pointed functor F by looking at its type expression, e.g.

$$F^{A,B} \triangleq ((\mathbb{1} + A \to \text{Int}) \to A \times B) + \text{String} \times A \times A \quad ?$$

The constructions shown in this section tell us that a co-product of two co-pointed functors is again co-pointed. Let us first check whether $\text{String} \times A \times A$ is co-pointed. We can certainly extract a value of type A out of $\text{String} \times A \times A$, but not a value of type B. So, there is no hope for $F^{A,B}$ to be co-pointed with respect to the parameter B.

It remains to consider A as the type parameter. The type constructor $\text{String} \times A \times A$ is co-pointed, but we still need to check $(\mathbb{1} + A \to \text{Int}) \to A \times B$, which is a function type. The function construction requires $\mathbb{1} + A \to \text{Int}$ to be a pointed contrafunctor and $A \times B$ to be a co-pointed functor (with respect to A). It is clear that $A \times B$ is co-pointed with respect to A since we have $\nabla_1 : A \times B \to A$. It remains to check that the contrafunctor $C^A \triangleq \mathbb{1} + A \to \text{Int}$ is pointed. A contrafunctor $C^\bullet$ is pointed if values of type $C^{\mathbb{1}}$ can be computed (see Section 7.3.8); this requires us to compute a value of type $\mathbb{1} + \mathbb{1} \to \text{Int}$. One such value is `{ _ => 0 }`, a constant function that always returns the integer 0.

We conclude that $F^{A,B}$ is co-pointed with respect to A but not co-pointed with respect to B.

7.3.8 Pointed contrafunctors

In the previous section, the function-type construction required a contrafunctor $C^\bullet$ to have a method pu_C of type $\forall A.\, C^A$; we called such contrafunctors **pointed**. We also needed to assume that the naturality law holds for all functions $f^{:A\to B}$,

$$\text{pu}_C \triangleright f^{\downarrow C} = \text{pu}_C \quad \text{or equivalently} \quad \text{cmap}_C(f^{:A\to B})(\text{pu}_C^{:C^B}) = \text{pu}_C^{:C^A} \quad . \tag{7.14}$$

$$\begin{array}{ccc} \text{pu}_C : C^B & & B \\ \Big\downarrow {\scriptstyle \text{cmap}_C(f)} & & \Big\uparrow {\scriptstyle f} \\ \text{pu}_C : C^A & & A \end{array}$$

We may simplify the formulation of the typeclass by setting $B = \mathbb{1}$ in the naturality law (7.14) and denoting $\text{wu}_C \triangleq \text{pu}_C^{\mathbb{1}}$. The law (7.14) then gives

$$\text{pu}_C^A = \text{wu}_C \triangleright (\_^{:A} \to 1)^{\downarrow C} \quad . \tag{7.15}$$

In this way, we express the `pure` method through a chosen value $\text{wu}_C : C^{\mathbb{1}}$. For the same reasons as in the case of pointed functors, pu_C and wu_C are computationally equivalent. The law (7.14) for pu_C will be satisfied automatically if pu_C is defined via Eq. (7.15). To verify that, compute

$$\begin{aligned} \text{expect to equal } \text{pu}_C^A : \quad & \text{pu}_C^B \triangleright f^{\downarrow C} \\ \text{use definition (7.15)} : \quad & = \text{wu}_C \triangleright \underline{(\_^{:B} \to 1)^{\downarrow C} \triangleright f^{\downarrow C}} \\ \text{composition law for contrafunctor } C : \quad & = \text{wu}_C \triangleright (\underline{f \mathbin{⨟} (\_^{:B} \to 1)})^{\downarrow C} \\ \text{compute function composition} : \quad & = \text{wu}_C \triangleright (\_^{:A} \to 1)^{\downarrow C} = \text{pu}_C^A \quad . \end{aligned}$$

So, a pointed contrafunctor instance for $C^\bullet$ is equivalent to a chosen value of type $C^{\mathbb{1}}$.

```
final case class Pointed[F[_]](wu: F[Unit])
def cpure[F[_]: Pointed : Contrafunctor, A]: F[A] = implicitly[Pointed[F]].wu.cmap(_ => ())
```

Here we used a `Contrafunctor` typeclass (see Example 7.4.1.7 below). We can now apply structural analysis to pointed contrafunctors, similarly to Section 6.2.4. The results are shown in Table 7.7.

Fixed types This construction is the same as for pointed functors (Section 7.3.6). A fixed type Z gives a constant contrafunctor $C^A \triangleq Z$. Since $C^{\mathbb{1}} = Z$, the constant contrafunctor is pointed if we have a chosen value of type Z; this will be the case, for instance, if $Z = \mathbb{1} + U$ for some type U.

Type parameters Since the identity functor $\text{Id}^A \triangleq A$ is not a contrafunctor, it remains to consider the functor compositions C^{F^A} and F^{C^A} where $C^\bullet$ is a contrafunctor and $F^\bullet$ is a functor.

If C is pointed, we can always obtain a value pu_C of type C^A for any type A, in particular for $A = F^{\mathbb{1}}$ (whether or not a value of type $F^{\mathbb{1}}$ can be computed). So, $C^{F^\bullet}$ is a pointed contrafunctor whenever $C^\bullet$ is one, for any (not necessarily pointed) functor F.

```
def pointedCoF[C[_]: Pointed: Contrafunctor, F[_]]: Pointed[Lambda[X => C[F[X]]]] =
  Pointed[Lambda[X => C[F[X]]]](cpure[C, F[Unit]])
```

Creating a value of type $F^{C^{\mathbb{1}}}$ requires F to have a `pure` method that could be applied to a value of type $C^{\mathbb{1}}$ to compute a value of type $F^{C^{\mathbb{1}}}$. So, $F^{C^\bullet}$ is pointed whenever both $C^\bullet$ and $F^\bullet$ are pointed.

```
def pointedFoC[C[_]: Pointed, F[_]: Pointed : Functor]: Pointed[Lambda[X => F[C[X]]]] =
  Pointed[Lambda[X => F[C[X]]]](pure[F, C[Unit]](implicitly[Pointed[C]].wu))
```

Products The construction is the same as for pointed functors: If we have values of type $C^{\mathbb{1}}$ and $D^{\mathbb{1}}$, we can compute the pair $C^{\mathbb{1}} \times D^{\mathbb{1}}$. This makes the product contrafunctor $L^A \triangleq C^A \times D^A$ pointed if both $C^\bullet$ and $D^\bullet$ are pointed contrafunctors.

Construction	Type signature to implement	Results
Constant functor returning a fixed type Z	value of type Z	Z has a default
Composition of pointed functors/contrafunctors	$\text{Pointed}^{F^\bullet} \times \text{Pointed}^{G^\bullet} \to \text{Pointed}^{F G^\bullet}$	one possibility
Product of pointed contrafunctors F and G	$\text{Pointed}^{F^\bullet} \times \text{Pointed}^{G^\bullet} \to \text{Pointed}^{F^\bullet \times G^\bullet}$	one possibility
Co-product of a pointed F and any G	$\text{Pointed}^{F^\bullet} \times \text{Contrafunctor}^{G^\bullet} \to \text{Pointed}^{F^\bullet + G^\bullet}$	one possibility
Function from a functor F to a pointed C	$\text{Pointed}^{C^\bullet} \times \text{Functor}^{F^\bullet} \to \text{Pointed}^{F^\bullet \to C^\bullet}$	one possibility
Recursive types	$\text{Pointed}^{C^\bullet} \to \text{Pointed}^{S^{\bullet, C^\bullet}}$ where $C^A \triangleq S^{A, C^A}$	$\text{Pointed}^{C^\bullet}$

Table 7.7: Type constructions for the `Pointed` contrafunctor typeclass.

Co-products The construction is the same as for pointed functors: If at least one of the contrafunctors $C^\bullet$ and $D^\bullet$ is pointed, we can create a `Pointed` instance for the co-product contrafunctor $L^A \triangleq C^A + D^A$ as either $\text{wu}_L = \text{wu}_C + \mathbb{0}^{:D^\mathbb{1}}$ or $\text{wu}_L = \mathbb{0}^{:C^\mathbb{1}} + \text{wu}_D$.

Functions The exponential contrafunctor construction is $L^A \triangleq F^A \to C^A$, where $C^\bullet$ is a contrafunctor and $F^\bullet$ is a functor. To create a value $\text{wu}_L : L^\mathbb{1}$ means to create a function of type $F^\mathbb{1} \to C^\mathbb{1}$. That function cannot use its argument of type $F^\mathbb{1}$ for computing a value $C^\mathbb{1}$ since F is an arbitrary functor. So, wu_L must be a constant function $(\_^{:F^\mathbb{1}} \to \text{wu}_C)$, where we assumed that a value $\text{wu}_C : C^\mathbb{1}$ is available. Thus, $F^A \to C^A$ is pointed when C is a pointed contrafunctor and F is any functor.

```
def pointedFuncFC[C[_]: Pointed, F[_]]: Pointed[Lambda[X => F[X] => C[X]]] =
      Pointed[Lambda[X => F[X] => C[X]]](_ => implicitly[Pointed[C]].wu)
```

Recursive types The recursive construction for contrafunctors (see Statement 6.2.4.3) is $C^A \triangleq S^{A, C^A}$ where $S^{A,R}$ is a contrafunctor in A and a functor in R. Values of type $C^\mathbb{1}$ will exist when the recursive type equation $T \triangleq S^{\mathbb{1},T}$ defines a non-void type T. This condition is similar to that for pointed functors, and the resulting construction is the same.

Summary Can we recognize a pointed contrafunctor C by looking at its type expression, e.g.

$$C^{A,B} \triangleq (\mathbb{1} + A \to B) + (\text{String} \times A \times B \to \text{String}) \text{ with respect to type parameter } A?$$

We need to set $A = \mathbb{1}$ and try to create a value $\text{wu} : C^{\mathbb{1},B}$. In this example, $C^{\mathbb{1},B} = (\mathbb{1} + \mathbb{1} \to B) + (\text{String} \times \mathbb{1} \times B \to \text{String})$. A value of this type is $\text{wu}_C \triangleq \mathbb{0} + (s^{:\text{String}} \times 1 \times b^{:B} \to s)$. So, the contrafunctor $C^{A,B}$ is pointed with respect to A.

7.4 Summary

What problems can we solve now?

- Define arbitrary PTVFs using typeclass constraints on type parameters.
- Define typeclasses and typeclass instances for types and for type constructors.
- Implement `Monoid`, `Functor`, and other standard typeclasses, and prove their laws.
- Use known constructions to derive typeclass instances from previous ones.

What problems cannot be solved with these tools?

- *Automatically* derive type class instances for given data types or type constructors.
- Combine typeclasses and express dependencies, e.g. typeclass `TC1` requires `TC2` and `TC3`.

We may want to write code such as

```
type F[A] = (A => Int) => A                    // Define a type constructor.
implicit val functorF: Functor[F] = implement  // Automatically implement typeclass instance for F.
implicit val pointedF: Pointed[F] = implement  // Automatically use the function-type construction.
```

However, no currently available library provides such functionality. Also, typeclass instances are not always derived uniquely, as we have seen in several cases (e.g. the co-product construction for monoids or pointed functors).

We will discuss how to combine typeclasses in Section 7.5.6 below.

7.4.1 Solved examples

Example 7.4.1.1 Define a PTVF with type signature `def bitsize[T]: Int` such that `bitsize[Short]` returns 16, `bitsize[Int]` returns 32, and `bitsize[Long]` returns 64. For all other types `T`, the expression `bitsize[T]` should remain undefined.

Solution The function `bitsize[T]` needs to take an additional implicit argument that will be available only for `T = Short`, `T = Int`, and `T = Long`. To implement that, we need to define a new type constructor, say `HasBitsize[T]`, and create implicit values of the corresponding types. The new type constructor defines a typeclass whose instances need to carry the information about the bit size:

```
final case class HasBitsize[T](size: Int)
object HasBitsize {
  implicit val bitsizeShort = HasBitsize[Short](16)
  implicit val bitsizeInt   = HasBitsize[Int]  (32)
  implicit val bitsizeLong  = HasBitsize[Long] (64)
}
```

Now we can define the function `bitsize` as a PTVF,

```
def bitsize[T](implicit ti: HasBitsize[T]): Int = ti.size
```

The instance argument such as `ti: HasBitsize[T]` is sometimes called the "evidence" argument because its presence provides "evidence" that the type `T` belongs to the type domain of the typeclass.

We can check that the function `bitsize` is defined only for supported types:

```
scala> bitsize[Long]
res0: Int = 64

scala> bitsize[String]
<console>:15: error: could not find implicit value for evidence parameter of type HasBitsize[String]
       bitsize[String]
               ^
```

The current implementation of `HasBitsize` allows the programmer to add new types to its type domain whenever necessary. For example, the following code will add support for the `Boolean` type so that `bitsize[Boolean]` will evaluate to `1`:

```
implicit val bitsizeBoolean = HasBitsize[Boolean](1)
```

In some applications, it is important that the type domain of a PTVF should remain fixed (e.g. as defined in a library). To prevent the programmer from creating any further values of type `HasBitsize`, we could make it a non-case class whose constructor is declared as a `private` function like this:

```
final class HasBitsize[T] private (val size: Int) // Not a case class; the constructor is private.
object HasBitsize {            // The companion object is allowed to call the private constructor.
  implicit val bitsizeShort = new HasBitsize[Short](16)
  implicit val bitsizeInt   = new HasBitsize[Int]  (32)
  implicit val bitsizeLong  = new HasBitsize[Long] (64)
}
```

The code of `bitsize[T]` remains unchanged. With these definitions, no further typeclass instances can be created by any code outside of the companion object `HasBitsize`:

```
scala> implicit val bitsizeBoolean = new HasBitsize[Boolean](1)
<console>:16: error: constructor HasBitsize in class HasBitsize cannot be accessed in object $iw
       implicit val bitsizeBoolean = new HasBitsize[Boolean](1)
                                     ^
```

An implementation via a `trait` requires longer code but brings no significant advantages:

```
trait HasBitsize[T] { def size: Int } // Declare the trait as ‘sealed‘ to prohibit further instances.
object HasBitsize {
  implicit val bitsizeShort = new HasBitsize[Short]{ def size: Int = 16 }
  implicit val bitsizeInt   = new HasBitsize[Int]  { def size: Int = 32 }
  implicit val bitsizeLong  = new HasBitsize[Long] { def size: Int = 64 }
}
```

Example 7.4.1.2 Define a `Monoid` instance for the type $\mathbb{1} + (\text{String} \to \text{String})$.

Solution We look for suitable monoid constructions (Section 7.3.4) that build up the given type expression from simpler parts. Since the type expression $\mathbb{1} + (\text{String} \to \text{String})$ is a co-product at the outer level, we must start with the co-product construction, which requires us to choose one of the parts of the disjunction, say $\text{String} \to \text{String}$, as the "preferred" monoid. Next, we need to produce `Monoid` instances for $\mathbb{1}$ and for $\text{String} \to \text{String}$. While the `Unit` type has a unique `Monoid` instance, there are several for $\text{String} \to \text{String}$. The function-type construction gives two possible monoid instances: the monoid $R \to A$ with $R = A = \text{String}$, and the function composition monoid. Let us choose the latter. The code using the monoid constructions from Section 7.3.4 can then be written as

```
val monoidX: Monoid[Either[Unit, String => String]] = {
  implicit val m1 = Monoid[Unit]( (x, y) => (), () )
  implicit val m2: Monoid[String => String] = monoidFunc1[String]
  monoidEitherPreferB[Unit, String => String]
)
```

We can translate the constructions into code for a `Monoid` instance for the type `Option[String => String]`:

```
val monoidX: Monoid[Option[String => String]] = Monoid( {
  case (None, None)          => None
  case (None, Some(f))       => Some(f)
  case (Some(f), None)       => Some(f)
  case (Some(f), Some(g))    => Some(f andThen g)
}, None )
```

Example 7.4.1.3 Show that if A is a monoid and B is a semigroup then $A + B$ is a monoid.

Solution The co-product construction for $A + B$ (where both A, B are monoids) has two implementations: one of the empty elements e_A or e_B must be chosen as the empty element for the monoid $A + B$. If B is not a monoid, the only choice is to set $e_{A+B} \triangleq e_A + \mathbb{0}$. This is the implementation in the function `monoidEitherPreferB` (Section 7.3.4). We just need to replace the `Monoid` typeclass constraint for B by `Semigroup`:

```
def monoidEitherSemigroup[A: Monoid, B: Semigroup] = Monoid[Either[A, B]]( {
  case (Left(a1), Left(a2))     => Left(a1 |+| a2)
  case (Left(a), Right(b))      => Right(b) // "Take B".
  case (Right(b), Left(a))      => Right(b)
  case (Right(b1), Right(b2))   => Right(b1 |+| b2)
}, Left(implicitly[Monoid[A]].empty) ) // The type B does not need an empty element.
```

The monoid laws hold here because the proofs of the laws do not depend on the existence of e_B.

Example 7.4.1.4 (the routing monoid) A (much simplified) web server is implemented as a number of "routes". Each route may respond to one or more URL paths by evaluating a custom function. The task is to implement a `combine` operation for routes. The combined route should respond to all paths that at least one of the previous routes responds to:

```
type Path = String              //    Here, the types ‘Path‘ and ‘Response‘ are defined only as an
type Response = (Int, String)   // illustration. The code will use these types as type parameters.
```

```
type Route = Path => Option[Response]

val r1: Route = { case "/get_users" => (200, "user1, user2, user3") }
val r2: Route = { case "/get_names" => (200, "name1, name2, name3") }
      // The task is to implement an extension method |+| such that this works correctly:
val route: Route = r1 |+| r2     // Should respond to both `/get_users` and `/get_names`.
```

Use the `cats` library for implementing a `Monoid` instance for routes; verify that the monoid laws hold.

Solution We will first figure out how to implement the required functionality, and then adapt the code to the `cats` library's definition of the `Monoid` typeclass.

A "route" is a function of type `Path => Option[Response]` that returns a non-empty option if the route responds to a given path value. A combination of two routes `r1` and `r2` needs to be a new route, i.e. a function `Path => Option[Response]`. The new function will first check whether `r1` responds to a given path. If so, it will evaluate the result of applying `r1`. Otherwise, it will try applying `r2` to the given path. We implement the required business logic in a function called `combineRoutes`:

```
def combineRoutes(r1: Route, r2: Route): Route = { path =>
  r1(path) match {
    case Some(response) => Some(response)
    case None => r2(path)
  }
}
```

A monoid also needs to have an empty element. An "empty route" can be combined with any other route and will not change the behavior of that route. If the empty route responds to any path, it would prevent another route from also responding to the same path. So, the only solution is to define the "empty route" as a function that *never* responds to any path:

```
val emptyRoute: Route = { _ => None }
```

The `cats` library defines the `Monoid` typeclass via a `trait` with methods `empty` and `combine`. We can define that typeclass using our existing code:

```
import $ivy.`org.typelevel::cats-core:1.5.0`, cats.Monoid      // Using `ammonite` for convenience.
implicit val catsMonoidRoute: Monoid[Route] = new Monoid[Route] {
  def empty: Route = emptyRoute
  def combine(x: Route, y: Route): Route = combineRoutes(x, y)
}
```

We can now check that the routes can be combined as we intended:

```
import cats.syntax.monoid._
val route: Route = r1 |+| r2

ammonite@ route("/get_users")
res0: Response = (200, "user1, user2, user3")

ammonite@ route("/get_names")
res1: Response = (200, "name1, name2, name3")
```

To verify that the monoid laws hold, we could look for direct proofs using the code of `emptyRoute` and `combineRoutes`. However, it is easier to figure out how to reduce the definition of the `Route` monoid to a number of constructions listed in Section 7.3.4.

For more convenient reasoning, we replace the types `Path` and `Response` by type parameters P and R. We note that these types are used fully parametrically by the code of our functions. So, we will write the type `Route` as $\text{Route} \triangleq P \to \mathbb{1} + R$. Since `Route` is a function type at the outer level, we start with the function-type construction, which requires $\mathbb{1} + R$ to be a monoid. This suggests using the co-product construction; however, R is not necessarily a monoid. There are several possible monoid instances for $\mathbb{1} + R$, so let us look at how the code of `combineRoutes` handles values of that type. The value $1 + \mathbb{0}^{:R}$ corresponds to a route that is not responding to a given path; in that case, `combineRoutes` will switch to the other route. So, the empty value of the monoid $\mathbb{1} + R$ must be $1 + \mathbb{0}^{:R}$. This is indeed

the value returned by `emptyRoute` when applied to any path.

A non-empty value $\mathbb{0} + r$ corresponds to a route that responds to a given path; given two such values, `combineRoutes` will take the first one. This corresponds to the binary operation $\oplus_R$ defined by $r_1 \oplus_R r_2 = r_1$. This operation makes R into a trivial semigroup (see Section 7.3.3). As Example 7.4.1.3 showed, $\mathbb{1} + R$ is a monoid if R is a semigroup (since the unit type $\mathbb{1}$ is a monoid).

So, we have reduced the monoid instance defined by `emptyRoute` and `combineRoutes` to a trivial semigroup, a co-product construction, and a function-type construction. Since all those constructions are guaranteed to produce lawful monoids, we do not need to prove the monoid laws by hand.

Example 7.4.1.5 Using the `cats` library, define a `Functor` instance for the type `Seq[Try[T]]`.

Solution The `cats` library defines the `Functor` typeclass as a `trait` with a `map` method,

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

The functor `Seq[Try[T]]` is a composition of functors `Seq` and `Try`, so we can define the `map` method for it using the functor composition (see Statement 6.2.3.6). We wrap `Seq[Try[T]]` into a case class:

```
import cats.Functor
final case class F[T](s: Seq[Try[T]]) // ‘type F[T] = Seq[Try[T]]‘ does not work with ‘.map‘ method.
implicit val functorF: Functor[F] = new Functor[F] {
  def map[A, B](fa: F[A])(f: A => B): F[B] = F(fa.s.map(_.map(f)))
}
```

One more `cats`-specific import is necessary to enable the `.map` extension method for functors:

```
import cats.syntax.functor._      // Enable the ‘.map‘ method.
val s = F(Seq(Try(1), Try(2), Try(3)))

ammonite@ s.map(_ * 10)
res0: F[Int] = F(List(Success(10), Success(20), Success(30)))
```

For the `.map` method to work, the type constructor `F` must be defined as a `class` or a `trait`. Defining `F` as a type alias would make the Scala compiler confused: if the type `F[T]` were *the same* as `Seq[Try[T]]`, the expression `Seq(Try(1)).map(_ * 10)` would mean to apply `Seq`'s built-in `.map` method rather than the extension method `.map` defined for the type constructor `F` via the `Functor` typeclass.

Example 7.4.1.6 Using the `cats` library, implement a `Bifunctor` instance for $Q^{X,Y} \triangleq X + X \times Y$.

Solution The `cats` library defines the `Bifunctor` typeclass as a `trait` with the `bimap` method. Implementing that method as an information-preserving, fully parametric function is straightforward:

```
final case class Q[X, Y](q: Either[X, (X, Y)])
implicit val bifunctorQ = new Bifunctor[Q] {
  def bimap[A, B, C, D](fab: Q[A, B])(f: A => C, g: B => D): Q[C, D] = fab.q match {
    case Left(a)         => Q(Left(f(a)))
    case Right((a, b))   => Q(Right((f(a), g(b))))
  }
}
```

Example 7.4.1.7 Define a `Contrafunctor` typeclass having the method `contramap` (see Section 6.1.7):

```
def contramap[A, B](c: C[A])(f: B => A): C[B]
```

Implement a `Contrafunctor` instance for the type constructor $C^A \triangleq A \to \text{Int}$.

Solution Since the typeclass method has type parameters, the instance value will have type

$$\text{contramap} : \forall(A, B).\, C^A \to (B \to A) \to C^B \quad .$$

So, the typeclass needs to be implemented as a `trait`:

```
trait Contrafunctor[C[_]] { def contramap[A, B](c: C[A])(f: B => A): C[B] }
```

A typeclass instance for the type constructor $C^A \triangleq A \to \text{Int}$ is created by

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

```
implicit val contrafunctorC = new Contrafunctor[C] {
  def contramap[A, B](c: A => Int)(f: B => A): B => Int = f andThen c
}
```

The `cats` library defines an equivalent typeclass named `Contravariant` with the method `contramap`.

Example 7.4.1.8 Define a `Functor` instance for recursive type constructor $Q^A \triangleq (\text{Int} \to A) + \text{Int} + Q^A$.

Solution Begin by defining Q^A as a recursive disjunctive type:

```
sealed trait Q[A]
final case class C1[A](i: Int => A) extends Q[A]
final case class C2[A](x: Int)      extends Q[A]
final case class C3[A](q: Q[A])     extends Q[A]
```

The methods of Section 6.2.3 show how to implement the `map` function for `Q[A]`. Without repeating those steps, we will write code for a `Functor` instance directly:

```
implicit val functorQ: Functor[Q] = new Functor[Q] { // The function ‘map‘ is recursive.
  def map[A, B](qa: Q[A])(f: A => B): Q[B] = qa match {
    case C1(i)   => C1(i andThen f)
    case C2(x)   => C2(x)
    case C3(q)   => C3(map(q)(f)) // Recursive case.
  }
}
```

Example 7.4.1.9 Using a function parameterized by the type constructors F and G (required to be functors), implement a `Functor` instance for $F^A + G^A$.

Solution The co-product construction (Statement 6.2.3.4) shows how to implement the `map` function for the functor $F^A + G^A$. We begin by writing code for a `Functor` instance assuming that the type constructors `F` and `G` are given:

```
type L[A] = Either[F[A], G[A]]
implicit val functorEither = new Functor[L] {
  def map[A, B](e: L[A])(f: A => B): L[B] = e match {
    case Left(fa)    => Left(fa.map(f))
    case Right(ga)   => Right(ga.map(f))
  }
}
```

We will now to rewrite this code by making `F` and `G` into type parameters. To achieve that, we need to use the "kind projector" plugin and replace the type constructor `L` by a nameless type function `Lambda[X => Either[F[X], G[X]]]`. The code becomes

```
implicit def functorEither[F[_], G[_]] = new Functor[Lambda[X => Either[F[X], G[X]]]] {
  type L[A] = Either[F[A], G[A]]         // We may use F and G to define a type alias in this scope.
  def map[A, B](e: L[A])(f: A => B): L[B] = e match {
    case Left(fa)    => Left(fa.map(f))
    case Right(ga)   => Right(ga.map(f))
  }
}
```

Example 7.4.1.10* **(a)** Implement a function with type signature $C^A + C^B \to C^{A \times B}$ parameterized by a type constructor C (required to be a contrafunctor) and by arbitrary types A, B. Show that the inverse type signature $C^{A \times B} \to C^A + C^B$ is not implementable for some contrafunctors C.

(b) Implement a function with type signature $F^{A \times B} \to F^A \times F^B$ parameterized by a type constructor F (required to be a functor) and by arbitrary types A, B. Show that the inverse type signature $F^A \times F^B \to F^{A \times B}$ is not implementable for some functors F.

Solution **(a)** We need to implement a function with type signature

$$\forall (A, B).\, C^A + C^B \to C^{A \times B} \quad .$$

Begin by looking at the types involved. We need to relate values $C^{A \times B}$, C^A, and C^B; can we relate $A \times B$, A, and B? There exist unique fully parametric functions ∇_1 and ∇_2 of types $A \times B \to A$ and

$A \times B \to B$. If we lift these functions to the contrafunctor C, we will get $\nabla_1^{\downarrow C} : C^A \to C^{A \times B}$ and $\nabla_2^{\downarrow C} : C^B \to C^{A \times B}$. The required type signature is then implemented via a `match` expression like this,

```
def f[C[_]: Contrafunctor, A, B]
                : Either[C[A], C[B]] => C[(A, B)] = {
  case Left(ca)    => ca.contramap { case (a, b) => a }
  case Right(cb)   => cb.contramap { case (a, b) => b }
}
```

The code notation for this function is

$$f^{:C^A + C^B \to C^{A \times B}} \triangleq \begin{array}{|c||c|} \hline & C^{A \times B} \\ \hline\hline C^A & \nabla_1^{\downarrow C} \\ \hline C^B & \nabla_2^{\downarrow C} \\ \hline \end{array} \quad .$$

To show that it is not possible to implement a function g with the inverse type signature,

$$g : \forall (A, B).\, C^{A \times B} \to C^A + C^B \quad ,$$

we choose the contrafunctor $C^A \triangleq A \to R$, where R is a fixed type. The type signature becomes

$$g : \forall (A, B).\, (A \times B \to R) \to (A \to R) + (B \to R) \quad .$$

To implement this function, we need to decide whether to return values of type $A \to R$ or $B \to R$. Can we compute a value of type $A \to R$ given a value of type $A \times B \to R$?

$$g^{:(A \times B \to R) \to A \to R} \triangleq q^{:A \times B \to R} \to a^{:A} \to ???^{:R}$$

We cannot compute a value of type R because that requires us to apply the function q to a pair $A \times B$, while we only have a value of type A. So, the typed hole $???^{:R}$ cannot be filled.

Similarly, we are not able to compute a value of type $B \to R$ from a value of type $A \times B \to R$. Whatever choice we make, $A \to R$ or $B \to R$, we cannot implement the required type signature.

(b) We need to implement a function with type signature

$$\forall (A, B).\, F^{A \times B} \to F^A \times F^B \quad .$$

The types $A \times B$, A, and B are related by the functions $\nabla_1 : A \times B \to A$ and $\nabla_2 : A \times B \to B$. Lifting these functions to the functor F, we obtain $\nabla_1^{\uparrow F} : F^{A \times B} \to F^A$ and $\nabla_2^{\uparrow F} : F^{A \times B} \to F^B$. It remains to take the product of the resulting values:

$$f^{:F^{A \times B} \to F^A \times F^B} \triangleq p^{:F^{A \times B}} \to (p \triangleright \nabla_1^{\uparrow F}) \times (p \triangleright \nabla_2^{\uparrow F}) \quad .$$

```
def f[F[_]: Functor, A, B](p: F[(A, B)]): (F[A], F[B]) =
  (p.map { case (a, b) => a }, p.map { case (a, b) => b })  // Or (p.map(_._1), p.map(_._2))
```

A shorter code for f via the "diagonal" function $\Delta \triangleq (q^{:Q} \to q \times q)$ and the function product $\boxtimes$ is

$$f^{:F^{A \times B} \to F^A \times F^B} \triangleq \Delta \fatsemi (\nabla_1^{\uparrow F} \boxtimes \nabla_2^{\uparrow F}) \quad .$$

This notation is sometimes easier to reason about when deriving properties of functions.

If we try implementing a function g with the inverse type signature,

$$g : \forall (A, B).\, F^A \times F^B \to F^{A \times B} \quad , \tag{7.16}$$

we will find that g *can* be implemented for functors such as $F^A \triangleq A \times A$, $F^A \triangleq \mathbb{1} + A$, and $F^A \triangleq P \to A$. It is not obvious how to find a functor F for which the function g has no implementation. By looking through the known functor constructions (Table 6.2) and trying various combinations, we eventually find a suitable functor: $F^A \triangleq (P \to A) + (Q \to A)$. The type signature of g becomes

$$g : \forall (A, B).\, ((P \to A) + (Q \to A)) \times ((P \to B) + (Q \to B)) \to (P \to A \times B) + (Q \to A \times B) \quad .$$

The argument of this function is of type

$$((P \to A) + (Q \to A)) \times ((P \to B) + (Q \to B)) \quad ,$$

which can be transformed equivalently into a disjunction of four cases,

$$(P \to A) \times (P \to B) + (P \to A) \times (Q \to B) + (Q \to A) \times (P \to B) + (Q \to A) \times (Q \to B) \quad .$$

Implementing the function g requires, in particular, to handle the case when we are given values of types $P \to A$ and $Q \to B$, and we are required to produce a value of type $(P \to A \times B) + (Q \to A \times B)$. The resulting type signature

$$(P \to A) \times (Q \to B) \to (P \to A \times B) + (Q \to A \times B)$$

cannot be implemented: If we choose to return a value of type $P \to A \times B$, we would need to produce a pair of type $A \times B$ from a value of type P. However, producing a pair $A \times B$ requires, in this case, to have values of *both* types P and Q, since the given arguments have types $P \to A$ and $Q \to B$. Similarly, we cannot return a value of type $Q \to A \times B$.

We find that the function g cannot be implemented for the functor $F^A \triangleq (P \to A) + (Q \to A)$. Functors F for which the type signature (7.16) *can* be implemented are called "applicative" (see Chapter 10 for precise conditions). The functor F is an example of a functor that is not applicative.

Example 7.4.1.11* (R. O'Connor [6]) Assume that a functor F admits a function p with type signature $p^{A,B} : A + F^B \to F^{A+B}$, additionally satisfying the special laws of identity and associativity,

```
def p[A, B, F[_]: Functor]: Either[A, F[B]] => F[Either[A, B]]
```

$$p^{\mathbb{0},B} = (b^{:B} \to \mathbb{0} + b)^{\uparrow F} \quad , \qquad p^{A+B,C} = \left|\begin{array}{c||cc} & A & F^{B+C} \\ \hline A & \mathrm{id} & \mathbb{0} \\ B + F^C & \mathbb{0} & p^{B,C} \end{array}\right| ; p^{A,B+C} \quad .$$

Show that the functor F is pointed if such a function p exists. Conversely, show that any pointed functor F admits a function p with these properties.

Solution To show that F is pointed, it is sufficient to find a value wu_F of type $F^{\mathbb{1}}$. We note that the given function p can create values of type F^{A+B} even when given input of type A. So, we set the type parameters $A = \mathbb{1}$ and $B = \mathbb{0}$ and apply p to the value $1 + \mathbb{0}^{:F^{\mathbb{0}}}$,

$$\mathrm{wu}_F \triangleq (1 + \mathbb{0}^{:F^{\mathbb{0}}}) \triangleright p^{\mathbb{1},\mathbb{0}} \triangleright (1 + \mathbb{0} \to 1)^{\uparrow F} \quad .$$

```
val wu[F[_]: Functor]: F[Unit] = p[Unit, Nothing](Left(())).map { case Left(_) => () }
```

Conversely, assuming that F is pointed, we use its pu_F function to define p as

```
def p[F[_]: Functor : Pointed, A, B]
              : Either[A, F[B]] => F[Either[A, B]] = {
  case Left(a)      => pure[F, Either[A, B]](Left(a))
  case Right(fb)    => fb.map { b => Right[A, B](b) }
}
```

$$p^{A,B} \triangleq \left|\begin{array}{c||c} & F^{A+B} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B}) ; \mathrm{pu}_F \\ F^B & (b^{:B} \to \mathbb{0}^{:A} + b)^{\uparrow F} \end{array}\right| \quad .$$

It remains to show that p satisfies the required laws. The identity law holds because

$$\text{expect to equal } (b^{:B} \to \mathbb{0} + b)^{\uparrow F} : \quad p^{\mathbb{0},B} = \left|\begin{array}{c||c} & F^{\mathbb{0}+B} \\ \hline \mathbb{0} & \text{(we can delete this line)} \\ F^B & (b^{:B} \to \mathbb{0} + b)^{\uparrow F} \end{array}\right| = (b^{:B} \to \mathbb{0} + b)^{\uparrow F} \quad .$$

[6] See https://mail.haskell.org/pipermail/haskell-cafe/2015-November/122357.html

To verify the associativity law, we begin with its right-hand side since it is more complicated:

$$\begin{array}{|c||cc|}\hline & A & F^{B+C} \\ \hline A & \mathrm{id} & \mathbb{0} \\ B+F^C & \mathbb{0} & p^{B,C} \\ \hline\end{array} \fatsemi p^{A,B+C} = \begin{array}{|c||cc|}\hline & A & F^{B+C} \\ \hline A & \mathrm{id} & \mathbb{0} \\ B+F^C & \mathbb{0} & p^{B,C} \\ \hline\end{array} \fatsemi \begin{array}{|c||c|}\hline & F^{A+B+C} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B+C}) \fatsemi \mathrm{pu}_F \\ F^{B+C} & (x^{:B+C} \to \mathbb{0}^{:A} + x)^{\uparrow F} \\ \hline\end{array}$$

$$= \begin{array}{|c||c|}\hline & F^{A+B+C} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B+C}) \fatsemi \mathrm{pu}_F \\ B+F^C & p^{B,C} \fatsemi (x^{:B+C} \to \mathbb{0}^{:A} + x)^{\uparrow F} \\ \hline\end{array} = \begin{array}{|c||c|}\hline & F^{A+B+C} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B+C}) \fatsemi \mathrm{pu}_F \\ B & (b^{:B} \to b + \mathbb{0}^{:C}) \fatsemi \mathrm{pu}_F \fatsemi (x^{:B+C} \to \mathbb{0}^{:A} + x)^{\uparrow F} \\ F^C & (c^{:C} \to \mathbb{0}^{:A+B} + c)^{\uparrow F} \\ \hline\end{array} .$$

In the last line, we have expanded the type matrix to three rows corresponding to the disjunctive type $A + B + F^C$. We need to show that the last matrix equals $p^{A+B,C}$; so let us rewrite $p^{A+B,C}$ as a similarly expanded type matrix, using the type isomorphisms such as $\mathbb{0}^{:A} + \mathbb{0}^{:B} \cong \mathbb{0}^{:A+B}$:

$$p^{A+B,C} = \begin{array}{|c||c|}\hline & F^{A+B+C} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B} + \mathbb{0}^{:C}) \fatsemi \mathrm{pu}_F \\ B & (b^{:B} \to \mathbb{0}^{:A} + b + \mathbb{0}^{:C}) \fatsemi \mathrm{pu}_F \\ F^C & (c^{:C} \to \mathbb{0}^{:A} + \mathbb{0}^{:B} + c)^{\uparrow F} \\ \hline\end{array} = \begin{array}{|c||c|}\hline & F^{A+B+C} \\ \hline A & (a^{:A} \to a + \mathbb{0}^{:B+C}) \fatsemi \mathrm{pu}_F \\ B & (b^{:B} \to \mathbb{0}^{:A} + b + \mathbb{0}^{:C}) \fatsemi \mathrm{pu}_F \\ F^C & (c^{:C} \to \mathbb{0}^{:A+B} + c)^{\uparrow F} \\ \hline\end{array} .$$

The only remaining difference is in the second lines of the matrices. We write those lines separately:

$$\begin{aligned} \text{expect to equal } (b^{:B} \to \mathbb{0}^{:A} + b + \mathbb{0}^{:C}) \fatsemi \mathrm{pu}_F : \quad & (b^{:B} \to b + \mathbb{0}^{:C}) \fatsemi \underline{\mathrm{pu}_F \fatsemi (x^{:B+C} \to \mathbb{0}^{:A} + x)^{\uparrow F}} \\ \text{use the naturality law of } \mathrm{pu}_F : \quad & = \underline{(b^{:B} \to b + \mathbb{0}^{:C}) \fatsemi (x^{:B+C} \to \mathbb{0}^{:A} + x)} \fatsemi \mathrm{pu}_F \\ \text{compute function composition} : \quad & = (b^{:B} \to \mathbb{0}^{:A} + b + \mathbb{0}^{:C}) \fatsemi \mathrm{pu}_F \quad . \end{aligned}$$

This completes the proof of the required laws.

7.4.2 Exercises

Exercise 7.4.2.1 Define a PTVF `def isLong[T]: Boolean` that returns `true` for `T = Long` or `Double` and returns `false` for `T = Int`, `Short`, or `Float`. The function should remain undefined for other types `T`.

Exercise 7.4.2.2 Implement a `Monoid` instance for the type String $\times$ ($\mathbb{1}$ + Int).

Exercise 7.4.2.3 **(a)** If A is a monoid and R any type, implement a `Monoid` instance for $R \to A$:

```
def monoidFunc[A: Monoid, R]: Monoid[R => A] = ???
```

Prove that the monoid laws hold for that instance.

(b) With the choice `R = Boolean`, use the type equivalence $(R \to A) = (\mathbb{2} \to A) \cong A \times A$ and verify that the monoid instance `monoidFunc[A, Boolean]` is the same as the monoid instance for $A \times A$ computed by `monoidPair[A, A]` in Section 7.3.4.

Exercise 7.4.2.4 Show that if `S` is a semigroup then `Option[S]` and `Option[(S, S)]` are monoids.

Exercise 7.4.2.5 Using the `cats` library, implement a `Functor` instance for `type F[T] = Try[Seq[T]]`.

Exercise 7.4.2.6 Using the `cats` library, implement a `Bifunctor` instance for $B^{X,Y} \triangleq (\text{Int} \to X) + Y \times Y$.

Exercise 7.4.2.7 Define a `Profunctor` typeclass having the method `xmap`:

```
def xmap[A, B](f: A => B, g: B => A): F[A] => F[B]
```

Implement a `Profunctor` instance for $P^A \triangleq A \to (\text{Int} \times A)$.

Exercise 7.4.2.8 Implement a `Functor` instance for the recursive type $Q^A \triangleq \text{String} + A \times A \times Q^A$.

Exercise 7.4.2.9 Show explicitly that a value $\text{wu}_C : C^{\mathbb{1}}$ is computationally equivalent to a value $\text{pu}_C : \forall A.\, C^A$ that satisfies the naturality law (7.14).

Exercise 7.4.2.10 Using a function parameterized by the type constructors F and G (required to be functors), implement a `Functor` instance for $F^A \times G^A$.

Exercise 7.4.2.11 Implement a `Functor` instance for $F^A \to G^A$ as a function parameterized by type constructors F and G, where F^A is required to be a contrafunctor and G^A is required to be a functor. For the contrafunctor F, use either the `Contrafunctor` typeclass from Example 7.4.1.7 or the `cats` library's typeclass `Contravariant`.

Exercise 7.4.2.12* **(a)** Implement a function with type signature $F^A + F^B \to F^{A+B}$ parameterized by a type constructor F (required to be a functor) and by arbitrary types A, B. Show that the inverse type signature $F^{A+B} \to F^A + F^B$ is not implementable for some functors F.

(b) Implement a function with type signature $C^{A+B} \to C^A \times C^B$ parameterized by a type constructor C (required to be a contrafunctor) and by arbitrary types A, B. Show that the inverse type signature $C^A \times C^B \to C^{A+B}$ is not implementable for some contrafunctors C.

Exercise 7.4.2.13* Implement a function with type signature $F^{A \to B} \to A \to F^B$ parameterized by a functor F and arbitrary types A, B. Show that the inverse type signature, $(A \to F^B) \to F^{A \to B}$, cannot be implemented for some functors F. (Functors admitting a function with that type signature are called "rigid"; see Section 13.6.5.)

Exercise 7.4.2.14** Assume that a functor F admits a function q with type signature

```
def q[A, B, F[_]: Functor]: F[(A, B)] => (A, F[B])
```

$q^{A,B} : F^{A \times B} \to A \times F^B$, additionally satisfying the special laws of identity and associativity,

$$q^{\mathbb{1},B} = f^{:F^{\mathbb{1}\times B}} \to \mathbb{1} \times (f \triangleright (\mathbb{1} \times b^{:B} \to b)^{\uparrow F}) \quad , \qquad q^{A,B\times C} \fatsemi (\text{id}^A \boxtimes q^{B,C}) = q^{A\times B,C} \quad .$$

This Scala code illustrates the required laws in full detail:

```
      // For any value f: F[(Unit, B)], we first compute
val (a1, b1) = q[Unit, B, F](f)._1
      // Then we must have:    a1 == ()   and   b1 == f.map { case (_, b) => b }

      // For any value g: F[(A, B, C)], we first compute
val ((a1: A, b1: B), f1: F[C]) = q[(A, B), C, F](g.map { case (a, b, c) => ((a, b), c) })
val (a2: A, f2: F[(B, C)]) = q[A, (B, C), F](g.map { case (a, b, c) => (a, (b, c)) }
      // Then we must have:    a1 == a2   and   (b1, f1) == g[B, C, F](f2)
```

The following naturality law should also hold for q,

$$(f^{:A\to C} \boxtimes g^{:B\to D})^{\uparrow F} \fatsemi q^{C,D} = q^{A,B} \fatsemi f \boxtimes (g^{\uparrow F}) \quad .$$

```
      // For any value k: F[(A, B)] and any functions f: A => C, g: B => D, first compute
val (a: A, fb: F[B]) = q[A, B, F](k)  // Then we must have:
(f(a), fb.map(g)) == q[C, D, F](k.map { case (a, b) => (f(a), g(b)) })
```

Show that the functor F is co-pointed if a function q with these properties exists. Conversely, show that any lawful co-pointed functor F admits a function q with these properties.

7.5 Discussion

7.5.1 Recursive types and the existence of their values

A recursive type T is defined by a type equation such as $T \triangleq \mathbb{1} + \text{Int} + T \times T$. Can we decide whether such a type equation has a "solution", i.e. a well-defined type T with values that we can create and manipulate in a program?

In all examples seen so far, the recursive type equations had the form $T \triangleq S^T$ where the type constructor S is a *functor*. Type equations with non-functor $S^\bullet$ (e.g. the equation $T \triangleq T \to \text{Int}$) do not seem to be useful in practice, and we will not consider them in this book.

In a rigorous approach, showing that T is a "solution" (called a "**fixpoint**") of the type equation $T \triangleq S^T$ means proving that the types T and S^T are equivalent (isomorphic). We must implement this type isomorphism as two functions, named e.g. `fix` and `unfix`, satisfying the conditions

$$\text{fix} : S^T \to T \quad , \qquad \text{unfix} : T \to S^T \quad , \qquad \text{fix} \fatsemi \text{unfix} = \text{id} \quad , \qquad \text{unfix} \fatsemi \text{fix} = \text{id} \quad .$$

Given a type constructor S, we can define the recursive type T with this Scala code,

```
final case class T(s: S[T])
def fix:    S[T] => T   =   { s => T(s) }
def unfix:  T => S[T]   =   { t => t.s }
```

We can generalize this code to a "fixpoint" constructor Fix^S parameterized by S:

```
final case class Fix[S[_]](s: S[Fix[S]])
def fix[S[_]]:    S[Fix[S]] => Fix[S]   =   { s => Fix(s) }
def unfix[S[_]]:  Fix[S] => S[Fix[S]]   =   { t => t.s }
```

In both implementations, the functions `fix` and `unfix` are inverses of each other because they merely wrap and unwrap values within a case class. So, we are always able to write code that *defines* the recursive type T. The remaining question is whether we will be able to create values of that type.

Applying structural analysis to the functor S, we begin with the constant functor $S^A \triangleq Z$, where Z is a fixed type. The type equation $T \triangleq S^T$ is just $T \triangleq Z$; so T is not a recursive type.

The next case is the identity functor $S^A \triangleq A$. The type equation $T \triangleq S^T$ has the form $T \triangleq T$; since all types T satisfy that equation trivially, we find that the identity functor does not define any specific type T. If we translate this type equation into Scala code, we will run into a problem with recursion:

```
type Id[A] = A
final case class T(s: Id[T])
// Equivalent to ‘final case class T(s: T)‘.
val x = T(x)                // Infinite loop!
```

A value of `case class T` can be created only if we supply a value of type `T`. Writing `def x: T = T(x)` instead of `val` does not help: the evaluation of the code for `x` will not terminate. It is clear that this recursive definition is invalid: we are unable to create values of the resulting type `T`, i.e. the type `T` is void.

Next, consider a product functor such as $S^A \triangleq A \times A \times \text{Int}$. Can we create values of type $T \triangleq S^T$?

```
final case class T(s: (T, T, Int))
val x = T(s = (x, x, 123))  // Infinite loop!
```

We again have an infinite loop when creating values of type `T`. As in the example with the identity functor, the case class `T` requires us to compute some values of type `T` before we can create a value of type `T`. This requirement is impossible to satisfy.

A disjunctive type, e.g. $S^A \triangleq \text{Int} + A \times A$, allows us to create values of type Fix^S with no difficulty:

```
final case class T(s: Either[Int, (T, T)])
val x: T = T(Left(123))     // OK
val y: T = T(Right(x, x))   // OK
```

We are able to create `x` of type $\text{Int} + \mathbb{0}^{:T \times T}$ without need for any previous values of T. We can then use `x` to create a value `y` of type T. This resembles defining a value by induction: the base case is the type $\text{Int} + \mathbb{0}^{:T \times T}$, which is a disjunctive part of S^A that does *not* contain any values of type A. The inductive step is the type $\mathbb{0}^{:\text{Int}} + T \times T$, which creates a new value of type T from two previous values. Type recursion terminates when the base case exists.

The examples we saw previously, $S^A \triangleq A$ and $S^A \triangleq \text{Int} \times A \times A$, do not have a base case where a value of type S^A could be computed without need for any previous values of type A.

Given a functor S^A, how can we determine whether the type recursion $T \triangleq S^T$ terminates? If S^A is a *polynomial* functor, we can view S^A as a polynomial function of A and reduce it to the form

$$S^A \cong C_0 + C_1 \times A + C_2 \times A \times A + \ldots \tag{7.17}$$

It is clear from this formula that the type C_0 is found by setting $A = \mathbb{0}$ in S^A, i.e. $C_0 \cong S^{\mathbb{0}}$. If we can compute a value c_0 of type C_0, a base-case value t_0 of type T is found as

$$t_0^{:T} \triangleq c_0^{:C_0} + \mathbb{0}^{:C_1 \times A} + \mathbb{0}^{:C_2 \times A \times A} + \ldots$$

Further values of type T could be computed as $c_1 \times t_0$ or $c_2 \times t_0 \times t_0$, etc.

We conclude that the recursive type $T \triangleq S^T$ defined via a *polynomial* functor S will be non-void (i.e. type recursion will terminate) when we can create values of type S^0.

In the examples $S^A \triangleq A$ and $S^A \triangleq \text{Int} \times A \times A$, we find $S^0 \cong \mathbb{0}$, so type recursion is invalid.

It remains to consider exponential functors S that cannot be reduced to the polynomial form (7.17). As an example, take $S^A \triangleq \text{String} \times (\text{Int} \to A)$. For this functor S, the condition $S^0 \not\cong \mathbb{0}$ does not hold:

$$\begin{aligned} S^0 &\cong \text{String} \times (\underline{\text{Int} \to \mathbb{0}}) \\ \text{use the type equivalence } (\text{Int} \to \mathbb{0}) \cong \mathbb{0}: \quad &\cong \text{String} \times \mathbb{0} \cong \mathbb{0} \quad . \end{aligned}$$

Nevertheless, we can write Scala code implementing values of the type T defined by $T \triangleq S^T$:

```
final case class T(message: String, next: Int => T)
val t: T = T("start", n => T(s"have $n", _ => t))   // A recursive ‘val‘.
```

The value `t` refers to itself recursively within a nested function `_ => t`, and there is no "base case" in the type S^T. However, the recursion does not lead to an infinite loop; we can use `t` safely:

```
scala> t.next(10)
res0: T = T(got 10,<function1>)

scala> t.next(10).next(10)
res1: T = T(start,<function1>)
```

A value of type T contains a function `next` that, when applied, returns a new value of the same type T. The new value of type T does not need to be computed in advance; its evaluation is *delayed* until some code decides to call `next`. For this reason, an infinite loop is avoided even though the structure functor S has no "base case". Values of type T can be viewed as an infinite stream of `String` values computed on demand; an `Int` value is required in order to produce the next element of the stream. A symbolic (and non-rigorous) representation of that type is

$$T = \text{String} \times (\text{Int} \to \text{String} \times (\text{Int} \to \text{String} \times (\text{Int} \to \text{String} \times ...))) \quad .$$

As another example, consider $S^A \triangleq \mathbb{1} \to \text{Int} \times A + \text{String}$. Using the type equivalence $P \cong (\mathbb{1} \to P)$, we *could* transform S^A into an equivalent functor $\tilde{S}^A$ in the polynomial form (7.17),

$$\tilde{S}^A \triangleq \text{String} + \text{Int} \times A \quad .$$

Although the types S^A and $\tilde{S}^A$ are equivalent, the recursive types Fix^S and $\text{Fix}^{\tilde{S}}$ are different. While

$$\begin{aligned} \text{Fix}^{\tilde{S}} &= \text{String} + \text{Int} \times (\text{String} + \text{Int} \times (\text{String} + \text{Int} \times ...)) \\ &\cong (\mathbb{1} + \text{Int} + \text{Int} \times \text{Int} + \text{Int} \times \text{Int} \times \text{Int} + ...) \times \text{String} \cong \text{String} \times \text{List}^{\text{Int}} \quad , \end{aligned}$$

the type Fix^S admits recursively defined values representing *unbounded* streams of integers, e.g.

```
final case class T(e: () => Either[String, (Int, T)]) // Define the type T ≜ String + Int × T.
val t1: T = T(() => Right((1, t1)))                   // Stream [1, 1, 1, ...].
def t2(n: Int): T = T(() => Right((n, t2(n+1))))      // Stream [n, n+1, n+2, ...].
```

The type $T \triangleq \text{Fix}^S$ also admits finite streams that may be defined as *non-recursive* values of type T:

```
val t0 = T(() => Right((10, T(() => Left("stop")))))  // Finite stream [10, "stop"].
```

We can recognize that Fix^S has non-recursive values by checking that the type S^0 is not void:

$$S^0 = \mathbb{1} \to \text{Int} \times \mathbb{0} + \text{String} \cong \mathbb{1} \to \text{String} \cong \text{String} \not\cong \mathbb{0} \quad .$$

How can we recognize functors S that admit valid recursive values of type Fix^S? The exponential functor construction (Statement 6.2.3.5) shows that $C^A \to P^A$ is a functor when C is a contrafunctor

and P is a functor. If the type expression for S^A contains a sub-expression of the form $C^A \to P^A$, we can use recursion to implement any values of A wrapped by P^A. It remains to create any other values that are required by P^A; effectively, we need P^A to be a pointed functor, so that we can create a value P^A given a value of A. The contrafunctor C^A is an argument of the function of type $C^A \to P^A$, so we are not required to produce values of type C^A— we *consume* those values. It follows that we can implement a value of type $C^T \to P^T$ (with $T = \text{Fix}^S$) as long as $P^\bullet$ is a pointed functor. As we saw in Section 7.3.5, a functor $P^\bullet$ is pointed if we can compute a value of type $P^{\mathbb{1}}$.

In the example $S^A \triangleq \text{String} \times (\text{Int} \to A)$, the functor S^A is pointed since

$$S^{\mathbb{1}} \cong \text{String} \times (\text{Int} \to \mathbb{1}) \cong \text{String} \not\cong \mathbb{0} \quad .$$

This consideration applies to any sub-expression of the form $C^A \to P^A$ within the type constructor S^A. The condition for values of Fix^S to exist is that every functor P^A involved in such sub-expressions should be pointed. To check that, we can set the type parameter $A = \mathbb{1}$ in all return types of *functions* within S^A. If the resulting type can be implemented, we will be able to implement a recursively defined value of type Fix^S.

If the functor S^A has the property $S^{\mathbb{0}} \not\cong \mathbb{0}$ (i.e. we have a base case for the inductive definition of Fix^S), we will also be able to implement non-recursive values of type Fix^S.

We conclude that $S^{\mathbb{0}} \not\cong \mathbb{0}$ is a *sufficient* condition for the type Fix^S to be non-void. (If S is a polynomial functor, this condition is also a necessary condition, as we have seen.) For exponential-polynomial functors S, recursive values of type Fix^S can be implemented if every function-type sub-expression $C^A \to P^A$ in S^A involves a pointed functor P.

7.5.2 Proofs of associativity of `concat` for lists and arrays

The `concat` function is defined for both lists and arrays, and works similarly for these data types:

```
scala> Array.concat(Array(1, 2), Array(3, 4), Array(5, 6))
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6)

scala> List.concat(List(1, 2), List(3, 4), List(5, 6))
res1: List[Int] = List(1, 2, 3, 4, 5, 6)
```

In this section, we will show rigorously that concatenation is an associative operation.

In Scala, `Array[A]` is a sequence whose elements can be accessed by index. The array access function (the `.apply` method defined on the `Array` class) is a partial function of type `Int => A` whose integer argument must be between 0 and $n - 1$, where n is the array length:

```
val x = Array("a", "b", "c")

scala> x(2)  // The syntax ‘x(2)‘ is the same as ‘x.apply(2)‘.
res2: String = c

scala> x(3)  // Applying the partial function ‘x.apply‘ to the value 3 will fail:
java.lang.ArrayIndexOutOfBoundsException: 3
```

We can denote the type of this function by $\text{Int}_{[0,n-1]} \to A$ to indicate the bounds of the index.

The `List` type constructor is a recursive disjunctive type defined by the type equation

$$\text{List}^A \triangleq \mathbb{1} + A \times \text{List}^A = \mathbb{1} + A + A \times A + A \times A \times A + \ldots$$

Although the definitions of types are different, lists and arrays are conceptually similar — they are both sequences of values of type A. However, proving properties is easier for non-recursive types than for recursive types. We begin by proving the associativity property of the array concatenation.

Statement 7.5.2.1 For arrays, $\text{Array}_n^A \triangleq \text{Int}_{[0,n-1]} \to A$, the `concat` function (denoted ++) defined by

$$a_1^{:\text{Array}_{n_1}^A} +\!\!+ a_2^{:\text{Array}_{n_2}^A} \triangleq i^{:\text{Int}_{[0,n_1+n_2-1]}} \to \begin{cases} 0 \le i < n_1 : & a_1(i) \\ n_1 \le i < n_1 + n_2 : & a_2(i - n_1) \end{cases}$$

satisfies the associativity law

$$(a_1{+\!+}a_2){+\!+}a_3 = a_1{+\!+}(a_2{+\!+}a_3) \quad .$$

Proof Both sides of the law evaluate to the same partial function of type $\text{Int} \to A$:

$$(a_1{+\!+}a_2){+\!+}a_3 = a_1{+\!+}(a_2{+\!+}a_3) = i^{:\text{Int}_{[0,n_1+n_2+n_3-1]}} \to \begin{cases} 0 \le i < n_1 : & a_1(i) \\ n_1 \le i < n_1 + n_2 : & a_2(i - n_1) \\ n_1 + n_2 \le i < n_1 + n_2 + n_3 : & a_3(i - n_1 - n_2) \end{cases} .$$

Now we establish a rigorous equivalence between the `List` and `Array` types.

Statement 7.5.2.2 The type $\text{List}^A \triangleq \mathbb{1} + A \times \text{List}^A$ is equivalent to $\text{Array}^A_n \triangleq \text{Int}_{[0,n-1]} \to A$, where $n \ge 0$ and $\text{Int}_{[0,n-1]}$ is the (possibly empty) subset of integers i within the range $0 \le i \le n - 1$.

Proof We need to implement two isomorphism maps f_1, f_2 and show that

$$f_1 : \text{Array}^A_n \to \text{List}^A \quad , \qquad f_2 : \text{List}^A \to \text{Array}^A_n \quad , \qquad f_1 \fatsemi f_2 = \text{id} \quad , \qquad f_2 \fatsemi f_1 = \text{id} \quad .$$

To implement f_1, we proceed by induction in n. The base case is $n = 0$, and we map Array_0 into an empty list. The inductive step assumes that f_1 is already defined on arrays of length n, and we now need to define f_1 for arrays of length $n + 1$. An array of length $n + 1$ is a partial function $g^{:\text{Int}_{[0,n]} \to A}$ defined on the integer interval $[0, n]$. We now split that array into its first element, $g(0)$, and the rest of the array, which needs to be represented by another partial function, say $g' \triangleq i \to g(i + 1)$, defined on the integer interval $[0, n - 1]$. The function g represents an array of length n. By the inductive assumption, f_1 is already defined for arrays of length n. So, we can compute $f_1(g') : \text{List}^A$ and thus create a value of type $\mathbb{0} + A \times \text{List}^A$, which is equivalent to a value of type List^A.

```
def f1[A](arr: Array[A]): List[A] =
  if (arr.length == 0) List()
  else arr(0) :: f1(arr.tail)
```

$$f_1(\text{Array}^A_0) \triangleq \mathbb{1} + \mathbb{0}^{:A \times \text{List}^A} \quad ,$$
$$f_1(g^{:\text{Int}_{[0,n]} \to A}) \triangleq \mathbb{0} + g(0)^{:A} \times \overline{f_1}(i \to g(i + 1)) \quad .$$

To implement f_2, we use induction in the structure of the list. The length n of the array is not known in advance and needs to be computed as we perform pattern-matching on the given `List[A]` value. The base case is an empty list, which yields an empty array (i.e. an array of length $n = 0$). In the inductive step, we assume that we already defined f_2 on lists of length n, and we now need to define f_2 for lists of length $n + 1$. Such a list must have the form $\mathbb{0} + x^{:A} \times s^{:\text{List}^A}$, where s is a list of length n. By the inductive assumption, we are allowed to apply f_2 to s and obtain an array of length n, i.e. a partial function $g^{:\text{Int}_{[0,n-1]} \to A}$. So we define $f_2(\mathbb{0} + x \times s)$ as a new array whose 0^{th} element is x and the i^{th} element is computed by applying the function $g \triangleq f_2(s)$ to $i - 1$:

```
def f2[A: ClassTag]: List[A] => Array[A] =
{  case List()   => Array()
   case x :: s   => Array(x) ++ f2.apply(s)
}  // This code is for illustration only.
```

$$f_2(\mathbb{1} + \mathbb{0}^{:A \times \text{List}^A}) \triangleq \text{Array}^A_0 \quad ,$$
$$f_2(\mathbb{0} + x^{:A} \times s^{:\text{List}^A}) \triangleq i^{:\text{Int}_{[0,n]}} \to \begin{cases} i = 0 : & x \\ i \ge 1 : & \overline{f_2}(s)(i - 1) \end{cases} .$$

To show that f_1 and f_2 are inverse functions for each other, we again need to use induction. The base case is the empty list and the empty array, which are indeed mapped isomorphically to each other. The inductive step for $f_1 \fatsemi f_2$ is an array of length $n + 1$ with the inductive assumption that $\overline{f_1 \fatsemi f_2} = \text{id}$ for arrays of length n. Writing out the code of $f_1 \fatsemi f_2$, we find that $g \triangleright f_1 \fatsemi f_2 = g$:

$$g^{:\text{Int}_{[0,n]} \to A} \triangleright f_1 \fatsemi f_2 = g \triangleright f_1 \triangleright f_2 = (\mathbb{0} + g(0) \times \overline{f_1}(i \to g(i + 1))) \triangleright f_2$$
$$= i \to \begin{cases} i = 0 : & g(0) \\ i \ge 1 : & \overline{f_2}(\overline{f_1}(i \to g(i + 1)))(i - 1) \end{cases} = i \to \begin{cases} i = 0 : & g(0) \\ i \ge 1 : & \underline{\text{id}}(i \to g(i + 1)))(i - 1) \end{cases}$$
$$= i \to \begin{cases} i = 0 : & g(0) \\ i \ge 1 : & g(\underline{(i - 1) + 1}) \end{cases} = (i \to g(i)) = g.$$

Similarly, we find that $(\mathbb{0} + x^{:A} \times s^{:\text{List}^A}) \triangleright f_2 \fatsemi f_1 = (\mathbb{0} + x^{:A} \times s^{:\text{List}^A})$ via this calculation:

$$(\mathbb{0} + x^{:A} \times s^{:\text{List}^A}) \triangleright f_2 \fatsemi f_1 = f_1 \left(i^{:\text{Int}_{[0,n]}} \to \begin{cases} i = 0 : & x \\ i \geq 1 : & \overline{f_2}(s)(i-1) \end{cases} \right)$$
$$= \mathbb{0} + x \times \overline{f_1}(i \to \overline{f_2}(s)(i+1-1)) = \mathbb{0} + x \times \overline{f_1}(\overline{f_2}(s)) = \mathbb{0} + x \times s \quad .$$

This concludes the proof of the isomorphism between `Array` and `List`.

Since arrays and lists are isomorphic as types, the concatenation for lists is associative as long as we show that the concatenation operation for lists is isomorphic to that we defined for arrays.

Statement 7.5.2.3 The `concat` function for lists is defined recursively as

```
def concat[A](p: List[A], q: List[A])
    : List[A] = p match {
  case List() => q
  case a :: t => a :: concat(t, q)
}
```

$$p^{:\text{List}^A} +\!\!+ q^{:\text{List}^A} \triangleq p \triangleright \begin{array}{|c||c|} \hline & \text{List}^A \\ \hline \mathbb{1} & q \\ \hline A \times \text{List}^A & a \times t \to \mathbb{0} + a \times (t +\!\!+ q) \\ \hline \end{array} \quad .$$

and is equivalent to the `concat` function on arrays defined in Statement 7.5.2.1:

$$\forall a^{:\text{Array}^A_{n_1}}, b^{:\text{Array}^A_{n_2}}. \quad f_1(a) +\!\!+ f_1(b) = f_1(a +\!\!+ b) \quad ,$$
$$\forall p^{:\text{List}^A}, q^{:\text{List}^A}. \quad f_2(p) +\!\!+ f_2(q) = f_2(p +\!\!+ q) \quad ,$$

where f_1, f_2 are the isomorphism maps defined in Statement 7.5.2.2.

Proof If we show the property for f_2, we can apply f_1 to both sides and obtain the other property.

The base case for lists is $[] +\!\!+ q = q$. This is clearly isomorphic to concatenating an empty array with another array since $f_2([])$ is an empty array.

The inductive step is $p +\!\!+ q$ where p is a non-empty list, $p = \mathbb{0} + a^{:A} \times t^{:\text{List}^A}$. We need to show that

$$f_2(\mathbb{0} + a \times t) +\!\!+ f_2(q) = f_2((\mathbb{0} + a \times t) +\!\!+ q) = f_2(\mathbb{0} + a \times (t +\!\!+ q)) \quad .$$

By definition of f_2, we have

$$f_2(\mathbb{0} + a \times t) = i \to \begin{cases} i = 0 : & a \\ i \geq 1 : & \overline{f_2}(t)(i-1) \end{cases} \quad ,$$
$$f_2(\mathbb{0} + a \times (t +\!\!+ q)) = i \to \begin{cases} i = 0 : & a \\ i \geq 1 : & \overline{f_2}(t +\!\!+ q)(i-1) \end{cases} \quad .$$

The inductive assumption guarantees that $\overline{f_2}(t +\!\!+ q) = f_2(t) +\!\!+ f_2(q)$. Using the definition of array concatenation and assuming that the length of t is n_1 and the length q is n_2, we get

$$f_2(\mathbb{0} + a \times (t +\!\!+ q)) = i \to \begin{cases} i = 0 : & a \\ 1 \leq i < n_1 + 1 : & f_2(t)(i-1) \\ n_1 + 1 \leq i < n_1 + 1 + n_2 : & f_2(q)(i - n_1 - 1) \end{cases} \quad .$$

This is the same array as the concatenation $f_2(\mathbb{0} + a \times t) +\!\!+ f_2(q)$. This concludes the proof.

7.5.3 "Kinds" and higher-order type functions

Type constructors are types parameterized by other types. We have also seen types parameterized by type constructors, e.g.

```
trait Functor[F[_]] { ... }
```

It is helpful to introduce terminology that distinguishes the different ways in which types can be parameterized.

A type constructor such as `List` can be understood as a type-to-type function (TTF): given a type, e.g. `Int`, it produces another type, `List[Int]`. It is a type error to apply `List` to a type parameter that is not a simple type:

```
scala> val x: List[List] = ???
<console>:11: error: type List takes type parameters
       val x: List[List] = ???
                   ^
```

To describe the restriction on possible type parameters of `List`, we say that `List` has a **kind signature** $* \to *$. The symbol $*$ means an ordinary type (not a type function), i.e. a type that can have values. In the type notation, we can write "kind signatures" as

$$\text{Int} : * \quad , \qquad \text{List} : * \to * \quad , \qquad \text{List}^{\text{Int}} : * \quad .$$

So, the concept of "**kind**" can be understood as the "type signature of a type". Types of kind $*$ (such as `Int` or `String`) can have values, but types of kind $* \to *$ cannot:

```
scala> val x: Seq = ???
<console>:11: error: type Seq takes type parameters
       val x: Seq = ???
              ^
```

Although Scala will check that all kinds match, there is no syntax in Scala to declare kinds.

The `Functor` typeclass is another example of a non-trivial "kind": a type whose type parameter is itself a type constructor,

$$\text{Functor} : (* \to *) \to * \quad .$$

The `Functor` type can be seen as a higher-order type-to-type function (TTF) since it takes a type parameter that is itself a TTF. Such types are often called "higher-kinded types".

For higher-order TTFs, Scala requires syntax such as `Functor[F[_]]` or `F[_[_]]`, for example:

```
// Ap1 and Ap2 will simply substitute type arguments into type constructors of various kinds.
type Ap1[F[_], A] = F[A]   // Ap1 : ((* → *) × *) → *
type Ap2[P[_[_], _], Q[_], R] = P[Q, R] // Ap2 : (((* → *) × * → *) × (* → *) × *) → *
type G[A] = Either[(A, A, String), A]   // G : * → *
type X = Ap2[Ap, G, Int]   // OK; X is now Either[(Int, Int, String), Int]
type Y = Ap2[Ap, Ap, Int]  // Type error: the second argument of Ap2 has wrong kind.
type Z = Ap2[G, G, Int]    // Type error: the first argument of Ap2 has wrong kind.
```

The "kind projector" plugin is often needed when writing code with higher-order TTFs:

```
   // ‘Twice‘ will apply the type constructor Q twice to its type argument, and substitute into P.
type Twice[P[_[_], _], Q[_], R] = P[Lambda[X => Q[Q[X]]], R] // Twice : (((* → *) × * → *) × (* → *) × *) → *
type O2[A] = Option[(A, A)]                    // O2 : * → *
type X2 = Twice[Ap1, O2, Int]        // X2 is now Option[(Option[(Int, Int)], Option[(Int, Int)])].
val x2: X2 = Some((Some((1, 2)), Some((3, 4))))   // Types match for ‘x2‘.
```

7.5.4 Inductive typeclasses and their properties

We have seen many examples of typeclasses that have similar structural properties. For instance, a product of semigroups is a semigroup, a product of monoids is a monoid, a product of functors is a functor, a product of pointed functors is a pointed functor, and so on. It turns out that all these typeclasses have a common structure that we can recognize and reason about. The common structure is that all typeclass methods can be expressed as a *single* uncurried function of a specific form $P^A \to A$, as Table 7.8 shows.

Functions of type $P^A \to A$ compute new values of type A from previous values (of type A or other types) wrapped by the functor P. This superficially resembles defining values of type A by induction

Typeclass	Instance type as a function	Inductive form	Structure functor
default value	$\mathbb{1} \to A$	$P^A \to A$	$P^A \triangleq \mathbb{1}$
semigroup	$A \times A \to A$	$P^A \to A$	$P^A \triangleq A \times A$
monoid	$\mathbb{1} + A \times A \to A$	$P^A \to A$	$P^A \triangleq \mathbb{1} + A \times A$
functor	$F^A \times (A \to B) \to F^B$	$S^{\bullet,F} \to F^\bullet$	$S^{B,F^\bullet} \triangleq \forall A.\, F^A \times (A \to B)$
pointed functor	$B + F^A \times (A \to B) \to F^B$	$S^{\bullet,F} \to F^\bullet$	$S^{B,F^\bullet} \triangleq \forall A.\, B + F^A \times (A \to B)$
contrafunctor	$F^A \times (B \to A) \to F^B$	$S^{\bullet,F} \to F^\bullet$	$S^{B,F^\bullet} \triangleq \forall A.\, F^A \times (B \to A)$
pointed contrafunctor	$\mathbb{1} + F^A \times (B \to A) \to F^B$	$S^{\bullet,F} \to F^\bullet$	$S^{B,F^\bullet} \triangleq \forall A.\, \mathbb{1} + F^A \times (B \to A)$

Table 7.8: Structure of typeclass instances values for various inductive typeclasses.

and so motivates the following definition: A typeclass is **inductive** if a type A's typeclass instance is a value of type $P^A \to A$ with some functor P called the **structure functor** of the typeclass.

A value of type $P^A \to A$ represents all the methods of the an inductive typeclass in a single function. We can implement this definition by the Scala code

```
final case class InductiveTypeclass[P[_]: Functor, A](methods: P[A] => A)
```

For example, the `Monoid` typeclass is inductive because its instances have type

$$\text{Monoid}^A = (A \times A \to A) \times A \cong (\mathbb{1} + A \times A \to A) = P^A \to A \quad \text{where } P^A \triangleq \mathbb{1} + A \times A \quad .$$

So, the `Monoid` typeclass can be declared as an inductive typeclass by code like this,

```
final case class MonoidStructure[A](s: Option[(A, A)])
val monoidStructureFunctor: Functor[MonoidStructure] = ... /* implementation */
type InductiveMonoid[A] = InductiveTypeclass[MonoidStructure, A]
```

Implementing the `Monoid` typeclass via a structure functor P and a function $P^A \to A$ is inconvenient for practical programming. The inductive form $P^A \to A$ is mainly useful for reasoning about general properties of typeclasses. To illustrate that kind of reasoning, we will show that the product-type, the function-type, and the recursive-type constructions work for all inductive typeclasses.

Consider an arbitrary inductive typeclass `TC` defined via a structure functor `P`,

```
type P[A] = ...
type TC[A] = P[A] => A
```

If two types A, B have an instance of the typeclass `TC`, an instance for the product $A \times B$ can be derived automatically using the function

$$\text{productTC} : (P^A \to A) \times (P^B \to B) \to P^{A \times B} \to A \times B \quad ,$$
$$\text{productTC} \triangleq (f^{:P^A \to A} \times g^{:P^B \to B}) \to p^{:P^{A \times B}} \to (p \triangleright \nabla_1^{\uparrow P} \triangleright f) \times (p \triangleright \nabla_2^{\uparrow P} \triangleright g) \quad .$$

```
def productTC[A, B](f: P[A] => A, g: P[B] => B): P[(A, B)] => (A, B) =
  p => ( f(p.map(_._1)), g(p.map(_._2)) )
```

This explains why product types have instances for all typeclasses listed in Table 7.8: those typeclasses are inductive. (This argument does not prove that the typeclass *laws* will also hold for the product types.)

It is interesting to note that the co-product construction cannot be derived in general for arbitrary inductive typeclasses: given $P^A \to A$ and $P^B \to B$, it is not guaranteed that we can compute $P^{A+B} \to A + B$. Not all inductive typeclasses support the co-product construction.

The function-type construction promises a typeclass instance for $E \to A$ if the type A has a typeclass instance. This construction works for any inductive typeclass because a value of type $P^{E \to A} \to E \to A$ can be computed from a value of type $P^A \to A$ when P is a functor:

$$q : (P^A \to A) \to P^{E \to A} \to E \to A \quad , \qquad q \triangleq h^{:P^A \to A} \to p^{:P^{E \to A}} \to e^{:E} \to p \triangleright (x^{:E \to A} \to e \triangleright x)^{\uparrow P} \triangleright h \quad .$$

As in the case of the product construction, the laws still need to be checked for the new instances.

Finally, let us show that the recursive type construction works for inductive typeclasses. Consider a recursive type T defined by a type equation $T \triangleq S^T$, where the functor $S^\bullet$ preserves typeclass instances (i.e. if A has an instance then S^A also does, as we saw in all our examples in this chapter). It means that we have a function $\text{tcS} : (P^A \to A) \to P^{S^A} \to S^A$ that creates typeclass instances of type $P^{S^A} \to S^A$ out of instances of type $P^A \to A$. Then we define an instance `tcT` for T as

$$\text{tcT}^{:P^T \to T} \triangleq p^{:P^T} \to \text{tcS}\,(\text{tcT})(p) \quad .$$

This recursive function terminates because it is implemented as an "expanded function". The types match since we can convert between the equivalent types T and S^T whenever necessary, so $p^{:P^T}$ can be converted to a value of type P^{S^T}, while the value $\text{tcS}\,(\text{tcT})(p)$ can be converted from type S^T to type T. In Scala code, the isomorphic conversions between T and S^T are implemented by the constructor `T(...)` and accessor `(_.s)` methods of the case class that wraps the type T.

```
type S[A] = ...              // Define a functor S as required.
final case class T(s: S[T]) // Define the recursive type T as T ≜ S^T.
def tcS: TC[A] => TC[S[A]] = ... // Compute instances for S[A] from instances of A.
def tcT: P[T] => T = p => T(tcS(tcT)(p.map(_.s))) // Define the recursive instance.
```

In this way, we can show that all inductive typeclasses support the recursive-type construction.

The `Functor` and `Pointed` typeclasses are also inductive. Instead of a structure functor, we need to use a higher-order type function denoted by $S^{A,F}$ and parameterized by a type constructor $F^\bullet$ as well as by a type parameter A. (The type S has kind $* \times (* \to *) \to *$.) The methods of these typeclasses are expressed as $S^{\bullet,F} \to F^\bullet$, which is analogous to $P^A \to A$ except for additional type parameters. Similar arguments can be made for these typeclasses, although it is more difficult to reason about type constructors (and the laws will not hold without additional assumptions). As we have seen, the product-type, the function-type, and the recursive-type constructions work for functors, contrafunctors, pointed functors, and pointed contrafunctors.

Important inductive typeclasses are filterables (Chapter 8), monads (Chapter 9), and applicatives (Chapter 10). We will see in Chapter 12 that another general construction also works for all inductive typeclasses, — the "free" type construction.

We have also seen examples of typeclasses that are not inductive. The `Extractor` typeclass has instances of type $A \to Z$, which is not of the form $P^A \to A$ for any functor P. Another such typeclass is `Copointed`, whose method $F^A \to A$ is not of the form $S^{\bullet,F} \to F^\bullet$. However, the methods of these typeclasses can be written in the inverted form $A \to P^A$ with some functor P. We call them **co-inductive typeclasses**. A motivation for this name is that the *co-product* construction (rather than the product construction) works with co-inductive typeclasses: given values of types $A \to P^A$ and $B \to P^B$, we can produce a value of type $A + B \to P^{A+B}$, i.e. a co-product instance, but not necessarily a value of type $A \times B \to P^{A \times B}$, which would be a product-type instance. The function-type construction is not guaranteed to work with co-inductive typeclasses, but the recursive-type construction and the "free" type construction can be implemented after appropriate modifications.

The `Eq` typeclass is "un-inductive" (neither inductive nor co-inductive) because its instance type, $A \times A \to 2$, is neither of the form $P^A \to A$ nor $A \to P^A$. The traversable functor (Chapter 11) is another example of an "un-inductive" typeclass. Un-inductive typeclasses usually support fewer type constructions. For example, only *polynomial* types can have instances of `Eq` or `Traversable` typeclasses.

7.5.5 Typeclasses with more than one type parameter (type relations)

A typeclass constraint in a function, such as `func[A: Monoid]`, restricts a type parameter to a certain type domain. Sometimes it is necessary to restrict *several* type parameters to satisfy some conditions together. Let us look at two simple examples of this.

The first example is converting integer numbers to floating point. The ranges of the available types allow us to convert a `Short` to a `Float` and an `Int` to a `Double`. Can we implement the type signature

```
def convertNumber[M, N](x: M): N
```

where the type parameters `M`, `N` are constrained to be either `M = Short` and `N = Float`, or `M = Int` and `N = Double`? (Of course, we will want to be able to add further supported pairs of types later.)

A condition that constrains several type parameters at once is called a **type relation**.

The second example is converting mutable data structures into the corresponding immutable ones. The Scala library contains data structures such as sequences, sets, and dictionaries, each having a mutable and an immutable version. Can we implement a function with type signature

```
def convertData[Mut[_], Immut[_], A](data: Mut[A]): Immut[A]
```

where the type parameters `Mut` and `Immut` are constrained to represent the mutable / immutable versions of a supported data structure (for example, `Mut = mutable.Set` and `Immut = immutable.Set`, etc.)?

As we have seen, a typeclass constraint for a single type parameter, such as `func[A: Monoid]`, is implemented by requiring an "evidence" value of type `Monoid[A]` as an additional argument of the function `func`. Implementing a type relation for *two* type parameters `A`, `B` is similar: We first define a type constructor, say `Rel[A, B]`, and create some evidence values of that type, with specific chosen type parameters `A`, `B`. Any function that needs a relation constraint on its type parameters `A`, `B` will require an implicit "evidence argument" of type `Rel[A, B]`. This will prevent our code from using that function with types `A`, `B` that are not in the required relation.

The code for `convertNumber` can be written like this:

```
final case class MayConvert[A, B](convert: A => B)  // Evidence value contains a conversion function.
implicit val ev1 = MayConvert[Short, Float](_.toFloat)          // Evidence value for [Short, Float].
implicit val ev2 = MayConvert[Int, Double](_.toDouble)          // Evidence value for [Int, Double].
def convertNumber[M, N](x: M)(implicit ev: MayConvert[M, N]): N = ev.convert(x)
```

With these definitions, it will be a compile-time error to use `convertNumber` with unsupported types:

```
scala> convertNumber(123)
res0: Double = 123.0

scala> convertNumber(123:Short)
res1: Float = 123.0

scala> convertNumber("abc")
<console>:17: error: could not find implicit value for parameter ev: MayConvert[String,N]
       convertNumber("abc")
                    ^
```

As we have just seen, a type relation is defined by creating a set of evidence values. The code defines `MayConvert` as a 1-to-1 relation because the evidence values (or "relation instances") `ev1` and `ev2` do not have any type parameters in common. So, `MayConvert` is equivalent to a type-to-type *function* that maps `Short` to `Float` and `Int` to `Double`. However, type relations are not limited to 1-to-1 relations or to type functions. By creating suitable implicit evidence values, we can implement 1-to-many or many-to-many relations when needed.

A practical example of a many-to-many type relation is the compatibility between physical units. Miles can be converted into kilometers; pounds can be converted into ounces or kilograms; but kilograms cannot be converted into miles. Type relations allow us to implement type-safe operations for quantities with units. Adding kilograms to pounds will automatically convert the quantity to a common unit, while adding kilograms to miles will raise a compile-time error.

Begin by declaring a type for "quantity with units" and some type names for the supported units:

```
trait KG; trait LB; trait OZ; trait KM; trait MI; trait FT        // Some supported units.
final case class Quantity[U](value: Double)   // Constant functor: No values of type U are stored.
def add[U1, U2](x: Quantity[U1], y: Quantity[U2]): Quantity[U2] = ???
```

The `add` function must have a type relation constraint for its type parameters `U1` and `U2`, so that only quantities with compatible units may be added. To implement that, we define a type constructor with two type parameters. A relation instance will contain a multiplier for converting the units:

```
final case class Convertible[U1, U2](multiplier: Double)
implicit val c1 = Convertible[KG, KG](1.0)
implicit val c2 = Convertible[LB, KG](0.453592)
implicit val c3 = Convertible[KM, KM](1.0)
implicit val c4 = Convertible[MI, KM](1.60934)
                    // Add many more definitions here...
```

Now we can implement the `add` function and verify that the type relation works as we intended:

```
def add[U1, U2](x: Quantity[U1], y: Quantity[U2])(implicit ev: Convertible[U1, U2]): Quantity[U2] =
  Quantity(x.value * ev.multiplier + y.value)

scala> add(Quantity[LB](1), Quantity[KG](1))   // 1 pound + 1 kg = 1.453592 kg.
res0: Quantity[KG] = Quantity(1.453592)

scala> add(Quantity[MI](1), Quantity[KG](1))   // Compile-time error: cannot add miles to kilograms.
<console>:25: error: could not find implicit value for parameter ev: Convertible[MI,KG]
       add(Quantity[MI](1), Quantity[KG](1))
          ^
```

To make this code more convenient for practical use, we can add extension methods to shorten the syntax from `Quantity[KG](1)` to `1.kg` and from `add(2.lb, 2.kg)` to a more readable `2.lb + 2.kg`:

```
implicit class QuantitySyntax(x: Double) {
  def kg = Quantity[KG](x)
  def lb = Quantity[LB](x)        // Add more definitions as needed.
}
implicit class QuantityAdd[U1](x: Quantity[U1]) {
  def +[U2](y: Quantity[U2])(implicit ev: Convertible[U1, U2]): Quantity[U2] = add(x, y)
}

scala> 2.lb + 2.kg
res1: Quantity[KG] = Quantity(2.907184)
```

Another necessary improvement is reducing the number of implicit values. The current code uses n^2 implicit values for every group of n compatible units. Adding support for a new unit (say, inches) requires adding implicit values for converting between inches and all previously defined units of length. To avoid this problem, the code must be reorganized to convert all compatible quantities to chosen units (e.g. all length to kilometers and all mass to kilograms). This makes the `Convertible` relation many-to-1 instead of many-to-many. The resulting code is shown in Figure 7.2.

7.5.6 Inheritance and automatic conversions of typeclasses

It often happens that one typeclass implies another; for example, a `Monoid` instance for a type T means that T already has the properties of both `Semigroup` and `HasDefault` typeclasses. One says that the `Monoid` typeclass **inherits** from the `Semigroup` and `HasDefault` typeclasses. We may want to express inheritance relations between typeclasses, so that e.g. a `Monoid` instance should automatically imply the presence of a `Semigroup` instance without extra code.

One way of inheriting a typeclass is to add a constraint to the new typeclass constructor:

```
final case class Semigroup[T](combine: (T, T) => T)
final case class Monoid[T: Semigroup](empty: T)  // Inherit ‘Semigroup‘.
```

```
trait KG; trait LB; trait OZ; trait KM; trait MI; trait FT            // Some supported units.

// This type relation is many-to-1: it relates all mass units to KG and all length units to KM.
final case class Convertible[U1, U2](multiplier: Double) extends AnyVal
implicit val cKG = Convertible[KG, KG](1.0)
implicit val cLB = Convertible[LB, KG](0.453592)
implicit val cOZ = Convertible[OZ, KG](0.0283495)
implicit val cKM = Convertible[KM, KM](1.0)
implicit val cMI = Convertible[MI, KM](1.60934)
implicit val cFT = Convertible[FT, KM](0.0003048)

final case class Quantity[U](value: Double) extends AnyVal { // Use `AnyVal` to reduce run-time cost.
  def +[U2, C](q: Quantity[U2])(implicit ev1: Convertible[U, C], ev2: Convertible[U2, C]) =
      Quantity[U2](value * ev1.multiplier / ev2.multiplier + q.value)

  def ==[U2, C](q: Quantity[U2])(implicit ev1: Convertible[U, C], ev2: Convertible[U2, C]) =
      value * ev1.multiplier == q.value * ev2.multiplier
}

implicit class QuantitySyntax(x: Double) {
  def kg = Quantity[KG](x)
  def lb = Quantity[LB](x)
  def oz = Quantity[OZ](x)
  def km = Quantity[KM](x)
  def mi = Quantity[MI](x)
  def ft = Quantity[FT](x)
  // A general extension method, e.g. `1.in[KM]`, that will also work for any units defined later.
  def in[U](implicit ev: Convertible[U, _]): Quantity[U] = Quantity[U](x)
}

scala> 1.in[KM]         // Use the general method `.in` with a type parameter.
res1: Quantity[KM] = Quantity(1.0)

scala> 10000.ft + 1.km  // Use the `.ft` and `.km` methods.
res2: Quantity[KM] = Quantity(4.048)

scala> 1.kg + 1.lb == 16.oz + 1.kg   // Compare values safely.
res2: Boolean = true

scala> 1.km + 1.lb      // Compile-time error: cannot add kilometers to pounds.
<console>:29: error: could not find implicit value for parameter ev2: Convertible[LB,C]
       1.km + 1.lb
            ^

scala> trait YD; implicit val cYD = Convertible[YD, KM](0.0009144)  // Easily add support for yards.
defined trait YD
cIN: Convertible[YD,KM] = Convertible(9.144E-4)

scala> 1.in[YD] + 1.ft // Use .in[YD] to compute 1 YD + 1 FT = 4 FT.
res3: Quantity[FT] = Quantity(4.0)
```

Figure 7.2: Implementing type-safe computations with units of length and mass.

The case class `Monoid` combines a previous `Semigroup` instance with the new method `empty`. Creating an instance of `Monoid` for a type `T` will then require having an instance of `Semigroup` for `T`:

```
implicit val c1 = Semigroup[Int](_ + _)
implicit val c2 = Monoid[Int](0)  // Works only if a ‘Semigroup[Int]‘ is available.
```

If we now have a `Monoid` instance, how can we recover the inherited `Semigroup` instance? We need to extract the implicit value of type `Semigroup[T]` that was passed to the `Monoid` class constructor. This is achieved by adding an implicit function and by redefining the `Monoid` class as follows,

```
final case class Monoid[T](empty: T)(implicit val semigroup: Semigroup[T])   // Use ‘implicit val‘.
implicit def semigroupFromMonoid[T](implicit ti: Monoid[T]): Semigroup[T] = ti.semigroup
```

Another possibility of implementing typeclass inheritance is to use object-oriented inheritance of traits. This approach is often used when defining typeclasses as traits with methods:

```
trait Semigroup[T] { def combine: (T, T) => T }
trait Monoid[T] extends Semigroup[T] { def empty: T } // ‘def combine‘ is inherited from ‘Semigroup‘.
```

Creating an instance of `Monoid` for a type `T` no longer requires having an instance of `Semigroup` for `T`:

```
implicit val c: Monoid[Int] = new Monoid[Int] {
  def empty: Int =  0
  def combine: (Int, Int) => Int =  _ + _
}
```

With this approach, we cannot avoid repeating the code for `def combine...` even if we *do* already have a `Semigroup` instance. However, conversion from `Monoid[T]` to `Semigroup[T]` is now automatic due to **object-oriented inheritance**: `Monoid[T]` is automatically considered a subtype of `Semigroup[T]` because the `Monoid` class is declared as `extends Semigroup`.

One problem with the object-oriented inheritance is that automatic conversions to parent typeclasses cannot be disabled. When several typeclasses inherit from the same parent typeclass, duplicate implicit instances of the parent typeclass may be present. This will give a compile-time error:

```
trait TC[A]                    // Parent typeclass.
trait TC1[A] extends TC[A]     // Two different typeclasses TC1 and TC2 inherit from TC.
trait TC2[A] extends TC[A]
// The function f requires A to have both TC1 and TC2 instances and then wants to access TC instance.
def f[A: TC1 : TC2]() = {
  implicitly[TC[A]]  // Compilation fails because two implicits of type TC[A] are found.
}
```

When typeclass inheritance is implemented by combining instances without object-oriented inheritance, conversions to parent typeclasses are not automatic – they are implemented by implicit functions that need to be imported into the current scope. The programmer's code can avoid producing duplicate instances by choosing which implicit conversions to import:

```
final case class TC[A]()

final case class TC1[A]()(implicit val tc: TC[A])                      // TC1 inherits TC.
object TC1 { implicit def toTC[A](implicit x: TC1[A]): TC[A] = x.tc }

final case class TC2[A]()(implicit val tc: TC[A])                      // TC2 inherits TC.
object TC2 { implicit def toTC[A](implicit x: TC2[A]): TC[A] = x.tc }

// The function f requires A to have both TC1 and TC2 instances and then wants to access TC instance.
def f[A: TC1 : TC2]() = {
      import TC1._  // Can import TC1._ or TC2._ but not both. If the next line is uncommented,
//    import TC2._  // compilation will fail because two implicits of type TC[A] will be found!
  implicitly[TC[A]] // This compiles successfully. One implicit instance of TC[A] can be found.
}
```

The problem of duplicate inherited instances can be solved with less work for the programmer if the typeclasses are implemented using a special, more complicated encoding.[7]

[7] See slides 5–14 of the talk by John de Goes: https://www.slideshare.net/jdegoes/scalaz-8-a-whole-new-game

18 "Applied functional type theory": A proposal

What exactly is the extent of "theory" that a software engineer must know in order to be a proficient functional programmer? This book proposes an answer to that question by presenting a coherent body of theoretical knowledge that, in the author's view, is the "theory that is practically useful for functional programming". This body of knowledge may be viewed as a new branch of computer science — **applied functional type theory** (AFTT). This is the area of theoretical computer science serving the practical needs of functional programmers working as software engineers.

It is for these practitioners, rather than for academic researchers, that this book sets out to examine the functional programming inventions over the last 30 years, – such as the "functional pearls" papers[1] – and to determine the scope of theoretical material that has demonstrated its pragmatic usefulness and thus belongs to AFTT, as opposed to material that is purely academic and may be tentatively omitted. This book is a first step towards formulating AFTT.

In this book, code is written in Scala because the author is fluent in that language. However, most of this material will work equally well in Haskell, OCaml, and other FP languages. This is so because the science of functional programming, call ed AFTT, is not a set of tricks specific to Scala or Haskell. An advanced user of any other functional programming language will have to face the same questions and struggle with the same practical issues.

18.1 AFTT is not covered by computer science curricula

Traditional courses of theoretical computer science (algorithms and data structures, complexity theory, distributed systems, databases, network systems, compilers, operating systems) are largely not relevant to AFTT. Courses in programming language theory are more relevant but are not presented at an appropriate level. As an example: To an academic computer scientist, the "science behind Haskell" is the theory of lambda-calculus, the type-theoretic "System $F\omega$", and formal semantics. These theories guided the design of the Haskell language and define rigorously what a Haskell program "means" in a mathematical sense. Academic computer science curricula typically present these theories at the graduate level.

However, a practicing Haskell or Scala programmer is not concerned with designing Haskell or Scala, or with proving any theoretical properties of those languages. A practicing programmer is mainly concerned with *using* a chosen programming language to *write code*.

Neither the theory of lambda-calculus, nor proofs of type-theoretical properties of "System $F\omega$", nor theories of formal semantics, nor domain theory will actually help a programmer to write code. So all these theories are not within the scope of AFTT. It appears that functional programming does not require graduate-level theoretical studies.

As an example of theoretical material that *is* within the scope of AFTT, consider applicative functors (see Chapter 10). It is essential for a practicing functional programmer to be able to recognize and use applicative functors. An applicative functor is a data structure specifying declaratively a set of operations that can run independently of each other. Programs may combine these operations, for example, to execute them in parallel, or to refactor the program for better maintainability.

To use this functionality, the programmer must begin by checking whether a given data structure satisfies the laws of applicative functors. In a given application, a data structure may be dictated in

[1] https://wiki.haskell.org/Research_papers/Functional_pearls

part by the business logic rather than by a programmer's choice. The programmer first writes down the type of that data structure and the code implementing the required methods, and then checks that the laws hold. The data structure may need to be adjusted in order to fit the definition of an applicative functor or its laws.

So, before starting to write the actual code, the programmer needs to perform a certain amount of symbolic reasoning and calculation. That work is best done using pen and paper, writing equations in a mathematical notation. Once the applicative laws are verified, the programmer proceeds to write code using that data structure.

Because of the mathematical proofs, it is assured that the data structure satisfies the properties of applicative functors, no matter how the rest of the program is written. So, for example, it is assured that the relevant effects can be automatically parallelized and will still work correctly. In this way, AFTT directly guides the programmer and helps to write correct code.

Applicative functors were discovered by practitioners who were using Haskell in applications such as parser combinators, compilers, and domain-specific languages for parallel computations. However, applicative functors are not a feature of Haskell: they can be used in Scala or any other functional programming language. And yet, no standard computer science textbook defines applicative functors, motivates their laws, explores their structure on examples, or shows data structures that are *not* applicative functors and explains why. Books on category theory and type theory rarely ever mention applicative functors (known in mathematics as "lax monoidal" functors[2]).

18.2 AFTT is not category theory, type theory, or formal logic

It appears that AFTT includes a selection of certain areas of category theory, formal logic, and type theory. However, software engineers would not benefit from traditional academic courses in these subjects, because their presentation is too abstract and at the same time lacks specific results necessary for practical programming. In other words, the traditional academic courses answer questions that academic computer scientists have, not questions that practicing software engineers have.

There exist several books intended as presentations of category theory "for computer scientists" or "for programmers". However, those books do not explain many concepts relevant to programming, such as applicative or traversable functors. Instead, those books dwell on concepts (e.g. limits, enriched categories, topoi) that have no applications in practical functional programming today.

Typical questions in academic books are "Is X an introduction rule or an elimination rule" and "Does property Y hold in non-small categories or only in the category of sets". Questions a Scala programmer might have are "Can we compute a value of type `Either[Z, R => A]` from a value of type `R => Either[Z, A]`" and "Is the type constructor `F[A] = Option[(A,A,A)]` a monad or only an applicative functor". The scope of AFTT includes answering the last two questions but *not* the first two.

A software engineer hoping to understand the foundations of functional programming will not find the concepts of filterable, applicative, or traversable functors in any books on category theory, including books intended for programmers. And yet, these concepts are necessary to obtain a mathematically correct implementation of such foundationally important operations as `filter`, `zip`, and `traverse` – operations that functional programmers often use in their code.

To compensate for the lack of AFTT textbooks, programmers have written many online tutorials for each other, trying to explain the theoretical concepts necessary for practical work. There are the infamous "monad tutorials", but also tutorials about applicative functors, traversable functors, free monads, and so on. These tutorials tend to be hands-on ("run this code now and see what happens") and narrow in scope, limited to one or two specific questions and specific applications. Such tutorials usually do not present sufficient mathematical insights.

For example, "free monads" became popular in the Scala community around 2015. Many talks about free monads were presented at Scala engineering conferences, each giving their own slightly different implementation but never formulating rigorously the properties required for a piece of

[2] https://en.wikipedia.org/wiki/Monoidal_functor

code to be a valid implementation of the free monad. Without knowing the required mathematical properties of free monads, a programmer cannot make sure that a given implementation is correct. However, books on category theory define free monads in a way that is unsuitable for programming applications: a free monad is an adjoint functor to a forgetful functor from a Kleisli category to the category of sets. ("*A monad is just a monoid in the category of endofunctors. What's the problem?*" as the joke[3] goes.) Such definitions are far too abstract for practical use: for instance, they do not help at all if our task is to implement the free monad in code and to verify its correctness.

Perhaps the best selection of AFTT tutorial material today can be found in the Haskell Wikibooks.[4] However, those tutorials are incomplete and limited to explaining the use of Haskell. Many of them are suitable neither as a first introduction nor as a reference on AFTT. Also, the Haskell Wikibooks tutorials rarely show any derivations of laws or explain the required techniques.

Apart from referring to some notions from category theory, AFTT also uses concepts from type theory and formal logic. However, existing textbooks on type theory and formal logic focus on domain theory and proof theory – which presents a lot of information that will be difficult to learn for practicing programmers and yet will never be applicable in their daily work. At the same time, those academic books never mention practical techniques used in many functional programming libraries today, such as reasoning about and implementing quantified types, types parameterized by type constructors, or partial type-level functions (known as "typeclasses").

The proper scope of AFFT is helping the programmer with practical tasks such as:

- Deciding whether two data types are equivalent and implementing the isomorphism transformations. For example, the Scala type `(A, Either[B,C])` is equivalent to `Either[(A,B), (A,C)]`.
- Checking whether a definition of a recursive type is "reasonable", i.e. does not lead to infinite loops. An example of an "unreasonable" recursive type definition is `case class Bad(x: Bad)`.
- Deciding whether a function with a given type signature can be implemented. For example,

```
def f[Z,A,R]: (R => Either[Z, A]) => Either[Z, R => A] = ???   // Cannot be implemented.
def g[Z,A,R]: Either[Z, R => A] => (R => Either[Z, A]) = ???   // Can be implemented.
```

- Deriving an implementation of a function from its type signature and checking the required laws; for example, deriving the `flatMap` method and checking the laws for the `Reader` monad,

```
def flatMap[Z, A, B](r: Z => A)(f: A => Z => B): Z => B = ???
```

These are real-world applications of type theory and the Curry-Howard correspondence, but existing books on type theory and logic do not show practical recipes for performing these tasks.

Books such as *Scala with Cats*[5], *Functional programming simplified*[6], and *Functional programming for mortals*[7] are primarily focused on explaining practical aspects of functional programming and do not explain the mathematical principles and laws for e.g. applicative, monadic, or traversable functors.

The only Scala-based AFTT textbook is *Functional Programming in Scala*[8], which balances practical coding with theoretical developments and equational laws. *Program design by calculation*[9] is another (Haskell-based) AFTT book in progress. The present book, *Science of Functional Programming*[10], is written at about the same level but aims at better motivation for mathematical concepts and a wider range of examples that help build the necessary intuition and technique.

[3] https://stackoverflow.com/questions/3870088/
[4] https://en.wikibooks.org/wiki/Haskell
[5] https://underscore.io/books/scala-with-cats/
[6] https://alvinalexander.com/scala/functional-programming-simplified-book
[7] http://www.lulu.com/shop/search.ep?contributorId=1600066
[8] https://www.manning.com/books/functional-programming-in-scala
[9] http://www4.di.uminho.pt/~jno/ps/pdbc.pdf
[10] https://github.com/winitzki/sofp

19 Essay: Software engineers and software artisans

Let us look at the differences between the kind of activities we ordinarily call engineering, as opposed to artisanship or craftsmanship. It will then become apparent that today's computer programmers are better understood as "software artisans" rather than software engineers.

19.1 Engineering disciplines

Consider what kinds of process a mechanical engineer, a chemical engineer, or an electrical engineer follows in their work, and what kind of studies they require for proficiency in their work.

A mechanical engineer studies[1] calculus, linear algebra, differential geometry, and several areas of physics such as theoretical mechanics, thermodynamics, and elasticity theory, and then uses calculations to guide the design of a bridge, say. A chemical engineer studies[2] chemistry, thermodynamics, calculus, linear algebra, differential equations, some areas of physics such as thermodynamics and kinetic theory, and uses calculations to guide the design of a chemical process, say. An electrical engineer studies[3] advanced calculus, linear algebra, and several areas of physics such as electrodynamics and quantum theory, and uses calculations to design an antenna or a microchip.

The pattern here is that an engineer uses mathematics and natural sciences in order to design new devices. Mathematical calculations and scientific reasoning are required *before* drawing a design, let alone building a real device or machine.

Some of the studies required for engineers include arcane abstract concepts such as a "rank-4 elasticity tensor"[4] (used in calculations of elasticity of materials), "Lagrangian with non-holonomic constraints"[5] (used in robotics), the "Gibbs free energy" (for chemical reactor design[6]), or the "Fourier transform of the delta function"[7] and the "inverse Z-transform"[8] (for digital signal processing).

To be sure, a significant part of what engineers do is not covered by any theory: the *know-how*, the informal reasoning, the traditional knowledge passed on from expert to novice, — all those skills that are hard to formalize are important. Nevertheless, engineering is crucially based on natural science and mathematics for some of its decision-making about new designs.

19.2 Artisanship: Trades and crafts

Now consider what kinds of things shoemakers, plumbers, or home painters do, and what they have to learn in order to become proficient in their profession.

A novice shoemaker, for example, begins by copying some drawings[9] and goes on to cutting leather in a home workshop. Apprenticeships proceed via learning by doing, with comments and

[1] https://www.colorado.edu/mechanical/undergraduate-students/curriculum
[2] https://www.colorado.edu/engineering/sample-undergraduate-curriculum-chemical
[3] http://archive.is/XYLyE
[4] https://serc.carleton.edu/NAGTWorkshops/mineralogy/mineral_physics/tensors.html
[5] https://arxiv.org/abs/math/0008147
[6] https://www.amazon.com/Introduction-Chemical-Engineering-Kinetics-Reactor/dp/1118368258
[7] https://www.youtube.com/watch?v=KAbqISZ6SHQ
[8] http://archive.is/SsJqP
[9] https://youtu.be/cY5MY0czMAk?t=141

instructions from an expert. After a few years of apprenticeship (for example, a painter apprenticeship in California[10] can be as short as 2 years), a new specialist is ready to start productive work.

All these trades operate entirely from tradition and practical experience. The trades do not require academic study because there is no formal theory from which to proceed. Of course, there is *a lot* to learn in the crafts, and it takes prolonged effort to become a good artisan in any profession. But there are no rank-4 tensors to calculate, nor any differential equations to solve; no Fourier transforms to apply to delta functions, and no Lagrangians to check for non-holonomic constraints.

Artisans do not study any formal science or mathematics because their professions do not make use of any *formal computation* for guiding their designs or processes.

19.3 Programmers today are artisans, not engineers

Programmers are *not engineers* in the sense we normally see the engineering professions.

19.3.1 No requirement of formal study

According to this recent Stack Overflow survey[11], about half of the programmers do not have a degree in Computer Science. The author of this book is a self-taught programmer who has degrees in physics but never formally studied computer science or taken any academic courses in algorithms, data structures, computer networks, compilers, programming languages, or other CS topics.

A large fraction of successful programmers have no college degrees and perhaps *never* studied formally. They acquired all their knowledge and skills through self-study and practical work. Robert C. Martin[12] is a prominent example; an outspoken guru in the arts of programming who has seen it all, he routinely refers to programmers as artisans[13] and uses the appropriate imagery: novices, trade and craft, the "honor of the guild", etc. He compares programmers to plumbers, electricians, lawyers, and surgeons, but never to mathematicians, physicists, or engineers of any kind. According to one of his blog posts[14], he started working at age 17 as a self-taught programmer, and then went on to more jobs in the software industry; he never mentions going to college. It is clear that R. C. Martin *is* an expert craftsman, and that he did *not* need academic study to master his craft.

Here is another opinion[15] (emphasis is theirs):

> Software Engineering is unique among the STEM careers in that it absolutely does *not* require a college degree to be successful. It most certainly does not require licensing or certification. *It requires experience*.

This description fits a career in crafts – but certainly not a career, say, in electrical engineering.

The high demand for software developers gave rise to "developer boot camps"[16] — vocational schools that educate new programmers in a few months through purely practical training, with no formal theory or mathematics involved. These vocational schools are successful[17] in job placement. But it is unimaginable that a 6-month crash course or even a 2-year vocational school could prepare an engineer to work successfully on say, quantum computers[18] without ever having studied quantum physics or calculus.

[10] http://www.calapprenticeship.org/programs/painter_apprenticeship.php
[11] https://thenextweb.com/insider/2016/04/23/dont-need-go-college-anymore-programmer/
[12] https://en.wikipedia.org/wiki/Robert_C._Martin
[13] https://blog.cleancoder.com/uncle-bob/2013/02/01/The-Humble-Craftsman.html
[14] https://blog.cleancoder.com/uncle-bob/2013/11/25/Novices-Coda.html
[15] http://archive.is/tAKQ3
[16] http://archive.is/GkOL9
[17] http://archive.is/E9FXP
[18] https://www.dwavesys.com/quantum-computing

19.3.2 No mathematical formalism guides software development

Most books on software engineering contain no formulas or equations, no mathematical derivations of any results, and no precise definitions of the various technical terms they are using (such as "object-oriented" or "software architecture"). Some of those books[19] have almost no program code in them; instead they are filled with words and illustrative diagrams. These books talk about how programmers should approach their job, how to organize the work flow and the code architecture, etc., in vague and general terms: "code is about detail", "you must never abandon the big picture", "you should avoid tight coupling in your modules", "a class must serve a single responsibility", and so on. Practitioners such as R. C. Martin never studied any formalisms and do not think in terms of formalisms; instead they think in vaguely formulated, heuristic "principles".[20]

In contrast, textbooks on mechanical or electrical engineering include a significant amount of mathematics. The design of a microwave antenna is guided not by an "open and closed module principle" but by solving the relevant differential equations[21] coming from electrodynamics.

Donald Knuth's classic textbook is called "*The Art of Programming*". It is full of tips and tricks about how to program; but it does not provide any formal theory that could guide programmers in actually *writing* programs. There is nothing in that book that would be similar to the way mathematical formalism guides designs in electrical or mechanical engineering. If Knuth's books were based on such formalism, they would have looked quite differently: some theory would be first explained and then applied to help us write code.

Knuth's books provide many rigorously derived algorithms. But algorithms are similar to patented inventions: they can be used immediately without further study. Understanding an algorithm is not similar to understanding a mathematical theory. Knowing one algorithm does not make it easier to develop another algorithm in an unrelated domain. In comparison, knowing how to solve differential equations will be applicable to thousands of different areas of science and engineering.

A book exists[22] with the title "Science of Programming", but the title is misleading. The author does not propose a science, similar to physics, at the foundation of the process of designing programs, similarly to how calculations in quantum physics predict the properties of a quantum device. The book claims to give precise methods that guide programmers in writing code, but the scope of proposed methods is narrow: the design of simple algorithms for iterative manipulation of data. The procedure suggested in that book is far from a formal mathematical *derivation* of programs from specifications. (A book with that title[23] also exists, and similarly disappoints.) In any case, programmers today are oblivious to these books and do not use the methods explained there.

Standard computer science courses today do not teach a true *engineering* aproach to software construction. They do teach analysis of programs using formal mathematical methods; the main such methods are complexity analysis[24] (the "big-O notation") and formal verification[25]. But programs are analyzed or verified only *after* they are complete. Theory does not guide the actual *process* of writing code, does not suggest good ways of organizing the code (e.g. choosing which classes or functions or modules should be defined), and does not tell programmers which data structures or APIs would be best to implement. Programmers make these design decisions purely on the basis of experience and intuition, trial-and-error, copy-paste, and guesswork.

The theory of program analysis and verification is somewhat analogous to writing a mathematical equation for the surface of a shoe made by a fashion designer. True, the "shoe surface equations" are mathematically rigorous and can be "analyzed" or "verified"; but the equations are written after the fact and do not guide the fashion designers in actually making shoes. It is understandable that fashion designers do not study the mathematical theory of surfaces.

[19] E.g. https://www.amazon.com/Object-Oriented-Software-Engineering-Unified-Methodology/dp/0073376256

[20] https://blog.cleancoder.com/uncle-bob/2016/03/19/GivingUpOnTDD.html

[21] https://youtu.be/8KpfVsJ5Jw4?t=447

[22] https://www.amazon.com/Science-Programming-Monographs-Computer/dp/0387964800

[23] https://www.amazon.com/Program-Derivation-Development-Specifications-International/dp/0201416247

[24] https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complexity.html

[25] https://en.wikipedia.org/wiki/Formal_verification

19.3.3 Programmers avoid academic terminology

Programmers jokingly grumble about terms such as "functor", "monad", or "lambda-functions":

> Those fancy words used by functional programmers purists really annoy me. Monads, functors... Nonsense!!! [26]

Perhaps only a small minority of software developers actually complain about this; the vast majority seems to remain unaware of "functors" and "monads".

However, chemical engineers accept the need for studying differential equations and do not mind using the terms "phase diagram" or "Gibbs free energy". Electrical engineers do not complain that the word "Fourier" is difficult to spell, or that "delta-function" is a weird thing to say. Mechanical engineers take it for granted that they need to calculate with "tensors" and "Lagrangians" and "non-holonomic constraints". The arcane terminology seems to be the least of their difficulties, as their textbooks are full of complicated equations and long derivations.

Similarly, software engineers would not complain about the word "functor", or about having to study the derivation of the algebraic laws for "monads," — if they were true engineers. Textbooks on true software engineering would have been full of equations and derivations, in order to teach engineers how to perform certain calculations that are required *before* starting to write code.

19.4 Towards software engineering

It is now clear that we do not presently have true software engineering. The people employed under that job title are actually artisans. They work using artisanal methods, and their culture and processes are that of a crafts guild.

One could point out that numerical simulations required for physics or the matrix calculations required for machine learning are "mathematical". True, these programming *tasks* are mathematical in nature and require formal theory to be *formulated*. However, mathematical *subject matter* (aerospace control, physics or astronomy experiments, mathematical statistics, etc.) does not mean that the process of programming is a form of engineering. Data scientists, aerospace engineers, and natural scientists all write code nowadays — and they are all working as artisans when they write code.

True software engineering means having a theory that guides and informs the process of creating programs, — not theory that describes or analyzes programs after they are *somehow* written.

We expect that software engineers' textbooks should be full of equations. What theory should those equations represent?

I believe this theory already exists, and I call it **applied functional type theory**. It is the algebraic foundation of the modern practice of functional programming, as implemented in languages such as OCaml, Haskell, and Scala. This theory is a blend of type theory, category theory, and logical proof theory. It has been in development since late 1990s and is still being actively worked on by a community of academic computer scientists and advanced software practitioners.

To appreciate that functional programming, unlike any other programming paradigm, *has a theory that guides coding*, we can look at some recent software engineering conferences such as "Scala By the Bay"[27] or BayHac[28], or at the numerous FP-related online tutorials and blogs. We cannot fail to notice that much time is devoted not to showing code but to a peculiar kind of mathematical reasoning. Rather than focusing on one or another API or algorithm, as it is often the case with other software engineering blogs or presentations, an FP speaker describes a *mathematical structure* — such as the "applicative functor"[29] or the "free monad"[30] — and illustrates its use for practical coding.

[26] http://archive.is/65K3D
[27] http://2015.scala.bythebay.io/
[28] http://bayhac.org/
[29] http://www.youtube.com/watch?v=bmIxIslimVY
[30] http://www.youtube.com/watch?v=U0lK0hnbc4U

These people are not graduate students showing off their theoretical research; they are practitioners, software engineers who use FP on their jobs. It is just the nature of FP that certain mathematical tools — coming from formal logic and category theory — are now directly applicable to practical programming tasks.

These mathematical tools are not mere tricks for a specific programming language; they apply equally to all FP languages. Before starting to write code, the programmer can jot down certain calculations in a mathematical notation (see Fig. 19.1). The results of those calculations will help design the code fragment the programmer is about to write. This activity is similar to that of an engineer who performs some mathematical calculations before embarking on a design project.A recent example of the hand-in-hand development of the functional type theory and its applications is seen in the "free applicative functor" construction. It was first described in a 2014 paper[31]; a couple of years later, a combined free applicative / free monad data type was designed and its implementation proposed [32] as well as in Haskell[33]. This technique allows programmers to implement declarative side-effect computations where some parts are sequential but other parts are computed in parallel, and to achieve the parallelism *automatically* while maintaining the composability of the resulting programs. The new technique has distinct advantages over using monad transformers, which was the previous method of composing declarative side-effects.

The "free applicative / free monad" combination was designed and implemented by true software engineers. They first wrote down the types and derived the necessary algebraic properties; the obtained results directly guided them about how to proceed writing the library API.

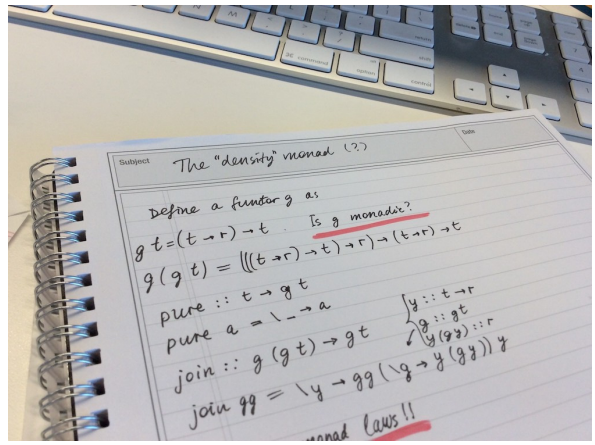


Figure 19.1: A programmer performs a derivation before writing Haskell code.

Another example of a development in functional type theory is the "tagless final" encoding of data types, first described in 2009. This technique, developed from category theory and type theory motivations, has several advantages over the free monad technique and can improve upon it in a number of cases — just as the free monad itself was designed to cure certain problems with monad transformers[34]. The new technique is also not a trick in a specific programming language; rather, it is a theoretical development that is available to programmers in any language (even in Java[35]).

This example shows that we may need several more years of work before the practical aspects of using "functional type theory" are sufficiently well understood by the FP community. The theory is in active development, and its design patterns — as well as the exact scope of the requisite theoretical material — are still being figured out. If the 40-year gap hypothesis[36] holds, we should expect functional type theory (perhaps under a different name) to become mainstream by 2030. This book is a step towards a clear designation of the scope of that theory.

[31] https://arxiv.org/pdf/1403.0749.pdf
[32] https://github.com/typelevel/cats/issues/983
[33] https://elvishjerricco.github.io/2016/04/08/applicative-effects-in-free-monads.html
[34] http://blog.ezyang.com/2013/09/if-youre-using-lift-youre-doing-it-wrong-probably/
[35] http://archive.is/rLAh9
[36] http://archive.is/rJc4A

19.5 Does software need engineers, or are artisans good enough?

The demand for programmers is growing. "Software developer" was #1 best job[37] in the US in 2018. But is there a demand for engineers or just for artisans?

We do not seem to be able[38] to train enough software artisans. So, it is probably impossible to train as many software engineers in the true sense of the word. Modern computer science courses do not actually train engineers in that sense; at best, they train academic researchers who write code as software artisans. Recalling the situation in construction business, with a few architects and hundreds of construction workers, we might also conclude that, perhaps, only a few software engineers are required per hundred software artisans.

What is the price of *not* having engineers, of replacing them with artisans?

Software practitioners have long bemoaned the mysterious difficulty of software development. Code "rots with time", its complexity grows "out of control", and operating systems have been notorious for ever-appearing security flaws[39] despite many thousands of programmers and testers employed. Clearly, we overestimated the capacity of the human brain for artisanal programming.

It is precisely in designing large and robust software systems that we would benefit from true engineering. Artisans has been building bridges and using chemical reactions by following tradition and by trial and error, long before mechanical or chemical engineering disciplines were developed and founded upon rigorous theory. But once the theory became available, engineers proceeded to design unimaginably more powerful and complicated structures and devices. It is clear that trial, error, and adherence to tradition is inadequate for the software development tasks in front of us.

To build large and reliable software, such as new mobile or embedded operating systems or distributed peer-to-peer trust architectures, we will most likely need the qualitative increase in productivity and reliability that can only come from replacing artisanal programming by a true engineering discipline. Functional type theory and functional programming are first steps in that direction.

[37] http://archive.is/cGJ2T
[38] http://archive.is/137b8
[39] http://archive.fo/HtQzw

20 Essay: Towards functional data engineering with Scala

Data engineering is among the most in-demand[1] novel occupations in the IT world today. Data engineers create software pipelines that process large volumes of data efficiently. Why did the Scala programming language emerge as a premier tool[2] for crafting the foundational data engineering technologies such as Spark or Akka? Why is Scala in such demand[3] within the world of big data?

There are reasons to believe that the choice of Scala was not accidental.

20.1 Data is math

Humanity has been working with data at least since Babylonian tax tables[4] and the ancient Chinese number books[5]. Mathematics summarizes several millennia's worth of data processing experience in a few fundamental tenets:

- Data is *immutable*, because facts are immutable.
- Each *type* of values (population count, land area, distances, prices, times, etc.) needs to be handled separately; it is meaningless to add a distance to a population count.
- Data processing should be performed according to *mathematical formulas*.

Violating these tenets produces nonsense (see Fig. 20.1 for a real-life illustration).

The power of the principles of mathematics extends over all epochs and all cultures; math is the same in San Francisco, in Rio de Janeiro, in Kuala-Lumpur, and in Pyongyang (Fig. 20.2).

Figure 20.1: Mixing incompatible data types produces nonsensical results.

20.2 Functional programming is math

The functional programming paradigm is based on mathematical principles: values are immutable, data processing is coded through formula-like expressions, and each type of data is required to match correctly during the computations. The type-checking process automatically prevents programmers from making many kinds of coding errors. In addition, programming languages such as Scala and Haskell have a set of features adapted to building powerful abstractions and domain-specific languages. This power of abstraction is not accidental. Since mathematics is the ultimate art of building abstractions, math-based functional programming languages capitalize on the advantage of millennia of mathematical experience.

[1] http://archive.is/mK59h
[2] https://www.slideshare.net/noootsab/scala-the-unpredicted-lingua-franca-for-data-science
[3] https://techcrunch.com/2016/06/14/scala-is-the-new-golden-child/
[4] https://www.nytimes.com/2017/08/29/science/trigonometry-babylonian-tablet.html
[5] http://quatr.us/china/science/chinamath.htm

A prominent example of how mathematics informs the design of programming languages is the connection between constructive logic[6] and the programming language's type system, called the Curry-Howard (CH) correspondence[7]. The main idea of the CH correspondence is to think of programs as mathematical formulas that compute a value of a certain type *A*. The CH correspondence is between programs and logical propositions: To any program that computes a value of type *A*, there corresponds a proposition stating that "a value of type *A* can be computed".

This may sound rather theoretical so far. To see the real value of the CH correspondence, recall that formal logic has operations "***and***", "***or***", and "***implies***". For any two propositions *A*, *B*, we can construct the propositions "*A* ***and*** *B*", "*A* ***or*** *B*", "*A* ***implies*** *B*". These three logical operations are foundational; without one of them, the logic is *incomplete* (you cannot derive some theorems).

A programming language **obeys the CH correspondence** with the logic if for any types *A*, *B*, the language also contains composite types corresponding to the logical formulas "*A* ***or*** *B*", "*A* ***and*** *B*", "*A* ***implies*** *B*". In Scala, these composite types are `Either[A,B]`, the tuple `(A,B)`, and the function type, `A⇒B`. All modern functional languages such as OCaml, Haskell, Scala, F#, Swift, Elm, and PureScript support these three type constructions and thus are faithful to the CH correspondence. Having a *complete* logic in a language's type system enables declarative domain-driven code design[8].

Figure 20.2: The Pyongyang method of error-free software engineering.

It is interesting to note that most older programming languages (C/C++, Java, JavaScript, Python) do not support some of these composite types. In other words, these programming languages have type systems based on an incomplete logic. As a result, users of these languages have to implement burdensome workarounds that make for error-prone code. Failure to follow mathematical principles has real costs (Figure 20.2).

20.3 The power of abstraction

Early adopters of Scala, such as Netflix, LinkedIn, and Twitter, were implementing what is now called "big data engineering". The required software needs to be highly concurrent, distributed, and resilient to failure. These software companies used Scala as their main implementation language and reaped the benefits of functional programming.

What makes Scala suitable for big data tasks? The only reliable way of managing massively concurrent code is to use sufficiently high-level abstractions that make application code declarative. The two most important such abstractions are the "resilient distributed dataset" (RDD) of Apache Spark and the "reactive stream" used in systems such as Kafka, Akka Streams, and Apache Flink. While these abstractions are certainly implementable in Java or Python, a fully declarative and type-safe usage is possible only in a programming language with a sophisticated functional type system. Among the currently available mature functional languages, only Scala and Haskell are technically adequate for that task, due to their support for typeclasses and higher-order types. The early adopters of Scala were able to reap the benefits of the powerful abstractions Scala supports. In this way, Scala enabled those businesses to engineer reliably and to scale up their massively concurrent computations.

It remains to see why Scala and not, say, Haskell became the *lingua franca* of big data.

[6] https://en.wikipedia.org/wiki/Intuitionistic_logic
[7] https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence
[8] https://fsharpforfunandprofit.com/ddd/

20.4 Scala is Java on math

The recently invented general-purpose functional programming languages can be grouped into "industrial" (F#, Scala, Swift) and "academic" (OCaml, Haskell).

The "academic" languages are clean-room implementations of well-researched mathematical principles of programming language design (the CH correspondence being one such principle). These languages are unencumbered by requirements of compatibility with any existing platform or libraries. Because of this, the "academic" languages are perfect playgrounds for taking various mathematical ideas to their logical conclusion. At the same time, software practitioners struggle to adopt these languages due to a steep learning curve, a lack of enterprise-grade libraries and tool support, and immature package management.

The languages from the "industrial" group are based on existing and mature software ecosystems: F# on .NET, Scala on JVM, and Swift on the MacOS/iOS platform. One of the important design requirements for these languages is 100% binary compatibility with their "parent" platforms and languages (F# with C#, Scala with Java, and Swift with Objective-C). Because of this, developers can immediately take advantage of the existing tooling, package management, and industry-strength libraries, while slowly ramping up the idiomatic usage of new language features. However, the same compatibility requirements necessitated certain limitations in the languages, making their design less than fully satisfactory from the functional programming viewpoint.

It is now easy to see why the adoption rate of the "industrial" group of languages is much higher[9] than that of the "academic" languages. The transition to the functional paradigm is also made smoother for software developers because F#, Scala, and Swift seamlessly support the familiar object-oriented programming paradigm. At the same time, these new languages still have logically complete type systems, which gives developers an important benefit of type-safe domain modeling.

Nevertheless, the type systems of these languages are not equally powerful. For instance, F# and Swift are similar to OCaml in many ways but omit OCaml's parameterized modules and some other features. Of all mentioned languages, only Scala and Haskell directly support typeclasses and higher-order types, which are helpful for expressing abstractions such as automatically parallelized data sets or asynchronous data streams.

To see the impact of these advanced features, consider LINQ, a domain-specific language for database queries on .NET, implemented in C# and F# through a special built-in syntax supported by Microsoft's compilers. Analogous functionality is provided in Scala as a *library*, without need to modify the Scala compiler, by several open-source projects such as Slick, Squeryl, or Quill. Similar libraries exist for Haskell – but not in languages with less powerful type systems.

20.5 Summary

The decisive advantages of Scala over other contenders (such as OCaml, Haskell, F#, or Swift) are:

1. Functional collections in the standard library.
2. A sophisticated type system with support for typeclasses and higher-order types.
3. Seamless compatibility with a mature software ecosystem (JVM).

Based on this assessment, we may be confident in Scala's great future as a main implementation language for big data engineering.

[9] https://www.tiobe.com/tiobe-index/, archived in 2019 at http://archive.is/RsNH8

Part V

Appendixes

A Notations

Certain notations and terms were chosen in this book differently from what the functional programming community currently uses. The proposed notation is well adapted to reasoning about types and code, and especially for designing data types and proving the equational laws of typeclasses.

A.1 Summary of notations for types and code

F^A type constructor F with type argument A. In Scala, `F[A]`

$x^{:A}$ value x has type A; in Scala, `x:A`

$\mathbb{1}$, 1 the unit type and its value; in Scala, `Unit` and `()`

$\mathbb{0}$ the void type. In Scala, `Nothing`

$A + B$ a disjunctive type. In Scala, this type is `Either[A, B]`

$x^{:A} + \mathbb{0}^{:B}$ a value of a disjunctive type $A + B$. In Scala, `Left(x)`

$A \times B$ a product (tuple) type. In Scala, this type is `(A,B)`

$a^{:A} \times b^{:B}$ value of a tuple type $A \times B$. In Scala, `(a, b)`

$A \to B$ the function type, mapping from A to B

$x^{:A} \to f$ a nameless function (as a value). In Scala, `{ x:A => f }`

id the identity function; in Scala, `identity[A]`

$\triangleq$ "equal by definition"

$\cong$ for types, a natural isomorphism between types; for values, "equivalent" values according to an established isomorphism

$A^{:F^B}$ type annotation, used for defining unfunctors (GADTs)

$\wedge$ logical conjunction; $\alpha \wedge \beta$ means "both α and β are true"

$\vee$ logical disjunction; $\alpha \vee \beta$ means "either α or β or both are true"

$\Rightarrow$ logical implication; $\alpha \Rightarrow \beta$ means "if α is true then β is true"

$\mathbf{fmap}_F$ the standard method fmap pertaining to a functor F. In Scala, `Functor[F].fmap`

$\mathbf{flm}_F$ the standard method flatMap pertaining to a monad F. In Scala, `Monad[F].flatMap`

$\mathbf{ftn}_F$ the standard method flatten pertaining to a monad F. In Scala, `Monad[F].flatten`

$\mathbf{pu}_F$ the standard method `pure` of a monad F. In Scala, `Monad[F].pure`

$F^\bullet$ the type constructor F understood as a type-level function. In Scala, `F[_]`

$F^\bullet \rightsquigarrow G^\bullet$ or $F^A \rightsquigarrow G^A$ a natural transformation between functors F and G. In Scala, `F ~> G`

$\forall A. P^A$ a universally quantified type expression. In Scala 3, `[A] => P[A]`

$\exists A. P^A$ an existentially quantified type expression. In Scala, `{ type A; val x: P[A] }`

$\fatsemi$ the forward composition of functions: $f \fatsemi g$ is $x \to g(f(x))$. In Scala, `f andThen g`

$\circ$ the backward composition of functions: $f \circ g$ is $x \to f(g(x))$. In Scala, `f compose g`

$\circ$ the backward composition of type constructors: $F \circ G$ is $F^{G^\bullet}$

$\triangleright$ use a value as the argument of a function: $x \triangleright f$ is $f(x)$. In Scala, `x.pipe(f)`

$f^{\uparrow G}$ a function f raised to a functor G; same as $\text{fmap}_G f$

$f^{\uparrow G \uparrow H}$ a function raised first to G and then to H. In Scala, `h.map(_.map(f))`

$f^{\downarrow H}$ a function f raised to a contrafunctor

$\diamond_M$ the Kleisli product operation for the monad M

$\oplus$ the binary operation of a monoid. In Scala, `x |+| y`

Δ the "diagonal" function of type $\forall A.\, A \to A \times A$

$\nabla_1, \nabla_2, \ldots$ (pronounced "nabla") the projections from a tuple to its first, second, ..., parts

$\boxtimes$ pair product of functions, $(f \boxtimes g)(a \times b) = f(a) \times g(b)$

$[a, b, c]$ an ordered sequence of values. In Scala, `Seq(a, b, c)`

$\left\| \begin{array}{cc} x \to x & \mathbb{0} \\ \mathbb{0} & a \to a \times a \end{array} \right|$ a function that works with disjunctive types

A.2 Detailed explanations

F^A means a type constructor F with a type parameter A. In Scala, this is `F[A]`. Type constructors with multiple type parameters are denoted by $F^{A,B,C}$.

$x^{:A}$ means a value x that has type A; this is a **type annotation**. In Scala, a type annotation is `x:A`. The colon symbol, :, in the superscript shows that A is not a type argument (as it would be in a type constructor, F^A). The notation $x : A$ can be used as well, but $x^{:A}$ is easier to read when x is inside a larger code expression.

$\mathbb{1}$ means the unit type, and 1 means the value of the unit type. In Scala, the unit type is `Unit`, and its value is `()`. Example of using the unit type is $\mathbb{1} + A$, which corresponds to `Option[A]` in Scala.

$\mathbb{0}$ means the void type (the type with no values). In Scala, this is the type `Nothing`. Example of using the void type is to denote the empty part of a disjunction. For example, in the disjunction $\mathbb{1} + A$ the non-empty part is $\mathbb{0} + A$, which in Scala corresponds to `Some[A]`. The empty part $\mathbb{1} + \mathbb{0}$ corresponds to `None`. Similarly, $A + \mathbb{0}$ denotes the left part of the type $A + B$ (in Scala, `Left[A]`), while $\mathbb{0} + B$ denotes its right part (in Scala, `Right[A]`). Values of disjunctive types are denoted similarly. For instance, $x^{:A} + \mathbb{0}^{:B}$ denotes a value of the left part of the type $A + B$; in Scala, this value is written as `Left[A,B](x)`.

$A + B$ means the disjunctive type made from types A and B (or, a disjunction of A and B). In Scala, this is the type `Either[A, B]`.

$x^{:A} + \mathbb{0}^{:B}$ denotes a value of a disjunctive type $A + B$, where x is the value of type A, which is the chosen case, and $\mathbb{0}$ stands for other possible cases. For example, $x^{:A} + \mathbb{0}^B$ is `Left[A,B](x)` in Scala. Type annotations $^{:A}$ and $^{:B}$ may be omitted if the types are unambiguous from the context.

$A \times B$ means the product type made from types A and B. In Scala, this is the tuple type `(A,B)`.

$a^{:A} \times b^{:B}$ means a value of a tuple type $A \times B$; in Scala, this is the tuple value `(a, b)`. Type annotations $^{:A}$ and $^{:B}$ may be omitted if the types are unambiguous from the context.

$A \to B$ means a function type from A to B. In Scala, this is the function type `A => B`.

$x^{:A} \to y$ means a nameless function with argument x of type A and function body y. (Usually, the body y will be an expression that uses x. In Scala, this is `{ x: A => y }`. Type annotation $^{:A}$ may be omitted if the type is unambiguous from the context.

id means the identity function. The type of its argument should be either specified as id^A or $\text{id}^{:A \to A}$, or else should be unambiguous from the context. In Scala, `identity[A]` corresponds to id^A.

$\triangleq$ means "equal by definition". Examples:

- $f \triangleq (x^{:\text{Int}} \to x + 10)$ is a definition of a function f. In Scala, this is `val f = { x: Int => x + 10 }`.
- $F^A \triangleq \mathbb{1} + A$ is a definition of a type constructor F. In Scala, this is `type F[A] = Option[A]`.

$\cong$ for types means an equivalence (an isomorphism) of types. For example, $A + A \times B \cong A \times (\mathbb{1} + B)$. The same symbol $\cong$ for *values* means "equivalent" according to an equivalence relation that needs to be established in the text. For example, if we have established an equivalence that allows nested tuples to be reordered whenever needed, we can write $(a \times b) \times c \cong a \times (b \times c)$, meaning that these values are mapped to each other by the established isomorphism functions.

$A^{:F^B}$ in type expressions means that the type constructor $F^\bullet$ assigns the type F^B to the type expression A. This notation is used for defining unfunctors (GADTs). For example, the Scala code

```
sealed trait F[A]
case class F1() extends F[Int]
case class F2[A](a: A) extends F[(A, String)]
```

defines an unfunctor, which is denoted by $F^A \triangleq \mathbb{1}^{:F^{\text{Int}}} + A^{:F^{A \times \text{String}}}$.

$\wedge$ (conjunction), $\vee$ (disjunction), and $\Rightarrow$ (implication) are used in formulas of Boolean as well as constructive logic in Chapter 5, e.g. $\alpha \wedge \beta$, where Greek letters stand for logical propositions.

fmap_F means the standard method fmap of the `Functor` typeclass, implemented for the functor F. In Scala, this may be written as `Functor[F].fmap`. Since each functor F has its own specific implementation of fmap_F, the subscript "F" is not a type parameter of fmap_F. The method fmap_F actually has *two* type parameters, which can be written out as $\text{fmap}_F^{A,B}$. Then the type signature of fmap is written in full as $\text{fmap}_F^{A,B} : (A \to B) \to F^A \to F^B$. For clarity, we may sometimes write out the type parameters A, B in the expression $\text{fmap}_F^{A,B}$, but in most cases these type parameters A, B can be omitted without loss of clarity.

pu_F denotes a monad F's method `pure`. This function has type signature $A \to F^A$ that contains a type parameter A. In the code notation, the type parameter may be either omitted or denoted as pu_F^A. If we are using the `pure` method with a complicated type, e.g. $\mathbb{1} + P^A$, instead of the type parameter A, we might want to write this type parameter for clarity and write $\text{pu}_F^{\mathbb{1}+P^A}$. The type signature of that function is then

$$\text{pu}_F^{\mathbb{1}+P^A} : \mathbb{1} + P^A \to F^{\mathbb{1}+P^A} \quad .$$

But in most cases we will not need to write out the type parameters.

flm_F denotes a monad F's method `flatMap` with the type signature

$$\text{flm}_F : \left(A \to F^B\right) \to F^A \to F^B \quad .$$

Note that Scala's standard `flatMap` type signature is not curried. The curried method flm_F is easier to use in calculations involving the monad's laws.

ftn_F denotes a monad F's method `flatten` with the type signature

$$\text{ftn}_F : F^{F^A} \to F^A \quad .$$

$F^\bullet$ means the type constructor F understood as a type-level function, – that is, with a type argument unspecified. In Scala, this is `F[_]`. The bullet symbol, $\bullet$, is used as a placeholder for the missing

type parameter. I also simply write F when no type argument is needed, and it means the same as $F^\bullet$. (For example, "a functor F" and "a functor $F^\bullet$" mean the same thing.) However, it is useful for clarity to be able to indicate the place where the type argument would appear. For instance, functor composition is denoted as $F^{G^\bullet}$; in Scala, this is `Lambda[X => F[G[X]]]` when using the "kind projector" plugin.[1] As another example, $T_L^{M,\bullet}$ denotes a monad transformer for the base monad L and the foreign monad M. The foreign monad M is a type parameter in $T_L^{M,\bullet}$, and so is the missing type parameter denoted by the placeholder symbol $\bullet$. (However, the base monad L is not a type parameter in $T_L^{M,\bullet}$ because the construction of the monad transformer depends sensitively on the internal details of L.)

$F^\bullet \rightsquigarrow G^\bullet$ or $F^A \rightsquigarrow G^A$ means a natural transformation between two functors F and G. In some Scala libraries, this is denoted by `F ~> G`.

$\forall A.P^A$ is a universally quantified type expression, in which A is a bound type parameter.

$\exists A.P^A$ is an existentially quantified type expression, in which A is a bound type parameter.

$\fatsemi$ means the forward composition of functions: $f \fatsemi g$ (reads "f before g") is the function defined as $x \to g(f(x))$.

$\circ$ means the backward composition of functions: $f \circ g$ (reads "f after g") is the function defined as $x \to f(g(x))$.

$\circ$ with type constructors means their (backward) composition, for example $F \circ G$ denotes the type constructor $F^{G^\bullet}$. In Scala, this is `F[G[A]]`.

$x \triangleright f$ (the **pipe notation**) means that x is inserted as the argument into the function f. The expression $x \triangleright f$ means the same as $f(x)$. In Scala, the expression $x \triangleright f$ is written as `x.pipe(f)` or, if `f` is a method, `x.f`. This syntax is used with many standard methods such as `.size` or `.toSeq`. Because the function f is to the *right* of x in this notation, forward compositions of functions such as $x \triangleright f \triangleright g$ are naturally grouped to the left as it is done in Scala code, for example `x.toSeq.sorted`. The operation $\triangleright$ (pronounced "pipe") groups weaker than the forward composition ($\fatsemi$), and so we have $x \triangleright f \fatsemi g = x \triangleright f \triangleright g$ in this notation. Reasoning about code in the pipe notation uses the identities

$$x \triangleright f = f(x), \qquad (x \triangleright f) \triangleright g = x \triangleright f \triangleright g \quad ,$$
$$x \triangleright f \fatsemi g = x \triangleright (f \fatsemi g), \qquad x \triangleright f \triangleright g = x \triangleright f \fatsemi g \quad .$$

Some examples of reasoning in the pipe notation:

$$(a \to a \triangleright f) = (a \to f(a)) = f \quad ,$$
$$f \triangleright (y \to a \triangleright y) = a \triangleright f = f(a) \quad ,$$
$$f(y(x)) = x \triangleright y \triangleright f \neq x \triangleright (y \triangleright f) = f(y)(x) \quad .$$

The correspondence between the forward composition and the backward composition:

$$f \fatsemi g = g \circ f \quad ,$$
$$x \triangleright (f \fatsemi g) = x \triangleright f \fatsemi g = x \triangleright f \triangleright g = g(f(x)) = (g \circ f)(x) \quad .$$

$f^{\uparrow G}$ means a function f lifted to a functor G. For a function $f^{:A \to B}$, the application of $f^{\uparrow G}$ to a value $g^{:G^A}$ is written as $f^{\uparrow G}(g)$ or as $g \triangleright f^{\uparrow G}$. In Scala, this is `g.map(f)`. Nested lifting (i.e. lifting to the functor composition $H \circ G$) can be written as $f^{\uparrow G \uparrow H}$, which means $\left(f^{\uparrow G}\right)^{\uparrow H}$ and produces a function of type $H^{G^A} \to H^{G^B}$. Applying a nested lifting to a value h of type H^{G^A} is written as $f^{\uparrow G \uparrow H} h$ or $h \triangleright f^{\uparrow G \uparrow H}$. In Scala, this is `h.map(_.map(f))`. The functor composition law is written as

$$p^{\uparrow G} \fatsemi q^{\uparrow G} = (p \fatsemi q)^{\uparrow G} \quad .$$

Note the similarity between Scala code `x.map(p).map(q)` and the notation $x \triangleright p^{\uparrow G} \triangleright q^{\uparrow G}$.

[1] https://github.com/typelevel/kind-projector

$f^{\downarrow H}$ means a function f lifted to a contrafunctor H. For a function $f^{:A\to B}$, the application of $f^{\downarrow H}$ to a value $h : H^B$ is written as $f^{\downarrow H} h$ or $h \triangleright f^{\downarrow H}$, and yields a value of type H^A. In Scala, this is `h.contramap(f)`. Nested lifting is denoted as $f^{\downarrow H\uparrow G} \triangleq (f^{\downarrow H})^{\uparrow G}$.

$\diamond_M$ means the Kleisli product operation for a given monad M. This is a binary operation working on two Kleisli functions of types $A \to M^B$ and $B \to M^C$ and yields a new function of type $A \to M^C$.

$\oplus$ means the binary operation of a monoid, for example $x \oplus y$. The specific monoid type should be defined for this expression to make sense. For example, in Scala the monoidal operation is usually denoted by `x |+| y`.

Δ means the "diagonal" function of type $\forall A.\, A \to A \times A$. There is only one implementation of this type signature,

```
def delta[A](a: A): (A, A) = (a, a)
```

$\nabla_1, \nabla_2, \ldots$ denote the standard projection functions from a tuple to its first, second, ..., parts. In Scala, ∇_1 is `_._1`.

$\boxtimes$ means the "pair product" of functions, where the result is a pair of the values of the two functions: $(f \boxtimes g)(a \times b) = f(a) \times g(b)$. In Scala, this operation can be defined by

```
def boxtimes[A,B,P,Q](f: A => P, g: B => Q): ((A, B)) => (P, Q) = {
  case (a, b) => (f(a), g(b))
}
```

The operations Δ, ∇_i (where $i = 1, 2, \ldots$), and $\boxtimes$ allow us to express any function operating on tuples. Useful properties for reasoning about code of such functions:

$$\begin{aligned}
\text{identity law}: &\quad \Delta \fatsemi \nabla_i = \text{id} \quad , \\
\text{duplication law}: &\quad f \fatsemi \Delta = \Delta \fatsemi (f \boxtimes f) \quad , \\
\text{projection law}: &\quad (f \boxtimes g) \fatsemi \nabla_1 = \nabla_1 \fatsemi f \quad , \\
\text{projection law}: &\quad (f \boxtimes g) \fatsemi \nabla_2 = \nabla_2 \fatsemi g \quad , \\
\text{composition law}: &\quad (f \boxtimes g) \fatsemi (p \boxtimes q) = (f \fatsemi p) \boxtimes (g \fatsemi q) \quad ,
\end{aligned}$$

as well as the naturality laws for Δ and ∇_i:

$$\begin{aligned}
& f^{\uparrow F} \fatsemi \Delta = \Delta \fatsemi f^{\uparrow (F \times F)} = \Delta \fatsemi (f^{\uparrow F} \boxtimes f^{\uparrow F}) \quad , \\
& (f^{\uparrow F} \boxtimes f^{\uparrow G}) \fatsemi \nabla_1 = f^{\uparrow (F \times G)} \fatsemi \nabla_1 = \nabla_1 \fatsemi f^{\uparrow F} \quad .
\end{aligned}$$

$[a, b, c]$ means an ordered sequence of values, such as a list or an array. In Scala, this can be `List(a, b, c)`, `Vector(a, b, c)`, `Array(a, b, c)`, or another collection type.

$f^{:Z+A\to Z+A\times A} \triangleq \left\| \begin{array}{cc} z \to z & \mathbb{0} \\ \mathbb{0} & a \to a \times a \end{array} \right|$ is the **matrix notation** for a function whose input and/or output type is a disjunctive type. In Scala, the function f is implemented as

```
def f[Z, A]: Either[Z, A] => Either[Z, (A, A)] = {
  case Left(z)   => Left(z) // Identity function on Z.
  case Right(a)  => Right((a, a)) // Delta on A.
}
```

The rows of the matrix indicate the different `case`s in the function's code, corresponding to the different parts of the input disjunctive type. If the input type is not disjunctive, there will be only one row. The columns of the matrix indicate the parts of the output disjunctive type. If the the output type is not disjunctive, there will be only one column.

A matrix may show all parts of the disjunctive types in separate "type row" and "type column":

$$f^{:Z+A\Rightarrow Z+A\times A} \triangleq \left| \begin{array}{c|cc} & Z & A \times A \\ \hline Z & \text{id} & \mathbb{0} \\ A & \mathbb{0} & a \to a \times a \end{array} \right| \quad . \tag{A.1}$$

This notation clearly indicates the input and the output types of the function and is useful at some stages of reasoning about the code. The vertical double line separates input types from the function code. In the code above, the "type column" shows the parts of the input disjunctive type $Z + A$. The "type row" shows the parts of the output disjunctive type $Z + A \times A$.

The matrix notation is adapted to *forward* function compositions ($f \fatsemi g$). Assume that A is a monoid type, and consider the composition of the function f shown above and the function g defined as

```
def g[Z, A: Monoid]: Either[Z, (A, A)] => A = {
  case Left(_)          => Monoid[A].empty
  case Right((a1, a2))  => a1 |+| a2
}
```

In the matrix notation, the function g is written (with and without types) as

$$g \triangleq \left\| \begin{array}{c|c} & A \\ \hline Z & \_ \to e^{:A} \\ A \times A & a_1 \times a_2 \to a_1 \oplus a_2 \end{array} \right\| \quad , \qquad g \triangleq \left\| \begin{array}{c} \_ \to e^{:A} \\ \hline a_1 \times a_2 \to a_1 \oplus a_2 \end{array} \right\| \quad .$$

The forward composition $f \fatsemi g$ is computed by forward-composing the matrix elements using the rules of the ordinary matrix multiplication, omitting any terms containing $\mathbb{0}$:

$$\begin{aligned} f \fatsemi g &= \left\| \begin{array}{cc} \mathrm{id} & \mathbb{0} \\ \mathbb{0} & a \to a \times a \end{array} \right\| \fatsemi \left\| \begin{array}{c} \_ \to e^{:A} \\ \hline a_1 \times a_2 \to a_1 \oplus a_2 \end{array} \right\| \\ &= \left\| \begin{array}{c} \mathrm{id} \fatsemi (\_ \to e^{:A}) \\ \hline (a \to a \times a) \fatsemi (a_1 \times a_2 \to a_1 \oplus a_2) \end{array} \right\| = \left\| \begin{array}{c} \_ \to e^{:A} \\ \hline a \to a \oplus a \end{array} \right\| \quad . \end{aligned}$$

Applying a function to a value of a disjunctive type such as $x : Z + A$ is computed by writing x as a single-row matrix, for example

$$x = z^{:Z} + \mathbb{0}^{:A} = \left\| \begin{array}{cc} z^{:Z} & \mathbb{0} \end{array} \right\| \quad ,$$

and the computation $x \triangleright f \fatsemi g$ again follows the rules of matrix multiplication:

$$x \triangleright f \fatsemi g = \left\| \begin{array}{cc} z^{:Z} & \mathbb{0} \end{array} \right\| \triangleright \left\| \begin{array}{c} \_ \to e^{:A} \\ \hline a \to a \oplus a \end{array} \right\| = z \triangleright (\_ \to e) = e \quad .$$

Since the standard rules of matrix multiplication are associative, the properties of the $\triangleright$-notation such as $x \triangleright (f \fatsemi g) = (x \triangleright f) \triangleright g$ are guaranteed to hold.

To use the matrix notation with *backward* compositions ($f \circ g$), all function matrices need to be transposed. (A standard identity of matrix calculus is that the transposition reverses the order of composition, $(AB)^T = B^T A^T$.) The argument types will then appear in the top row and the result types in the left column; the double line is above the matrix since that is where the function inputs come from. The above calculations are then rewritten as

$$g \circ f = \left| \begin{array}{c|cc} & Z & A \times A \\ \hline A & \_ \to e^{:A} & a_1 \times a_2 \to a_1 \oplus a_2 \end{array} \right| \circ \left| \begin{array}{c|cc} & Z & A \\ \hline Z & \mathrm{id} & \mathbb{0} \\ A \times A & \mathbb{0} & a \to a \times a \end{array} \right|$$

$$= \left| \begin{array}{cc} \mathrm{id} \fatsemi (\_ \to e^{:A}) & (a \to a \times a) \fatsemi (a_1 \times a_2 \to a_1 \oplus a_2) \end{array} \right| = \left| \begin{array}{cc} \_ \to e^{:A} & a \to a \oplus a \end{array} \right| \quad .$$

$$(g \circ f)(x) = \left| \begin{array}{cc} \_ \to e^{:A} & a \to a \oplus a \end{array} \right| \left| \begin{array}{c} z^{:Z} \\ \mathbb{0} \end{array} \right| = (\_ \to e^{:A})(z) = e \quad .$$

The *forward* composition seems to be easier to read and to reason about in the matrix notation.

B Glossary of terms

Code notation A mathematical notation developed in this book for deriving properties of code in functional programs. Variables have optional type annotations, such as $x^{:A}$ or $f^{:A \to B}$. Nameless functions are denoted by $x^{:A} \to f$, products by $a \times b$, and values of a disjunctive type $A + B$ are written as $x^{:A} + \mathbb{0}^{:B}$ or $\mathbb{0}^{:A} + y^{:B}$. Functions working with disjunctive types are denoted by matrices. Lifting of functions to functors, such as $\text{fmap}_L(f)$, is denoted by $f^{\uparrow L}$; function compositions are denoted by $f \fatsemi g$ (forward composition) and $f \circ g$ (backward composition); and function applications by $f(x)$ or equivalently $x \triangleright f$. See Appendix A for details.

Contrafunctor A type constructor having the properties of a contravariant functor with respect to a type parameter. Instead of saying "contravariant functor", this book uses the shorter name "contrafunctor".

Disjunctive type A type representing one of several distinct possibilities. In Scala, this is usually implemented as a sealed trait extended by several case classes. The standard Scala disjunction types are `Option[A]` and `Either[A, B]`. Also known as **sum** type, **tagged union** type, **co-product** type, and variant type (in Object Pascal and in OCaml). The shortest name is "sum type," but the English word "sum" is more ambiguous to the ear than "disjunctive".

Exponential-polynomial type A type constructor built using products, disjunctions (sums or co-products), and function types ("exponentials"), as well as type parameters and fixed types. For example, `type F[A] = Either[(A,A), Int=>A]` is an exponential-polynomial type constructor. Such type constructors are always profunctors and can also be functors or contrafunctors.

Functor block A short syntax for composing several `.map`, `.flatMap`, and `.filter` operations applied to a functor-typed value. The type constructor corresponding to that value must therefore be fixed throughout the entire functor block. (The type constructor *must* be a functor and may additionally be filterable and/or monadic.) For example, in Scala the code

```
for { x <- List(1,2,3); y <- List(10, x); if y > 2 }
  yield 2 * y
```

is equivalent to the code

```
List(1, 2, 3).flatMap(x => List(10, x))
  .filter(y => y > 1).map(y => 2 * y)
```

and computes the value `List(20, 20, 20, 6)`. This is a functor block that "raises" computations to the `List` functor. Similar syntax exists in a number of languages and is called a **"for-comprehension"** or a "list comprehension" in Python, **"do-notation"** in Haskell, and **"computation expressions"** in F#. I use the name "functor block" in this book because it is shorter and more descriptive. (The type constructor used in a functor block needs to be at least a functor but does not have to be a monad.)

Kleisli function Also called a Kleisli morphism. A function with type signature $A \to M^B$ for some fixed monad M. More verbosely, "a morphism from the Kleisli category corresponding to the monad M". The standard monadic method $\text{pure}_M : A \to M^A$ has the type signature of a Kleisli function. The Kleisli product operation, $\diamond_M$, is a binary operation that combines two Kleisli functions (of types $A \to M^B$ and $B \to M^C$) into a new Kleisli function (of type $A \to M^C$).

Method This word is used in two ways: 1) A method$_1$ is a Scala function defined as a member of a typeclass. For example, `flatMap` is a method defined in the `Monad` typeclass. 2) A method$_2$ is a Scala function defined as a member of a data type declared as a Java-compatible `class` or `trait`. Trait methods$_2$ are necessary in Scala when implementing functions whose arguments have type parameters (because Scala function values defined via `val` cannot have type parameters). So, many typeclasses such as `Functor` or `Monad`, whose methods$_1$ require type parameters, will use Scala `traits` with methods$_2$ for their implementation. The same applies to type constructions with quantified types, such as the Church encoding.

Nameless function An expression of function type, representing a function. For example, `(x: Int) => x * 2`. Also known as function expression, function literal, anonymous function, closure, lambda-function, lambda-expression, or simply a "lambda".

Partial type-to-value function (PTVF) A function with a type parameter but defined only for a certain subset of types. In Scala, PTVFs are implemented via a typeclass constraint:

```
def double[T: Semigroup](t: T): T = implicitly[Semigroup[T]].combine(t, t)
```

This PTVF is defined only for types `T` for which a `Semigroup` typeclass instance is available.

Polynomial functor A type constructor built using disjunctions (sums), products (tuples), type parameters and fixed types. For example, in Scala, `type F[A] = Either[(Int, A), A]` is a polynomial functor with respect to the type parameter `A`, while `Int` is a fixed type (not a type parameter). Polynomial functors are also known as **algebraic data types**.

Product type A type representing several values given at once. In Scala, product types are the tuple types, for example `(Int, String)`, and case classes. Also known as **tuple** type, **struct** (in C and C++), and **record**.

Profunctor A type constructor whose type parameter occurs in both covariant and contravariant positions and satisfying the appropriate laws; see Section 6.4.1.

Type notation A mathematical notation for type expressions developed in this book for easier reasoning about types in functional programs. Disjunctive types are denoted by $+$, product types by $\times$, and function types by $\to$. The unit type is denoted by $\mathbb{1}$, and the void type by $\mathbb{0}$. The function arrow $\to$ groups weaker than $+$, which in turn groups weaker than $\times$. This means

$$Z + A \to Z + A \times A \quad \text{is the same as} \quad (Z + A) \to (Z + (A \times A)) \quad .$$

Type parameters are denoted by superscripts. As an example, the Scala definition

```
type F[A] = Either[(A, A => Option[Int]), String => List[A]]
```

is written in the type notation as

$$F^A \triangleq A \times (A \to \mathbb{1} + \text{Int}) + (\text{String} \to \text{List}^A) \quad .$$

Unfunctor A type constructor that cannot possibly be a functor, nor a contrafunctor, nor a profunctor. An example is a type constructor with explicitly indexed type parameters, such as $F^A \triangleq (A \times A)^{:F^{\text{Int}}} + (\text{Int} \times A)^{:F^{\mathbb{1}}}$. The Scala code for this type constructor is

```
sealed trait F[A]
final case class F1[A](x: A, y: A)   extends F[Int]
final case class F2[A](s: Int, t: A) extends F[Unit]
```

This can be seen as a **GADT** (generalized algebraic data type) that uses specific values of type parameters instead of the type parameter A in at least some of its case classes.

B.1 On the current misuse of the term "algebra"

This book avoids using the terms "algebra" or "algebraic" because these terms are too ambiguous. The functional programming community uses the word "algebra" in at least *four* incompatible ways.

Definition 0. In mathematics, an "algebra" is a vector space with multiplication and certain standard properties. For example, we need $1 * x = x$, the addition must be commutative, the multiplication must be distributive over addition, etc. The set of all 10×10 matrices with real coefficients is an "algebra" in this sense. These matrices form a 100-dimensional vector space, and they can be multiplied and added. This definition of "algebra" is not used in functional programming.

Definition 1. An "algebra" is a function with type signature $F^A \to A$, where F^A is some fixed functor. This definition comes from category theory, where such types are called **F-algebras**. There is no direct connection between this "algebra" and Definition 0, except when the functor F is defined by $F^A \triangleq A \times A$, and then a function of type $A \times A \to A$ may be interpreted as a "multiplication" operation (however, A is a type and not a vector space, and there are no distributivity or commutativity laws). It is better to call such functions "F-algebras", emphasizing that they characterize and depend on a chosen functor F. However, knowing how to reason about the theoretical properties of F-algebras does not help in practical programming, and so they are not used in this book.

Definition 2. Polynomial functors are often called "algebraic data types". However, they are not "algebraic" in the sense of Definitions 0 or 1. For example, consider the "algebraic data type" `Either[Option[A], Int]`, which is $F^A \triangleq 1 + A + \text{Int}$ in the type notation. The set of all values of the type F^A does not have the addition and multiplication operations required by the mathematical definition of "algebra". The type F^A may admit some binary or unary operations (e.g. that of a monoid), but these operations will not be commutative or distributive. Also, there is not necessarily a function with type $F^A \to A$, as required for Definition 1. Rather, the the word "algebra" refers to "school-level algebra" with polynomials, since these data types are built from sums and products of types. If the data contains a function type, e.g. `Option[Int => A]`, the type is no longer polynomial. So this book uses more precise terms: "polynomial type" and "exponential-polynomial type".

Definition 3. People talk about the "algebra" of properties of functions such as `map` or `flatMap`, meaning that these functions satisfy certain laws (e.g. the composition, naturality, or associativity laws). But these laws do not make functions `map` or `flatMap` into an "algebra" in the sense of Definition 0 or in the sense of Definition 1. There is also no relation to "algebraic data types" of Definition 2. So, this is a different usage of the word "algebra". However, there is no general "algebra of laws" that we can use; every derivation proceeds in a different way, specific to the laws being proved. In mathematics, "algebraic equations" are distinguished in another sense from "differential" or "integral" equations. But the laws in functional programming are always "algebraic: they are equations with compositions and applications of functions. We find that it is not useful to refer to "algebraic" laws in any of these two senses. This book talks about "equational laws" or just "laws".

Definition 4. The Church encoding of a free monad (also known as the "final tagless style of programming") is the type expression $\forall E^\bullet. (S^{E^\bullet} \rightsquigarrow E^\bullet) \to E^A$ that uses a higher-order type constructor S parameterized by a *type constructor* parameter E. The sub-expression $S^{E^\bullet} \rightsquigarrow E^\bullet$ can be viewed as an S-algebra in the category of type constructors (functors in a category of types without non-trivial morphisms). So, Definition 4 is related to Definition 1, with a specific choice of a category. However, the knowledge that $S^{E^\bullet} \rightsquigarrow E^\bullet$ is an S-algebra in the category of type constructors does not provide any help or additional insights for practical work with the Church encoding of a free monad.

The higher-order type constructor S is used to parameterize the effects described by a Church-encoded free monad, so this book calls it the "effect constructor".

So, it seems that the current usage of the word "algebra" in functional programming is both inconsistent and unhelpful to practitioners. In this book, the word "algebra" always denotes the branch of mathematics, as in "school-level algebra". Instead of "algebra" as in Definitions 1 to 4, this book talks about "F-algebras" with a specific functor F; "polynomial types" or "polynomial functors" or "exponential-polynomial functors" etc.; "equational laws"; and an "effect constructor" S.

E A humorous disclaimer

The following text is quoted in part from an anonymous source ("Project Guten Tag") dating back at least to 1997. The original text is no longer available on the Internet.

WARRANTO LIMITENSIS; DISCLAMATANTUS DAMAGENSIS
Solus exceptus "Rectum Replacator Refundiens" describitus ecci,

1. Projectus (etque nunquam partum quis hic etext remitibus cum PROJECT GUTEN TAG-tm identificator) disclamabat omni liabilitus tuus damagensis, pecuniensisque, includibantus pecunia legalitus, et

2. REMEDIA NEGLIGENTITIA NON HABET TUUS, WARRANTUS DESTRUCTIBUS CONTRACTUS NULLIBUS NI LIABILITUS SUMUS, INCLUTATIBUS NON LIMITATUS DESTRUCTIO DIRECTIBUS, CONSEQUENTIUS, PUNITIO, O INCIDENTUS, NON SUNT SI NOS NOTIFICAT VOBIS.

Sit discubriatus defectus en etextum sic entram diariam noventam recibidio, pecuniam tuum refundatorium receptorus posset, sic scribatis vendor. Sit veniabat medium physicalis, vobis idem reternat et replacator possit copius. Sit venitabat electronicabilis, sic viri datus chansus segundibus.

HIC ETEXT VENID "COMO-ASI". NIHIL WARRANTI NUNQUAM CLASSUM, EXPRESSITO NI IMPLICATO, LE MACCHEN COMO SI ETEXTO BENE SIT O IL MEDIO BENE SIT, INCLUTAT ET NON LIMITAT WARRANTI MERCATENSIS, APPROPRIATENSIS PURPOSEM.

Statuen varias non permitatent disclamabaris ni warranti implicatoren ni exclusioni limitatio damagaren consequentialis, ecco lo qua disclamatori exclusatorique non vobis applicant, et potat optia alia legali.

F GNU Free Documentation License

Version 1.2, November 2002

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

F.0.0 Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

F.0.1 Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section F.0.2.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

F.0.2 Copying in quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

F.0.3 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections F.0.1 and F.0.2 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section F.0.2 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section F.0.3. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warrany Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section F.0.3) to Preserve its Title (section F.0.0) will typically require changing the actual title.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) <year> <your name>. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being <list their titles>, with the Front-Cover Texts being <list>, and with the Back-Cover Texts being <list>.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Copyright

List of Tables

List of Figures

Index