The Science of Functional Programming

A Tutorial with Examples in Scala

by Sergei Winitzki, Ph.D.

draft version 0.2, May 8, 2019

Published by **lulu.com** 2019

Published and printed by **lulu.com**
ISBN XXXXXX

The PDF file `sofp.pdf` contains the full source code of the book as an "attachment". To extract the source file `soft-src.tar.bz2` to the current directory, run the command ‘`pdftk sofp.pdf unpack_files output .`‘

This book presents the theoretical knowledge that helps to write code in the functional programming paradigm. Detailed explanations and derivations are logically developed and accompanied by worked examples tested in the Scala interpreter as well as exercises. Readers need to have a working knowledge of basic Scala (e.g. be able to write code that reads a small text file and prints all word counts, sorted in descending order by count). Readers should also have a basic command of school-level mathematics; for example, be able to simplify the expressions such as $\frac{1}{x-2} - \frac{1}{x+2}$ 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 here at medium to advanced level. It requires a certain amount of mathematical experience and is not suitable for people unfamiliar with school-level algebra, or for people who are generally allergic to mathematics, or for people who are unwilling to learn new and difficult concepts through prolonged mental concentration.

The first two chapters are introductory and may be suitable for beginners in programming. Starting from the middle of chapter 3, the material becomes unsuitable for beginners.

The presentation in this book is self-contained. I define and explain all the required notations, concepts, and Scala language features. The emphasis is on clarity and understandability of all examples, mathematical notions, derivations, and code. I introduce a fair amount of *non-standard* terminology and notations because this allows me to to achieve a clearer and more logically coherent presentation of the material, especially for readers not familiar with today's theoretical literature.

The main intent 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.

Each mathematical construction is accompanied by code examples and unit tests to illustrate its usage and check correctness. For example, the equational laws for every standard typeclass (functor, applicative, monad, etc.) are first motivated heuristically before deriving a set of mathematical equations and formulating the laws in terms of axioms of a category. A software engineer needs to know only that part of theory which is developed in order to write some new code or generally to answer a question arising from programming practice. So I kept the theoretical material from category theory, type theory, and formal logic to the absolute minimum necessary. I do not pursue mathematical generalizations beyond practical relevance or immediate pedagogical usefulness.

This limits the scope of required mathematical knowledge to bare rudiments of category theory, type theory, and formal logic. For instance, I never mention "introduction/elimination rules", "strong normalization", "complete partial order domains", "adjoint functors", "pullbacks", "pushouts", "limits", "co-limits", "topoi", or even the word "algebra", because using these concepts will not help a functional programmer to write code. Instead, I focus on practically useful material – including some little-known but useful constructions such as the "partial type function", "filterable functor" and "applicative contrafunctor" typeclasses, and the "free pointed" monad.

Each new concept or technique is fully explained via solved examples and drilled via provided 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 (*).

In this book, I show code in Scala because this is what I am most familiar with. 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 is not specific to Scala or Haskell. A serious user of any other functional programming language will have to face the same questions and struggle with the same practical issues.

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.

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`. If we need to define a name for this value (e.g. to use it later), we use the "`val`" syntax and write

```
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 two things: (1) it can be computed (e.g. with the Scala interpreter) and yields a value, and (2) the code can be inserted anywhere as part of a larger expression; for example, we could write

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

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 quite 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. I denote "nameless" mathematical functions like this:

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

Then the mathematical notation for the nameless factorial function is

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

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 `fac` 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 argument and returns an integer result value. What is the value of the function `fac` *itself*? As we have just seen, the Scala interpreter

[1] In Scala, the two ASCII characters => and the single Unicode character ⇒ have the same meaning. I use the symbol ⇒ (pronounced "maps to") in this book. However, when doing calculations *by hand*, I tend to write → instead of ⇒ since it is faster. Several programming languages, such as OCaml and Haskell, use the symbols -> or the Unicode equivalent, →, for the function arrow.

prints `<function1>` as the "value" of `fac`. An alternative Scala interpreter called `ammonite`[2] prints something like this,

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

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

I usually imagine that a function value represents a block of compiled machine code, – code that will actually run and evaluate the function's body when the function is applied.

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

```
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 have written

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

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

[2] See https://ammonite.io/

```
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 **local value** definition.

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

Checking integers for being prime Define a function that takes an integer n and determines whether n is a prime number.

A simple mathematical formula for this function can be written as

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

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, $n \neq 0 \mod k$.

This mathematical expression is translated into the 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`. This is because 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 Scala code for `is_prime` differs from the mathematical formula (1.1) in two ways.

One difference is that the interval $[2, n-1]$ comes first in the Scala expression. To understand this, look at the ways Scala allows programmers to define syntax.

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 also widely used in mathematics, for instance 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 notation, such as

$$\forall k \in [2, n-1] \ : \ n \neq 0 \mod k \quad ,$$

does not seem to involve any nameless functions. Instead, the mathematical formula defines the symbol k that "goes over a certain set," as we might say. The symbol k is then used for writing the predicate $n \neq 0 \mod k$.

However, let us investigate the role of the symbol k more closely.

The symbol k is a mathematical variable that 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

$$\forall k \in [2, n-1] \ : \ n \neq 0 \mod k$$

gives two pieces of information: first, we are examining all values from the given range; second, we chose the name k for the values from the given range, and for each of those k we need to evaluate the expression $n \neq 0 \mod k$, which is a certain given *function of* k that returns a `Boolean` value. Translating the mathematical notation into code, it is therefore natural to use a nameless function of k,

$$k \Rightarrow n \neq 0 \mod k \quad ,$$

and to write Scala code that applies this nameless function to each element of the range $[2, n-1]$ and then requires 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 $k \Rightarrow n \neq 0 \mod k$ (which is a nameless function), the variable k is bound (because it is named and 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 x and write instead of Eq. (1.1) an equivalent definition

$$\text{is\_prime}\,(n) = \forall x \in [2, n-1] \ : \ n \neq 0 \mod x \quad ,$$

or in Scala code,

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

In the nameless function `k => n % k != 0`, the argument k may be renamed to x or to anything else, without changing the value of the entire program. No code outside this expression needs to be changed after renaming k to x. But the value `n` is defined outside and thus cannot be renamed locally (i.e. only within the sub-expression). If, for any reason, we wanted to rename `n` in the sub-expression `k => n % k != 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 $\forall k : p(k)$, $\exists k : p(k)$, $\sum_{k=a}^{b} f(k)$, $\int_0^1 k^2 dk$, $\lim_{n\to\infty} f(n)$, $\text{argmax}_k f(k)$, and so on. 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. $k \Rightarrow p(k)$ or $k \Rightarrow f(k)$ 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

$$\forall k \in [1, n] : p(k) \quad ,$$

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 the following simplification trick to this code. The nameless function $k \Rightarrow p(k)$ 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 $x \Rightarrow f(x)$ to just f is always possible for functions f of a single argument.[3]

[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 conversion for its argument `x`. In those cases, replacing the code `x => f(x)` by just `f` may fail to compile in Scala. But these complications do not arise when working with simple functions.

1.1.4 Aggregating data from lists

Consider the task of finding how many even numbers there are in a given list L of integers. For example, the list $[1, 2, 3, 4, 5]$ contains *two* even numbers: 2 and 4.

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

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

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

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.

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

```
def is_even(k: Int): Int = (k % 2) match {
  case 1 => 1
  case _ => 0
}
```

For such a simple computation, we could also write a shorter code and use 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, we used the nameless function $x \Rightarrow x^2 + 100x$ as the argument of `.map`. This function is repeatedly applied by `.map` to transform each of the values from a given list, creating a new list as a result.

It is equally possible to define the transforming function separately, give it a name, and then pass 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)
```

Usually, short and simple functions are defined inline, while longer functions are given a name and defined separately.

An infix method, such as `.map`, can be also used with a "dot-less" 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 $x \Rightarrow x^2 + 100x$, 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, I 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. The interpreter will automatically insert the visual continuation characters. (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 defined using `def`. It prints `(s: List[Int])Int` for the function type that one would normally write as `List[Int] => Int`.

1.1.5 Filtering

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` iff the predicate is true on all values in the list; the `.exists` method returns `true` iff 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, mathematical notation does not exist for operations such as "finding elements in a list", so we will focus on the Scala syntax for these operations.

The `.filter` method returns a *new 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 returns a *new list* that contains the initial portion of values from the original list, which is truncated when the predicate first returns `false`:

```
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). By **transformation** I mean a function from a list of values to another list of values; examples of transformation functions are `.filter` and `.map`. By **aggregation** I mean a function from a list of values to a *single* value; examples of aggregation functions are `.max` and `.sum`.

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

1.2 Solved examples: transformation and aggregation

1. Improve the code for `is_prime` by limiting the search to $k^2 \leq n$,

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

The solution is 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)
```

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

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

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

```
(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 $\cos k > 0$:

```
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)
```

4. Compute the average of a list of numbers of type `Double` (assuming that the list is not empty).

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

scala> average(List(1.0, 2.0, 3.0))
res4: Double = 2.0
```

5. Given n, compute the Wallis product truncated up to $\frac{2n}{2n+1}$:

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

We will define the helper function `wallis_frac(i)` that computes the i^{th} fraction. The method `.toDouble` is used to convert 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.
res5: Double = 3.9267453954401036E-5
scala> math.cos(wallis(100000)) // Should be even closer to 0.
res6: 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.

6. Another well-known series related to π is

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} \quad .$$

Define a function of n that computes a partial sum of this series until $k = n$. Compute the result for a large value of n and compare with the limit value.

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

scala> euler_series(100000)
res10: Double = 1.6449240668982423

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

scala> pi*pi/6
res9: Double = 1.6449340668482264
```

7. Check numerically the infinite product formula

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

We will compute the product up to $k = n$ for a fixed $x = 0.1$ and 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(100000, 0.1)
res11: Double = 0.9836317414461351

scala> math.sin(pi*0.1)/pi/0.1
res12: Double = 0.9836316430834658
```

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.

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

   Here is an example of using this function:

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

9. 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]]`.
   To "select the inner lists" satisfying a condition 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`. Then the solution is

   ```
   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) ))
   res13: List[List[Int]] = List(List(1, 2, 3), List(1, 2, 3, 4))
   ```

   The predicate in the argument of `.filter` is a nameless function `t => t.size >= 3` whose argument `t` is of type `List[Int]`. The Scala compiler deduces the type of `t` from the code; no other type would work with the way we use `.filter` on a *list of lists* of integers.

10. 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 $j^2 > 2k$.

```
scala> (1 to 10).toList.filter(k => (1 to k).
     |  filter(j => j*j > 2*k).size >= 3)
res14: List[Int] = List(6, 7, 8, 9, 10)
```

The argument of the outer `.filter` is a nameless function that itself uses another `.filter`. The inner expression,

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

computes the list of j's that satisfy the condition $j^2 > 2k$, and then computes the length of that list and imposes the requirement that there should be at least 3 such values of j. We can see how the Scala code closely follows the mathematical formulation of the problem.

1.3 Summary

The following table translates mathematical formulas into code.

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`
$\forall k$ such that $1 \leq k \leq n : p(k)$ holds	`(1 to n).forall(k ⇒ p(k))`
$\exists k$, $1 \leq k \leq n$ such that $p(k)$ holds	`(1 to n).exists(k ⇒ p(k))`
$\sum_{k \in S:\, p(k) \text{ holds}} f(k)$	`s.filter(p).map(f).sum`

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: $r_n = \sum_{k=0}^{n} s_k$.

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

These problems are not yet solvable because the required formulas involve *mathematical induction*, which we cannot yet 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 use mathematical induction to write the solution of Example 1 as a formula, but we have not yet seen how to implement that in Scala code.

Similarly, Example 2 defines a new list r from s by induction,

$$r_0 = s_0; \quad r_i = s_i + r_{i-1}, \forall i > 0 \quad .$$

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 solve these problems by translating mathematical induction into code using recursion.

1.4 Exercises

1. 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.

2. Machin's formula computes π more efficiently than the Wallis fraction:

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239} \quad ,$$
$$\arctan \frac{1}{n} = \frac{1}{n} - \frac{1}{3}\frac{1}{n^3} + \frac{1}{5}\frac{1}{n^5} - ...$$

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.

3. Using the function `is_prime`, check numerically the Euler product formula for the Riemann zeta function $\zeta(s)$ at $s = 4$, since it is known that $\zeta(4) = \frac{\pi^4}{90}$,

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

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

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

5. 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 $n \in [1, ..., 1000]$.

6. 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 5 using that function.

7. Define a function `sel_100` of type `List[List[Int]] => List[List[Int]]` that selects only those inner lists whose largest value is at least 100. Test with:

```
scala> sel_100( List( List(0, 100), List(60, 80), List(1000) ) )
res0: List[List[Int]] = List(List(0, 100), List(1000))
```

1.5 Discussion

1.5.1 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 thousands of years of human history. It took centuries to invent flexible and powerful notation such as $\forall k \in S : p(k)$ and to develop the corresponding rules of reasoning. Functional programmers are fortunate to have at their disposal such a superior reasoning tool.

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 the mathematical notation leaves out.) Just as in mathematics, large code expressions may be split into parts in a suitable way, so that the parts can be easily reused, 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 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 programming 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 principles in detail, developing them through intuition and practical coding tasks.

1.5.2 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 apply only a small number of the idioms of FP, such as the map/reduce operations. More advanced FP constructions are impractical in these languages because the corresponding code becomes too complicated to read, and mistakes are too easy to make, which negates the advantage of easier reasoning about the FP code.

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, PureScript, and Rust, had different design goals but still support enough FP features to be considered FP languages. I will be using Scala in this book, but exactly the same constructions could 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, PureScript, or Rust will not play a significant role.

1.5.3 The mathematical meaning of variables

The usage of variables in functional programming closely corresponds 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 a number 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 "modifying" 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 modify 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 $k \Rightarrow k^2 + k$. 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 want to "modify the value 4". In the same way, we cannot modify k.

So, a variable in mathematics does not actually vary *within the expression* where it is defined; in that expression, a variable is essentially a *named constant* value. 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 it is computed.

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

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

```
def f(x: Int) = x*x + x // Can't modify x here.
```

The *type* of each mathematical variable (say, integer, string, etc.) is also fixed in advance. In mathematics, each variable is a value from a specific set, known in advance (the set of all integers, the set of all strings, etc.). Mathematical formulas such as $x^2 + x$ do not express any "checking" that x is indeed an integer and not, say, a string, before starting to evaluate $x^2 + x$.

Functional programming adopts the same view: Each argument of each function must have a **type**, which represents *the set of possible allowed values* for that function argument. The programming language's compiler will automatically check all types of all arguments. 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. Types for "`val`"s are optional in Scala, for instance we could write

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

or more concisely,

```
val x = 123
```

because it is clear that this `x` is an integer. However, when types are complicated, it helps to write them out. The compiler will check that the types match correctly everywhere and give an error message if we use wrong types:

```
scala> val x: Int = "123" // A String instead of an Int.
<console>:11: error: type mismatch;
 found   : String("123")
 required: Int
       val x: Int = "123" // A String instead of an Int.
                    ^
```

1.5.4 Iteration without loops

Another distinctive feature of functional programming is the absence of explicit loops or repetition.

Iterative computations are ubiquitous in mathematics; one can see formulas such as

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

To compute these expressions, we need to iterate over various values of i, j, etc. And yet, no mathematics textbook ever mentions "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, 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 repetitive 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.

1.5.5 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 quite important in functional programming because, in particular, they allow programmers to write code more concisely and use a straightforward, consistent syntax.

Nameless functions have the property that their bound variables are invisible outside their 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 these formulas define the same function f, and that one may rename the integration variable at will.

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 $z \Rightarrow \frac{1}{1+z}$, and the integral rewritten as code such as

$$\text{integration}\,(0, x, g) \text{ where } g = \left(z \Rightarrow \frac{1}{1+z}\right) \quad .$$

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

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

In sums, 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:

$$\sum_{0}^{x} k \Rightarrow \frac{1}{1+k} \text{ instead of } \sum_{k=0}^{x} \frac{1}{1+k} \quad ,$$
$$\int_0^x z \Rightarrow \frac{1}{1+z} \text{ instead of } \int_0^x \frac{dz}{1+z} \quad .$$

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 a function as argument. The above summation may be written in a consistent and straightforward manner as a function:

$$\text{summation}\,(0, x, f) \text{ where } f = (y \Rightarrow \frac{1}{1+y}) \quad .$$

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

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

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

Numerical integration requires longer code, since the formulas are more com-

plicated. For instance, Simpson's rule can be written as

$$\begin{aligned}
\text{integration}\,(a,b,g) &= \frac{\delta}{3}\left(g(a)+g(b)+4s_1+2s_2\right) \quad , \\
n &= 2\left\lfloor\frac{b-a}{\varepsilon}\right\rfloor, \quad \delta = \frac{b-a}{n} \quad , \\
s_1 &= \sum_{i=1,3,\ldots,n-1} g(a+i\delta) \quad , \\
s_2 &= \sum_{i=2,4,\ldots,n-2} g(a+i\delta) \quad .
\end{aligned}$$

A straightforward translation of this formula into Scala is

```
def integration(a: Double, b: Double,
  g: Double => Double, eps: Double) = {
   // Define some helper values and functions.
   val n: Int = (math.round((b-a)/eps/2)*2).toInt
   val delta_x = (b - a) / n
   val g_i = (i: Int) => g(a + i*delta_x)
   val s1 = (1 to (n-1) by 2).map(g_i).sum
   val s2 = (2 to (n-2) by 2).map(g_i).sum
   // Compute the final result.
   delta_x / 3 * (g(a) + g(b) + 4*s1 + 2*s2)
}
scala> integration(0, 2, x => x*x*x, 0.001) // Exact answer is 4
res0: Double = 4.000000000000003

scala> integration(0, 7, x => x*x*x*x*x*x, 0.001) // Exact answer is 117649
res1: Double = 117649.00000000023
```

The entire code is one large *expression*, with a few sub-expressions defined for convenience as a few helper values and helper functions. In other words, this code is written in the FP paradigm.

1.5.6 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 most programming languages today, we can directly write arithmetical expressions such as `(x+123)*y/(2+x)`. Here, `x` and `y` are variables with names. Note, however, that the entire expression does not need to have a name. Parts of that

expression (such as `x+123` or `2+x`) also do not need to have separate names. It would be quite inconvenient if we *needed* to assign a name separately to each sub-expression. The code for `(x+123)*y/(2+x)` would then look like this:

```
r1 = 123
r2 = x + r1
r3 = r2 * y
r4 = 2
r5 = r4 + x
r6 = r3 / r5
return r6
```

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 gain productivity when their programming language supports expressions without names (which I call "nameless expressions").

This is also common practice in mathematics; names are assigned when needed, but many expressions remain nameless.

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

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

This is a nameless expression representing a dictionary. Without this construction, programmers have to write cumbersome, repetitive code that creates an initially empty named dictionary and then fills it step by step with values:

```
// Scala code for creating a dictionary:
val myMap = Map("a" -> 1, "b" -> 2, "c" -> 3)
// Java code:
Map<String, Int> myMap = new HashMap<String, Integer>() {{
  put("a", 1);
  put("b", 2);
  put("c", 3);
}}; // The shortest Java code for creating the same dictionary.
```

Nameless functions are useful for the same reason as nameless data values: they allow the programmer to build a larger program from simpler parts in a uniform and concise way.

1.5.7 Nameless functions: historical perspective

Nameless functions were first used in 1936 in a theoretical programming language called "λ-calculus". In that language,[4] 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 $x \Rightarrow x + 1$ 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, possibly because programming in the map/reduce style (which invites frequent use of nameless functions) turned out to be quite useful. Table 1.1 shows when nameless functions were introduced in each language.

What I call a "nameless function" is also elsewhere called anonymous function, function expression, function literal, closure, lambda function, lambda expression, or just a "lambda". I use the term "nameless function" in this book because it is the most descriptive and unambiguous both in speech and in writing.

[4] Although called a "calculus," it is in reality 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 3 of this book.

Language	**Year**	**Code for $k^{\mathbf{Int}} \Rightarrow k + k$**
λ-calculus	1936	$\lambda k.\ add\ k\ k$
typed λ-calculus	1940	$\lambda k : int.\ add\ k\ k$
LISP	1958	`(lambda (k) (+ k k))`
Standard ML	1973	`fn (k: int) => k+k`
OCaml	1985	`fun (k: int) -> k+k`
Haskell	1990	`\ (k: Int) -> k+k`
Oz	1991	`fun {$ K} K+K`
Ruby	1993	`lambda {|k| k+k }`
R	1993	`function(k) k+k`
Python	1994	`lambda k: k+k`
JavaScript	1995	`function(k) { return k+k; }`
Scala	2003	`(k: Int) => k+k`
F#	2005	`fun (k: int) -> k+k`
C++ 11	2011	`[] (int k) { return k+k; }`
Go	2012	`func(k int) { return k+k }`
Kotlin	2012	`{ k: Int -> k+k }`
Swift	2014	`{(k: int) -> int in return k+k}`
Java 8	2014	`(int k) -> k+k`
Rust	2015	`|k: i32| k+k`

Table 1.1: Nameless functions in various programming languages.

2 Mathematical formulas as code. II. Mathematical induction

In this chapter, we will see more functionality that Scala gives us to treat collections in a functional way – that is, in a way that is more suitable for mathematical thinking. In particular, the Scala standard library has methods for performing quite general iterative computations – including those that represent mathematical quantities defined by induction. Translating mathematical induction into code is the main topic of this chapter.

Before we begin that, we need to become fluent in using tuple types with the Scala collections library.

2.1 Tuple types

2.1.1 First examples

Many standard library methods in Scala require working 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 this tuple type.

We can also have a *triple* of values:

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

Pairs and triples are examples of tuples. A tuple can contain any number of values, which I call **parts** 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 this,

```
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 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
         ^
```

These **type errors** are detected at compile time, before any computations begin.

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

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

scala> c._1
res2: Boolean = true

scala> c._2
```

```
res3: (String, Int) = (abc,3)
```

To define functions whose arguments are tuples, we could use the syntax

```
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 to compute its 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 `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 on tuples

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

- destructuring definition: `val $pattern$ = ...`
- partial function: `case $pattern$ => ...`

An example of a **destructuring definition** is

```
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
```

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, and then assigns the names `x`, `y`, `z` to these parts. The types of the new values are also assigned automatically.

The left-hand side of the destructuring definition contains the 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 in defining functions on tuples:

```
def s(p: (Int, Int, Int)): Int = p match { case (x, y, z) => x + y + z }
scala> s(g)
res0: Int = 6
```

The **case expression** `case (x, y, z) => ...` performs pattern matching on the tuple argument `p`. The pattern matching process will "destructure" (i.e. decompose) the tuple and try to match it to the given pattern `(x, y, z)`. In this pattern, `x`, `y`, `z` are as yet undefined new variables, – that is, they are pattern variables. If the pattern matching succeeds, the pattern variables `x`, `y`, `z` are assigned their values, and the function body can proceed to perform its computation. In this example, the pattern variables `x`, `y`, `z` will be assigned values 1, 2, and 3, so the function returns 6 as its result value.

Pattern matching is especially convenient when working with nested tuples. Here is an example of such code:

```
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 is visually repeated in the pattern. It is easy to see that `x` and `y` become integers and `str` becomes a string after pattern matching. If we rewrite the same code using the tuple accessor methods instead of pattern matching, the could will look like this:

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

This code is more difficult to read: for example, it is not immediately clear what `p._2._1` refers to. It is also more difficult to maintain this code. To see this, suppose we want to change the type of the tuple `p` to, say, `((Int, String), Int)`. Then the code will have to be changed to

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

```
}
```

It is not immediately clear, without going through every accessor method, that the function `t3` computes the same expression as `t2`. In contrast, the code change is straightforward to make when using the pattern matching approach instead of the accessor methods:

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

Since the only change in the function body, compared to `t1`, is in the pattern matcher, 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. To express this, the following syntax is used,

```
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 values regardless of their types.

2.1.3 Using tuples with collections

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

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

We can have a list of tuples,

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

We can have 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. Note how the chosen names "fruit", "count" 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 Using dictionaries (Scala's `Map`s) as collections

In the Scala standard library, tuples are frequently used as types of intermediate values. For instance, 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**. They represent unknown types that are chosen later, when creating specific dictionary values.

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

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

This is equivalent to first creating a sequence of key/value *pairs* and then converting that sequence into a dictionary.

Pairs are used so often that the Scala library defines a special infix syntax for pairs, using 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 looks visually clearer:

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

A list of pairs can be converted to a dictionary using the method `.toMap`. The same method works for other one-dimensional collection types such as `Seq`, `Vector`, `Iterator`, or `Array`.

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 defined in the Scala library. All of these data structures are gathered under the common "sequence" type, `Seq`, and the Scala standard library may change the implementation of the "sequence" for reasons of performance.

The standard library has several useful methods that work with tuple types:

- `.map` for dictionaries
- `.filter` for dictionaries
- `.zip` and `.zipWithIndex`

Additionally, the following methods work with collections, including dictionaries:

- `.flatten` and `.flatMap`
- `.groupBy`
- `.sliding`

It is important to familiarize yourself with these methods, because it will make programming with sequences and dictionaries much easier. Let us now look at these methods one by one.

We have seen in Chapter 1 how the `.map` method works on sequences: the expression `xs.map(f)` applies a given function `f` to each element of the sequence `xs`, and gathers the results into 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)]`.[1] 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.

```
scala> 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 can first create a sequence of pairs and then convert it to a dictionary with the `.toMap` method:

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

The `.filter` method works similarly on dictionaries; the filtering predicate needs to be a function of type `(K, V) => Boolean`. For example:

```
scala> m1.filter { case (fruit, count) => count > 0 }.toMap
res2: scala.collection.immutable.Map[String,Int] = Map(apples -> 6, pears ->
    4)
```

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, I used the equivalent "dotless" infix syntax (`s zip t`) just to illustrate the flexibility of syntax conventions in Scala.

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

[1] A dictionary can be also converted into a sequence of pairs using the `.toSeq` method.

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 all inner sequences. In the Scala standard library, sequences are concatenated using the operation `++`, for example:

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

Applying the `.flatten` method is the same as inserting 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 doing first `.map` and then `.flatten`:

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

scala> List(1,2,3,4).map(n => (1 to n).toList)
res10: 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
res11: List[Int] = List(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)
```

It transforms a sequence by mapping each element to a potentially different number of new elements.

At first sight, it is probably not clear why `.flatMap` is at all useful. (Why not also combine `.filter` and `.flatten` into a `.flatFilter`, or combine `.zip` and `.flatten` into a

`.flatZip`?) However, later in this book I will show that the use of `.flatMap` (which is related to "monads") turns out to be 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: scala.collection.immutable.Map[Int,Seq[String]] = Map(1 ->
    List(yoghurt), 2 -> List(wombat, xanthan, zebra))
```

The `.groupBy` method has an argument, which is a function that computes a "key" out of each sequence element. The key can have an arbitrarily chosen type. (In the current example, the type of the key 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 unchanged from the order of 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: scala.collection.immutable.Map[Int,List[Int]] = Map(2 -> List(2, 5),
    1 -> List(1, 4), 0 -> List(3))
```

The `.sliding` method creates a sliding window of a given width and returns a sequence of nested sequences:

```
scala> (1 to 10).sliding(4).toList
res14: List[scala.collection.immutable.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, to find the sliding-window average:

```
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)
```

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. So a straightforward solution is

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

We could also use the solution of Exercise 2.1.5.1 that computed the cosine and the sine values. The code would then become longer. In particular, doing a `.map` on a sequence of pairs would be normally written using pattern matching for readability:

```
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 use `.map` on this sequence and compute the difference between the elements. So the full code is

```
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, we get a sequence of pairs:

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

Since the second sequence (`ps.tail`) is shorter, 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 we can 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 $c_i = \max(b_{i-k}, ..., b_{i+k})$.
Hint: use `.sliding`.

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))
```

What we need is to get, for each b_i, a list of nearby elements $(b_{i-k}, ..., b_{i+k})$. There are $2k+1$ such elements; so we need to use `.sliding(2*k+1)` to obtain a window of the required size. Now we need to compute the maximum of each of the nested lists. We can use the `.map` method on the outer list, with the `.max` method applied to the nested list. So the argument of the `.map` method must be the function `nested => nested.max`. The final code is

```
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 the syntax `(_.max)` is equivalent to `(q => q.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 set of *keys* for that dictionary. How can we create a list of pairs `(x, y)` of the form `List((1, 1), (1, 2), ..., (2, 1), (2,2), ...)`? We can start with a sequence of values of `x`, and for each `x` from that sequence, iterate over another sequence. 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 => (x, y)` that converts a value to a tuple,

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

Here the curly braces `{ y => (x, y) }` are used only for clarity; we could equivalently use parenteses, `(y => (x, y))`.

Using this `.map` operation, we obtain the code for a nested list of tuples:

```
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 close to what we need, except that the nested lists need to be concatenated into a single big 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))
```

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, which should map each *pair* of integers `(x, y)` to `x*y`. In Scala, a dictionary is usually created by applying `.toMap` to a sequence of pairs `(key, value)`. So we need to create a sequence of nested tuples of the form `((x, y), product)`. To achieve this, we use `.map` with a function that computes the product and creates a nested tuple:

```
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 converting each `y` to a tuple `(x, y)`, only to later convert each tuple `(x, y)` to a nested tuple `((x, y), x * y)`. Instead, let us perform the entire computation 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 the pair `(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, $\cos x_i$, $\sin x_i$.

Hint: use `.flatMap`, `.max`.

Solution: We will compute what is required if we take `.max` of the 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> List(0.1, 0.5, 0.9)
```

```
res0: List[Double] = List(0.1, 0.5, 0.9)

scala> res0.map { x => Seq(x, math.cos(x), math.sin(x)) }
res1: List[Seq[Double]] = List(List(0.1, 0.9950041652780258,
    0.09983341664682815), List(0.5, 0.8775825618903728, 0.479425538604203),
    List(0.9, 0.6216099682706644, 0.7833269096274834))
```

The list of values required for our task is almost the same as this `res1`, except that `res1` is a nested list. So we need to `.flatten` it:

```
scala> res1.flatten
res2: List[Double] = List(0.1, 0.9950041652780258, 0.09983341664682815, 0.5,
    0.8775825618903728, 0.479425538604203, 0.9, 0.6216099682706644,
    0.7833269096274834)
```

Now we just need to take the maximum of the resulting numbers:

```
scala> res2.max
res3: 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.

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 using the fixed type `String`. Test it with `Name = Boolean` and `Addr = Set[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 the final code is

```
def rev[Name, Addr](dict: Map[Name, Addr]): Map[Addr, Name] = {
```

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

Test with some values of type `Set[String]`:

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

scala> rev(d)
res4: Map[Set[String], Boolean] = Map(Set(x, y) -> true, Set(z, t) -> false)
```

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 types (`String`) into arbitrary type parameters in this function.

Example 2.1.5.10* Given a sequence `words:Seq[String]` of words, compute a sequence of pairs, of type `Seq[(Seq[String], Int)]`, where each inner sequence contains all the words with the same length, and the integer value shows the length. The resulting sequence must be ordered by increasing length of words. For example, the input `Seq("the", "food", "is", "good")` should produce the output

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

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

```
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 already 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) }
```

2.1.6 Exercises: Tuples and collections

Exercise 2.1.6.1 Find all pairs i, j within $(0, 1, ..., 9)$ such that $i + 4 * j > i * j$.
Hint: use `.flatMap` and `.filter`.

Exercise 2.1.6.2 Same task as in Exercise 2.1.6.1, but for i, j, k and the condition $i + 4 * j + 9 * k > i * j * k$.

Exercise 2.1.6.3 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 2.1.6.4 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, `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 2.1.6.5 Given `p: Seq[String]` and `q: Seq[Int]` of equal length, and assuming that elements of `q` do not repeat, compute a `Map[Int, String]` that maps numbers from `q` to their corresponding strings from `p`.

Exercise 2.1.6.6 Write the solution of Exercise 2.1.6.5 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 2.1.6.7 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`.
Hint: use `.sortBy`.

Exercise 2.1.6.8 Write the solution of Exercise 2.1.6.7 as a function with type parameter `S` instead of the fixed type `String`.

Exercise 2.1.6.9 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 2.1.6.10 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 `.sortBy`, `.take`.

Exercise 2.1.6.11 Given two sets `p`, `q` both of type `Set[Int]`, compute a `Set[(Int, Int)]` representing the Cartesian product of the sets `p` and `q` (that is, the set of all pairs `(x, y)` where `x` is from `p` and `y` is from `q`).

Implement this computation as a function with type parameters `I`, `J` instead of `Int`, and arguments of types `p:Set[I]` and `q:Set[J]`.

Hint: use `.flatMap` on sets.

Exercise* 2.1.6.12 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.

Hint: use `.flatMap`, `.groupBy`.

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 some tasks still remain unsolvable with these methods.

A simple computation that is impossible to do using `.map` is to compute the sum of a sequence of numbers. The standard library method `.sum` already does this;

but we cannot *implement* `.sum` *ourselves* by using `.map`, `.zip`, `.filter` or other operations on sequences. Indeed, these operations will compute *new sequences*, while we need to compute a single value (the sum of all elements) from a sequence.

We have seen a few standard 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 any other necessary 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`, `.zip` and other methods we have seen so far? In fact, the same task for *integer* numbers (not for floating-point numbers) is solvable using `.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
```

The 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 is computed separately and "zipped" with the sequence of digits d_k. However, for floating-point numbers, the sequence of powers of 10 is not determined in advance but depends on the position of the "dot" character. Methods such as `.map` and `.zip` cannot compute a sequence whose next elements are not known in advance but depend on previous elements via a custom function.

2.2.1 Inductive definitions of functions on sequences

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. a function `f:Seq[Int]=>Int`) by using mathematical induction, we need to specify two computations:

- (**Base case** of the induction.) The function `f` must return some value for an empty sequence, `Seq()`. Here we need to specify that value. If the function 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)`, where `x` is a given value.
- (**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 a one-element sequence of digits, `Seq(x)`, the result is `x`
- if `digitsToInt` is already known for a sequence `xs` of digits, and we have one more digit `x`, the result is

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

It is straightforward to 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 already 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 already computed then `(xs ++ Seq(x)).count(p) = xs.count(p) + c` where `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 Defining 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**.

We translate mathematical induction into code by first writing a condition to decide whether we are in the base case or in the inductive step. As an example, consider how we would 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 {
// To split s = Seq(x) ++ xs, compute x and xs.
  val x = s.head
  val xs = s.tail
  sum(prev) + next // Call sum() recursively.
}
```

In this example, we use the `if/then/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 the elements of a numerical sequence, the order of summation does not matter. However, the order of operations *will* matter for many other computational tasks.

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

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

In this example, it is important to split the sequence `s = xs ++ Seq(x)` into a concatenation of a subsequence `xs` and a one-element sequence `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 will always work, 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 the "tail recursion". The second problem is that much of the code is repeated in each inductively defined function: namely, 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 standard library already provides such functions. We will look at using them in Section 2.2.4.

2.2.3 Tail recursion

The code of `lengthS` will work unless the sequences become large. Consider an inductive definition of the `.length` method as a function:

```
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)
...
```

It is not a problem with insufficient memory: we *are* able to compute and hold in memory the 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/then/else`", about 100000 times. Each time, the sub-expression with nested `1+(1+(...))` functions gets larger. This intermediate sub-expression has 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. This sub-expression needs to be held in memory of the computer, in a special area 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 the program may crash.

A way to solve this problem 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 expression in the function body. Recursive calls cannot be placed inside other expressions.

As an example, we can rewrite the code of `lengthS` in this way:

```
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/then/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 do not occur within sub-expressions such as `1 + lengthT(...)`, unlike in the code of `lengthS`.

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/then/else`. This shows that the code of `lengthT()` is tail-recursive.

A tail-recursive function can have many `if/then/else` or `match/case` branches, 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. This is done with the `@tailrec` annotation. (If a function has a `@tailrec` annotation but 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 a short 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
```

I have written out the sub-expressions such as `1 + 2`, but they are computed each time *before* each recursive call 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`? The main difference between `lengthS` and `lengthT` is the additional argument, `res`, of `lengthT`, called the **accumulator argument**. Typically, this argument is equal to an intermediate result of the computation. The next intermediate result 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`.

Rewriting code to achieve tail recursion by adding an accumulator argument 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. The correct initial value for the accumulator is `0`, since in the base case (an empty sequence `s`) we need to return `0`. So, the complete tail-recursive implementation of `lengthT` requires us to define *two* functions: the tail-recursive `lengthT` and the "adapter" function that will set the initial value of the accumulator argument. To emphasize that `lengthT` is merely 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)
}
```

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)
```

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

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

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

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

Using a default argument value, we can define the tail-recursive 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 is its initial value, and how to define the induction step for the accumulator. In the example with the `lengthT` function, the accumulator trick works because of the following math-

ematical property of the expression being computed:

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

This property is called the **associativity law** of addition. Due to this law, we can rearrange the computation so that additions associate to the left. In code, it means that intermediate expressions are computed immediately before making the recursive calls; this avoids the growth of 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 in tail positions, – that is, outside all other sub-expressions. 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. In that code, the recursive call is within a sub-expression `digitsToInt(xs) * 10 + x`, which means that the code of `digitsToInt` is *not* tail-recursive. 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 fromDigitsT(s: Seq[Int], res: Int = 0): Int =
  // ‘res‘ is the accumulator.
  if (s == Seq()) res
  else fromDigitsT(s.tail, 10 * res + s.head)
```

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

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

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

Proof: We prove this by induction. The base case is a single-element sequence `s = Seq(x)` where the statement holds because

```
fromDigitsT(Seq(x), r) = 10 * r + x
  = fromDigits(Seq(x)) + r * math.pow(10, 1)
```

The induction step assumes `fromDigitsT(xs, r) = fromDigits(xs) + r` and needs to show that

```
fromDigitsT(Seq(x) ++ xs, r) = fromDigits(Seq(x) ++ xs) + r * math.pow(10,
    xs.length + 1)
```

while assuming that the property holds for sequences of the same length as `xs`. By expanding the code of `fromDigitsT`, we find

```
fromDigitsT(Seq(x) ++ xs, r) = fromDigitsT(xs, 10 * r + x)
```

By the inductive assumption, we know that

```
fromDigitsT(xs, 10 * r + x) = fromDigits(xs) + (10 * r + x) * math.pow(10,
    xs.length)
```

It remains to show that the right-hand side equals

```
fromDigits(Seq(x) ++ xs) + r * math.pow(10, xs.length + 1)
```

This will be proved if we show that

```
fromDigits(Seq(x) ++ xs) = fromDigits(xs) + x * math.pow(10, xs.length)
```

But this is an obvious property of decimal numbers: a digit x in front of n other digits has the value $x \cdot 10^n$. This concludes the proof of equivalence of `fromDigitsT` and `fromDigits`.

When it is impossible to transform recursive code into tail-recursive code via the accumulator trick, other methods can be used (for instance, the "continuation-passing transform" or "trampolines") that are beyond the scope of this chapter.

2.2.4 Implementing the general aggregation function

An "aggregation" converts a sequence of values into a single value. In general, the type of the result value may be different from the type of elements in the sequence. To describe this 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 fixed initial value.
- (Induction 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 then 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 general utility function, by making `b0` and `g` into parameters. A possible implementation is

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

However, this implementation is not tail-recursive. When applied to a sequence of, say, three elements, `Seq(x, y, z)`, the resulting expression is `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. (For this reason, the function is called a "left" fold.) In other words, 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 signature of `.foldLeft` in the Scala library. The resulting code is now 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)
```

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 may write very concise implementations of methods such as `.sum` or `.max`, and of many other similar aggregation methods. The method `leftFold` already implements all the code necessary to set up the base case and the induction 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` function for integer sequences:

```
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.

The type of the accumulator value can be different from the type of the sequence elements. As an example, here is an implementation of `count`:

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

The accumulator value is integer (type `Int`), while the sequence elements can have arbitrary type, parameterized by `A`. The aggregation function `leftFold` works in the same way for all types of accumulators and all types of sequence elements.

In Scala's standard library, the `.foldLeft` method has a slightly different type signature and, in particular, requires its arguments to be in separate argument groups. For example, the implementation of `sum` using our `leftFold` function is

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

With the Scala library's `.foldLeft` method, the code is written as

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

This syntax makes it more convenient to write longer function bodies, since the function argument of `.foldLeft` is separated from other arguments by curly braces. We will use the standard `.foldLeft` method now; the `leftFold` function was implemented only as an illustration.

The method `.foldLeft` is available in the Scala standard library for all collections (including dictionaries). It is safe to use `.foldLeft`, in the sense that no stack overflows will occur even for very large sequences.

The Scala library contains several other methods similar to `.foldLeft`, such as `.foldRight`, `.fold`, `.reduceLeft`, `.reduce` and so on.[2]

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 `Integer.MIN_VALUE` for empty sequences.

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

- (Base case.) For an empty sequence, return `Integer.MIN_VALUE`.
- (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(Integer.MIN_VALUE) { (b, x) => math.max(b, x) }
```

If we are sure that the function is never 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]`.

[2] It is not the purpose of this book to explain the use of all these methods in detail; this is done in the Scala library documentation. However, I note that in Scala up to version 2.12, the method `.foldRight` is not tail-recursive and so cannot be used for large sequences.

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 translated into code:

```
def digitsToInt(d: Seq[Int]): Int = d.foldLeft(0){ (i, x) => i*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. you must use a single `xs.foldLeft` function call.

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

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

However, this expression traverses `xs` at least three times, since each method call, such as `xs.min` or `xs.max`, traverses `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 the return value of that function? We need to accumulate intermediate values of *all four* numbers in a tuple. So the required type of the accumulator is `(Double, Double, Double, Double)`. To avoid repeating this 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 triple(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> triple(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]`. For example,

```
digitsToDouble(Seq('3', '4', '.', '2', '5')) = 34.25
```

Assume that all characters are either digits or the dot character (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 to "remember" anything in a folding aggregation 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 number constructed so far and a `Boolean` flag showing whether we have already seen the dot character.

To visualize what `digitsToDouble` must do, let us consider how the evaluation of `digitsToDouble(Seq('3', '4', '.', '2', '5'))` should go. We will write down what the resulting number should be at each iteration. This will hopefully help us figure out what the accumulator and the updater function must be:

Current digit c	**Previous number** n	**New number** n'
'3'	0.0	3.0
'4'	3.0	34.0
'.'	34.0	34.0
'2'	34.0	34.2
'5'	34.2	34.25

Until the dot character is found, the updater function multiplies the previous number by 10 and adds the current digit. After the dot character, the updater function must divide the current digit by a factor that represents increasing powers of 10, and add the result to the previous number. In other words, the update computation $n' = g(n, c)$ is defined by these formulas,

- before the dot character: $g(n, c) = n * 10 + c$
- after the dot character: $g(n, c) = n + c / f$ where f is 10, 100, 1000, etc., for each subsequent 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.

Let us choose the accumulator type as a triple `(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 is to 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)
    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 value, `n`, is of interest to us. We could extract the first element using the accessor method `._1`, but the code is more readable if we show what the different parts of the tuple mean. In Scala syntax, the underscore symbol can be used to fill the unused places in a pattern matching expression.

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 `.map` method should build a new sequence by applying a given function 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 ++ Seq(f(x)) }
```

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 each pair of adjacent elements. If the initial sequence has an odd number of elements, a given default value of type `A` is used:

```
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. However, we iterate over elements of the input sequence one by one. So, we will be able to add a new pair 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 contains the previous element for odd-numbered iterations and will be empty for even-numbered iterations. The accumulator is initially 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. The code calls `.foldLeft` and then needs to do 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). 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)
  // This is the last expression that will be returned.
  remainder match {
    case Seq() => result
    case Seq(x) => result ++ Seq((x, default))
  }
}
```

This code shows examples of a partial function that is used safely. Here, the partial function is the expression

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

This expression works only when `xyz` is an empty sequence or a one-element sequence, and fails for longer sequences. The code is safe as long as this expression is always used on sequences of length at most 1, as it is done in the code of `toPairs`.

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 Implement the `zipWithIndex` method for sequences by using `.foldLeft`. 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 method 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 batching[A](xs: Seq[A], size: Int): Seq[Seq[A]] = ???
```

```
scala> batching(Seq(1, 2, 3, 4, 5, 6, 7), 3)
res0: Seq[Seq[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7))
```

Exercise 2.2.6.6* Implement `groupBy` using `.foldLeft`. The required type signature and a sample 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(0 -> List(2, 4), 1 -> List(1, 3, 5))
```

Hint: The accumulator should be of type `Map[K, Seq[A]]`. Use the method `.updated` that produces a new dictionary, either with a new key or with a new value for an existing key:

```
scala> Map("a" -> 1, "b" -> 2).updated("c", 3)
res0: Map[String,Int] = Map(a -> 1, b -> 2, c -> 3)

scala> Map("a" -> 1, "b" -> 2).updated("a", 4)
res1: Map[String,Int] = Map(a -> 4, 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 of a given integer:

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

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

We cannot implement this function using `.map`, `.zip`, or even with `.foldLeft`: these methods work on a given sequence that we must *already have*. We could try making a new sequence using a function such as `(1 to n)` from the Scala library, but we do not know in advance how long the resulting sequence must be. The length of the resulting sequence is determined by a condition that we cannot easily evaluate in advance.

One way of implementing a general "unfolding" operation is first to build a sequence whose length is not determined in advance. This sort of sequence is

realized by a function, called an **iterator**, that computes each following element of a sequence whenever necessary. The unfolding operation will call the iterator until a certain given condition holds. We can use the function `.takeWhile` on an iterator, to stop the sequence when a certain condition holds. In this way, we can generate a sequence according to the given requirements, even if we cannot know the length of the sequence in advance. The length of the sequence will be determined later, as we compute each element.

In Scala, the standard iterator function is `Iterator.iterate`; this function has two arguments, the initial value and a function that computes the next value out of the previous one:

```
scala> val iter1 = Iterator.iterate(1){x => x + 10}
iter1: Iterator[Int] = non-empty iterator
```

The iterator is ready to start computing the next elements of the sequence (but it has not yet computed any). In order to see these elements, we first need to stop the sequence at a finite size and then convert the result to a list:

```
scala> iter1.take(10).toList
res0: List[Int] = List(1, 11, 21, 31, 41, 51, 61, 71, 81, 91)
```

It is a mistake to evaluate `iter1.toList` without limiting the size of the sequence. If we do that, the computer will keep producing more and more elements of the list, until it runs out of memory.

To obtain the solution for `digitsOf`, let us first formulate this function mathematically. To compute the sequence of digits of, say, 2405, we will compute a sequence of pairs (n_k, d_k) where n_k are intermediate numbers (such as 2405, 240, etc.), and d_k are the consecutive digits. The sequence (n_k, d_k) is defined using mathematical induction:

- Base case: $(n_0, d_0) = (n, 0)$ where n is the given initial integer.
- Inductive step: $(n_{k+1}, d_{k+1}) = \left(\left\lfloor \frac{n_k}{10} \right\rfloor, n \bmod 10\right)$ for $k = 1, 2, \ldots$

Here $\left\lfloor \frac{n_k}{10} \right\rfloor$ is the mathematical notation for the *integer* division by 10 (whose result is an integer).

If we keep evaluating the next values in the sequence (n_k, d_k), eventually both numbers will become 0. At this point, we need to stop the sequence. Let us trace the evaluation of this sequence for $n = 2405$:

$k =$	0	1	2	3	4	5	6
$n_k =$	2405	240	24	2	0	0	0
$d_k =$	0	5	0	4	2	0	0

The sequence (n_k, d_k) will generate zeros forever after $k = 4$. It is clear that the useful part of the sequence (n_k, d_k) is before it becomes all zeros. In this example, the sequence (n_k, d_k) needs to be stopped after $k = 4$. We need to obtain the sequence of digits as `Seq(2, 4, 0, 5)`. What we have as d_k is `Seq(0, 5, 0, 4, 2)`. So, it remains to exclude the first element (i.e. to start the sequence at $k = 1$) and to reverse the sequence (for this, Scala has the standard method `.reverse`). The code is

```
def digitsOf(x: Int): Seq[Int] = if (x == 0) Seq(0) else {
  Iterator.iterate((x,0)){ case (n, d) => (n / 10, n % 10) }
    .takeWhile { case (n, d) => ! (n == 0 && d == 0) }
    .map { case (n, d) => d }.toList.tail.reverse
}
```

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

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

This shows that `Iterator.iterate` implements a general case where a sequence is defined by 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 flexible way of generating sequences whose length is not determined in advance.

2.4 Transforming a sequence into another sequence

2.5 Summary

What problems can we solve now?

- Compute mathematical expressions involving arbitrary recursion
- Use the accumulator trick to enforce tail recursion
- Use arbitrary inductive (i.e. recursive) formulas to:
  - convert sequences to numbers (“aggregation”)

  - create new sequences from scratch
  - transform existing sequences

What problems are not solved with these tools?

- Compute non-tail recursive functions without expression overflow
  - The accumulator trick does not always work

2.5.1 Worked examples

2.5.2 Exercises

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 partial function is only well-defined on a subset of possible argument values.

A simple example of a partial function in Scala is the `.max` method on sequences: it is only defined on non-empty sequences. Trying to evaluate it on an empty sequence generates a run-time 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 happens during evaluation of the program and may crash the entire program. It is desirable to write code that does not generate such errors. For instance, if we know that a certain sequence is always non-empty, it is safe to call `.max` on it.

Functions that use pattern matching are sometimes partial because pattern matching may fail. In most cases, programs can be written to avoid the possibility of match errors.

If the pattern matching 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. An example of a potentially fallible pattern matching 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
```

In this case, 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. Therefore, `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 the pattern variables will always match whatever values the tuple has. On the other hand, the compiler will generate a compile-time error if a function is applied to a tuple of incorrect type. For this reason, pattern matching will a pattern such as `(x, y, z)` is **infallible** (never fails).

2.6.2 Scope of pattern matching variables

Pattern matching constructions introduce locally scoped variables. Consider this code,

```
def f(x: (Int, Int)): Int = x match {
  case (x, y) => x + y
}
```

2.6.3 Case classes as “named tuple” types

3 The formal logic of types. I. Higher-order functions

3.1 Types of higher-order functions

3.1.1 Curried functions

Consider a function with type signature `Int => (Int => Int)`. This is a function that takes an integer and returns a *function* that again takes an integer and then returns an integer. So, we obtain an integer result only after we apply the function to *two* integer values, one after another. This is, in a sense, equivalent to a function having two arguments, except the application of the function needs to be done in two steps, applying it to one argument at a time. Functions of this sort are called **curried** functions.

One way of defining such a function is

```
def f0(x: Int): Int => Int = { y => x - y }
```

The function takes an integer argument `x` and returns the expression `y => x - y`, which is a function of type `Int => Int`. So the type of `f0` is written as `Int => (Int => Int)`.

To use `f0`, we must apply it to an integer value. The result is a value of function type:

```
scala> val r = f0(20)
```

The value r can be now applied to another integer argument:

```
scala> r(4)
```

In Scala, `Int => Int => Int` means the same as `Int => (Int => Int)`, and `x => y => x - y` means the same as `x => (y => x - y)`. In other words, the function symbol `=>` associates to the right. Thus, the type signature of `f0` may be equivalently written

as `Int => Int => Int`.

An equivalent way of defining a function with the same type signature is

```
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. Such a function will have a different type signature, for instance

```
def f2(x: Int, y: Int): Int = x - y
```

has type signature `(Int, Int) => Int`.

The syntax for calling the functions `f1` and `f2` is different:

```
scala> f1(20)(4)
scala> f2(20, 4)
```

The main difference between the usage of `f1` and `f2` is that `f2` must be applied *at once* to both arguments, while `f1(20)` can be evaluated separately, – that is, applied only to its first argument, `20`. The result of `f1(20)` is a *function* that can be later applied to another argument:

```
scala> val r = f1(20)
scala> r(4)
```

Applying a curried function to some but not all of possible arguments is called **partial application**.

More generally, a curried function may have a type signature of the form `A => B => C => P => Z`, where `A`, `B`, `C`, ..., `Z` are some types. I prefer to think about this kind of type signature as having arguments of types `A`, `B`, ..., `P`, called the **curried arguments** of the function, and the "final" return value of type `Z`. The "final" return value of the function is returned after supplying all arguments.

A function with this type signature is, in a sense, equivalent to an **uncurried** function with type signature `(A,B,C,...,P) => Z`. The uncurried function takes all arguments at once. The equivalence of curried and uncurried functions is not *equality* – these functions are *different*; but one of them can be easily reconstructed from the other if necessary. One says that a curried function is **isomorphic** or **equivalent** to an uncurried function.

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 sometimes harder to read.

In the syntax used e.g. in OCaml and Haskell, a curried function such as `f2` is applied to its arguments as `f2 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 significantly harder to read, compared with the syntax where commas are used to separate the arguments. (Consider, for example, the Haskell expression `f2 (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 give a curried argument using the curly brace syntax. Compare the two definitions of the function `summation` described earlier:

```
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
summation1(1, 10, x => x*x*x + 2*x)
summation2(1, 10) { x => x*x*x + 2*x }
```

The code that calls `summation2` may be 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, since the Scala curly braces syntax allows function bodies to contain their own local definitions (`val` or `def`).

A feature of Scala is the "dotless" method syntax: for example, `xs map f` is equivalent to `xs.map(f)`. The "dotless" syntax works only for infix methods, such as `.map`, defined on specific types such as `Seq`. Do not confuse Scala's "dotless" method syntax with the short function application syntax such as `fmap f xs`, used in Haskell and some other languages.

3.1.2 Calculations with nameless functions

We now need to gain experience working with nameless 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 $x \Rightarrow x + 10$ 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,

$$(x \Rightarrow x + 10)(2) = 2 + 10 = 12 \quad .$$

Nameless functions are *values* and 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 $f \Rightarrow f(2)$ to a nameless function $x \Rightarrow x + 4$:

$$(f \Rightarrow f(2))\,(x \Rightarrow x + 4) = (x \Rightarrow x + 4)(2) = 6 \quad .$$

In the nameless function $f \Rightarrow f(2)$, the argument f has to be itself a function, otherwise the expression $f(2)$ would make no sense. In this example, f must have type `Int => Int`.

There are some standard conventions for reducing the number of parentheses when writing expressions involving nameless functions, especially curried functions:

- Function expressions group everything to the right:
  $x \Rightarrow y \Rightarrow z \Rightarrow e$ means the same as $x \Rightarrow (y \Rightarrow (z \Rightarrow e))$.
- Function applications group everything to the left:
  $f(x)(y)(z)$ means $((f(x))(y))(z)$.
- Function applications group stronger than infix operations:
  $x + f(y)$ means $x + (f(y))$, just like in mathematics.

To specify the type of the argument, I will use a colon in the superscript, for example: $x^{:\mathrm{Int}} \Rightarrow x + 2$.

The convention of grouping functions to the right reduces the number of parentheses for curried function types used most often. It is rare to find use for function types such as $((a \Rightarrow b) \Rightarrow c) \Rightarrow d$ that require many parentheses with this convention.

Here are some more examples of performing function applications symbolically. I will omit types for brevity, since every non-function value is of type `Int` in

these examples.

$$\left(x^{:\mathrm{Int}} \Rightarrow x * 2\right)(10) = 10 * 2 = 20 \quad .$$
$$(p \Rightarrow z \Rightarrow z * p)\,(t) = (z \Rightarrow z * t) \quad .$$
$$(p \Rightarrow z \Rightarrow z * p)\,(t)(4) = (z \Rightarrow z * t)(4) = 4 * t \quad .$$

Some results of these computation are integer values such as 20; in other cases, results are *function values* such as $z \Rightarrow z * t$.

In the following examples, some function arguments are themselves functions:

$$(f \Rightarrow p \Rightarrow f(p))\,(g \Rightarrow g(2)) = (p \Rightarrow p(2)) \quad .$$
$$\begin{aligned}(f \Rightarrow p \Rightarrow f(p))\,(g \Rightarrow g(2))\,(x \Rightarrow x + 4) &= (p \Rightarrow p(2))\,(x \Rightarrow x + 4) \\ &= 2 + 4 = 6 \quad .\end{aligned}$$

Here I have been performing calculations step by step, as usual in mathematics. A Scala program is evaluated in a similar way at run time.

3.1.3 Short syntax for function applications

In mathematics, function applications are sometimes written without parentheses, for instance $\cos x$ or $\arg z$. There are also cases where formulas such as $\sin 2x = 2 \sin x \cos x$ imply parentheses as $\sin(2x) = 2 \cdot \sin(x) \cdot \cos(x)$.

Many programming languages (such as ML, OCaml, F#, Haskell, Elm, PureScript) have adopted this "short syntax", in which parentheses are optional for function arguments. The result is a concise notation where `f x` means the same as `f(x)`. Parentheses are still used where necessary to avoid ambiguity or for readability.[1]

The conventions for nameless functions in the short syntax become:

- Function expressions group everything to the right:
  $x \Rightarrow y \Rightarrow z \Rightarrow e$ means $x \Rightarrow (y \Rightarrow (z \Rightarrow e))$.
- Function applications group everything to the left:
  $f\ x\ y\ z$ means $((f\ x)\ y)\ z$.

[1] The no-parentheses syntax is also used in Unix shell commands, for example `cp file1 file2`, as well as in the programming language Tcl. In LISP and Scheme, each function application is enclosed in parentheses but the arguments are separated by spaces, for example `(+ 1 2 3)`.

- Function applications group stronger than infix operations: $x + f\ y$ means $x + (f\ y)$, just like in mathematics $x + \cos y$ groups $\cos y$ stronger than the infix "+" operation.

So, $x \Rightarrow y \Rightarrow a\,b\,c + p\,q$ means $x \Rightarrow (y \Rightarrow ((a\,b)\ c) + (p\,q))$. When this notation becomes hard to read correctly, one needs to add parentheses, e.g. to write $f(x \Rightarrow g\ h)$ instead of $f\ x \Rightarrow g\ h$.

In this book, I will sometimes use this "short syntax" when reasoning about code. Scala does not support the short syntax; in Scala, parentheses need to be put around every curried argument. The infix method syntax such as `List(1,2,3) map func1` does not work with curried functions in Scala.

3.1.4 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 function arrow, the function is called a **higher-order** function. A higher-order function takes a function as argument and/or returns a function as its result value.

Examples:

```
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 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
```

The Scala library defines methods to transform between curried and uncurried functions:

```
scala> def f2u(x: Int, z: Int): Int = z + x
scala> val f2c = (f2u _).curried
scala> val f2u1 = Function.uncurried(f2c)
```

The syntax `(f2u _)` is used 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 methods `.curried` and `.uncurried` can be easily implemented in Scala code, as we will see in the worked examples.

Unlike `f2`, the function `f3` cannot be converted to a non-higher-order function because `f3` has an argument of function type, rather than a return value of function type. Converting to an uncurried form cannot eliminate an argument of function type.

3.1.5 Worked examples: higher-order functions

1. Using both `def` and `val`, define a function that...
   a) ...adds 20 to its integer argument.

      ```
      def fa(i: Int): Int = i + 20
      val fa_v: (Int ⇒ Int) = k ⇒ k + 20
      ```

      It is not necessary to specify the type of the argument `k` because we already fully specified the type `(Int => Int)` of `fa_v`. The parentheses around the type of `fa_v` are optional, I added them for clarity.

   b) ...takes an integer x, and returns a *function* that adds x to *its* argument.

      ```
      def fb(x: Int): (Int ⇒ Int) = k ⇒ k + x
      val fb_v: (Int ⇒ Int ⇒ Int) = x ⇒ k ⇒ k + x
      def fb_v2(x: Int)(k: Int): Int = k + x
      ```

      Since functions are values, we can directly return new functions. When defining the right-hand sides as function expressions in `fb` and `fb_v`, it is not necessary to specify the type of the arguments `x` and `k` because we already fully specified the type signatures of `fb` and `fb_v`. The last version, `fb_v2`, may be easier to read and is equivalent to `fb_v`.

c) ...takes an integer x and returns `true` iff $x + 1$ is a prime. Use the function `is_prime` defined previously.

```
def fc(x: Int): Boolean = is_prime(x + 1)
val fc_v: (Int ⇒ Boolean) = x ⇒ is_prime(x + 1)
```

d) ...returns its integer argument unchanged. (This is called the **identity function** for integer type.)

```
def fd(i: Int): Int = i
val fd_v: (Int ⇒ Int) = k ⇒ k
```

e) ...takes x and always returns 123, ignoring its argument x. (This is called a **constant function**.)

```
def fe(x: Int): Int = 123
val fe_v: (Int ⇒ Int) = x ⇒ 123
```

To emphasize the fact that the argument x is ignored, use the special syntax where x is replaced by the underscore:

```
val fe_v1: (Int ⇒ Int) = _ ⇒ 123
```

f) ...takes x and returns a constant function that always returns the fixed value x. (This is called the **constant combinator**.)

```
def ff(x: Int): Int ⇒ Int = _ ⇒ x
val ff_v: (Int ⇒ Int ⇒ Int) = x ⇒ _ ⇒ x
def ff_v2(x: Int)(y: Int): Int = x
```

The syntax of `ff_v2` may be easier to read, but then we cannot omit the name `y` for the unused argument.

2. Define a function `comp` that takes two functions f :`Int ⇒ Double` and g :`Double ⇒ String` as arguments, and returns a new function that computes $g(f(x))$. What is the type of the function `comp`?

```
def comp(f: Int ⇒ Double, g: Double ⇒ String): (Int ⇒ String) =
  x ⇒ g(f(x))
scala> val f: Int ⇒ Double = x ⇒ 5.67 + x
scala> val g: Double ⇒ String = x ⇒ f"x=%3.2f"
scala> val h = comp(f, g)
scala> h(10)
```

The function `comp` has two arguments, of types `Int ⇒ Double` and `Double ⇒ String`. The result value of `comp` is of type `Int ⇒ String`, because `comp` returns a new function that takes an argument x of type `Int` and returns a `String`. So the full type signature of the function `comp` is written as

```
/// (Int ⇒ Double, Double ⇒ String) ⇒ (Int ⇒ String)
```

This is an example of a function that both takes other functions as arguments *and* returns a new function.

3. Define a function `uncurry2` that takes a curried function of type `Int => Int => Int` and returns an uncurried equivalent function of type `(Int, Int) => Int`.

```
def uncurry2(f: Int ⇒ Int ⇒ Int): (Int, Int) ⇒ Int =
  (x, y) ⇒ f(x)(y)
```

3.2 Discussion

3.2.1 Scope of bound variables

A bound variable is invisible outside the scope of the expression (often called **local scope** whenever it is clear which expression is being considered). This is why bound variables may be renamed at will: no outside code could possibly use them and depend on their values. However, outside code may define variables that (by chance or by mistake) have the same name as a bound variable inside the scope.

Consider this example from calculus: In the integral

$$f(x) = \int_0^x \frac{dx}{1+x} \quad ,$$

a bound variable named x is defined in *two* local scopes: in the scope of f and in the scope of the nameless function $x \Rightarrow \frac{1}{1+x}$. 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 **shadowed**, i.e. hidden, by the definition of the inner x. For this reason, mathematicians expect that evaluating $f(10)$ will give

$$f(10) = \int_0^{10} \frac{dx}{1+x} \quad ,$$

rather than $\int_0^{10} \frac{dx}{1+10}$, because the outer definition $x = 10$ is shadowed, within the expression $\frac{1}{1+x}$, by the closer definition of x in the local scope of $x \Rightarrow \frac{1}{1+x}$.

Since this is the prevailing mathematical convention, the same convention is adopted in FP. A variable defined in a local scope (i.e. a bound variable) is invisible outside that scope but will shadow any outside definitions of a variable with the same name.

It is better to avoid name shadowing, because it usually decreases the clarity of code and thus invites errors. Consider this function,

$$x \Rightarrow x \Rightarrow x \quad .$$

Let us decipher this confusing syntax. The symbol $\Rightarrow$ associates to the right, so $x \Rightarrow x \Rightarrow x$ is the same as $x \Rightarrow (x \Rightarrow x)$. So, it is a function that takes x and returns $x \Rightarrow x$. Since the returned nameless function, $(x \Rightarrow x)$, may be renamed to $(y \Rightarrow y)$ without changing its value, we can rewrite the code to

$$x \Rightarrow (y \Rightarrow y) \quad .$$

It is now easier to understand this code and reason about it. For instance, it becomes clear that this function actually ignores its argument x.

3.3 Exercises

1. Define a function of type `Int => List[List[Int]] => List[List[Int]]` similar to Exercise 7 except that the hard-coded number 100 must be a *curried* first argument. Implement Exercise 7 using this function.

2. Define a function q that takes a function `f: Int => Int` as its argument, and returns a new function that computes $f(f(f(x)))$. What is the required type of the function q?

3. Define a function `curry2` that takes an uncurried function of type `(Int, Int) => Int` and returns a curried equivalent function of type `Int => Int => Int`.

4 The formal logic of types. II. Disjunctive types

4.0.1 Discussion

5 The formal logic of types. III. The Curry-Howard correspondence

5.0.1 Discussion

6 Functors

6.1 Discussion

6.2 Practical use

6.3 Laws and structure

7 Type-level functions and typeclasses

7.1 Combining typeclasses

7.2 Inheritance

7.3 Functional dependencies

7.4 Discussion

8 Computations in functor blocks. I. Filterable functors

8.1 Practical use

8.1.1 Discussion

8.2 Laws and structure

8.2.1 Discussion

9 Computations in functor blocks. II. Semimonads and monads

9.1 Practical use

9.1.1 Discussion

9.2 Laws and structure

9.2.1 Discussion

10 Applicative functors, contrafunctors, and profunctors

10.1 Practical use

10.1.1 Discussion

10.2 Laws and structure

11 Traversable functors and profunctors

11.1 Discussion

12 “Free” type constructions

12.1 Discussion

13 Computations in functor blocks. III. Monad transformers

13.1 Practical use

13.2 Laws and structure

A monad transformer $T_L^{M,A}$ is a type constructor with a type parameter A and a monad parameter M, such that the following five laws hold:

1. **Monad construction law**: $T_L^{M,\bullet}$ is a lawful monad for any monad M. For instance, the transformed monad $T_L^{M,\bullet}$ has methods pu_T and ftn_T that satisfy the monad laws.

2. **Identity law**: $T_L^{\text{Id},\bullet} \cong L^\bullet$ via a monadic isomorphism, where Id is the identity monad, $\text{Id}^A \triangleq A$.

3. **Lifting law**: For any monad M, the function $\text{lift} : M^A \Rightarrow T_L^{M,A}$ is a monadic morphism. (In a shorter notation, $\text{lift} : M \rightsquigarrow T_L^M$.)

4. **Runner law**: For any monads M, N and any monadic morphism $\phi : M \rightsquigarrow N$, the runner $\text{mrun}\,\phi : T_L^M \rightsquigarrow T_L^N$ is a monadic morphism. Moreover, the function `mrun` is a lawful lifting of monadic morphisms from $M \rightsquigarrow N$ to $T_L^M \rightsquigarrow T_L^N$:

$$\text{mrun}\,\text{id} = \text{id} \quad , \qquad \text{mrun}\,\phi \fatsemi \text{mrun}\,\chi = \text{mrun}\,(\phi \fatsemi \chi) \quad .$$

5. **Base runner law**: For any monadic morphism $\theta : L \rightsquigarrow \text{Id}$ and for any monad M, the base runner $\text{brun}\,\theta : T_L^M \rightsquigarrow M$ is a monadic morphism. This morphism must satisfy the nondegeneracy law

$$\text{lift} \fatsemi \text{brun}\,\theta = \text{id} \quad .$$

(Since it is not possible to transform the base monad L into an arbitrary other monad, there are no lifting laws for `brun`, unlike `mrun`.)

***

13.2.1 Properties of monadic morphisms

Statement 13.2.1.1 If $\phi : M^\bullet \rightsquigarrow N^\bullet$, equivalently written as $\phi : M^A \Rightarrow N^A$, is a monadic morphism between monads M and N, then ϕ is also a natural transformation between functors M and N.

Statement 13.2.1.2 If L, M, N are monads and $\phi : L^\bullet \rightsquigarrow M^\bullet$ and $\chi : M^\bullet \rightsquigarrow N^\bullet$ are monadic morphisms then the composition $\phi \fatsemi \chi : L^\bullet \rightsquigarrow N^\bullet$ is also a monadic morphism.

Statement 13.2.1.3 For any monad M, the function $\triangle : M^A \Rightarrow M^A \times M^A$ is a monadic morphism between monads M and $M \times M$.

Statement 13.2.1.4 For any monads K, L, M, N and monadic morhpisms $\phi : K^\bullet \rightsquigarrow M^\bullet$ and $\chi : L^\bullet \rightsquigarrow N^\bullet$, the componentwise function product $\phi \boxtimes \chi : K^\bullet \times L^\bullet \rightsquigarrow M^\bullet \times N^\bullet$ is a monadic morphism.

Statement 13.2.1.5 For any monads M and N, the function $\nabla_1 : M^\bullet \times N^\bullet \rightsquigarrow M^\bullet$ is a monadic morphism. Same for $\nabla_2 : M^\bullet \times N^\bullet \rightsquigarrow N^\bullet$.

Statement 13.2.1.6 For any monads M and N, the component-swapping function $\sigma : M^\bullet \times N^\bullet \rightsquigarrow N^\bullet \times M^\bullet$ is a monadic morphism.

Proof: The code for σ can be written as $\sigma = \triangle \fatsemi \left(\nabla_2 \boxtimes \nabla_1\right)$. When σ is written in this way, it is easy to show that σ is a monadic morphism since σ is a composition of $\triangle$, which is a monadic morphism, and a componentwise function product of monadic morphisms ∇_1 and ∇_2.

13.3 Functor composition using transformed monads

Suppose we are working with a base monad L and a foreign monad M, and we have constructed the transformed monad T_L^M. In this section, let us denote this monad simply by T.

A useful property of monad transformers is that the monad T adequately describes the effects of both monads L and M at the same time. Suppose we are

working with a deeply nested type constructor involving many functor layers of monads L, M, and T such as

$$T^{M^{T^{L^{M^{L^A}}}}} \quad .$$

The properties of the transformer allow us to convert this type to a single layer of the transformed monad T. In this example, we will have a natural transformation

$$T^{M^{T^{L^{M^{L^A}}}}} \Rightarrow T^A \quad .$$

To achieve this, we first use the methods blift and lift to convert each layer of L or M to a layer of T, raising to functors as necessary. The result will be a number of nested layers of T. Second, we use ftn_T as many times as necessary to flatten all layers of T into a single layer.

13.4 Stacking composition of monad transformers

13.4.1 Stacking two monads

Suppose we know the transformers T_P and T_Q for some given monads P and Q. We can transform Q with P and obtain a monad $R^A \triangleq T_P^{Q,A}$. What would be the monad transformer T_R for the monad R?

A simple solution is to first transform the foreign monad M with T_Q, obtaining a new monad $T_Q^{M,\bullet}$, and then to transform that new monad with T_P. So the formula for the transformer T_R is

$$T_R^{M,A} = T_P^{T_Q^{M,\bullet},A} \quad .$$

Here the monad $T_Q^{M,\bullet}$ was substituted into $T_P^{M,A}$ as the foreign monad M (not as the type parameter A). This way of composition is called **stacking** the monad transformers.

In Scala code, this "stacking" composition is written as

```
type RT[M, A] = PT[QT[M, ?], A]
```

The resulting monad is a **stack** of three monads P, Q, and M. The order of monads in the stack is significant since, in general, there will be no monadic isomorphism between monads stacked in a different order.

We will now show that the transformer T_R is lawful (satisfies all five monad transformer laws), as long as the same five laws are satisfied by both T_P and T_Q. To shorten the notation, I will talk about a "monad T_P^M" meaning the monad defined as $T_P^{M,\bullet}$ or, more verbosely, the monad $G^A \triangleq T_P^{M,A}$.

Monad construction law We need to show that $T_P^{T_Q^M}$ is a monad for any monad M. The monad law for T_Q says that T_Q^M is a monad. Denote this monad temporarily by $S \triangleq T_Q^M$. The monad law for T_P says that T_P^M is a monad for any monad M; in particular, for $M = S$. Therefore, T_P^S is a monad, as required.

Identity law We need to show that $T_P^{T_Q^{\text{Id}}} \cong T_P^Q$ via a monadic isomorphism. The identity law for T_Q says that $T_Q^{\text{Id}} \cong Q$ via a monadic isomorphism. So, we have a monadic morphism $\phi : Q \rightsquigarrow T_Q^{\text{Id}}$ and its inverse, $\chi : T_Q^{\text{Id}} \rightsquigarrow Q$. The runner law for T_P can be applied to both ϕ and χ since they are monadic morphisms. So we obtain two new monadic morphisms,

$$\text{mrun}_P\, \phi : T_P^Q \rightsquigarrow T_P^{T_Q^{\text{Id}}} \quad ; \qquad \text{mrun}_P\, \chi : T_P^{T_Q^{\text{Id}}} \rightsquigarrow T_P^Q \quad .$$

Are these two monadic morphisms inverses of each other? Since mrun_P is a lifting of monadic morphisms, it lifts identity into identity and composition into composition. So the composition $\phi \fatsemi \chi = \text{id}$ is lifted into

$$\text{mrun}\, \phi \fatsemi \text{mrun}\, \chi = \text{id} \quad ,$$

and similarly for the composition $\chi \fatsemi \phi$. Thus we obtain a monadic isomorphism between monads T_P^Q and $T_P^{T_Q^{\text{Id}}}$.

Lifting law We need to show that there exists a monadic morphism $M \rightsquigarrow T_P^{T_Q^M}$ for any monad M. The lifting law for T_Q gives a monadic morphism $\text{lift}_Q : M \rightsquigarrow T_Q^M$. The lifting law for T_P can be applied to the monad T_Q^M, which gives a monadic morphism

$$\text{lift}_P : T_Q^M \rightsquigarrow T_P^{T_Q^M} \quad .$$

The composition of this with lift_Q is a monadic morphism of the required type $M \rightsquigarrow T_P^{T_Q^M}$. (We use the fact that a composition of monadic morphisms is again a monadic morphism.)

Runner law We need to show that there exists a lawful lifting

$$\text{mrun}_R : (M \rightsquigarrow N) \Rightarrow T_P^{T_Q^M} \rightsquigarrow T_P^{T_Q^N} \quad .$$

First, we have to define $\text{mrun}_R \phi$ for any given $\phi : M \rightsquigarrow N$. We use the lifting law for T_Q to get a monadic morphism

$$\text{lift}_Q \phi : T_Q^M \rightsquigarrow T_Q^N \quad .$$

Now we can apply the lifting law for T_P to this monadic morphism and obtain

$$\text{lift}_P \left(\text{lift}_Q \phi \right) : T_P^{T_Q^M} \rightsquigarrow T_P^{T_Q^N} \quad .$$

This function has the correct type signature. So we can define

$$\text{lift}_R \triangleq \text{lift}_P \circ \text{lift}_Q \quad .$$

It remains to prove that lift_R is a lawful lifting. We use the fact that both lift_P and lift_Q are lawful liftings; we need to show that their composition is also a lawful lifting. To verify the identity law of lifting, apply lift_R to an identity function $\text{id} : M \rightsquigarrow M$,

$$\begin{aligned} & \text{lift}_R \left(\text{id}^{:M \rightsquigarrow M} \right) = \text{lift}_P \left(\text{lift}_Q \text{id}^{:M \rightsquigarrow M} \right) \\ \text{identity law for } \text{lift}_Q : \quad & = \text{lift}_P (\text{id}^{:T_Q^M \rightsquigarrow T_Q^M}) \\ \text{identity law for } \text{lift}_P : \quad & = \text{id} \quad . \end{aligned}$$

To verify the composition law of lifting, apply lift_R to a composition of two monadic morphisms $\phi : L \rightsquigarrow M$ and $\chi : M \rightsquigarrow N$,

$$\begin{aligned} & \text{lift}_R \left(\phi \fatsemi \chi \right) = \text{lift}_P \left(\text{lift}_Q \left(\phi \fatsemi \chi \right) \right) \\ \text{composition law for } \text{lift}_Q : \quad & = \text{lift}_P \left(\text{lift}_Q \phi \fatsemi \text{lift}_Q \chi \right) \\ \text{composition law for } \text{lift}_P : \quad & = \text{lift}_P \left(\text{lift}_Q \phi \right) \fatsemi \text{lift}_P \left(\text{lift}_Q \phi \right) \\ & = \text{lift}_R \phi \fatsemi \text{lift}_R \chi \quad . \end{aligned}$$

Base runner law We need to show that for any monadic morphism $\theta : T_P^Q \rightsquigarrow \text{Id}$ and for any monad M, there exists a monadic morphism $\text{brun}_R \theta : T_P^{T_Q^M} \rightsquigarrow M$. To

define this morphism for a given θ, we clearly need to use the base runners for T_P and T_Q. The base runner for T_Q has the type signature

$$\text{brun}_Q : (Q \rightsquigarrow \text{Id}) \Rightarrow T_Q^M \rightsquigarrow M \quad .$$

We can apply the base runner for T_P to T_Q^M as the foreign monad,

$$\text{brun}_P : (P \rightsquigarrow \text{Id}) \Rightarrow T_P^{T_Q^M} \rightsquigarrow T_Q^M \quad .$$

It is now clear that we could obtain a monadic morphism $T_P^{T_Q^M} \rightsquigarrow M$ if we had some monadic morphisms $\phi : P \rightsquigarrow \text{Id}$ and $\chi : Q \rightsquigarrow \text{Id}$,

$$\text{brun}_P \phi \fatsemi \text{brun}_Q \chi : T_P^{T_Q^M} \rightsquigarrow M \quad .$$

However, we are only given a single monadic morphism $\theta : T_P^Q \rightsquigarrow \text{Id}$. How can we compute ϕ and χ out of θ? We can use the liftings $\text{blift}_P : P \rightsquigarrow T_P^Q$ and $\text{lift}_P : Q \rightsquigarrow T_P^Q$, which are both monadic morphisms, and compose them with θ:

$$\text{blift}_P \fatsemi \theta : P \rightsquigarrow \text{Id} \quad ; \qquad \text{lift}_P \fatsemi \theta : Q \rightsquigarrow \text{Id} \quad .$$

So it follows that we can define a monadic morphism

$$\text{brun}_R \theta = \text{brun}_P\ (\text{blift}_P \fatsemi \theta) \fatsemi \text{brun}_Q\ (\text{lift}_P \fatsemi \theta) : T_P^{T_Q^M} \rightsquigarrow M \quad .$$

Since we have defined $\text{brun}_R \theta$ as a composition of monadic morphisms, it is itself a monadic morphism.

13.4.2 Stacking multiple monads

The monad transformer for T_P^Q can be applied to another monad K; the result is the transformed monad

$$S^A \triangleq T_P^{T_Q^K, A}.$$

What is the monad transformer for the monad S? Assuming that we know the monad transformer T_K, we could stack the transformers one level higher:

$$T_S^{M,A} \triangleq T_P^{T_Q^{T_K^M}, A} \quad .$$

This looks like a stack of four monads P, Q, K, and M. Note that the type parameter A is used as $T_P^{(\ldots),A}$, that is, it belongs to the outer transformer T_P.

We can define a transformer stack for any number of monads P, Q, ..., Z in the same way,

$$T_S^{M,A} \triangleq T_P^{T_Q^{\cdots^{T_Z^M}},A} \quad . \tag{13.1}$$

The type parameter A will always remain at the outer transformer level, while the foreign monad M will be in the innermost nested position.

It appears plausible that T_S is a lawful monad transformer for *any* number of stacked monads. We can prove this by induction on the number of monads. In the previous section, we have derived the transformer laws for any *three* stacked monads (two monads P, Q within the transformer and one foreign monad M). Now we need to derive the same laws for a general transformer stack, such as that in Eq. (13.1). Let us temporarily denote by J the monad

$$J \triangleq T_Q^{\cdots^{T_Z^{\mathrm{Id}}}} \quad ,$$

where we replaced the foreign monad M by the identity monad. The monad J is a shorter transformer stack than S, so the inductive assumption tells us that the transformer laws already hold for the transformer T_J defined by

$$T_J^M \triangleq T_Q^{\cdots^{T_Z^M}} \quad .$$

Since both T_P and T_J are lawful transformers, their stacking composition $T_P^{T_J^M,A}$ is also a lawful transformer (this was shown in the previous section). In our notation, $T_S^{M,A} = T_P^{T_J^M,A}$, and so we have shown that $T_S^{M,A}$ is a lawful transformer.

13.5 Monad transformers that use functor composition

We have seen examples of monad transformers that work via functor composition, either from inside or from outside. The simplest such examples are the `OptionT` transformer,

$$L^A \triangleq 1 + A, \qquad T_L^{M,A} \triangleq M^{L^A} = M^{1+A} \quad ,$$

which puts the base monad L *inside* the monad M, and the `ReaderT` transformer,

$$L^A \triangleq R \Rightarrow A, \qquad T_L^{M,A} \triangleq L^{M^A} = R \Rightarrow M^A \quad ,$$

which puts the base monad L *outside* the foreign monad M.

We can obtain many properties of both kinds of monad transformers at once from a single derivation if we temporarily drop the distinction between the base monad and the foreign monad. We simply assume that we are given two different monads, L and M, whose functor composition $T^\bullet \triangleq L^{M^\bullet}$ also happens to be a monad. Since the assumptions on the monads L and M are the same, the resulting properties of the composed monad T will apply equally to both kinds of monad transformers.

To interpret the results, we will assume that L is the base monad for the composed-outside transformers, and that M is the base monad for the composed-inside transformers. For instance, we will be able to prove the laws of liftings $L \rightsquigarrow T$ and $M \rightsquigarrow T$ regardless of the choice of the base monad.

What properties of monad transformers will *not* be derivable in this way? Monad transformers depend on the structure on the base monad, but not on the structure of the foreign monad; the transformer's methods `pure` and `flatten` are generic in the foreign monad. This is expressed via the monad transformer laws for the runners `mrun` and `brun`, which we will need to derive separately for each of the two kinds of transformers.

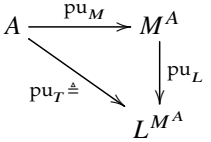
13.5.1 Motivation for the `swap` function

The first task is to show that the composed monad $T^\bullet \triangleq L^{M^\bullet}$ obeys the monad laws. For this, we need to define the methods for the monad T, namely `pure` (short notation "pu_T") and `flatten` (short notation "ftn_T"), with the type signatures

$$\text{pu}_T : A \Rightarrow L^{M^A} \quad , \quad \text{ftn}_T : L^{M^{L^{M^A}}} \Rightarrow L^{M^A} \quad .$$

How can we implement these methods? *All we know* about L and M is that they are monads with their own methods pu_L, ftn_L, pu_M, and ftn_M. We can easily implement

$$\text{pu}_T \triangleq \text{pu}_M \fatsemi \text{pu}_L \quad . \tag{13.2}$$



It remains to implement ftn_T. In the type $L^{M^{L^{M^A}}}$, we have two layers of the functor L and two layers of the functor M. We could use the available method ftn_L to flatten the two layers of L if we could *somehow* bring these nested layers together. However, these layers are separated by a layer of the functor M. To show this layered structure in a more visual way, let us employ another notation for the functor composition,

$$L \circ M \triangleq L^{M^\bullet} \quad .$$

In this notation, the type signature for `flatten` is written as

$$\text{ftn}_T : L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

If we had $L \circ L \circ ...$ here, we would have applied ftn_L and flattened the two layers of the functor L. Then we would have flattened the remaining two layers of the functor M. How can we achieve this? The trick is to *assume* that we can interchange the order of the layers in the middle, using a function called `swap` (short notation "sw"), with the type signature

$$\text{sw} : M \circ L \rightsquigarrow L \circ M \quad ,$$

which is equivalently written in the short type notation as

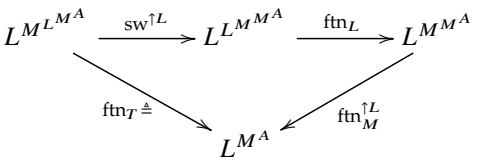
$$\text{sw} : M^{L^A} \Rightarrow L^{M^A} \quad .$$

If this operation were *somehow* defined for the two monads L and M, we could implement ftn_T by first swapping the order of the inner layers M and L as

$$L \circ M \circ L \circ M \rightsquigarrow L \circ L \circ M \circ M$$

and then applying the `flatten` methods of the monads L and M. The resulting code for the function ftn_T and the corresponding type diagram are

$$\text{ftn}_T \triangleq \text{sw}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \quad . \tag{13.3}$$



It turns out that in *both* cases (the composed-inside and the composed-outside transformers), the new monad's `flatten` method can be defined through the `swap` operation. For the two kinds of transformers, the type signatures of these functions are

$$\text{composed-inside}: \quad \text{ftn}_T : M^{L^{M^{L^A}}} \Rightarrow M^{L^A} \quad , \quad \text{sw} : L^{M^A} \Rightarrow M^{L^A} \quad ,$$

$$\text{composed-outside}: \quad \text{ftn}_T : L^{M^{L^{M^A}}} \Rightarrow L^{M^A} \quad , \quad \text{sw} : M^{L^A} \Rightarrow L^{M^A} \quad .$$

The operations `swap` and `sequence` There is a certain similarity between the `swap` operation introduced here and the `sequence` operation introduced in Chapter 11 for traversable functors. Indeed, the type signature of the sequence operation is

$$\text{seq} : L^{F^A} \Rightarrow F^{L^A} \quad ,$$

where F is an arbitrary applicative functor (which could be M, since monads are applicative functors) and L is a traversable functor. However, the similarity ends here. The laws required for the `swap` operation to yield a monad T are much stronger than the laws of traversable functors. In particular, if we wish $M^{L^\bullet}$ to be a monad, it is insufficient to require the monad L to be a traversable functor. A simple counterexample is found with $L^A \triangleq A \times A$ and $M^A \triangleq 1 + A$. Both L and M are traversable (since they are polynomial); but their composition $Q^A \triangleq 1 + A \times A$ is not a monad.

Another difference between `swap` and `sequence` is that the `swap` operation needs to be generic in the foreign monad, which may be either L or M according to the type of the monad transformer; whereas `sequence` is always generic in F. To denote more clearly the monad with respect to which `swap` is generic, we may write

$$\text{sw}_L^M : L^{M^A} \Rightarrow M^{L^A} \quad \text{for the composed-inside transformers,}$$

$$\text{sw}_L^M : M^{L^A} \Rightarrow L^{M^A} \quad \text{for the composed-inside transformers.}$$

The superscript M in sw_L^M shows that M is a *type parameter* in `swap`; that is, `swap` is generic in the monad M. The subscript L in sw_L^M shows that the implementations of `swap` may need to use completely different code for different monads L.

To avoid confusion, I use the name "swap" rather than "sequence" for the function $M^{L^\bullet} \rightsquigarrow L^{M^\bullet}$ in the context of monad transformers. Let us now find out what laws are required for the `swap` operation.[1]

13.5.2 Laws for `swap`

The first law is that `swap` must be a natural transformation. Since `swap` has only one type parameter, there is one naturality law: for any function $f : A \Rightarrow B$,

$$f^{\uparrow L\uparrow M} \fatsemi \text{sw} = \text{sw} \fatsemi f^{\uparrow M\uparrow L} \quad . \tag{13.4}$$

$$\begin{array}{ccc} M^{L^A} & \xrightarrow{f^{\uparrow L\uparrow M}} & M^{L^B} \\ {\scriptstyle \text{sw}}\downarrow & & \downarrow{\scriptstyle \text{sw}} \\ L^{M^A} & \xrightarrow{f^{\uparrow M\uparrow L}} & L^{M^B} \end{array}$$

To derive further laws for `swap`, consider the requirement that the transformed monad T should satisfy the monad laws:

$$\text{pu}_T \fatsemi \text{ftn}_T = \text{id} \quad , \qquad \text{pu}_T^{\uparrow T} \fatsemi \text{ftn}_T = \text{id} \quad ,$$
$$\text{ftn}_T^{\uparrow T} \fatsemi \text{ftn}_T = \text{ftn}_T \fatsemi \text{ftn}_T \quad .$$

Additionally, T must satisfy the laws of a monad transformer. We will now discover the laws for `swap` that make the laws for ftn_T hold automatically, as long as ftn_T is derived from `swap` using Eq. (13.3).

We substitute Eq. (13.3) into the left identity law for ftn_T and simplify:

$$\begin{aligned} \text{id} &= \text{pu}_T \fatsemi \underline{\text{ftn}_T} \\ \text{replace } \text{ftn}_T \text{ using Eq. (13.3)} : \quad &= \text{pu}_T \fatsemi \text{sw}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{replace } \text{pu}_T \text{ using Eq. (13.2)} : \quad &= \text{pu}_M \fatsemi \underline{\text{pu}_L \fatsemi \text{sw}^{\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{naturality of } \text{pu}_L : \quad &= \text{pu}_M \fatsemi \text{sw} \fatsemi \underline{\text{pu}_L \fatsemi \text{ftn}_L} \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{left identity law for } L : \quad &= \text{pu}_M \fatsemi \text{sw} \fatsemi \text{ftn}_M^{\uparrow L} \quad . \end{aligned} \tag{13.5}$$

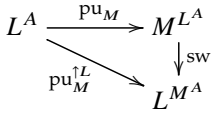
[1] The "swap" operation was first introduced in a 1993 paper "Composing monads" by M. P. Jones and L. Duponcheel. They studied various ways of composing monads and also gave some arguments to show that no generic transformer could compose all monads L, M. The impossibility of a generic monad composition is demonstrated by the `State` monad that, as I show in this chapter, does not compose with arbitrary other monads M – either from inside or from outside.

How could the last expression in Eq. (13.5) be equal to id? We know nothing about the `pure` and `flatten` methods of the monads L and M, except that these methods satisfy their monad laws. We could satisfy the law in Eq. (13.5) if we somehow reduce that expression to

$$\text{pu}_M^{\uparrow L} \fatsemi \text{ftn}_M^{\uparrow L} = \left(\text{pu}_M \fatsemi \text{ftn}_M\right)^{\uparrow L} = \text{id} \quad .$$

This will be possible only if we are able to interchange the order of function compositions with sw and eliminate `swap` from the expression. So, we must require the "M-identity law" for `swap`,

$$\text{pu}_M \fatsemi \text{sw} = \text{pu}_M^{\uparrow L} \quad . \tag{13.6}$$



Intuitively, this law says that a pure layer of the monad M remains pure after interchanging the order of layers with `swap`.

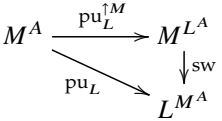
With this law, we can finish the derivation in Eq. (13.5) as

$$\begin{aligned}
& \text{pu}_M \fatsemi \text{sw} \fatsemi \text{ftn}_M^{\uparrow L} \\
M\text{-identity law for sw}: \quad & = \text{pu}_M^{\uparrow L} \fatsemi \text{ftn}_M^{\uparrow L} \\
\text{functor composition law for } L: \quad & = \left(\text{pu}_M \fatsemi \text{ftn}_M\right)^{\uparrow L} \\
\text{left identity law for } M: \quad & = \text{id}^{\uparrow L} \\
\text{functor identity law for } L: \quad & = \text{id} \quad .
\end{aligned}$$

So, the M-identity law for `swap` entails the left identity law for T.

In the same way, we motivate the "L-identity" law for `swap`,

$$\text{pu}_L^{\uparrow M} \fatsemi \text{sw} = \text{pu}_L \quad . \tag{13.7}$$



This law expresses the idea that a pure layer of the functor L remains pure after swapping the order of layers.

Assuming this law, we can derive the right identity law for T:

$$\begin{aligned}
& \mathrm{pu}_T^{\uparrow T} \fatsemi \mathrm{ftn}_T \\
\left(\text{note that } f^{\uparrow T} \triangleq f^{\uparrow M \uparrow L}\right): \quad & = \left(\mathrm{pu}_T\right)^{\uparrow M \uparrow L} \fatsemi \mathrm{ftn}_T \\
\text{definitions of } \mathrm{pu}_T \text{ and } \mathrm{ftn}_T: \quad & = \mathrm{pu}_M^{\uparrow M \uparrow L} \fatsemi \underline{\mathrm{pu}_L^{\uparrow M \uparrow L} \fatsemi \mathrm{sw}^{\uparrow L}} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
L\text{-identity law for sw, under } {}^{\uparrow L}: \quad & = \mathrm{pu}_M^{\uparrow M \uparrow L} \fatsemi \underline{\mathrm{pu}_L^{\uparrow L} \fatsemi \mathrm{ftn}_L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{right identity law for } L: \quad & = \mathrm{pu}_M^{\uparrow M \uparrow L} \fatsemi \mathrm{ftn}_M^{\uparrow L} = \underline{\left(\mathrm{pu}_M^{\uparrow M} \fatsemi \mathrm{ftn}_M\right)^{\uparrow L}} \\
\text{right identity law for } M: \quad & = \mathrm{id}^{\uparrow L} = \mathrm{id} \quad .
\end{aligned}$$

Deriving the monad associativity law for T,

$$\mathrm{ftn}_T^{\uparrow T} \fatsemi \mathrm{ftn}_T = \mathrm{ftn}_T \fatsemi \mathrm{ftn}_T \quad ,$$

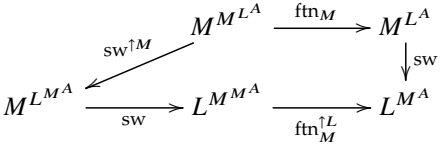
turns out to require *two* further laws for `swap`. Substituting the definition of ftn_T into the associativity law, we get

$$\begin{aligned}
& \left(\mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L}\right)^{\uparrow M \uparrow L} \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
& \quad = \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \quad . \qquad (13.8)
\end{aligned}$$

The only hope of proving this law is being able to interchange ftn_L as well as ftn_M with sw. In other words, the `swap` function should be in some way adapted to the `flatten` methods of *both* monads L and M.

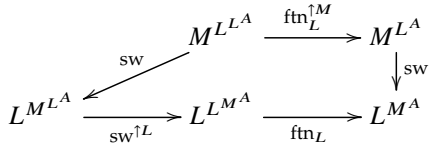
Let us look for such interchange laws. One possibility is to have a law involving $\mathrm{ftn}_M \fatsemi \mathrm{sw}$, which is a function of type $M^{M^{L^A}} \Rightarrow L^{M^A}$ or, in another notation, $M \circ M \circ L \rightsquigarrow L \circ M$. This function first flattens the two adjacent layers of M, obtaining $M \circ L$, and then swaps the two remaining layers, moving the L layer outside. Let us think about what law could exist for this kind of transformation. It is plausible that we may obtain the same result if we first swap the layers *twice*, so that the L layer moves to the outside, obtaining $L \circ M \circ M$, and then flatten the two inner M layers. Writing this assumption in code, we obtain the "M-interchange" law

$$\mathrm{ftn}_M \fatsemi \mathrm{sw} = \mathrm{sw}^{\uparrow M} \fatsemi \mathrm{sw} \fatsemi \mathrm{ftn}_M^{\uparrow L} \quad . \qquad (13.9)$$



The analogous "L-interchange" law involving two layers of L and a transformation $M \circ L \circ L \rightsquigarrow L \circ M$ is written as

$$\mathrm{ftn}_L^{\uparrow M} \fatsemi \mathrm{sw} = \mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \quad . \tag{13.10}$$



At this point, we have simply written down these two interchange laws, hoping that they will help us derive the associativity law for T. It remains to prove that this is indeed so.

Both sides of the law in Eq. (13.8) involve compositions of several `flattens` and `swaps`. The heuristic idea of the proof is to use various laws to move all `flattens` to right of the composition, while moving all `swaps` to the left. In this way we will transform both sides of Eq. (13.8) into a similar form, hoping to prove that they are equal.

We begin with the right-hand side of Eq. (13.8) since it is simpler than the left-hand side, and look for ways of using the interchange laws. At every step of the calculation, there happens to be only one place where some law can be applied:

$$\begin{aligned}
& \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \underline{\mathrm{ftn}_M^{\uparrow L} \fatsemi \mathrm{sw}^{\uparrow L}} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{composition for } L: \quad & = \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \underline{(\mathrm{ftn}_M \fatsemi \mathrm{sw})^{\uparrow L}} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
M\text{-interchange for sw}: \quad & = \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \underline{\left(\mathrm{sw}^{\uparrow M} \fatsemi \mathrm{sw} \fatsemi \mathrm{ftn}_M^{\uparrow L}\right)^{\uparrow L}} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{composition for } L: \quad & = \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \underline{\mathrm{sw}^{\uparrow M \uparrow L} \fatsemi \mathrm{sw}^{\uparrow L}} \fatsemi \underline{\mathrm{ftn}_M^{\uparrow L \uparrow L} \fatsemi \mathrm{ftn}_L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{naturality of } \mathrm{ftn}_L: \quad & = \mathrm{sw}^{\uparrow L} \fatsemi \underline{\mathrm{ftn}_L \fatsemi \left(\mathrm{sw}^{\uparrow M} \fatsemi \mathrm{sw}\right)^{\uparrow L}} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{naturality of } \mathrm{ftn}_L: \quad & = \mathrm{sw}^{\uparrow L} \fatsemi \underline{\left(\mathrm{sw}^{\uparrow M} \fatsemi \mathrm{sw}\right)^{\uparrow L \uparrow L} \fatsemi \mathrm{ftn}_L} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \quad .
\end{aligned}$$

Now all `swap`s are on the left and all `flatten`s on the right of the expression.

Transform the right-hand side of Eq. (13.8) in the same way as

$$
\begin{aligned}
& \left(\mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \underline{\mathrm{ftn}_M^{\uparrow L}}\right)^{\uparrow M \uparrow L} \fatsemi \underline{\mathrm{sw}^{\uparrow L}} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{functor composition} : &= \left(\mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L\right)^{\uparrow M \uparrow L} \fatsemi \underline{\left(\mathrm{ftn}_M^{\uparrow L \uparrow M} \fatsemi \mathrm{sw}\right)}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{naturality of sw} : &= \left(\mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L\right)^{\uparrow M \uparrow L} \fatsemi (\mathrm{sw} \fatsemi \underline{\mathrm{ftn}_M^{\uparrow M \uparrow L}})^{\uparrow L} \fatsemi \underline{\mathrm{ftn}_L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{naturality of } \mathrm{ftn}_L : &= \mathrm{sw}^{\uparrow L \uparrow M \uparrow L} \fatsemi \mathrm{ftn}_L^{\uparrow M \uparrow L} \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \underline{\mathrm{ftn}_M^{\uparrow M \uparrow L}} \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{associativity of } \mathrm{ftn}_M : &= \mathrm{sw}^{\uparrow L \uparrow M \uparrow L} \fatsemi \underline{\left(\mathrm{ftn}_L^{\uparrow M} \fatsemi \mathrm{sw}\right)}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
L\text{-interchange for sw} : &= \mathrm{sw}^{\uparrow L \uparrow M \uparrow L} \fatsemi (\mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \underline{\mathrm{ftn}_L})^{\uparrow L} \fatsemi \underline{\mathrm{ftn}_L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{associativity of } \mathrm{ftn}_L : &= (\mathrm{sw}^{\uparrow L \uparrow M} \fatsemi \mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow L})^{\uparrow L} \fatsemi \underline{\mathrm{ftn}_L \fatsemi \mathrm{ftn}_L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \fatsemi \mathrm{ftn}_M^{\uparrow L} \quad .
\end{aligned}
$$

We have again managed to move all `swap`s to the left and all `flatten`s to the right of the expression.

Comparing now the two sides of the associativity law, we see that all the `flatten`s occur in the same combination: $\mathrm{ftn}_L \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \fatsemi \mathrm{ftn}_M^{\uparrow L}$. It remains to show that

$$
\mathrm{sw}^{\uparrow L} \fatsemi \left(\mathrm{sw}^{\uparrow M} \fatsemi \mathrm{sw}\right)^{\uparrow L \uparrow L} = (\mathrm{sw}^{\uparrow L \uparrow M} \fatsemi \mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow L})^{\uparrow L} \quad .
$$

or equivalently

$$
\left(\mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow M \uparrow L} \fatsemi \mathrm{sw}^{\uparrow L}\right)^{\uparrow L} = (\mathrm{sw}^{\uparrow L \uparrow M} \fatsemi \mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow L})^{\uparrow L} \quad .
$$

The two sides are equal due to the naturality law of `swap`,

$$
\mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow M \uparrow L} = \mathrm{sw}^{\uparrow L \uparrow M} \fatsemi \mathrm{sw}.
$$

This completes the proof of the following theorem:

Theorem 13.5.2.1 If two monads L and M are such that there exists a natural transformation

$$
\mathrm{sw} : M^{L^A} \Rightarrow L^{M^A}
$$

(called "`swap`") satisfying the four additional laws,

$$\begin{aligned} L\text{-identity}: &\quad \mathrm{pu}_L^{\uparrow M} \fatsemi \mathrm{sw} = \mathrm{pu}_L \quad , \\ M\text{-identity}: &\quad \mathrm{pu}_M \fatsemi \mathrm{sw} = \mathrm{pu}_M^{\uparrow L} \quad , \\ L\text{-interchange}: &\quad \mathrm{ftn}_L^{\uparrow M} \fatsemi \mathrm{sw} = \mathrm{sw} \fatsemi \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \quad , \\ M\text{-interchange}: &\quad \mathrm{ftn}_M \fatsemi \mathrm{sw} = \mathrm{sw}^{\uparrow M} \fatsemi \mathrm{sw} \fatsemi \mathrm{ftn}_M^{\uparrow L} \quad , \end{aligned}$$

then the functor composition

$$T^A \triangleq L^{M^A}$$

is a monad with the methods `pure` and `flatten` defined by

$$\mathrm{pu}_T \triangleq \mathrm{pu}_M \fatsemi \mathrm{pu}_L \quad , \tag{13.11}$$

$$\mathrm{ftn}_T \triangleq \mathrm{sw}^{\uparrow L} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \quad . \tag{13.12}$$

13.5.3 Intuition behind the laws of `swap`

The interchange laws for `swap` guarantee that in any functor composition built up from L and M, e.g. like this,

$$M \circ M \circ L \circ M \circ L \circ L \circ M \circ M \circ L \quad ,$$

we can flatten the layers using ftn_L, or ftn_M, ftn_T, or interchange the layers with `swap`, in any order. We will always get the same final value, which we can transform to the monad type $T = L \circ M$.

In other words, the monadic effects of the monads L and M can be arbitrarily interleaved, swapped, and flattened in any order, with no change to the final results. The programmer is free to refactor a monadic program, say, by computing some L-effects in a separate functor block of L-`flatMap`s and only then combining the result with the rest of the computation in the monad T. Regardless of the refactoring, the monad T computes all the effects correctly. This is what programmers would certainly expect of the monad T, if it is to be a useful monad transformer.

We will now derive the properties of ftn_T that correspond to the interchange laws. We will find that it is easier to formulate these laws in terms of `swap` than in terms of ftn_T. In practice, all known examples of compositional monad transformers (the "linear" and the "rigid" monads) are defined via `swap`.

13.5.4 Deriving `swap` from `flatten`

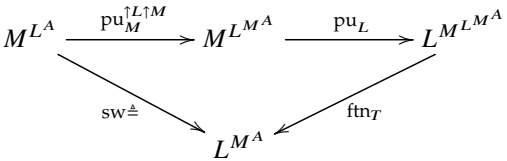
We have seen that the `flatten` method for the monad $T^\bullet = L^{M^\bullet}$ can be defined via the `swap` method. However, we have seen examples of some composable monads (such as `Reader` and `Option`) where we already know the definitions of the `flatten` method for T. How do we know whether a suitable `swap` function exist for these examples? In other words, if a `flatten` function for the monad $T^\bullet = L^{M^\bullet}$ is already given, can we establish whether a `swap` function exists such that the given `flatten` function is expressed via Eq. (13.3)?

To answer this question, let us look at the type signature of `flatten` for T:

$$\text{ftn}_T : L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

This type signature is different from $\text{sw} : M \circ L \rightsquigarrow L \circ M$ only because the argument of ftn_T has extra layers of the functors L and M that are placed outside the $M \circ L$ composition. We can use the `pure` methods of M and L to add these extra layers to a value of type $M \circ L$, without modifying any monadic effects present in $M \circ L$. This will allow us to apply ftn_T and to obtain a value of type $L \circ M$. The resulting code for the function ftn_T and the corresponding type diagram are

$$\text{sw} = \text{pu}_M^{\uparrow L \uparrow M} \fatsemi \text{pu}_L \fatsemi \text{ftn}_T \quad . \tag{13.13}$$



We have expressed ftn_T and sw through each other. Are these functions always equivalent? To decide this, we need to answer two questions:

1. If we first define ftn_T using Eq. (13.3) through a given implementation of sw, and then substitute that ftn_T into Eq. (13.13), will we always recover the initially given function sw? (Yes.)

2. If we first define sw using Eq. (13.13) through a given implementation of ftn_T, and then substitute that sw into Eq. (13.3), will we always recover the initially given function ftn_T? (No, not without additional assumptions for ftn_T.)

To answer the first question, substitute ftn_T from Eq. (13.3) into Eq. (13.13):

$$\begin{aligned}
& \text{pu}_M^{\uparrow L\uparrow M} \fatsemi \text{pu}_L \fatsemi \text{ftn}_T \\
\text{substitute } \text{ftn}_T : \quad & = \text{pu}_M^{\uparrow L\uparrow M} \fatsemi \underline{\text{pu}_L \fatsemi \text{sw}^{\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\
\text{naturality of } \text{pu}_L : \quad & = \text{pu}_M^{\uparrow L\uparrow M} \fatsemi \text{sw} \fatsemi \underline{\text{pu}_L \fatsemi \text{ftn}_L} \fatsemi \text{ftn}_M^{\uparrow L} \\
\text{left identity law for } L : \quad & = \underline{\text{pu}_M^{\uparrow L\uparrow M} \fatsemi \text{sw}} \fatsemi \text{ftn}_M^{\uparrow L} \\
\text{naturality of sw} : \quad & = \text{sw} \fatsemi \underline{\text{pu}_M^{\uparrow M\uparrow L} \fatsemi \text{ftn}_M^{\uparrow L}} \\
\text{functor composition for } L : \quad & = \text{sw} \fatsemi \underline{\left(\text{pu}_M^{\uparrow M} \fatsemi \text{ftn}_M\right)^{\uparrow L}} \\
\text{right identity law for } M : \quad & = \text{sw} \quad .
\end{aligned}$$

So, yes, we do always recover the initial `swap` function.

To answer the second question, substitute sw from Eq. (13.13) into Eq. (13.3):

$$\begin{aligned}
& \text{sw}^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\
\text{substitute sw} : \quad & = \left(\text{pu}_M^{\uparrow L\uparrow M} \fatsemi \text{pu}_L \fatsemi \text{ftn}_T\right)^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\
\text{functor composition} : \quad & = \text{pu}_M^{\uparrow L\uparrow M\uparrow L} \fatsemi \text{pu}_L^{\uparrow L} \fatsemi \text{ftn}_T^{\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \quad . \qquad (13.14)
\end{aligned}$$

At this point, we are stuck: no laws can be applied to transform the last expression any further. Without additional assumptions, it does not follow that this expression is equal to ftn_T. Let us derive the necessary additional assumptions.

A simplification of Eq. (13.14) clearly requires a law involving the function

$$\text{ftn}_T^{\uparrow L} \fatsemi \text{ftn}_L \quad : L \circ L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

This function flattens the two layers of $(L \circ M)$ and then flattens the remaining two layers of L. An alternative transformation with the same type signature is to first flatten the two *outside* layers of L and then to flatten the two remaining layers of $(L \circ M)$:

$$\text{ftn}_L \fatsemi \text{ftn}_T \quad : L \circ L \circ M \circ L \circ M \rightsquigarrow L \circ M \quad .$$

So we conjecture that a possibly useful additional law for ftn_T is

$$\text{ftn}_L \fatsemi \text{ftn}_T = \text{ftn}_T^{\uparrow L} \fatsemi \text{ftn}_L \quad .$$

$$\begin{array}{ccc} L^{L^{M^{L^{M^A}}}} & \xrightarrow{\text{ftn}_L} & L^{M^{L^{M^A}}} \\ {\scriptstyle \text{ftn}_T^{\uparrow L}}\downarrow & & \downarrow{\scriptstyle \text{ftn}_T} \\ L^{L^{M^A}} & \xrightarrow[\text{ftn}_L]{} & L^{M^A} \end{array}$$

This law shows a kind of "compatibility" between the monads L and T.

With this law, the right-hand side of Eq. (13.14) becomes

$$\begin{aligned} & \text{pu}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \underline{\text{pu}_L^{\uparrow L} \fatsemi \text{ftn}_L} \fatsemi \text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{right identity law of } L: \quad & = \text{pu}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} \quad . \end{aligned}$$

Again, we cannot proceed unless we have a law involving the function

$$\text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} \quad : L \circ M \circ L \circ M \circ M \rightsquigarrow L \circ M \quad .$$

This function first flattens the two layers of $(L \circ M)$ and then flattens the remaining two layers of M. An alternative order of flattenings is to first flatten the *innermost* two layers of M:

$$\text{ftn}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_T \quad : L \circ M \circ L \circ M \circ M \rightsquigarrow L \circ M \quad .$$

The second conjectured law is therefore

$$\text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} = \text{ftn}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_T \quad .$$

$$\begin{array}{ccc} L^{M^{L^{M^{M^A}}}} & \xrightarrow{\text{ftn}_T} & L^{M^{M^A}} \\ {\scriptstyle \text{ftn}_M^{\uparrow L \uparrow M \uparrow L}}\downarrow & & \downarrow{\scriptstyle \text{ftn}_M^{\uparrow L}} \\ L^{M^{L^{M^A}}} & \xrightarrow[\text{ftn}_T]{} & L^{M^A} \end{array}$$

This law expresses "compatibility" between the `flatten` methods of M and T.

With this law, we can finally complete the derivation:

$$\begin{aligned} & \text{pu}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{substitute the second conjecture}: \quad & = \underline{\text{pu}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_M^{\uparrow L \uparrow M \uparrow L}} \fatsemi \text{ftn}_T \\ \text{functor composition}: \quad & = \left(\text{pu}_M \fatsemi \text{ftn}_M\right)^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_T \\ \text{left identity law of } M: \quad & = \text{ftn}_T \quad . \end{aligned}$$

We recovered the initial ftn_T by assuming two additional laws.

It turns out that these additional laws will always hold when ftn_T is defined via `swap` (see Exercise 13.5.4.1).

Given some monads L and M, it is cumbersome to verify the associativity and the interchange laws for T directly because of the deeply nested type constructors, such as $L \circ M \circ L \circ M \circ L \circ M$. If the monad T is defined via a suitable `swap` method (in practice, this is always the case), it is simpler to verify the laws of T by checking the laws of `swap`. This is what we will do next.

Exercise 13.5.4.1 Assuming that

- L and M are monads,
- the method `swap` is a natural transformation $M \circ L \rightsquigarrow L \circ M$,
- the method ftn_T of the monad $T = L \circ M$ is defined via `swap` by Eq. (13.3),

show that the two interchange laws hold for T:

$$\begin{aligned} L\text{-interchange} &: \quad \text{ftn}_L \fatsemi \text{ftn}_T = \text{ftn}_T^{\uparrow L} \fatsemi \text{ftn}_L \quad , \\ M\text{-interchange} &: \quad \text{ftn}_T \fatsemi \text{ftn}_M^{\uparrow L} = \text{ftn}_M^{\uparrow L \uparrow M \uparrow L} \fatsemi \text{ftn}_T \quad . \end{aligned}$$

Exercise 13.5.4.2 With the same assumptions as Exercise 13.5.4.1 and additionally assuming the L-identity and the M-identity laws for `swap` (see Theorem 13.5.2.1), show that the monad $T^\bullet \triangleq L^{M^\bullet}$ satisfies two "pure compatibility" laws,,

$$\begin{aligned} L\text{-pure-compatibility} &: \quad \text{ftn}_L = \text{pu}_M^{\uparrow L} \fatsemi \text{ftn}_T \quad : L^{L^{M^A}} \Rightarrow L^{M^A} \quad , \\ M\text{-pure-compatibility} &: \quad \text{ftn}_M^{\uparrow L} = \text{pu}_L^{\uparrow T} \fatsemi \text{ftn}_T \quad : L^{M^{M^A}} \Rightarrow L^{M^A} \quad , \end{aligned}$$

or, expressed equivalently through the flm methods instead of ftn,

$$\begin{aligned} \text{flm}_L f^{:A \Rightarrow L^{M^B}} &= \text{pure}_M^{\uparrow L} \fatsemi \text{flm}_T f^{:A \Rightarrow L^{M^B}} \quad , \\ \left(\text{flm}_M f^{:A \Rightarrow M^B}\right)^{\uparrow L} &= \text{pure}_L^{\uparrow T} \fatsemi \text{flm}_T \left(f^{\uparrow L}\right) \quad . \end{aligned}$$

13.5.5 Laws of monad transformer liftings

We will now derive the laws of liftings from the laws of `swap`, using Eqs. (13.11)–(13.12) as definitions of the methods of T.

To be specific, let us assume that L is the base monad of the transformer. Only names will need to change for the other choice of the base monad.

The lifting morphisms of a compositional monad transformer are defined by

$$\begin{aligned} \text{lift} &= \text{pu}_L : M^A \Rightarrow L^{M^A} \quad , \\ \text{blift} &= \text{pu}_M^{\uparrow L} : L^A \Rightarrow L^{M^A} \quad . \end{aligned}$$

Their laws of liftings (the identity and the composition laws) are

$$\begin{aligned} \text{pu}_M \fatsemi \text{lift} = \text{pu}_T \quad , \quad & \text{pu}_L \fatsemi \text{blift} = \text{pu}_T \quad , \\ \text{ftn}_M \fatsemi \text{lift} = \text{lift}^{\uparrow M} \fatsemi \text{lift} \fatsemi \text{ftn}_T \quad , \quad & \text{ftn}_L \fatsemi \text{blift} = \text{blift}^{\uparrow L} \fatsemi \text{blift} \fatsemi \text{ftn}_T \quad . \end{aligned}$$

The identity laws are verified quickly,

$$\begin{aligned} \text{expect to equal } \text{pu}_T : \quad & \text{pu}_M \fatsemi \text{lift} = \text{pu}_M \fatsemi \text{pu}_L \\ \text{definition of } \text{pu}_T : \quad & = \text{pu}_T \quad , \\ \text{expect to equal } \text{pu}_T : \quad & \text{pu}_L \fatsemi \text{blift} = \text{pu}_L \fatsemi \text{pu}_M^{\uparrow L} \\ \text{naturality of } \text{pu}_L : \quad & = \text{pu}_M \fatsemi \text{pu}_L = \text{pu}_T \quad . \end{aligned}$$

To verify the composition laws, we need to start from their right-hand sides because the left-hand sides cannot be simplified, and we substitute the definition of ftn_T in terms of `swap`. The composition law for lift:

$$\begin{aligned} \text{expect to equal } \text{ftn}_M \fatsemi \text{pu}_L : \quad & \text{lift}^{\uparrow M} \fatsemi \text{lift} \fatsemi \text{ftn}_T \\ \text{definitions of lift and } \text{ftn}_T : \quad & = \text{pu}_L^{\uparrow M} \fatsemi \underline{\text{pu}_L \fatsemi \text{sw}^{\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{naturality of } \text{pu}_L : \quad & = \text{pu}_L^{\uparrow M} \fatsemi \text{sw} \fatsemi \underline{\text{pu}_L \fatsemi \text{ftn}_L} \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{left identity law of } L : \quad & = \underline{\text{pu}_L^{\uparrow M} \fatsemi \text{sw}} \fatsemi \text{ftn}_M^{\uparrow L} \\ L\text{-identity law of sw} : \quad & = \text{pu}_L \fatsemi \text{ftn}_M^{\uparrow L} \\ \text{naturality of } \text{pu}_L : \quad & = \text{ftn}_M \fatsemi \text{pu}_L \quad . \end{aligned}$$

The composition law for blift:

$$\begin{aligned}
\text{expect to equal } \mathrm{ftn}_L \fatsemi \mathrm{pu}_M^{\uparrow L} : &\quad \mathrm{blift}^{\uparrow L} \fatsemi \mathrm{blift} \fatsemi \mathrm{ftn}_T \\
\text{definitions of blift and } \mathrm{ftn}_T : &\quad = \mathrm{pu}_M^{\uparrow L \uparrow L} \fatsemi \underline{\mathrm{pu}_M^{\uparrow L} \fatsemi \mathrm{sw}^{\uparrow L}} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{functor composition in } L : &\quad = \mathrm{pu}_M^{\uparrow L \uparrow L} \fatsemi \underline{\left(\mathrm{pu}_M \fatsemi \mathrm{sw}\right)^{\uparrow L}} \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
M\text{-identity law of sw} : &\quad = \left(\mathrm{pu}_M^{\uparrow L \uparrow L} \fatsemi \mathrm{pu}_M^{L \uparrow L}\right) \fatsemi \mathrm{ftn}_L \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{naturality of } \mathrm{ftn}_L : &\quad = \mathrm{ftn}_L \fatsemi \left(\mathrm{pu}_M^{\uparrow L} \fatsemi \mathrm{pu}_M^{\uparrow L}\right) \fatsemi \mathrm{ftn}_M^{\uparrow L} \\
\text{right identity law of } M : &\quad = \mathrm{ftn}_L \fatsemi \mathrm{pu}_M^{\uparrow L} \quad .
\end{aligned}$$

So, the lifting laws for T follow from the laws of `swap`.

13.5.6 Laws of monad transformer runners

The laws of runners are not symmetric with respect to the base monad and the foreign monad: the runner, `mrun`, is generic in the foreign monad (but not in the base monad). In each case, the `swap` function must be monadically natural in the *foreign* monad. This law needs to be written differently, depending on the choice of the base monad. So we need to consider separately the cases when either L or M is the base monad.

If the base monad is L, the runners are

$$\begin{aligned}
\mathrm{mrun}\, \phi^{: M^\bullet \rightsquigarrow N^\bullet} : L^{M^\bullet} \rightsquigarrow L^{N^\bullet} \quad , \qquad \mathrm{mrun}\, \phi = \phi^{\uparrow L} \quad , \\
\mathrm{brun}\, \theta^{: L^\bullet \rightsquigarrow \bullet} : L^{M^\bullet} \rightsquigarrow M^\bullet \quad , \qquad \mathrm{brun}\, \theta = \theta \quad .
\end{aligned}$$

The monadic naturality of the `swap` function is written as

$$\mathrm{sw}_L^M \fatsemi \phi^{\uparrow L} = \phi \fatsemi \mathrm{sw}_L^N \quad .$$

$$\begin{array}{ccc}
M^{L^A} & \xrightarrow{\mathrm{sw}_L^M} & L^{M^A} \\
{\scriptstyle \phi}\downarrow & & \downarrow{\scriptstyle \phi^{\uparrow L}} \\
N^{L^A} & \xrightarrow[\mathrm{sw}_L^N]{} & L^{N^A}
\end{array}$$

The laws of runners require that $\text{mrun}\,\phi$ and $\text{brun}\,\theta$ must be monadic morphisms, i.e. the identity and composition laws of monadic morphisms must hold for $\text{mrun}\,\phi$ and $\text{brun}\,\theta$:

$$\begin{aligned}
\text{pu}_{L\circ M} \fatsemi \text{mrun}\,\phi &= \text{pu}_{L\circ N} \quad , \\
\text{ftn}_{L\circ M} \fatsemi \text{mrun}\,\phi &= (\text{mrun}\,\phi)^{\uparrow M\uparrow L} \fatsemi \text{mrun}\,\phi \fatsemi \text{ftn}_{L\circ N} \quad , \\
\text{pu}_{L\circ M} \fatsemi \text{brun}\,\theta &= \text{pu}_M \quad , \\
\text{ftn}_{L\circ M} \fatsemi \text{brun}\,\theta &= (\text{brun}\,\theta)^{\uparrow M\uparrow L} \fatsemi \text{brun}\,\theta \fatsemi \text{ftn}_M \quad .
\end{aligned}$$

To derive these laws, we may use the identity and composition laws of monadic morphisms for ϕ and θ. We also use Eqs. (13.11)–(13.12) as definitions of the methods of T.

The first law to be shown is the identity law for $\text{mrun}\,\phi$:

$$\begin{aligned}
\text{expect this to equal } \text{pu}_{L\circ N}: \quad & \text{pu}_{L\circ M} \fatsemi \text{mrun}\,\phi \\
\text{definitions of mrun and } \text{pu}_{L\circ M}: \quad & = \text{pu}_M \fatsemi \underline{\text{pu}_L \fatsemi \phi^{\uparrow L}} \\
\text{naturality of } \text{pu}_L: \quad & = \underline{\text{pu}_M \fatsemi \phi} \fatsemi \text{pu}_L \\
\text{identity law for } \phi: \quad & = \text{pu}_N \fatsemi \text{pu}_L \\
\text{definition of } \text{pu}_{L\circ N}: \quad & = \text{pu}_{L\circ N} \quad .
\end{aligned}$$

The next law to be shown is the composition law for $\text{mrun}\,\phi$:

$$\begin{aligned}
\text{expect this to equal } \text{ftn}_T \fatsemi \phi^{\uparrow L}: \quad & (\text{mrun}\,\phi)^{\uparrow M\uparrow L} \fatsemi \text{mrun}\,\phi \fatsemi \text{ftn}_{L\circ N} \\
\text{definitions of mrun and } \text{ftn}_{L\circ N}: \quad & = \phi^{\uparrow L\uparrow M\uparrow L} \fatsemi \underline{\phi^{\uparrow L} \fatsemi \text{sw}_L^{N\uparrow L}} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_N^{\uparrow L} \\
\text{monadic naturality of } \text{sw}_L: \quad & = \underline{\phi^{\uparrow L\uparrow M\uparrow L} \fatsemi \text{sw}_L^{M\uparrow L}} \fatsemi \phi^{\uparrow L\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_N^{\uparrow L} \\
\text{naturality of } \text{sw}_L: \quad & = \text{sw}_L^{M\uparrow L} \fatsemi \underline{\phi^{\uparrow M\uparrow L\uparrow L} \fatsemi \phi^{\uparrow L\uparrow L} \fatsemi \text{ftn}_L} \fatsemi \text{ftn}_N^{\uparrow L} \\
\text{naturality of } \text{ftn}_L: \quad & = \text{sw}_L^{M\uparrow L} \fatsemi \text{ftn}_L \fatsemi \underline{\left(\phi^{\uparrow M} \fatsemi \phi \fatsemi \text{ftn}_N\right)^{\uparrow L}} \\
\text{composition law for } \phi: \quad & = \underline{\text{sw}_L^{M\uparrow L} \fatsemi \text{ftn}_L \fatsemi \text{ftn}_M^{\uparrow L}} \fatsemi \phi^{\uparrow L} \\
\text{definition of } \text{ftn}_T: \quad & = \text{ftn}_T \fatsemi \phi^{\uparrow L} \quad .
\end{aligned}$$

***

13.6 Composed-inside transformers: Linear monads

13.7 Composed-outside transformers: Rigid monads

A monad R is **rigid** if its monad transformer can be defined by $T_R^{M,\bullet} \triangleq R^{M^\bullet}$ for a foreign monad M.

***I do not know whether one could derive the properties 1-3 from each other. However, I will show that these properties hold for the four known rigid monad constructions, namely

$$\begin{aligned} \text{contrafunctor choice}: &\quad R^A \triangleq H^A \Rightarrow A \\ \text{generalized selector}: &\quad S^A \triangleq F^{A \Rightarrow R^Q} \Rightarrow R^A \text{ if } R \text{ is rigid} \\ \text{product}: &\quad S^A \triangleq P^A \times Q^A \text{ if both } P, Q \text{ are rigid} \\ \text{composition}: &\quad S^A \triangleq P^{Q^A} \text{ if both } P, Q \text{ are rigid} \end{aligned}$$

13.7.1 Rigid monad construction 1: contrafunctor choice

The construction I call **contrafunctor choice**, $R^A \triangleq H^A \Rightarrow A$, defines a rigid monad R for *any* given contrafunctor H.

This monad's effect is to choose and return a value of type A, given a contrafunctor H that *consumes* values of type A (and presumably can check some conditions on those values). The contrafunctor H could be a constant contrafunctor $H^A \triangleq Q$, a function such as $H^A \triangleq A \Rightarrow Q$, or a more complicated contrafunctor.

Different choices of the contrafunctor H will yield simple rigid monad examples such as $R^A \triangleq 1$ (the unit monad), $R^A \triangleq A$ (the identity monad), $R^A \triangleq Z \Rightarrow A$ (the reader monad), as well as the **search**[2] **monad** $R^A \triangleq (A \Rightarrow Q) \Rightarrow A$.

The search monad represents the effect of searching for a value of type A that satisfies a condition expressed through a function of type $A \Rightarrow Q$. The simplest example of a search monad is found by setting $Q \triangleq$ Bool. One may implement a function of type $(A \Rightarrow \text{Bool}) \Rightarrow A$ that *somehow* finds a value of type A that might satisfy the given predicate of type $A \Rightarrow$ Bool. The intention is to return a value

[2] See http://math.andrej.com/2008/11/21/

that satisfies the predicate if possible. If no such value can be found, some value of type A is still returned.

A closely related monad is the search-with-falure $R^A \triangleq (A \Rightarrow \text{Bool}) \Rightarrow 1 + A$, which is of the "selector" type. This (non-rigid) monad will return an empty value $1 + 0$ if no value of type A can be found that satisfies the predicate. Given a search monad, one can compute a search-with-failure monad by checking whether the value returned by the search monad does actually satisfy the predicate.

Assume that H is a contrafunctor and M is a monad, and denote for brevity

$$T^A \triangleq R^{M^A} \triangleq H^{M^A} \Rightarrow M^A \quad .$$

We need to prove that (1) a non-degenerate fi method is defined for R^A; (2) T^A is a monad, and (3) T satisfies the compatibility laws.

Statement 2. $T^\bullet \triangleq R^{M^\bullet} \triangleq H^{M^\bullet} \Rightarrow M^\bullet$ is a monad if M is any monad and H is any contrafunctor. (If we set $M^A \triangleq A$, this will also prove that R itself is a monad.)

Proof. We need to define the monad instance for T and prove the identity and the associativity laws for T, assuming that the monad M satisfies these laws.

To define the monad instance for T, it is convenient to use the Kleisli category formulation of the monad. In this formulation, we consider Kleisli morphisms of type $A \Rightarrow T^B$ and then define the Kleisli identity morphism, $\text{pure}_T : A \Rightarrow T^A$, and the Kleisli product operation $\diamond_T$,

$$f^{:A \Rightarrow T^B} \diamond_T g^{:B \Rightarrow T^C} : A \Rightarrow T^C \quad .$$

We are then required to define the operation $\diamond_T$ and to prove identity and associativity laws for it.

We notice that since the type constructor R is itself a function type $H^A \Rightarrow A$, the type of the Kleisli morphism $A \Rightarrow T^B$ is actually $A \Rightarrow T^B \triangleq A \Rightarrow H^{M^B} \Rightarrow M^B$. While proving the monad laws for T, we will need to use the monad laws for M (since M is an arbitrary, unknown monad). In order to use the monad laws for M, it would be helpful if we had the Kleisli morphisms for M of type $A \Rightarrow M^B$ more easily available. If we flip the curried arguments of the Kleisli morphism type $A \Rightarrow H^{M^B} \Rightarrow M^B$ and instead consider the **flipped Kleisli** morphisms of type $H^{M^B} \Rightarrow A \Rightarrow M^B$, the type $A \Rightarrow M^B$ will be easier to reason about. Since the type $A \Rightarrow H^{M^B} \Rightarrow M^B$ is equivalent to $A \Rightarrow H^{M^B} \Rightarrow M^B$, any laws we prove for the flipped Kleisli morphisms will yield the corresponding laws for

the standard Kleisli morphisms. The use of flipped Kleisli morphisms makes the proof significantly shorter.

We temporarily denote by $\tilde{\text{pure}}_T$ and $\tilde{\diamond}_T$ the flipped Kleisli operations:

$$\begin{aligned} \tilde{\text{pure}}_T &: H^{M^A} \Rightarrow A \Rightarrow M^A \\ f^{:H^{M^B} \Rightarrow A \Rightarrow M^B} \tilde{\diamond}_T g^{:H^{M^C} \Rightarrow B \Rightarrow M^C} &: H^{M^C} \Rightarrow A \Rightarrow M^C \quad . \end{aligned}$$

To define the operations $\tilde{\text{pure}}_T$ and $\tilde{\diamond}_T$, we may use the methods pure_M and flm_M as well as the Kleisli product $\diamond_M$ for the given monad M. The definitions are

$$\begin{aligned} \tilde{\text{pure}}_T &= q \Rightarrow \text{pure}_M \quad \text{(the argument } q \text{ is unused)}, \\ f \tilde{\diamond}_T g &= q \Rightarrow (f\ p) \diamond_M (g\ q) \quad \text{where} \\ p^{:H^{M^B}} &= (\text{flm}_M\ (g\ q))^{\downarrow H}\ q \quad . \end{aligned}$$

This definition works by using the Kleisli product $\diamond_M$ on values $f\ p : A \Rightarrow M^B$ and $g\ q : B \Rightarrow M^C$. To obtain a value $p : H^{M^B}$, we use the function $\text{flm}_M\ (g\ q) : M^B \Rightarrow M^C$ to H-contramap $q : H^{M^C}$ into $p : H^{M^B}$.

Written as a single expression, the definition of $\tilde{\diamond}_T$ is

$$f \tilde{\diamond}_T g = q \Rightarrow f \left((\text{flm}_M\ (g\ q))^{\downarrow H}\ q \right) \diamond_M (g\ q) \quad . \tag{13.15}$$

Checking the left identity law:

$$\begin{aligned} & \tilde{\text{pu}}_T \tilde{\diamond}_T g \\ \text{definition of } \tilde{\diamond}_T : \quad & = q \Rightarrow \tilde{\text{pu}}_T \left((\text{flm}_M\ (g\ q))^{\downarrow H}\ q \right) \diamond_M (g\ q) \\ \text{definition of } \tilde{\text{pure}}_T : \quad & = q \Rightarrow \text{pu}_M \diamond_M g\ q \\ \text{left identity law for } M : \quad & = q \Rightarrow g\ q \\ \text{function expansion} : \quad & = g \end{aligned}$$

Checking the right identity law:

$$
\begin{aligned}
& f \tilde{\diamond}_T \tilde{\text{pure}}_T \\
\text{definition of } \tilde{\diamond}_T : \quad & = q \Rightarrow f \left(\left(\text{flm}_M \left(\tilde{\text{pure}}_T q \right) \right)^{\downarrow H} q \right) \diamond_M \left(\tilde{\text{pure}}_T q \right) \\
\text{definition of } \tilde{\text{pure}}_T : \quad & = q \Rightarrow f \left(\left(\text{flm}_M \left(\text{pure}_M \right) \right)^{\downarrow H} q \right) \diamond_M \text{pure}_M \\
\text{right identity law for } M : \quad & = q \Rightarrow f \left((\text{id})^{\downarrow H} q \right) \\
\text{identity law for } H : \quad & = q \Rightarrow f \, q \\
\text{function expansion} : \quad & = f
\end{aligned}
$$

Checking the associativity law: $(f \tilde{\diamond}_T g) \tilde{\diamond}_T h$ must equal $f \tilde{\diamond}_T (g \tilde{\diamond}_T h)$. We have

$$
\begin{aligned}
& (f \tilde{\diamond}_T g) \tilde{\diamond}_T h \\
& = \left(r \Rightarrow f \left((\text{flm}_M \, (g \, r))^{\downarrow H} r \right) \diamond_M (g \, r) \right) \tilde{\diamond}_T h \\
& = q \Rightarrow f \left((\text{flm}_M \, (g \, r))^{\downarrow H} r \right) \diamond_M (g \, r) \diamond_M (h \, q) \quad \text{where} \\
& \qquad r \triangleq (\text{flm}_M \, (h \, q))^{\downarrow H} q \quad ;
\end{aligned}
$$

while

$$
\begin{aligned}
& f \tilde{\diamond}_T (g \tilde{\diamond}_T h) \\
& = f \tilde{\diamond}_T \left(q \Rightarrow g \left((\text{flm}_M \, (h \, q))^{\downarrow H} q \right) \diamond_M (h \, q) \right) \\
& = q \Rightarrow f \left((\text{flm}_M \, k)^{\downarrow H} q \right) \diamond_M k \quad \text{where} \\
& \qquad r \triangleq (\text{flm}_M \, (h \, q))^{\downarrow H} q \quad \text{and} \\
& \qquad k \triangleq (g \, r) \diamond_M (h \, q) \quad .
\end{aligned}
$$

It remains to show that the following two expressions are equal,

$$
\begin{aligned}
& f \left((\text{flm}_M \, (g \, r))^{\downarrow H} r \right) \diamond_M (g \, r) \diamond_M (h \, q) \quad \text{and} \\
& f \left((\text{flm}_M \, ((g \, r) \diamond_M (h \, q)))^{\downarrow H} q \right) \diamond_M (g \, r) \diamond_M (h \, q) \, , \quad \text{where} \\
& \qquad r \triangleq (\text{flm}_M \, (h \, q))^{\downarrow H} q \quad .
\end{aligned}
$$

These long expressions differ only by the following two sub-expressions, namely

$$
(\text{flm}_M \, (g \, r))^{\downarrow H} r
$$

and

$$(\text{flm}_M\ ((g\,r) \diamond_M (h\,q)))^{\downarrow H}\ q \quad ,$$

where $r \triangleq (\text{flm}_M\ (h\,q))^{\downarrow H}\ q$. Writing out the value r in the last argument of $(\text{flm}_M\ (g\,r))^{\downarrow H}\ r$ but leaving r unexpanded everywhere else, we now rewrite the differing sub-expressions as

$$(\text{flm}_M\ (g\,r))^{\downarrow H}\ (\text{flm}_M\ (h\,q))^{\downarrow H}\ q \quad \text{and}$$
$$(\text{flm}_M\ ((g\,r) \diamond_M (h\,q)))^{\downarrow H}\ q \quad .$$

Now it becomes apparent that we need to put the two "flm_M"s closer together and to combine them by using the associativity law of the monad M. Then we can rewrite the first sub-expression and transform it into the second one:

$$\begin{aligned}
& (\text{flm}_M\ (g\,r))^{\downarrow H}\ (\text{flm}_M\ (h\,q))^{\downarrow H}\ q \\
\text{composition law for } H: \quad & = (\text{flm}_M\ (g\,r) \fatsemi \text{flm}_M\ (h\,q))^{\downarrow H}\ q \\
\text{associativity law for } M: \quad & = (\text{flm}_M\ ((g\,r) \fatsemi \text{flm}_M\ (h\,q)))^{\downarrow H}\ q \\
\text{definition of } \diamond_M \text{ via } \text{flm}_M: \quad & = (\text{flm}_M\ ((g\,r) \diamond_M (h\,q)))^{\downarrow H}\ q \quad .
\end{aligned}$$

This proves the associativity law for $\tilde{\diamond}_T$.

Statement 3. The monad instance for T defined in Statement 3 can be defined equivalently as

$$\begin{aligned}
\text{pure}_T\ \left(a^{:A}\right) &: \ H^{M^A} \Rightarrow M^A \\
\text{pure}_T\ a &\triangleq _ \Rightarrow \text{pure}_M\ a \\
\text{flm}_T\ (f^{:A \Rightarrow H^{M^B} \Rightarrow M^B}) &: \ \left(H^{M^A} \Rightarrow M^A\right) \Rightarrow H^{M^B} \Rightarrow M^B \\
\text{flm}_T\ f &\triangleq t^{:R^{M^A}} \Rightarrow q^{:H^{M^B}} \Rightarrow p^{\uparrow R}\ t\ q \quad \text{where} \\
p^{:M^A \Rightarrow M^B} &\triangleq \text{flm}_M\ \left(x^{:A} \Rightarrow f\ x\ q\right) \quad .
\end{aligned}$$

Proof. The definition of $\tilde{\diamond}_T$ in Statement 3 used the flipped types of Kleisli morphisms, which is not the standard way of defining the methods of a monad. To restore the standard types, we need to unflip the arguments:

$$\begin{aligned}
& f^{:A \Rightarrow H^{M^B} \Rightarrow M^B} \diamond_T g^{:B \Rightarrow H^{M^C} \Rightarrow M^C} : \ A \Rightarrow H^{M^C} \Rightarrow M^C \quad ; \\
& f \diamond_T g = t \Rightarrow q \Rightarrow \left(\tilde{f}\ \left((\text{flm}_M\ (b \Rightarrow g\,b\,q))^{\downarrow H}\ q\right) \diamond_M (b \Rightarrow g\,b\,q)\right) t \quad ,
\end{aligned}$$

where $\tilde{f} \triangleq h \Rightarrow k \Rightarrow f\, k\, h$ is the flipped version of f. To replace $\diamond_M$ by flm_M, express $x \diamond_M y = x \fatsemi \text{flm}_M\, y$ to find

$$f \diamond_T g = t \Rightarrow q \Rightarrow \left(\tilde{f}\left(p^{\downarrow H} q\right) \fatsemi p\right) t \quad \text{where } p = \text{flm}_M\,(x \Rightarrow g\, x\, q) \quad .$$

To obtain an implementation of flm_T, express flm_T through $\diamond_T$ as

$$\text{flm}_T\, g^{:A \Rightarrow T^B} = \text{id}^{:T^A \Rightarrow T^A} \diamond_T g \quad .$$

Now we need to substitute $f^{:T^A \Rightarrow T^A} = \text{id}$ into $f \diamond_T g$. Noting that $\tilde{f}$ will then become

$$\tilde{f} = (h \Rightarrow k \Rightarrow \text{id}\, k\, h) = (h \Rightarrow k \Rightarrow k\, h) \quad ,$$

we get

$$\begin{aligned}
& \text{flm}_T\, g^{:A \Rightarrow T^B} = \text{id} \fatsemi \text{flm}_T\, g \\
\text{definition of } \diamond_T : \quad & = t^{:T^A} \Rightarrow q^{H^{M^B}} \Rightarrow \left(\tilde{f}\left(p^{\downarrow H} q\right) \fatsemi p\right) t \\
& \qquad \text{where } p = \text{flm}_M\,(x \Rightarrow g\, x\, q) \\
\text{substitute } f = \text{id} : \quad & = t \Rightarrow q \Rightarrow \left((h \Rightarrow k \Rightarrow k\, h)\left(p^{\downarrow H} q\right) \fatsemi p\right) t \\
\text{apply } k \text{ to } p^{\downarrow H} q : \quad & = t \Rightarrow q \Rightarrow \left(\left(k \Rightarrow k\left(p^{\downarrow H} q\right)\right) \fatsemi p\right) t \\
\text{definition of } \fatsemi : \quad & = t \Rightarrow q \Rightarrow p\left(t\left(p^{\downarrow H} q\right)\right) \quad .
\end{aligned}$$

By definition of the functor $R^A \triangleq H^A \Rightarrow A$, we lift any function $p^{:A \Rightarrow B}$ as

$$\begin{aligned}
p^{\uparrow R} &: \left(H^A \Rightarrow A\right) \Rightarrow H^B \Rightarrow B \quad , \\
p^{\uparrow R} r^{H^A \Rightarrow A} &\triangleq p^{\downarrow H} \fatsemi r \fatsemi p \\
&= q^{H^B} \Rightarrow p\left(r\left(p^{\downarrow H} q\right)\right) \quad .
\end{aligned}$$

Finally, renaming g to f, we obtain the desired code,

$$\text{flm}_T\, f = t \Rightarrow q \Rightarrow p^{\uparrow R} t\, q \quad \text{where } p \triangleq \text{flm}_M\,(x \Rightarrow f\, x\, q) \quad .$$

Statement 4. The rigid monad $R^A \triangleq H^A \Rightarrow A$ satisfies the two compatibility laws with respect to any monad M.

Proof. Denote for brevity $T^\bullet \triangleq R^{M^\bullet}$ and rewrite the compatibility laws in terms of flm instead of ftn (since our expressions for the monad T always need to involve flm_M). Then the two laws that we need to prove are

$$\text{flm}_R f^{:A \Rightarrow R^{M^B}} = \text{pure}_M^{\uparrow R} \fatsemi \text{flm}_T f \quad \text{as functions } R^A \Rightarrow R^{M^B},$$
$$(\text{flm}_M f^{A \Rightarrow M^B})^{\uparrow R} = \text{pure}_R^{\uparrow T} \fatsemi \text{flm}_T (f^{\uparrow R}) \quad \text{as functions } R^{M^A} \Rightarrow R^{M^B}.$$

A definition of flm_R is obtained from flm_T by choosing the identity monad $M^A \triangleq A$ instead of an arbitrary monad M. This replaces flm_M by id:

$$\text{flm}_T f^{:A \Rightarrow R^{M^B}} = t^{:R^{M^A}} \Rightarrow q^{H^{M^B}} \Rightarrow \left(\text{flm}_M \left(x^{:A} \Rightarrow f\, x\, q\right)\right)^{\uparrow R} t\, q \quad ;$$
$$\text{flm}_R f^{:A \Rightarrow R^B} = r^{:R^A} \Rightarrow q^{H^B} \Rightarrow \left(x^{:A} \Rightarrow f\, x\, q\right)^{\uparrow R} r\, q \quad .$$

To prove the first compatibility law, rewrite its right-hand side as

$$\begin{aligned}
& \text{pure}_M^{\uparrow R} \fatsemi \text{flm}_T f \\
\text{definition of } \text{flm}_T: \quad & = \left(r^{:R^A} \Rightarrow \text{pure}_M^{\uparrow R} r\right) \fatsemi \left(r \Rightarrow q \Rightarrow (\text{flm}_M\ (x \Rightarrow f\, x\, q))^{\uparrow R}\, r\, q\right) \\
\text{expand } \fatsemi \text{ and simplify}: \quad & = r \Rightarrow q \Rightarrow (\text{flm}_M\ (x \Rightarrow f\, x\, q))^{\uparrow R} \left(\text{pure}_M^{\uparrow R} r\right)\, q \\
\text{composition law for } R: \quad & = r \Rightarrow q \Rightarrow \left(\text{pure}_M \fatsemi \text{flm}_M\ (x \Rightarrow f\, x\, q)\right)^{\uparrow R} r\, q \\
\text{left identity law for } M: \quad & = r \Rightarrow q \Rightarrow (x \Rightarrow f\, x\, q)^{\uparrow R}\, r\, q \\
\text{definition of } \text{flm}_R: \quad & = \text{flm}_R f \quad .
\end{aligned}$$

To prove the second compatibility law, rewrite its right-hand side as

$$\begin{aligned}
& \text{pure}_R^{\uparrow T} \fatsemi \text{flm}_T (f^{\uparrow R}) \\
\text{definition of } \text{flm}_T (f^{\uparrow R}): \quad & = \left(t^{:T^A} \Rightarrow \text{pure}_R^{\uparrow T} t\right) \fatsemi \left(t \Rightarrow q \Rightarrow \left(\text{flm}_M \left(x \Rightarrow f^{\uparrow R} x\, q\right)\right)^{\uparrow R} t\, q\right) \\
\text{expand } \fatsemi \text{ and simplify}: \quad & = t^{:T^A} \Rightarrow q \Rightarrow \left(\text{flm}_M \left(x^{:R^A} \Rightarrow f^{\uparrow R} x\, q\right)\right)^{\uparrow R} \left(\text{pure}_R^{\uparrow M \uparrow R} t\right)\, q \\
\text{composition law for } R: \quad & = t \Rightarrow q \Rightarrow \left(\text{pure}_R^{\uparrow M} \fatsemi \text{flm}_M \left(x^{:R^A} \Rightarrow f^{\uparrow R} x\, q\right)\right)^{\uparrow R} t\, q \\
\text{left naturality of } \text{flm}_M: \quad & = t \Rightarrow q \Rightarrow \left(\text{flm}_M \left(x^{:A} \Rightarrow \left(\text{pure}_R \fatsemi f^{\uparrow R}\right) x\, q\right)\right)^{\uparrow R} t\, q \\
\text{naturality of } \text{pure}_R: \quad & = t \Rightarrow q \Rightarrow \left(\text{flm}_M \left(x^{:A} \Rightarrow \text{pure}_R\ (f\, x)\ q\right)\right)^{\uparrow R} t\, q \\
\text{definition of } \text{pure}_R: \quad & = t \Rightarrow q \Rightarrow \left(\text{flm}_M \left(x^{:A} \Rightarrow f\, x\right)\right)^{\uparrow R} t\, q \\
\text{unapply } f \text{ and } \text{flm}_M f: \quad & = t \Rightarrow q \Rightarrow (\text{flm}_M f)^{\uparrow R}\, t\, q = (\text{flm}_M f)^{\uparrow R} \quad .
\end{aligned}$$

Statement 5. Assume that R and M are arbitrary (not necessarily rigid) monads such that the functor composition $T^\bullet \triangleq R^{M^\bullet}$ is also a monad whose method pure_T is defined by

$$\text{pure}_{R^M} = \text{pure}_M \fatsemi \text{pure}_R \quad .$$

Then the two "liftings" and the two "runners" defined by

$$\begin{aligned} \text{lift} : M^\bullet \rightsquigarrow R^{M^\bullet} &\triangleq \text{pure}_R \quad , \\ \text{blift} : R^\bullet \rightsquigarrow R^{M^\bullet} &\triangleq \text{pure}_M^{\uparrow R} \quad , \\ \text{mrun}\,(\phi^{:M^\bullet \rightsquigarrow N^\bullet}) : R^{M^\bullet} \Rightarrow R^{N^\bullet} &\triangleq \phi^{\uparrow R} \quad , \\ \text{brun}\,\theta^{:R^\bullet \rightsquigarrow \bullet} : R^{M^\bullet} \Rightarrow M^\bullet &\triangleq \theta^{:R^{M^\bullet} \rightsquigarrow M^\bullet} \quad , \end{aligned}$$

will satisfy the identity laws

$$\begin{aligned} \text{pure}_M \fatsemi \text{lift} &= \text{pure}_T \quad , \\ \text{pure}_R \fatsemi \text{blift} &= \text{pure}_T \quad , \\ \text{pure}_{R^M} \fatsemi (\text{mrun}\,\phi) &= \text{pure}_N \fatsemi \text{pure}_R \quad , \\ \text{pure}_{R^M} \fatsemi (\text{brun}\,\theta) &= \text{pure}_M \quad , \end{aligned}$$

where we assume that $\phi : M^\bullet \rightsquigarrow N^\bullet$ and $\theta : M^\bullet \rightsquigarrow \bullet$ are monadic morphisms, and N is an arbitrary monad.

Proof. The first law is a direct consequence of the definition of lift. The second law follows from the naturality of pure_R,

$$\text{pure}_R \fatsemi \text{pure}_M^{\uparrow R} = \text{pure}_M \fatsemi \text{pure}_R \quad .$$

The third law follows from the identity law for ϕ,

$$\text{pure}_M \fatsemi \phi = \text{pure}_N ,$$

if we write

$$\begin{aligned} & \text{pure}_{R^M} \fatsemi (\text{mrun}\,\phi) \\ \text{definitions of } \text{pure}_{R^M} \text{ and mrun} : \quad & = \text{pure}_M \fatsemi \text{pure}_R \fatsemi \phi^{\uparrow R} \\ \text{naturality of } \text{pure}_R : \quad & = \text{pure}_M \fatsemi \phi \fatsemi \text{pure}_R \\ \text{identity law for } \phi : \quad & = \text{pure}_N \fatsemi \text{pure}_R \quad . \end{aligned}$$

The fourth law follows from the identity law for θ,

$$\mathrm{pure}_R \fatsemi \theta = \mathrm{id},$$

if we write

$$\begin{aligned} & \mathrm{pure}_{R^M} \fatsemi (\mathrm{brun}\,\theta) \\ \text{definitions of } \mathrm{pure}_{R^M} \text{ and brun}: \quad & = \mathrm{pure}_M \fatsemi \mathrm{pure}_R \fatsemi \theta \\ \text{identity law for } \theta: \quad & = \mathrm{pure}_M \quad . \end{aligned}$$

In this proof, we have not actually used the assumption that R, M, $R^{M^\bullet}$, and N are monads. We only used the naturality of pure_R. So, it is sufficient to assume that these functors are pointed, and that ϕ and θ are morphisms of pointed functors. However, the four laws we just derived are mainly useful as identity laws for monad transformer methods.

Statement 6. A monad transformer for the rigid monad $R^A \triangleq H^A \Rightarrow A$ is $T_R^{M,A} \triangleq R^{M^A}$, with the four required methods defined as

$$\begin{aligned} \mathrm{lift}^{:M^\bullet \rightsquigarrow T^\bullet} &\triangleq \mathrm{pure}_R \quad , \\ \mathrm{blift}^{:R^\bullet \rightsquigarrow T^\bullet} &\triangleq \mathrm{pure}_M^{\uparrow R} \quad , \\ \mathrm{mrun}\,\phi^{:M^\bullet \rightsquigarrow N^\bullet} &\triangleq \phi^{\uparrow R} \quad , \\ \mathrm{brun}\,\theta^{:R^\bullet \rightsquigarrow \bullet} &\triangleq \theta^{:R^{M^\bullet} \rightsquigarrow M^\bullet} \quad . \end{aligned}$$

Each of these four methods is a monadic morphism for any monadic morphism $\phi : M^\bullet \rightsquigarrow N^\bullet$, where M and N are arbitrary (not necessarily rigid) monads, and for any monadic morphism $\theta : R^A \rightsquigarrow A$.

Proof. Since blift is a special case of mrun, we only need to prove the monadic morphism laws for the three methods: lift, mrun, and brun.

The identity laws follow from Statement 5, since $T_R^{M,\bullet} = R^{M^\bullet}$ is a functor composition. It remains to verify the composition laws,

$$\begin{aligned} \left(f^{:A \Rightarrow M^B} \fatsemi \mathrm{lift}\right) \diamond_{R^M} \left(g^{:B \Rightarrow M^C} \fatsemi \mathrm{lift}\right) &= (f \diamond_M g) \fatsemi \mathrm{lift} \quad , \\ \left(f^{:A \Rightarrow R^{M^B}} \fatsemi \mathrm{mrun}\,\phi\right) \diamond_{R^N} \left(g^{:B \Rightarrow R^{M^C}} \fatsemi \mathrm{mrun}\,\phi\right) &= (f \diamond_{R^M} g) \fatsemi \mathrm{mrun}\,\phi \quad , \\ \left(f^{:A \Rightarrow R^{M^B}} \fatsemi \mathrm{brun}\,\theta\right) \diamond_M \left(g^{:B \Rightarrow R^{M^C}} \fatsemi \mathrm{brun}\,\theta\right) &= (f \diamond_{R^M} g) \fatsemi \mathrm{brun}\,\theta \quad . \end{aligned}$$

For the first law, we need to use the definition of lift $\triangleq \text{pure}_R$, which is

$$\text{pure}_R\, x^{:A} \triangleq \_^{:H^A} \Rightarrow x \quad .$$

So the definition of lift can be written as

$$f \fatsemi \text{lift} = x^{:A} \Rightarrow \_^{:H^{M^B}} \Rightarrow f\, x$$

as a function that ignores its second argument (of type H^{M^B}). It is convenient to use the flipped Kleisli product $\tilde{\diamond}_{R^M}$ defined before in eq. (13.15). We compute

$$\begin{aligned}
& \left(f^{:A\Rightarrow M^B} \fatsemi \text{lift}\right) \tilde{\diamond}_{R^M} \left(g^{:B\Rightarrow M^C} \fatsemi \text{lift}\right) \\
\text{definition of lift :} \quad & = (_ \Rightarrow f) \,\tilde{\diamond}_{R^M}\, (_ \Rightarrow g) \\
\text{eq. (13.15) :} \quad & = q^{:H^{M^B}} \Rightarrow (_ \Rightarrow f) \left((\text{flm}_M\, ((_ \Rightarrow g)\; q))^{\downarrow H}\; q\right) \diamond_M\, ((_ \Rightarrow g)\; q) \\
\text{expanding the functions :} \quad & = q \Rightarrow f \diamond_M g \\
\text{unflipping the definition of lift :} \quad & = q \Rightarrow x \Rightarrow ((f \diamond_M g) \fatsemi \text{lift})\, a\, x \quad .
\end{aligned}$$

Note that the last function, of type $H^{M^B} \Rightarrow A \Rightarrow M^B$, ignores its first argument. Therefore, after unflipping this Kleisli function, it will ignore its second argument. This is precisely what the function $(f \diamond_M g) \fatsemi \text{lift}$ must do.

For the second and third laws, we need to use the composition laws for ϕ and θ, which can be written as

$$\begin{aligned}
\left(f^{:A\Rightarrow M^B} \fatsemi \phi\right) \diamond_N \left(g^{:B\Rightarrow M^C} \fatsemi \phi\right) &= (f \diamond_M g) \fatsemi \phi \quad , \\
\left(f^{:A\Rightarrow R^B} \fatsemi \theta\right) \fatsemi \left(g^{:B\Rightarrow M^C} \fatsemi \theta\right) &= (f \diamond_M g) \fatsemi \theta \quad .
\end{aligned}$$

13.7.2 Rigid monad construction 2: composition

Statement 1. The composition $R_1^{R_2^\bullet}$ is a rigid functor if both R_1 and R_2 are rigid functors.

Proof. We will show that the non-degeneracy law $\text{fi}_T \fatsemi \text{fo}_T = \text{id}$ holds for the rigid functor $T^\bullet \triangleq R^{M^\bullet}$ as long as M is rigid.

Since it is given that M is rigid, we may use its method fi_M satisfying the non-degeneracy law $\text{fo}_M\, (\text{fi}_M\, f) = f$.

Flip the curried arguments of the function type $A \Rightarrow T^B \triangleq A \Rightarrow H^{M^B} \Rightarrow M^B$, to obtain $H^{M^B} \Rightarrow A \Rightarrow M^B$, and note that $A \Rightarrow M^B$ can be mapped to $M^{A\Rightarrow B}$ using

fi_M. So we can implement $\tilde{\text{fi}}_T$ using fi_M:

$$\tilde{\text{fi}}_T : (H^{M^B} \Rightarrow A \Rightarrow M^B) \Rightarrow H^{M^{A \Rightarrow B}} \Rightarrow M^{A \Rightarrow B}$$
$$\tilde{\text{fi}}_T = f \Rightarrow h \Rightarrow \text{fi}_M \left(f ((b \Rightarrow \_ \Rightarrow b)^{\uparrow M \downarrow H} h) \right)$$
$$\tilde{\text{fo}}_T : \left(H^{M^{A \Rightarrow B}} \Rightarrow M^{A \Rightarrow B} \right) \Rightarrow H^{M^B} \Rightarrow A \Rightarrow M^B$$
$$\tilde{\text{fo}}_T = g \Rightarrow h \Rightarrow a \Rightarrow \text{fo}_M \left(g (\left(p^{:A \Rightarrow B} \Rightarrow p\, a \right)^{\uparrow M \downarrow H} h) \right) a$$

To show the non-degeneracy law for T, compute

$$\begin{aligned}
& \tilde{\text{fo}}_T \left(\tilde{\text{fi}}_T f \right) h^{:H^{M^B}} a^{:A} \\
\text{insert the definition of } \tilde{\text{fo}}_T: \quad & = \text{fo}_M \left(\left(\tilde{\text{fi}}_T f \right) ((p \Rightarrow p\, a)^{\uparrow M \downarrow H} h) \right) a \\
\text{insert the definition of } \tilde{\text{fi}}_T: \quad & = \text{fo}_M \left(\text{fi}_M \left(f ((b \Rightarrow _ \Rightarrow b)^{\uparrow M \downarrow H} (p \Rightarrow p\, a)^{\uparrow M \downarrow H} h) \right) \right) a \\
\text{nondegeneracy law for } \text{fi}_M: \quad & = f ((b \Rightarrow _ \Rightarrow b)^{\uparrow M \downarrow H} (p \Rightarrow p\, a)^{\uparrow M \downarrow H} h) a \\
\text{composition laws for } M, H: \quad & = f ((b \Rightarrow _ \Rightarrow b \fatsemi p \Rightarrow p\, a)^{\uparrow M \downarrow H} h) a \\
\text{simplify :} \quad & = f \left((b \Rightarrow b)^{\uparrow M \downarrow H} h \right) a \\
\text{identity laws for } M, H: \quad & = f\, h\, a \quad .
\end{aligned}$$

We obtained $\tilde{\text{fo}}_T \left(\tilde{\text{fi}}_T\, f \right) h\, a = f\, h\, a$. Therefore the non-degeneracy law $\tilde{\text{fi}}_T \fatsemi \tilde{\text{fo}}_T = \text{id}$ holds.

Statement 2. The composition $R_1^{R_2^\bullet}$ has property 2 of a rigid monad if both R_1 and R_2 have that property.

Proof. To prove property 2 for $R_1^{R_2^\bullet}$, first consider that $R_1^{R_2^\bullet}$ is itself a monad, which is property 2 of the rigid monad R_1. Now consider an arbitrary monad M and the type constructor $Q^\bullet \triangleq R_1^{R_2^{M^\bullet}}$. Property 2 of rigid monads requires that Q be a monad. Since R_2 is rigid, $R_2^{M^\bullet}$ is a monad, and since R_1 is rigid, Q is a monad.

To prove property 3 for $R_1^{R_2^\bullet}$, assume that this property already holds for R_1 and R_2 separately, and consider an arbitrary monad M and the type constructor $Q^\bullet \triangleq R_1^{R_2^{M^\bullet}}$. ???

13.7.3 Rigid monad construction 3: product

The product of rigid monads, $R_1^A \times R_2^A$, is a rigid monad.

13.7.4 Rigid monad construction 4: selector monad

The **selector monad** $S^A \triangleq F^{A \Rightarrow R^Q} \Rightarrow R^A$ is rigid if $R^\bullet$ is a rigid monad, $F^\bullet$ is any functor, and Q is any fixed type.

13.8 Irregular monad transformers

13.8.1 The state monad transformer `StateT`

13.8.2 The continuation monad transformer `ContT`

13.8.3 The codensity monad transformer `CodT`

The **codensity monad** over a functor F is defined as

$$\text{Cod}^{F,A} \triangleq \forall B.\, \left(A \Rightarrow F^B\right) \Rightarrow F^B$$

Properties:

- $\text{Cod}^{F,\bullet}$ is a monad for any functor F
- If $F^\bullet$ is itself a monad then we have monadic morphisms $\text{inC} : F^\bullet \rightsquigarrow \text{Cod}^{F,\bullet}$ and $\text{outC} : \text{Cod}^{F,\bullet} \rightsquigarrow F^\bullet$ such that $\text{inC} \fatsemi \text{outC} = \text{id}$
- A monad transformer for the codensity monad is

$$T^{M,A}_{\text{Cod}} = \forall B.\, \left(A \Rightarrow M^{F^B}\right) \Rightarrow M^{F^B}$$

  However, this transformer does not have the base lifting morphism

$$\text{blift} : \left(\forall B.\, \left(A \Rightarrow F^B\right) \Rightarrow F^B\right) \Rightarrow \forall B.\, \left(A \Rightarrow M^{F^B}\right) \Rightarrow M^{F^B}$$

  since this type signature cannot be implemented. The codensity transformer also does not have any of the required "runner" transformations mrun and brun,

$$\text{mrun} : (M^\bullet \rightsquigarrow N^\bullet) \Rightarrow \left(\forall B.\, \left(A \Rightarrow M^{F^B}\right) \Rightarrow M^{F^B}\right) \Rightarrow \forall B.\, \left(A \Rightarrow N^{F^B}\right) \Rightarrow N^{F^B} \quad ,$$
$$\text{brun} : \left(\left(\forall B.\, \left(A \Rightarrow F^B\right) \Rightarrow F^B\right) \Rightarrow A\right) \Rightarrow \left(\forall B.\, \left(A \Rightarrow M^{F^B}\right) \Rightarrow M^{F^B}\right) \Rightarrow M^A \quad .$$

13.9 Discussion

13.10 Rigid functors

The properties of rigid monads can be extended to a slightly larger class of rigid functors.

R is a **rigid functor** if there exists a natural transformation `fuseIn` (short notation "fi_R") with the type signature

$$\text{fi}_R : \left(A \Rightarrow R^B\right) \Rightarrow R^{A \Rightarrow B} \quad .$$

Note that any functor admits the natural transformation `fuseOut` (short notation fo_R), defined by

$$\text{fo} : R^{A \Rightarrow B} \Rightarrow A \Rightarrow R^B \quad ,$$
$$\text{fo}\,(r) = a \Rightarrow \left(f^{:A \Rightarrow B} \Rightarrow f\,(a)\right)^{\uparrow R} r \quad .$$

The method `fuseIn` must satisfy the nondegeneracy law

$$\text{fi}_R \fatsemi \text{fo}_R = \text{id} \quad .$$

***

Statement 1. The nondegeneracy law $\text{fi} \fatsemi \text{fo} = \text{id}$ holds for the rigid functor $R^A \triangleq H^A \Rightarrow A$ where H^A is any contrafunctor.

Proof. The fi method for R is defined by

$$\text{fi} : (A \Rightarrow H^B \Rightarrow B) \Rightarrow H^{A \Rightarrow B} \Rightarrow A \Rightarrow B$$
$$\text{fi} = f^{:A \Rightarrow H^B \Rightarrow B} \Rightarrow h^{:H^{A \Rightarrow B}} \Rightarrow a \Rightarrow f(a)(\,(b \Rightarrow \_ \Rightarrow b)^{\downarrow H}\, h)$$

It is easier to manipulate the methods fi and fo if we flip the curried arguments of the function type $A \Rightarrow R^B \triangleq A \Rightarrow H^B \Rightarrow B$ and instead consider the equivalent methods $\tilde{\text{fi}}$ and $\tilde{\text{fo}}$ defined by

$$\tilde{\text{fi}} : (H^B \Rightarrow A \Rightarrow B) \Rightarrow H^{A \Rightarrow B} \Rightarrow A \Rightarrow B$$
$$\tilde{\text{fi}} = f^{:H^B \Rightarrow A \Rightarrow B} \Rightarrow h^{:H^{A \Rightarrow B}} \Rightarrow f(\,(b \Rightarrow \_ \Rightarrow b)^{\downarrow H}\, h)$$
$$\tilde{\text{fo}} : \left(H^{A \Rightarrow B} \Rightarrow A \Rightarrow B\right) \Rightarrow H^B \Rightarrow A \Rightarrow B$$
$$\tilde{\text{fo}} = g^{:H^{A \Rightarrow B} \Rightarrow A \Rightarrow B} \Rightarrow h^{:H^B} \Rightarrow a^{:A} \Rightarrow g(\,\left(p^{:A \Rightarrow B} \Rightarrow p\,a\right)^{\downarrow H}\, h)a$$

To show the non-degeneracy law for R, compute

$$\begin{aligned}
& \tilde{\text{fo}}\left(\tilde{\text{fi}}\, f\right) h^{:H\,B}\, a^{:A} \\
\text{definition of } \tilde{\text{fo}}: \quad & = \left(\tilde{\text{fi}}\, f\right)\left((p \Rightarrow p\, a)^{\downarrow H}\, h\right) a \\
\text{definition of } \tilde{\text{fi}}: \quad & = f\left((b \Rightarrow _ \Rightarrow b)^{\downarrow H}\, (p \Rightarrow p\, a)^{\downarrow H}\, h\right) a \\
\text{composition law for } H: \quad & = f\left((b \Rightarrow _ \Rightarrow b \fatsemi p \Rightarrow p\, a)^{\downarrow H}\, h\right) a \\
\text{simplify } b \Rightarrow _ \Rightarrow b \fatsemi p \Rightarrow p\, a \text{ to } : \quad & = f\left((b \Rightarrow b)^{\downarrow H}\, h\right) a \\
\text{identity law for } H: \quad & = f\, h\, a \quad .
\end{aligned}$$

We obtained $\tilde{\text{fo}}\left(\tilde{\text{fi}}\, f\right) h\, a = f\, h\, a$, so the non-degeneracy law $\tilde{\text{fi}} \fatsemi \tilde{\text{fo}} = \text{id}$ holds.

14 Recursive types

14.1 Fixpoints and type recursion schemes

14.2 Row polymorphism and OO programming

14.3 Column polymorphism

14.4 Discussion

15 Co-inductive typeclasses. Comonads

16 Irregular typeclasses*

16.1 Distributive functors

16.2 Monoidal monads

16.3 Lenses and prisms

16.4 Discussion

17 Conclusions and discussion

18 Essay: Software engineers and software artisans

Published on 2018-07-23

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.

18.1 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 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 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 advanced calculus, linear algebra, as well as several areas of physics such as electrodynamics and quantum physics, and uses calculations to guide the design of 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" (used in calculations of elasticity of materials), "Lagrangian with non-holonomic constraints" (used in robotics), the "Gibbs free energy" (for chemical reactor design), or the "Fourier transform of the delta function" and the "inverse Z-transform" (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. Nevertheless, engineering is crucially based on natural science and mathematics for some of its decision-making about new designs.

18.2 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, would begin by copying some drawings and then cutting leather in a home workshop. Apprenticeships proceed via learning by doing while listening to comments and instructions from an expert. After a few years of apprenticeship (for example, a painter apprenticeship in California 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 any academic study because there is no formal theory from which to proceed. To be sure, there is *a lot* to learn in the crafts, and it takes a large amount of 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.

18.3 Programmers today are artisans, not engineers

Now I will argue that programmers are *not engineers* in the sense we normally see the engineering professions.

18.3.1 No requirement of formal study

According to this recent Stack Overflow survey, about half of the programmers do not have a degree in Computer Science. I am one myself; my degrees are in

physics, and I have never formally studied computer science. I took no academic courses in algorithms, data structures, computer networks, compilers, programming languages, or any other topics ordinarily included in the academic study of "computer science". None of the courses I took at university or at graduate school were geared towards programming. I am a completely self-taught software developer.

There is a large number of successful programmers who *never* studied at a college, or perhaps never studied formally in any sense. They acquired all their knowledge and skills through self-study and practical work. Robert C. Martin is one such prominent example; an outspoken guru in the arts of programming who has seen it all, he routinely refers to programmers as artisans 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 not to mathematicians, physicists, or engineers of any kind. According to one of his blog posts, 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 (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 is a description that 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" – vocational schools that prepare new programmers very quickly, with no formal theory or mathematics involved, through purely practical training. These vocational schools are successful 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 designing, say, quantum computers, without ever having studied quantum physics or calculus.

18.3.2 No mathematical formalism to guide 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 books on software engineering even have no program code in them – just words and illustrative diagrams. These books talk about how programmers should approach their job, how to organize the work flow and the code architecture, 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".

In contrast, every textbook on mechanical engineering or electrical engineering has a significant amount of mathematics in it. The design of a microwave antenna is guided not by the principle of "serving a single responsibility" but by calculations of wave propagation, based on theoretical 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 while 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 algorithms, including mathematical ones. 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 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 specification. (A book with that title also exists, and similarly disappoints.) Programmers today are mostly 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 (the so-called "big O notation"), and formal verification. But programs are analyzed 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 analogous to writing a mathematical equation for the surface of a shoe made by a fashion designer. True, the "shoe surface equations" are mathematically unambiguous 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.

18.3.3 Programmers avoid academic terminology

Programmers appear to be taken aback by such terminology as "*functor*", "*monad*", or "*lambda-functions*".

> Those fancy wordsused by functional programmers purists really annoy me. Monads, functors... Nonsense!!!

In my experience, only a tiny minority of programmers actually complain about this. The vast majority has never heard these words and are unaware of functors or monads.

However, chemical engineers do not wince at "phase diagram" or "Gibbs free energy", and apparently accept the need for studying differential equations. Electrical engineers do not complain that the word "Fourier" is foreign and difficult to spell, or that "delta-function" is such a weird thing to say. Mechanical engineers take it for granted that they need to calculate with "tensors" and "Lagrangians" and "non-holonomic constraints". Actually, it seems that the arcane

terminology is the least of their difficulties! Their textbooks are full of complicated equations and long, difficult 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 actually *engineers*. True software engineers' textbooks would be full of equations and derivations, which would be used to perform calculations required *before* starting to write code.

18.4 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 automatically make the *process of programming* into 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 would be achieved if we had theory that guides and informs our 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 **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 or BayHac, 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.

Figure 18.1: Example calculation in functional type theory.

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" or the "free monad" – and illustrates its use for practical coding.

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. 18.1). The results of those calculations will help design the code fragment the programmer is about to write. This activity is quite similar to that of an engineer who first performs some mathematical calculations and only then embarks on a real-life 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; a couple of years later, a combined free applicative / free monad data type was designed and its implementation proposed in Scala as well as in Haskell. This technique allows programmers to work with declarative side-effect computations where some parts are sequential but other parts can be 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.

Another example of a development in functional type theory is the so called "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. 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).

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 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.

18.5 Do we need software *engineering*, or are artisans good enough?

The demand for programmers is growing. "Software developer" was #1 best job in the US in 2018. But is there a demand for engineers, or just for artisans?

We do not seem to be able to train enough software artisans. Therefore, it is probably impossible to train as many software engineers in the true sense of the word. Modern courses in Computer Science do not actually train engineers in that sense; at best, they train academics who act as software artisans when writing code. The few existing true software *engineers* are all self-taught. 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 "becomes rotten with time", programs grow in size "out of control", and operating systems have been notorious for ever-appearing security flaws despite many thousands of programmers and testers employed. I think this shows we are overestimating the artisanal creative capacity of the human brain.

It is precisely in designing very large and robust software systems that we would benefit from true engineering. Consider that humanity has been using chemical reactions and building bridges by trial, error, and adherence to tradition, long before mechanical or chemical engineering disciplines were developed and founded upon rigorous theory. Once the theory became available, human-

ity proceeded to create unimaginably more complicated and powerful structures and devices than ever before.

For building 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 transforming artisanal programming into a proper engineering discipline. Functional type theory and functional programming are first steps in that direction.

19 Essay: Towards functional data engineering with Scala

Published on 2017-09-29

Data engineering is among the most in-demand 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 for crafting the foundational data engineering technologies such as Spark or Akka? Why is Scala in such demand within the world of big data software?

There are reasons to believe that the choice of Scala was quite far from pure happenstance.

19.1 Data is math

Humanity has been working with data at least since Babylonian tax tables and the ancient Chinese number books. Mathematics summarizes several millennia's worth of data processing experience into a few tenets:

- Data is *immutable*, because facts are immutable.
- Each *type* of values – population count, land area, distances, prices, dates, times, – needs to be handled separately; e.g. it is meaningless to add a distance to a population count.
- Data processing is to be codified by *mathematical formulas*.

Violating these tenets produces nonsense (see Fig. 19.1).

The power of the basic principles of mathematics extends over all epochs and all cultures; they are the same in Rio de Janeiro, in Kuala-Lumpur, and even in Pyongyang (see Fig. 19.2).

Figure 19.1: A nonsensical calculation arises when mixing incompatible data types.

19.2 Functional programming is math

The functional programming paradigm is based on similar 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. A flexible system of data types helps programmers automatically prevent many kind of coding errors. In addition, modern 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 several millennia of mathematical experience.

A prominent example of how mathematics informs the design of programming languages is the connection between constructive logic and the programming language's type system, called the Curry-Howard (CH) correspondence. 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** to the logic if for any two 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, Rust, Elm, 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.

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.

19.3 The power of abstraction

Data engineering at scale poses problems of such complexity that many software companies adopt functional programming languages as their main implementation tool. Netflix, LinkedIn, Twitter started using Scala early on and were able to reap the benefits of the powerful abstractions Scala affords, such as asynchronous streams and parallelized functional collections. In this way, Scala enabled these businesses to engineer and scale up their massively concurrent computations. What exactly makes Scala suitable for big data processing?

The only way to manage 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, Apache Storm, Akka Streams, and Apache Flink. While these abstractions are certainly implementable in Java or Python, true declarative and type-safe usage is possible only

Figure 19.2: The Pyongyang method of error-free programming.

in a programming language with a sufficiently sophisticated functional type system. Among the currently available mature functional languages, only Scala and Haskell would be technically adequate for that task, due to their support for typeclasses and higher-order generic collections.

It remains to see why Scala became the *lingua franca* of big data and not, say, Haskell.

19.4 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 strug-

gle 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 Apple's Cocoa/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 requirement 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 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, Scala and Haskell are the only ones that directly support typeclasses and higher-order generics, which are necessary for expressing abstractions such as an automatically parallelized data set or an asynchronous data stream.

To see the impact of these advanced features of Scala and Haskell, consider LINQ, a domain-specific language for database queries on .NET, implemented in C# and F# through a special 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 are impossible to implement in languages with less powerful type systems.

19.5 Conclusion

The decisive advantages of Scala over other contenders (such as OCaml, Haskell, F#, Swift, or Rust) are

1. functional collections in the standard library;
2. a highly sophisticated type system, with support for typeclasses and higher-order generics;
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.

20 "Applied functional type theory": A proposal

What exactly is the extent of "theory" that a practicing functional programmer should know in order to be effective at writing code in the functional paradigm? In my view, this question is not yet resolved. In this book, I present a coherent body of theoretical knowledge that I believe fits the description of "practicable functional programming theory". This body of knowledge is, or should be, understood as a branch of computer science, and I propose to call it **applied functional type theory** (AFTT). This is the area of theoretical computer science that should serve the needs of functional programmers working as software engineers.

It is for these practitioners (I am one myself), rather than for academic researchers, that I set out to examine the functional programming inventions over the last 30 years, – such as the "functional pearls" papers – 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.

20.1 AFTT is not yet part of computer science curricula

Traditional courses of theoretical computer science (algorithms and data structures, complexity theory, distributed systems, formal languages, formal semantics, compiler techniques, databases, networks, operating systems) are largely not relevant to AFTT.

Here is 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 courses teach these theories.

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 will actually help a programmer to write code. So all these theories are not within the scope of AFTT.

As an example of theoretical material that *is* within the scope of AFTT, consider the equational laws imposed on 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 can specify declaratively a set of operations that do not depend on each other's effects. Programs can then easily combine these operations and effects, for example, in order 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 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.

This work is done using pen and paper, 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 known 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 for writing code, in applications such as parser combinators, compilers, and domain-specific languages for parallel computations. However, applicative functors are not a feature of Haskell: they are the same in Scala, OCaml, or any other functional programming language. And yet, no standard computer science textbook defines applicative functors, motivates their laws, explores their structure

on basic examples, or shows data structures that are *not* applicative functors and explains why. (Books on category theory and type theory also do not mention applicative functors.)

20.2 AFTT is not category theory, type theory, or formal logic

So far it appears that AFTT includes a selection of certain areas of category theory, formal logic, and type theory. However, software engineers would not derive much benefit from following traditional academic courses in these subjects, because their presentation is too abstract and at the same time lacks specific results necessary for practical software engineering. In other words, the traditional academic courses answer questions that academic computer scientists have, not questions that software engineers have.

There exist several books intended as presentations of category theory "for computer scientists" or "for programmers". However, these books fail to give examples vitally relevant to everyday programming, such as applicative or traversable functors. Instead, these books contain purely theoretical topics such as limits, adjunctions, or toposes, – concepts 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 the property Y hold in non-small categories, or only in the category of sets". Typical 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 proper 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 foldable, 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 `fold`, `filter`, `zip`, and `traverse` – operations that functional programmers use often 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 very hands-on and narrow in scope, limited to one or two specific questions and specific applications. Such tutorials usually do not present a sufficiently broad picture and do not illustrate deeper connections between these mathematical constructions.

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 required properties for a piece of code to be a valid implementation of the free monad.

Without knowledge of mathematical principles behind free monads, a programmer cannot make sure that a given implementation is correct. However, books on category theory present free monads in a way that is unsuitable for programming applications: a free monad is just an adjoint functor to a forgetful functor into the category of sets.[1] This definition is too abstract and, for instance, cannot be used to check whether a given implementation of the free monad in Scala is correct.

Perhaps the best selection of AFTT tutorial material can be found in the Haskell Wikibooks. 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 proofs or derivations of equational laws.

Apart from referring to some notions from category theory, AFTT also uses some concepts from type theory and formal logic. However, existing textbooks on type theory and formal logic focus on domain theory and proof theory – which is a lot of information that practicing programmers will have difficulty assimilating and yet will have no chance of ever applying in their daily work. At the same time, these books never mention practical techniques used in many functional programming libraries today, such as quantified types, types parameterized by type constructors, or partial type-level functions (known as "typeclasses").

Type theory and formal logic can, in principle, help the programmer with certain practical tasks, such as:

- deciding whether two data structures are equivalent as types, and implementing the isomorphism transformation; for example, the Scala type `(A, Either[B, C])` is equivalent to `Either[(A, B), (A, C)]`

[1] "What's your problem?" as the joke would go.

- detecting whether a definition of a recursive type is "reasonable", i.e. does not lead to a useless infinite recursion; an example of a useless recursive type definition in Scala is `case class Bad(x: Bad)`
- deriving an implementation of a function from its type signature and required laws; for example, deriving the `flatMap` method for the `Reader` monad from the type signature

  ```
  def flatMap[Z, A, B](r: Z => A)(f: A => Z => B): Z => B
  ```

  and verifying that the monad laws hold
- deciding whether a generic pure function with a given signature can be implemented; for example, `def f[A, B]: (A => B) => A` cannot be implemented but `def g[A, B]: A => (B => A)` can be implemented

I mention these practical tasks as examples because they are actual real-world-coding applications of domain theory and the Curry-Howard correspondence theory. However, existing books on type theory and logic do not give practical recipes for resolving these questions.

On the other hand, books such as "Scala with Cats" and "Functional programming, simplified" are focused on explaining the practical aspects of programming and do not adequately treat the equational laws that the mathematical structures require (such as the laws for applicative or monadic functors).

The only existing Scala-based AFTT textbook aiming at the proper scope is the Bjarnason-Chiusano book, which balances practical considerations with theoretical developments such as equational laws. This book is written at about the same level but goes deeper into the mathematical foundations and at the same time gives a wider range of examples.

This book is an attempt to delineate the proper scope of AFTT and to develop a rigorous yet clear and approachable presentation of the chosen material.

A Notations

I chose certain notations in this book to be different from the notations currently used in the functional programming community. The proposed notation seems to be well adapted to reasoning about types and code, and especially for designing data types and proving the equational laws of typeclasses.

A.1 Summary table

F^A type constructor F with type argument A

$x^{:A}$ value x has type A

$A + B$ a disjunctive type; in Scala, this type is `Either[A, B]`

$A \times B$ a product type; in Scala, this type is `(A,B)`

$A \Rightarrow B$ the function type, mapping from A to B

$x^{:A} \Rightarrow f$ a nameless function (as a value); in Scala, `{x:A => f}`

id the identity function

1 the unit type or its value; in Scala, `Unit` and `()`

0 the void type; in Scala, `Nothing`

$\triangleq$ "equal by definition"

$\cong$ "equivalent" according to an established isomorphism of types

$A^{:F^B}$ type annotation, used for defining unfunctors (GADTs)

$\mathbf{fmap}_F$ the standard method fmap pertaining to a functor F

$\mathbf{pu}_F$ the standard method `pure` of a monad F

$F^\bullet$ 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

$\forall A.P^A$ a universally quantified type expression

$\exists A.P^A$ an existentially quantified type expression

$\fatsemi$ the forward composition of functions: $f \fatsemi g$ is $x \Rightarrow g(f(x))$.

$\circ$ the backward composition of functions: $f \circ g$ is $x \Rightarrow f(g(x))$.

$\circ$ functor composition: $F \circ G$ is $F^{G^\bullet}$

$f^{\uparrow G}$ a function f lifted to a functor G; same as $\text{fmap}_G f$

$f^{\uparrow G \uparrow H}$ a function lifted first to G and then to H; In Scala, `h.map(_.map(f))`

$f^{\downarrow H}$ a function f lifted to a contrafunctor

$\diamond_M$ the Kleisli product operation for the monad M

A.2 Explanations

F^A means a type constructor F with a type parameter A. In Scala, this is `F[A]`.

$x^{:A}$ means a value x that has type A. The colon, :, in the superscript is to show that it is not a type argument. In Scala, this is `x : 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.

$A + B$ means the disjunctive type made of types A and B (or, a disjunction of A and B). In Scala, this is the type `Either[A, B]`.

$A \times B$ means the product type made of types A and B. In Scala, this is the tuple type `(A,B)`.

$A \Rightarrow B$ means a function type from A to B. In Scala, this is the function type `A => B`.

$x^{:A} \Rightarrow f$ means a nameless function (a literal function value) with argument x of type A and function body f.

id means the identity function. The type of its argument should be either specified as id^A or $\text{id}^{:A \Rightarrow A}$, or else should be unambiguous from the context.

1 means the unit type, and also the value of the unit type. In Scala, the unit type is `Unit`, and its value is denoted by `()`. Example of using the unit type is $1 + A$, which in Scala corresponds to `Option[A]`.

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 $1 + A$ the non-empty option type is $0 + A$, which in Scala corresponds to `Some[A]`. The empty option type $1 + 0$ corresponds to `None`. Similarly, $A + 0$ may denote the left part of the type $A + B$ (in Scala, `Left[A]`), while $0 + B$ denotes its right part (in Scala, `Right[A]`).

$\triangleq$ means "equal by definition". Examples:

- $f \triangleq x^{:\text{Int}} \Rightarrow x + 1$ is a definition of the function f. In Scala, this is `val f = (x: Int) => x + 1`.
- $F^A \triangleq 1 + A$ is a definition of a type constructor F. In Scala, this is `type F[A] = Option[A]`.

$\cong$ means "equivalent" according to an equivalence relation that needs to be established in the text. For example, if we have established the equivalence that allows nested tuples to be reordered whenever needed, we can write $(a \times b) \times c \cong a \times (b \times c)$.

$A^{:F^B}$ 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 $F^A \triangleq 1^{:F^{\text{Int}}} + A^{:F^{A \times \text{String}}}$.

fmap_F means the standard method fmap pertaining to the functor F. In Scala, this is typically implemented as `Functor[F].fmap`. Since each functor F has its own specific definition of fmap, the subscript "F" is not a type parameter of fmap_F. The method fmap_F actually has *two* type parameters, and its type signature written in full is $\text{fmap}_F^{A,B} : (A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$. For clarity, one may sometimes write explicitly 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. As another example, a monad's standard method `pure` is denoted by pu_F, where the subscript refers to the monad F. This function has type signature $A \Rightarrow F^A$ that contains a type parameter A. In the short notation, this is denoted by pu_F^A. If we are using the `pure` method with a complicated type, e.g. $1 + P^A$, instead of the type parameter A, we might want to indicate this type for clarity as a type parameter

value and write $\text{pu}_F^{1+P^A}$. The type signature of that function is then

$$\text{pu}_F^{1+P^A} : 1 + P^A \Rightarrow F^{1+P^A} \quad .$$

$F^\bullet$ 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 clearly denoted as $F^{G^\bullet}$; in Scala, this is `F[G[?]]` 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 $x \Rightarrow g(f(x))$.

$\circ$ means the backward composition of functions: $f \circ g$ (reads "f after g") is the function defined as $x \Rightarrow f(g(x))$.

$\circ$ with type constructors means their functor composition, for example $F \circ G$ denotes the functor $F^{G^\bullet}$. In Scala, this is `F[G[A]]`.

$f^{\uparrow G}$ means a function f lifted to a functor G. For a function $f^{:A \Rightarrow B}$, the application of $f^{\uparrow G}$ to a value $g : G^A$ is written as $f^{\uparrow G} g$. In Scala, this is `g.map(f)`. Nested liftings can be written as $f^{\uparrow G \uparrow H}$, which means $\left(f^{\uparrow G}\right)^{\uparrow H}$ and produces a function of type $H^{G^A} \Rightarrow H^{G^B}$. Applying a nested lifting to a value h of type H^{G^A} is written as $f^{\uparrow G \uparrow H} h$. In Scala, this is `h.map(_.map(f))`.

$f^{\downarrow H}$ means a function f lifted to a contrafunctor H. For a function $f^{:A \Rightarrow B}$, the application of $f^{\downarrow H}$ to a value $h : H^B$ is written as $f^{\downarrow H} h$ and yields a value of type

[1] https://github.com/typelevel/kind-projector

H^A. In Scala, this is `h.contramap(f)`.

$\diamond_M$ means the Kleisli product operation for the monad M. This is a binary operation working on two Kleisli functions of types $A \Rightarrow M^B$ and $B \Rightarrow M^C$ and yields a new function of type $A \Rightarrow M^C$.

B Glossary of terms

I chose certain terms in this book to be different from the terms currently used in the functional programming community. My proposed terminology is designed to help readers understand and remember the concepts behind the terms.

Nameless function An expression of function type, representing a function. For example, `x  x * 2`. Also known as function expression, function literal, anonymous function, closure, lambda-function, lambda-expression, or simply a "lambda".

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**.

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 OCaml and in Object Pascal). It is shorter to say "sum type," but the English word "sum" is more ambiguous to the ear than "disjunction".

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 called **algebraic data types**. A polynomial type constructor is always a functor with respect to any of its type parameters, hence I use the shorter name "polynomial functor" instead of "polynomial type constructor".

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 $F^A \triangleq (A \times A)^{:F^{\text{Int}}} + (\text{Int} \times A)^{:F^1}$. Also called a **GADT** (generalized algebraic data type).

Functor block A short syntax for composing various `map`, `flatMap`, and `filter` operations on a functor value. The `Functor` typeclass instance must be fixed for the entire functor block. 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)`. Similar syntax exists in a number of languages and is called a **for-comprehension** or **list comprehension** in Python, **do-notation** or do-block in Haskell, and **computation expressions** in F#. I use the name "functor block" in this book because it is shorter and more descriptive.

Method 1) A function defined as a member of a typeclass. For example, `flatMap` is a method defined in the `Monad` typeclass. 2) In Scala, a function defined as a member of a data type declared as a Java-compatible `class` or `trait`. Trait methods are necessary in Scala when implementing functions whose arguments have type parameters (because ordinary Scala function values 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.

Kleisli function Also called a Kleisli morphism or a Kleisli arrow. A function with type signature $A \Rightarrow M^B$ for some fixed monad M. More verbosely, "a morphism from the Kleisli category corresponding to the monad M". The standard monadic method $\text{pure}_M : A \Rightarrow M^A$ 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 $A \Rightarrow M^B$ and $B \Rightarrow M^C$) into a new Kleisli function (of type $A \Rightarrow M^C$).

Exponential-polynomial type A type constructor built using disjunctions (sums), products, and function types, as well as type parameters or fixed types. For brevity, I call them "exp-poly" types. For example, in Scala, `type F[A] =`

`Either[(A, A), Int  A]` is an exp-poly type constructor. Such type constructors can be functors, contrafunctors, or profunctors.

Short type notation A mathematical notation for type expressions developed in this book for the purpose of quicker and more practical reasoning about types in functional programs. Disjunction types are denoted by +, product types by ×, and function types by ⇒. The unit type is denoted by 1, and the void type by 0. The function arrow ⇒ has weaker precedence than +, which is in turn weaker than ×. Type parameters are denoted by superscripts. As an example of using these conventions, the Scala definition

```
type F[A] = Either[(A, A  Option[Int]), String  List[A]]
```

is written in the short type notation as

$$F^A \triangleq A \times (A \Rightarrow 1 + \text{Int}) + \left(\text{String} \Rightarrow \text{List}^A\right) \quad .$$

B.1 On the current misuse of the term "algebra"

In this book, I do not use the terms "algebra" or "algebraic", because these terms are too ambiguous. In the current practice, the functional programming community is using 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, you need $1 * x = x$, the addition must be commutative, the multiplication must be distributive over addition, and so on. As an example, 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 can be multiplied and added. This standard definition of "algebra" is not actually used in functional programming.

Definition 1. An "algebra" is a function with type signature $F^A \Rightarrow A$, 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 $A \times A \Rightarrow A$ may be interpreted as a "multiplication" operation (but, in any case, A is a type and not a vector space, and there are no distributivity or commutativity laws). I prefer to call such functions "F-algebras", emphasizing

that they characterize and depend on a chosen functor F. However, F-algebras are not mentioned in this book: knowing how to reason about their properties does not give much help in practical work.

Definition 2. Polynomial functors are often called "algebraic data types". However, they are not "algebraic" in the sense of Definition 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 short type notation. The set of all values of the type F^A does not admit 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 cannot be a function with type $F^A \Rightarrow A$, as required for Definition 1. It seems that the usage of the word "algebra" here is to refer to "school-level algebra" with polynomials; these data types are built from sums and products of types. In this book, I call such types "polynomial". However, if the data type contains a function type, e.g. `Option[Int => A]`, the type is no longer polynomial. So I use the 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`, referring to the fact that these functions must satisfy certain equational laws (e.g. the identity, composition, or associativity laws). But these laws do not form an "algebra" in the sense of Definition 0, nor do the functions such as map or `flatMap` themselves (there are no binary operations on them). Neither do they form an algebra in the sense of Definition 1. The laws for `map` or `flatMap` are in no way related to "algebraic data types" of Definition 2. So here the word "algebra" is used in a way that is unrelated to the three previous definitions. To me, it does not seem helpful to say the word "algebra" or "algebraic" when talking about equational laws. These laws are "algebraic" in a trivial sense – i.e. they are written as equations. In mathematics, "algebraic" equations are different from "differential" or "integral" equations. In functional programming, all equational laws are of the same kind: some code on the left-hand side must be equal to some code on the right-hand side of the equation. So calling them "algebraic" does not help and does not clarify anything. I call them "equational laws" or just "laws".

Definition 4. In the Church encoding of a free monad (nowadays known as the "final tagless" encoding), the term "algebra" refers to the *type constructor* param-

eter F. This definition has nothing to do with any of the previous definitions. Clearly, Definition 0 cannot apply to a type constructor. Definition 1 does not apply since F is not itself a function type of the form $G^A \Rightarrow A$. (A function of type $F^A \Rightarrow A$, which I call a "runner" for the type constructor F, is not usually called an "algebra" in discussions of the "final tagless" encoding.) Definition 2 seems to be the most closely related meaning, since F is *sometimes* a polynomial functor in practical usage (although in most cases F will be an unfunctor). However, it is not helpful to call the polynomial functor F an "algebraic data type" and, at the same time, an "algebra". Definition 3 does not apply since the free monad construction does not assume that any laws hold about F, nor has any means of imposing such laws. The type constructor F is used to parameterize the effects described by the free monad, so it seems more reasonable to call 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, I reserve the word "algebra" to denote the branch of mathematics, as in "school-level algebra" or "graduate-level algebra". Instead of "algebra" as in Definitions 1 to 4, I talk about "F-algebras" with a specific functor F; "polynomial types" or "polynomial functors" or "exponential-polynomial functors" etc.; "equational laws"; and an "effect constructor" F.

C Scala syntax and features

C.0.1 Function syntax

Functions have arguments, body, and type. The function type lists the type of all arguments and the type of the result value.

```
def f(x: Int, y: Int): Int ⇒ Int = { z ⇒ x + y + z }
```

Functions may be used with infix syntax as well. For this syntax to work, the function must be defined **as a Scala method**, that is, using `def` within the declaration of `x`'s class, or as an extension method. The infix syntax cannot work with functions defined using `val`. For clarity, I call Scala functions **infix methods** when defined and used in this way.

The syntax `List[Int]` means "a list of integer values." In the type expression `List[Int]`, the "`Int`" is called the **type parameter** and `List` is called the **type constructor**.

A list can contain values of any type; for example, `List[List[List[Int]]]` means a list of lists of lists of integers. So, a type constructor can be seen as a function from types to types. A type constructor takes a type parameter as an argument, and produces a new type as a result.

C.0.2 Scala collections

The Scala standard library defines collections of several kinds, the main ones being sequences, sets, and dictionaries. These collections have many map/reduce-style methods defined on them.

Sequences are "subclasses" of the class `Seq`. The standard library will sometimes choose automatically a suitable subclass of `Seq`, such as `List`, `IndexedSeq`, `Vector`, `Range`, etc.; for example:

```
scala> 1 to 5
scala> (1 to 5).map(x ⇒ x*x)
scala> (1 to 5).toList
```

```
scala> 1 until 5
scala> (1 until 5).toList
```

For our purposes, all these "sequence-like" types are equivalent.

Sets are values of class `Set`, and dictionaries are values of class `Map`.

```
scala> Set(1, 2, 3).filter(x ⇒ x % 2 == 0)
```

D Intuitionistic propositional logic (IPL)

The intuitionistic propositional logic (sometimes also called the "constructive" propositional logic) describes how programs in functional programming languages may be able to compute values of different types.

The main formal difference between IPL and the classical (Boolean) logic is that IPL does not include the axiom of excluded middle ("*tertium non datur*"), which is

$$\forall A : (A \text{ or } (\text{not } (A))) \text{ is true} \quad .$$

It is not easy to figure out the consequences of *not having* this axiom. The reason this axiom is not included in IPL is that IPL propositions such as $\mathcal{CH}(A)$ correspond to the *practical possibility* of values of type A *to be computed* by a program. For the proposition $\mathcal{CH}(A)$ to be true in IPL, a program needs to actually compute a value of type A. It is not sufficient merely to show that the non-existence of such a value would be somehow contradictory. In classical logic, the axiom of excluded middle says that either $\mathcal{CH}(A)$ or not $(\mathcal{CH}(A))$ is true. So showing that "not $\mathcal{CH}(A)$" is contradictory would be sufficient for proving $\mathcal{CH}(A)$ without ever computing any values of type A. For this reason, classical (Boolean) logic does not adequately predict which typed values can be computed by functional programs; one needs to use the IPL.

D.1 Example: Implementable types not described by Boolean logic

Here is an explicit example of obtaining an incorrect result when using classical logic to reason about values computed by functional programs. Consider the formula

$$(A \Rightarrow B + C) \Rightarrow (A \Rightarrow B) + (A \Rightarrow C) \tag{D.1}$$

or, putting in all the parentheses for clarity,

$$(A \Rightarrow (B + C)) \Rightarrow ((A \Rightarrow B) + (A \Rightarrow C)) \quad .$$

This formula is a true theorem in classical logic. To prove this, we only need to show that Eq. (D.1) is always equal to *true* (i.e. Boolean value 1) for any Boolean values of the variables A, B, C. Consider that the only way an implication $A \Rightarrow B$ could be *false* (that is, equal to 0) in Boolean logic is when $A = 1$ and $B = 0$. So, Eq. (D.1) can be false only if $(A \Rightarrow B + C) = 1$ and $(A \Rightarrow B) + (A \Rightarrow C) = 0$. The disjunction can be false only when both parts are false; so we must have $(A \Rightarrow B) = 0$ and $(A \Rightarrow C) = 0$. This is only possible if $A = 1$ and $B = C = 0$. But, with these value assignments, we find $(A \Rightarrow B + C) = 0$ rather than 1. So, we cannot ever make Eq. (D.1) equal to 0 as a Boolean formula. This shows Eq. (D.1) to be a "classically valid" formula, i.e. a theorem that holds in classical Boolean logic.

If we use the Curry-Howard correspondence and apply Eq. (D.1) to propositions such as $\mathcal{CH}(A)$, $\mathcal{CH}(B)$, $\mathcal{CH}(C)$, we obtain the statement that a program should be able to compute a value of type $(A \Rightarrow B) + (A \Rightarrow C)$ given a value of type $A \Rightarrow B + C$. In Scala, such a program would be written as a function with the following type signature,

```
def bad[A, B, C](g: A => Either[B, C]): Either[A=>B, A=>C] = ???
```

However, it is impossible to implement this function in Scala as a total function.

To help build an intuition for the impossibility of implementing `bad`, consider that the only available data is a function $g : A \Rightarrow B + C$, which may return values of type B or C depending on the input value of type A. The function `bad` must return either a function of type $A \Rightarrow B$ or a function of type $A \Rightarrow C$. Can we create a function of type $A \Rightarrow B$? Given a value of type A, we need to compute a value of type B. Since the type B is completely 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 input function g. However, g may produce values of type C for some values of type A. So, in general, we cannot build a function of type $A \Rightarrow B$ out of the function g. Similarly, we find that we cannot build a function of type $A \Rightarrow C$ out of g.

The decision about whether to return $A \Rightarrow B$ or $A \Rightarrow C$ must be somehow made in the code of `bad`. The only input data is the function g that takes an argument of type A. We could imagine calling g on various arguments of type A and to see whether g returns a B or a C. However, the type A is unknown, so the function `bad`

cannot produce any values of that type and call g. So the decision about whether to return $A \Rightarrow B$ or $A \Rightarrow C$ must be made regardless of the function g. Whichever we choose to return, $A \Rightarrow B$ or $A \Rightarrow C$, we will not be able to return a result value of the required type.

What we *could* do is to switch between $A \Rightarrow B$ and $A \Rightarrow C$ depending on a given value of type A. This, however, corresponds to a different type signature:

$$(A \Rightarrow B + C) \Rightarrow A \Rightarrow (A \Rightarrow B) + (A \Rightarrow C) \quad .$$

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 type signature that corresponds to Eq. (D.1) via the Curry-Howard correspondence.

In the IPL, it turns out that Eq. (D.1) is not a valid theorem, i.e. it is impossible to find a proof of Eq. (D.1) in the IPL. To *prove* that there is no proof, one needs to use methods of proof theory that are beyond the scope of this book. A good introduction to the required technique is the book "*Proof and Disproof in Formal Logic*" by R. Bornat.[1]

This example illustrates that it is precisely the valid theorems in the IPL, and not the valid theorems in the Boolean logic, that correspond to implementable functional programs.

D.2 Using truth values in Boolean logic and in IPL

Another significant difference between IPL and the Boolean logic is that propositions in IPL cannot be assigned a fixed set of "truth values". This was proved by Gödel in 1935. It means that a proposition in IPL cannot be decided by writing out a truth table, even if we allow more than two truth values.

[1] R. Bornat, "Proof and Disproof in Formal Logic", Oxford, 2005 - link to Amazon.com

E Category theory

Examples of categories

1. Objects: types Int, String, ...; morphisms (arrows) are functions Int $\to$ String etc. – this is the "standard" category corresponding to a given programming language
2. Objects: types A, B, ...; morphisms are pairs of functions $(A \to B)$, $(B \to A)$
3. * Objects: types List^A, List^B, ...; morphisms are functions of type $\text{List}^A \to \text{List}^B$
4. Objects: types A, B, ...; morphisms are functions of type $\text{List}^A \to \text{List}^B$
5. Objects: types A, B, ...; morphisms are functions of type $A \to \text{List}^B$
6. * Objects: types List^A, List^B, ...; morphisms are functions $A \to B$
7. Objects: types A, B, ...; morphisms are $\text{List}^{A \to B}$
8. Objects: types A, B, ...; morphisms are functions $B \to A$
9. * Objects: things A, B, ...; morphisms are pairs (A, B) of things – this is the same as a preorder

Examples marked with * are for illustration only, they are probably not very useful

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

G A humorous disclaimer

The following text is quoted in part from an anonymous source ("Project Guten Tag") dating back 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-ASÍ". 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.

Index