The science of functional programming

With examples in Scala

by Sergei Winitzki, Ph.D.

draft version 0.1, November 7, 2018

Published by **lulu.com** 2019

Published and printed by **lulu.com**

ISBN XXXXXX

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. Exercises with solutions are provided. 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 algebra: for example, be able to compute and simplify $\frac{d}{dx}\left((x+1)\,e^{-x}\right)$.

Contents

TO DO:

Implement scripts for Scala syntax highlighting.

If LyX-Code has `scala>` prepended to its first line, the result of running the code will be shown in the book text.

If LyX-Code has `!scala>` prepended to its first line, the result of running the code will be shown in the book text, and we expect it to be an error.

If LyX-Code has `///` prepended to it, the code should not be run at all but just shown with syntax highlighting.

All LyX-Code fragments must have font "smaller"; also, all inline code fragments must have font "smaller".

Special symbols must be replaced in all LyX-Code fragments: curly quotes by straight quotes; mathematical ⇒, →, ←, as well as ASCII =>,->, <- by the corresponding Unicode symbols.

Syntax highlighting in color must be introduced in all code fragments, both inline and LyX-Code.

1 Values, types, expressions, functions

1.1 Translating mathematics into code

1.1.1 First examples

We will consider some computational tasks and write Scala code for them.

Factorial of 10 Compute 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 now write Scala code in a way that resembles the formula:

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

The Scala interpreter indicates that the result is the value 3628800 of type `Int`. If we need 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
scala> assert(fac10 == 3628800)
```

The code `(1 to 10).product` is an **expression**, which means that (1) it can be computed and yields a value, as we have just seen, 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
```

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 ($\mathbb{N}$). The mention of this set is similar to specifying the type `Int`. So, the mathematical domain of a function corresponds in Scala code to the specification of the argument's type.

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

```
scala> f(10)
```

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

```
!scala> f("abc")
```

Nameless functions The formula and the code, shown above, both involve *naming* the function as "f". What if we do not need to name that function, for instance, if f will be used only once? Some mathematics books use a notation for nameless functions that looks like this:

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

In this book, I will use the double arrow symbol $\Rightarrow$ instead of $\mapsto$, because the symbol $\Rightarrow$ is what Scala uses for functions.[1] So, our notation

[1] In Scala, the two characters => and the single Unicode symbol $\Rightarrow$ mean the same thing. I will use the Unicode symbol $\Rightarrow$ for brevity.

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 k for k from 1 to n".

The Scala expression implementing this mathematical formula is

```
(n: Int) ⇒ (1 to n).product
```

This expression is Scala's syntax for a nameless function. 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
```

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. The Scala interpreter may print something like

```
/// Lambda$1924/1256837057@363a52f
```

as the "value" of `fac`. This of course is not of much help to us, except to indicate that the value is not easily printable. We can visualize a function value as representing a *block of compiled code*, — code that will actually run and evaluate the function body when the function is applied. There is no easy way of printing that code for us to look at.

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

```
scala> fac(10)
```

However, functions can be used even without naming them. We can directly apply a nameless factorial function to an integer argument 10, like this:

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

One would not usually write code like this, for two reasons: (1) the syntax is harder to read, and (2) there is no advantage in creating a nameless function that we apply right away to a known argument, because we can then simplify the code and replace the expression

```
((n: Int) ⇒ (1 to n).product)(10)
```

by substituting $n = 10$ and writing an equivalent expression

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

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.

A simple mathematical formula for this function can be written as

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

This formula has two clearly separated parts: First we take a range of integers between 2 and n, and then we impose the requirement that each of 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 closely similar to the mathematical notation, except for the arrow after k.

We can apply this function now to some integer values:

```
scala> is_prime(12)
scala> is_prime(13)
```

As we can see, the function returns a value of type `Boolean`. Therefore, the type of `is_prime` is `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 have type `Int`.

1.1.2 Nameless functions and bound variables

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

The first, superficial difference is that the interval $[2, n-1]$ comes first in the Scala expression. To understand this, look at the way 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 range `(2 to n-1)` and 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 usual mathematical notation, this would be $f(x, z)$. The second argument, `z`, may be missing, and then the syntax becomes simply `x.f`, as in the `(1 to n).product` example. 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. The infix syntax does not work with functions defined using `val`. For clarity, I call Scala functions **infix methods** when defined and used in this way.

The infix methods `.product` and `.forall` are already provided in the Scala standard library, so it is natural to use them. If we did not want to use the infix syntax, we would instead have to define a function `for_all` with two arguments and write code like this,

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

This brings the Scala 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, this is a peculiarity of the mathematical notation; 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, that we are examining all values from the given range; second, that we choose 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. It is natural to write this function explicitly as a nameless function

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

and to write Scala code saying explicitly that we are applying this nameless function to each element of the range $[2, n-1]$ and requiring that all result values are `true`. The Scala code is therefore

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

Just as the mathematical notation makes the variable k defined only on 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 as the argument of the nameless function, which happens in this expression) but the variable n is free (it is defined outside this expression).

The main difference between free and bound variables is that bound variabes can be 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

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

or in Scala code,

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

This renaming does not change the result values. In the nameless function `k ⇒ n % k != 0`, the variable k is bound and thus may be renamed to x or to anything else, without changing the value of the expression. No code outside this expression needs to be changed after renaming k to x. But the variable n is free and cannot be renamed at will. If, for any reason, we wanted to rename n in the expression `k ⇒ n % k != 0`, we would also need to change some code *outside* this expression, or else the program would become incorrect.

Mathematical formulas introduce bound variables in constructions such as $\forall k : p(k)$, $\exists k : p(k)$, $\sum_{k=a}^{b} f(k)$, $\int_0^1 k^2 dk$, $\text{argmax}_k f\,(k)$, and so on. When we translate mathematical expressions into code, we need to recognize the presence of bound variables, which the mathematical notation does not make quite 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

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

where p is a given predicate, has a bound variable k and is translated into Scala code as

```
(1 to n).forall(k ⇒ p(k))
```

At this point we can also apply a simplification trick to this code. Note that 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 to

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

The simplification of $x \Rightarrow f(x)$ to just f is always possible in Scala code, for functions f of a single argument.[2]

1.1.3 Aggregating data from arrays

Consider the task of counting how many even numbers there are in a given set S of integers. For example, the set $(10, 20, 30, 40, 50)$ contains *two* even numbers (20 and 40).

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

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

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

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 functions.

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
}
```

[2] The great flexibility of Scala syntax makes it possible to write code that looks like `f(x)` but actually involves additional implicit or default arguments to the function `f`, or an implicit conversion for its argument `x`. In those cases, replacing `x⇒f(x)` by `f` may fail to compile in Scala. But these complications do not arise when reasoning about simple functions on integers.

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 $\sum_{k \in S} \text{is\_even}(k)$. Instead of using a set S, for simplicity we will use the class `List` from the Scala standard library and consider S to be a list of integers.

To compute $\sum_{k \in S} \text{is\_even}(k)$, we need to apply the function `is_even` to each element of the list S, 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 on the class `List`.

The method `.sum` is defined for any `List` of numerical types (`Int`, `Float`, `Double`, etc.) and computes the sum of all numbers in the list. For example,

```
scala> List(1, 2, 3).sum
```

We have seen a similar method `.product` before.

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 + 10*x)
```

In this example, we used the nameless function $x \Rightarrow x^2 + 10x$ as the argument of `.map()`. This function is repeatedly applied by `.map` to transform 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 + 10*x
scala> List(1, 2, 3).map(func1)
```

An infix method, such as `.map`, can be also used in a shorter syntax:

```
scala> List(1, 2, 3) map func1
```

If the transforming function `func1` is used only once, this longer code will not have any advantages over the shorter code that uses a nameless function, especially for a simple function such as $x \Rightarrow x^2 + 10x$.

It is now clear how to use the methods `.map` and `.sum` to define `count_even`. The code looks like this:

```
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 to use infix methods to chain 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 as shown.

1.1.4 Filtering

The Scala standard library defines several useful methods for lists. In addition to the methods `.sum`, `.product`, `.map`, `.forall` that we have already seen, there are methods `.max`, `.min`, `.exists`, `.size`, `.filter`, `.takeWhile`, and many others.

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

```
scala> List(10, 20, 30).max
scala> List(10, 20, 30).min
scala> List(10, 20, 30).size
```

The methods `.forall`, `.exists`, `.filter`, and `.takeWhile` take 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 is true on at least one value in the list. These methods can be defined via 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 now 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)
```

The `.takeWhile` method returns a *new list* that contains the initial subset of values from the original list, until the predicate first returns `false`:

```
scala> List(1, 2, 3, 4, 5).takeWhile(k ⇒ k % 3 != 0)
```

In all these cases, the predicate must be a *function* whose argument is of the same type as the values in the list. In the examples shown above, lists have integer values (i.e. the lists have type `List[Int]`), therefore the predicate argument `k` must be also of type `Int`.

The methods `.max`, `.min`, `.sum`, and `.product` are defined on lists of *numeric types*, such as `Int`, `Double`, and so on. 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 **aggregating** I mean applying a function from a list of values to a single value; examples of aggregation functions are `.max` and `.sum`. By **transforming** I mean applying a function from a list of values to another list of values; examples of transformation functions are `.filter` and `.map`. Writing programs by chaining together various operations of transformation and aggregation is known as programming in the **map/reduce** style.

1.1.5 Calculations with nameless functions

We now need to gain some experience working with nameless functions.

The main thing a function does is evaluating its result value when applied to an argument. In mathematics, functions are evaluated by simply substituting their argument values into their body. When applied to an argument, 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:

- 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 the underlined superscript, for example: $x^{\underline{\text{Int}}} \Rightarrow x * x + 20$.

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

$$\begin{aligned}(x \Rightarrow x * 2)\,(10) &= 10 * 2 = 20 \quad .\\ (p \Rightarrow z \Rightarrow z * p)\,(t) &= (z \Rightarrow z * t) \quad .\\ (p \Rightarrow z \Rightarrow z * p)\,(t)(4) &= (z \Rightarrow z * t)(4) = 4 * t \quad .\end{aligned}$$

In some cases, the result of a computation is an integer such as 20; in other cases, it is a *function* value such as $z \Rightarrow z * t$.

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

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

Here I have been performing calculations **symbolically**, imitating the computation that the running code would perform at run time.

1.1.6 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 $\sin(2x) = 2 \cdot \sin(x) \cdot \cos(x)$.

Several programming languages (LISP, ML, OCaml, F#, Haskell, Elm, PureScript) have adopted this "short syntax" and made parentheses optional for function arguments. The result is a very concise notation where `f x` means the same as `f(x)`. Parentheses are still used where necessary to avoid ambiguity.

The syntax 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 $((f\ x)\ y)\ z$.

- 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 $f(x \Rightarrow g\ x)$ instead of $f\ x \Rightarrow g\ x$.

In this book, I will sometimes use the short syntax when reasoning about code. Scala does not permit short syntax, so parentheses need to be put around every argument, except when using the infix method syntax such as `List(1,2,3) map func1`.

1.1.7 Curried functions

Consider a function of type `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, this function will return an integer value only after we apply it to *two* integer values. 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. This usage is suggested by the equivalent syntax `Int ⇒ Int ⇒ Int` with the parentheses omitted.

Scala supports both variants: a function with type signature `(Int, Int) ⇒ Int` that takes two arguments at once,

```
def f1(x: Int, y: Int): Int = x - y
```

and a function with type signature `Int ⇒ Int ⇒ Int` that takes one argument at a time,

```
def f2: Int ⇒ Int ⇒ Int = x ⇒ y ⇒ x - y
```

Functions of this latter sort are called **curried** functions.

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

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

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

```
scala> val r = f2(20)
scala> r(4)
```

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

More generally, a curried function has a type signature of the form $A \Rightarrow B \Rightarrow C \Rightarrow ... \Rightarrow P \Rightarrow Z$ where A, B, C, ..., Z are some types. This function is, in a sense, equivalent to an **uncurried** function with type signature $(A, B, C, ..., P) \Rightarrow Z$. Their equivalence is not equality — these functions are *different*; but one can be always reconstructed from the other if necessary.

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 not often used and may be harder to read.

In the short notation (used e.g. in OCaml and Haskell), a curried function such as `f2` is applied to its arguments as `f2 20 4`. This departs significantly from the mathematical notation, 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, since no commas are used to separate the arguments. For this reason, Haskell code tends to first define complicated expressions as short names if they are used as arguments of a curred function.

In Scala, the choice of whether to use curried or uncurried function signatures is largely a matter of syntactic convenience. Most Scala code seems to be written with uncurried functions, while curried functions are used when they produce more easily readable code. One of the syntactic conveniences for curried functions is when an argument of a function is itself a function. It is then convenient to supply that argument as a curried argument, with the curly brace syntax supported in Scala. 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 for `summation2` may be easier to read.

1.1.8 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 its 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, slightly inequivalent, ways of defining a function: one as a method (using `def`), another as a function value (using `val`). The main difference between them is that methods can have type parameters (e.g. `def f[A](x: A): A`) while function values cannot.

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.

1.1.9 Worked examples: transformation and aggregation

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

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

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

   ```
   scala> (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).filter(k ⇒ math.cos(k) > 0)
   ```

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

```
def average(s: List[Double]): Double = s.sum / s.size
scala> average(List(1.0, 2.0, 3.0))
```

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

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

   We create a list of `Double` numbers from integers, using the method `.toDouble`:

```
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.
```

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

5. Given a list of lists, `s: List[List[Int]]`, compute the list containing only the inner lists of size at least 3. The result must be again of type `List[List[Int]]`.

```
def f(s: List[List[Int]]): List[List[Int]] =
   s.filter(t ⇒ t.size >= 3)
scala> f(List( List(1,2), List(1,2,3), List(1,2,3,4) ))
```

   The predicate in the argument of `.filter` is a function `t ⇒ t.size >= 3` whose argument `t` is of type `List[Int]`.

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

```
scala> (1 to 10)
  .filter(k ⇒ (1 to k).filter(j ⇒ j*j > k).size >= 1)
```

   The argument of the outer `.filter` is a nameless function that itself uses a `.filter` to compute the list of j's that satisfy the condition, and then to compute the length of that list.

1.1.10 Worked examples: nameless 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)
   ```

   Since functions are values, we can directly return new functions. It is not necessary to specify the type of the arguments `x` and `k` because we already fully specified the types of `fb` and `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)
```

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))
```

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 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
```

1.2 Summary

This 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 we 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.
- Define **higher-order functions**, that is, functions that take other functions as arguments and/or return new functions.

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

- Example 1: Compute the smallest $n \geq 1$ such that $f(f(f(...f(0)...) > 1000$, 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 array.

These problems are not yet solvable because the corresponding 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 only a restricted class of iterative operations on lists: namely, operations that process each element of a given list independently. 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 to previous values until a given condition holds—that is, an *initially unknown* number of times. So it is impossible to write an expression containing `.map`, `.filter`, `.takeWhile`, etc., that solves Example 1. We need mathematical induction to write the solution of Example 1 as a formula.

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

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

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

Example 3 defines the search result by induction and again requires an initially unknown number of steps. Chapter 2 explains how to solve these problems by translating mathematical induction into code using recursion.

1.3 Exercises

1. Define a function of type `List[Double]` ⇒ `List[Double]` that "normalizes" the list: finds the element having the max. 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. Define a function of type `List[List[Int]]` ⇒ `List[List[Int]]` that adds 20 to every element of every inner list. A test:

```
/// add20( List( List(1), List(2, 3) ) )
///     == List( List(21), List(22, 23) )
```

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

4. 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? Rewrite Exercise 3 using that function.

5. Define a function q that takes a function $f : \text{Int} \Rightarrow \text{Int}$ as an argument, and returns a new function that computes $f(f(f(x)))$. What is the required type of the function q?

1.4 Discussion

1.4.1 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 programs as mathematical expressions or formulas*. This allows programmers to write 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 simple principles, because code is written as formulas.

Mathematical intuition is backed by the vast experience accumulated while working with data over thousands of years of human history. Functional programmers are fortunate to have at their disposal such a superior reasoning tool.

As we have seen, the Scala code corresponds quite closely to mathematical formulas. Scala conventions and syntax, of course, require programmers to spell out certain things that the mathematical notation does not spell out. Large expressions need to be split into parts in a suitable way, so that the parts can be easily reused, flexibly composed together, and developed independently from each other. The FP community has developed a standard toolkit of functions (such as `.map`, `.filter`, etc.) that proved especially useful in real-life programming.

Mastering FP thus 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 conventions of academic mathematics. This book explains the required mathematical principles in detail, developing them through intuition and practical coding tasks.

1.4.2 Functional programming languages

It is possible to apply the FP paradigm while writing code in any programming language. However, some languages have 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 the simplest of the idioms of FP, such as the map/reduce operations. More advanced FP constructions are impractical in these languages because the corresponding code is too complicated to read, and mistakes are too easy to make, which will cancel 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 fully-fledged 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.

1.4.3 The mathematical meaning of variables

In mathematics, a **variable** is usually an argument of some function—perhaps, of a nameless function. For example, the mathematical formula

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

contains the variable x and can be seen as a function 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 x cannot change within the function body. Indeed, there is no widely used 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 this formula, we modify x by adding 4 to it". If the "last x" in $x^2 + x$ needs to have 4 added to it, a mathematics textbook just writes the formula $x^2 + x + 4$.

An example involving nameless functions is

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

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.

So, a variable in mathematics does not actually "vary" *within* the expression where it is used. We could describe a variable as a "named constant value of a fixed type," as seen within the expression where it is used. One could apply the function f to different values x, and each time compute a different result $f(x)$, but the value of x will remain unmodified within f while the body of the function f is being computed.

Functional programming adopts the same convention: variables are immutable named constants.

In Scala, function arguments remain 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 be labeled by 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, and an error will be given if a function is called on arguments of incorrect types. So, the programmer does not need to write any code that checks types of arguments before starting to evaluate the function.

The second usage of "variables" in mathematics is to denote partial expressions for brevity. For example, one writes: let z_0 = ... and now compute $\cos z_0 + \cos 2z_0 + \cos 3z_0$, etc. Again, in this case z_0 remains immutable, and its type remains fixed.

In Scala, this construction is 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 obvious that this `x` is of type `Int`. 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 write

```
val x: Int = "123" // A String instead of an Int.
```

1.4.4 Iteration without loops

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

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

And yet, no mathematics textbook ever mentions "`for` loops" or says "now repeat this formula 10 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 $\sum_{i=1}^{n} s_i$, which are defined using mathematical induction. The FP 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 concise and largely error-free 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 which, perhaps, will contain loops; but the programmer does not need to see that code or to reason about it.

1.4.5 Nameless functions in mathematical notation

Functions in mathematics are mappings from one set to another. Viewed in this way, it is clear that 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 some

function. It follows that the notations $\frac{dx}{1+x}$ and $\frac{dz}{1+z}$ correspond to the same nameless function whose argument was renamed from x to z. In FP, this nameless function would be denoted as $z \Rightarrow \frac{1}{1+z}$, and the integral rewritten as code such as

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

If we compare the mathematical notations for sums and for integrals, an analogous sum would be

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

The traditional mathematical notation is somewhat inconsistent here: For sums, the bound variable k is introduced under the $\sum$ symbol; for integrals, the bound variable follows the symbol "d". This notational inconsistency can be removed from the mathematical notation if we were to use nameless functions explicitly, for example like this:

$$\sum_{0}^{x} k \Rightarrow \frac{1}{1+k} \text{ instead of } \sum_{k=0}^{x} \frac{1}{1+k}$$
$$\int_{0}^{x} z \Rightarrow \frac{1}{1+z} \text{ instead of } \int_{0}^{x} \frac{dz}{1+z}$$

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" but takes a function as an argument. In other words, summation and integration become *functions* that take a function as argument (i.e. higher-order functions). The summation shown above may be expressed as

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

In this way, nameless functions remove inconsistency from the traditional mathematical notation and express summation and integration in a uniform and straightforward manner as higher-order functions.

Expressed as functions, these operations are easy to implement and to use.

The Scala standard library does not define functions `integration` or `summation`. We could implement `summation(a,b,g)` as `(a to b).map(g).sum`. Numerical integration requires more code; a simple implementation using Simpson's rule is

```
def integration(a: Double, b: Double,
  g: Double ⇒ Double, eps: Double = 0.001) = {
   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)
   delta_x / 3 * (g(a) + g(b) +
    4*(1 to (n-1) by 2).map(g_i).sum +
    2*(2 to (n-2) by 2).map(g_i).sum
  )
}
```

Ths code is written in the FP paradigm: the entire code is a large expression, whose sub-expressions are defined for convenience as a few helper values and helper functions.

1.4.6 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 meant). This is why bound variables may be renamed at will; no outside code could possibly depend on them. However, outside code may (by chance or by mistake) define variables that have the same name.

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 subexpressions where both of these bound variables are visible, priority is given to

the bound variable defined in the closest 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 invites errors. Consider this function,

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

Since the inner nameless function, $(x \Rightarrow x)$, may be renamed to $(y \Rightarrow y)$ without changing its value, we can rewrite the function to

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

which is a function that takes an x and returns an identity function. It is now easier to understand this code and reason about it. For instance, it becomes immediately clear that this function actually ignores its argument x.

1.4.7 Named and nameless expressions and their uses

It seems to be an advantage if a programming language supports unnamed (or "nameless") expressions. 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. Indeed, it would be quite cumbersome if we had to assign a name separately to each part of an expression. We would have to write code like this:

```
/// a = 123
/// b = x + a
/// c = b * y
/// d = 2
/// e = d + x
/// f = c / e
/// return f
```

This style of programming resembles assembly language, where *every* sub-expression, every step of any calculation must be computed separately and labeled with a name (or a memory address). Clearly, this is a less productive style of programming than writing

```
/// f = (x+123)*y/(2+x)
```

Clearly, programmers gain productivity when their programming language supports expressions without names ("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 (or a "hashmap") is declared in Scala as

```
scala> 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 fills an initially empty dictionary step by step (this is the case with Java).

Nameless functions are useful for the same reason: they allow the programmer to build the entire application as one expression from simpler parts, with more concise code.

1.4.8 Nameless functions: historical perspective

Nameless functions were first invented in 1936 within a theoretical programming language. That programming language was called "λ-calculus", and it used the letter λ as a syntax separator to denote function arguments in a nameless function. For example, the nameless function $x \Rightarrow x + 1$ was denoted as $\lambda x.x + 1$ in λ-calculus.

Many programming languages were invented since that time. However, in most programming languages that were in use until around 1990, all functions required names. By 2015, most languages added support for anonymous functions: Table 1.1 shows when that feature was introduced in each language and gives example syntax for the nameless function $k \to k * k$ defined for integer k.

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

Language	Year	Code for $k^{\underline{\text{Int}}} \Rightarrow k * k$
λ-calculus	1936	$\lambda k.\ k * k$
typed λ-calculus	1940	$\lambda k : int.\ k * k$
LISP	1958	`(lambda (k) (* k k))`
Standard ML	1973	`fn (k: int) => k*k`
OCaml, later F#	1985	`fun (k: int) -> k*k`
Haskell	1990	`\ (k: Int) -> 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`
C++	2011	`[] (int k) { return k*k; }`
Swift	2014	`{(k: int) -> int in return k*k}`
Java	2014	`(int k) -> k*k`

Table 1.1: The nameless function $k^{\underline{\text{Int}}} \Rightarrow k * k$ in various programming languages.

2 Mathematical induction

2.1 Discussion

3 The formal logic of types

3.1 Higher-order functions

3.1.1 Discussion

3.2 Disjunction types

3.2.1 Discussion

3.3 The Curry-Howard correspondence

3.3.1 Discussion

4 Functors

4.1 Discussion

4.2 Practical use

4.3 Laws and structure

5 Type-level functions and type classes

5.1 Discussion

6 Filterable functors

6.1 Practical use

6.1.1 Discussion

6.2 Laws and structure

6.2.1 Discussion

7 Monads and semimonads

7.1 Practical use

7.1.1 Discussion

7.2 Laws and structure

7.2.1 Discussion

8 Applicative functors and profunctors

8.1 Practical use

8.1.1 Discussion

8.2 Laws and structure

9 Traversable functors and profunctors

9.1 Discussion

10 Free constructions

11 Recursive types

11.1 Fixpoints

11.2 Row polymorphism

11.3 Column polymorphism

11.4 Discussion

12 Monad transformers

12.1 Practical use

12.2 Laws and structure

12.3 Discussion

13 Comonads

13.1 Practical use

13.2 Laws and structure

13.3 Co-free comonad

13.4 Discussion

14 Irregular type classes

14.1 Distributive functors

14.2 Rigid functors

14.3 Discussion

15 Conclusions and discussion

16 Essay: Software engineers and software artisans

Published on July 23, 2018

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 "software artisans" rather than software engineers. Several important conclusions will follow.

16.1 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 profession.

A mechanical engineer studies calculus, linear algebra, basics of differential geometry, and several areas of physics such as theoretical mechanics or elasticity theory, and then uses calculations to guide the design of a bridge, say. A chemical engineer studies chemistry, thermodynamics, calculus, geometry, 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, 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 science and mathematics in order to guide new designs. Mathematical calculations and scientific reasoning are required *before* anyone starts drawing a design, let alone building a real device or construction.

Some of the studies required for engineers include somewhat ar-

cane mathematical concepts such as a "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 large part of what engineers do will not be 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 science and mathematics for some of its decision-making about new designs.

16.2 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.

All these trades operate *entirely* from tradition and practical experience. A novice shoemaker needs only to take interest in how leather is cut and put together, say, and to start trying it out 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.

The trades do not require an 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 is no "rank-4 elasticity tensor" to take into account, 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.

16.3 Programmers today are artisans, not engineers

Now I will argue that programmers are *not engineers* in the sense we normally understand the engineering professions.

16.3.1 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 with plumbers, electricians (but not electrical engineers!), lawyers, and surgeons (but not mathematicians, physicists, or chemists). 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 he did not need years of academic study in order to become an expert craftsman.

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 electronic 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 pure 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 who goes on successfully to work on designing, say, quantum computers, without ever having studied quantum physics or advanced calculus.

16.3.2 No mathematical formalism to guide the *development* of software

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 or 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", etc. 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 advanced mathematics in it.

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 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 electrical engineering or mechanical engineering designs. If Knuth's book were based on such formalism, it would have looked very differently: we would first study some theory and then apply it to start writing code.

Knuth's book provides many algorithms, including mathematical ones. But algorithms are similar to patented inventions: They can be used immediately without further study. Algorithms are not similar to theory. Knowing one algorithm does not make it easier to develop another, unrelated algorithm. In comparison, knowing how to solve differential equations will be applicable to thousands of different problems in science and engineering.

A book exists with the title "Science of programming", but the title is misleading. It does not propose a science, similar to physics, that underlies the process of designing programs, similarly to how calculations in quantum physics derive the predicted 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.

To be sure, standard computer science courses do teach analysis of programs using formal mathematical methods. The main such methods are complexity analysis (the so-called "big O notation" is used there), 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. 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 would be analogous to writing a mathematical equation describing the shape of the surface of a shoe made by a fashion designer. True, the equation is mathematically rigorous and can be "analyzed" or "verified"; but the equation in no way guides the fashion designer in actually making shoes. It is understandable that fashion designers do not study differential geometry of surfaces.

16.3.3 Programmers dislike 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 complain about this. The vast majority has never heard these words and are unaware of functional programming.

But 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 would appear that the arcane terminology is the least of their difficulties! Their textbooks are full of complicated equations and long, difficult derivations.

Software engineers would similarly 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 the engineers would need to learn and use to perform some mathematical derivations; and derivations would be required *before* starting to write code.

It is now clear that we do not presently have true software engineering. The people employed under that job title are actually artisans. They employ artisanal methods for their work, and their culture is that of a crafts guild.

16.4 Towards software engineering

True software engineering would be achieved if we had theory that guides and informs the writing of code, — not theory that describes or "analyzes" programs *after* they are written.

One could point out that certain programming tasks, for example numerical simulations in physics or machine learning, are mathematical and require formal theory. However, mathematical subject matter (aerospace control, physics or astronomy experiments, statistics, data science) does not automatically make the *process of programming* into engineering. Most data scientists as well as aerospace engineers and natural scientists are artisans at programming.

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 propose to call it **functional type theory**. It is the algebraic foundation of modern 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 writing code but to a peculiar kind of mathematical reasoning. Rather than focusing on this or that 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 some 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 something that resembles calculations in abstract algebra (see Fig. 16.1), which will help design the code fragment they are about to write. This activity is quite similar to that of a modern engineer who first performs some mathematical calculations and only then embarks on a real-life

Figure 16.1: Example calculation in functional type theory.

design project.

A concrete example of this hand-in-hand development of the functional type theory and its applications is the so called "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 asin Haskell. This technique allows programmers to work with declarative side-effect computations where some parts can be computed in parallel, and to achieve the parallelism *automatically* while keeping the composability of the resulting programs. The new technique has distinct advantages over using monad transformers, which was the previous method of combining 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 the programmers about how to proceed writing code.

Another example of a development in functional type theory is the so called "final tagless" 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 the issues 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 functional programmers in any language (even in Java).

This example shows that we may need several years of work before the practical aspects of using "functional type theory" are sufficiently well understood by the functional programming community. The theory is in active development, and its design patterns — as well as the exact scope of its 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.

16.5 Do we need software *engineering*, or are the artisans good enough?

The demand for programmers is growing. Software developer is #1 best job in the US in 2018. But is there a demand for engineers, or just for artisans?

We don't seem to be able to train enough software artisans. Therefore, it is probably impossible to train enough engineers. Modern courses in Computer Science do not actually train software engineers in the true sense of the word; at best, they train academics who are software artisans when writing code. The few software *engineers* we do have are all self-taught. Recalling the situation in construction business, with a few architects and hundreds of construction workers, we might also conclude that 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 Microsoft's operating systems have been notorious for ever-appearing security flaws despite many thousands of programmers and testers Microsoft employed. I think this shows we are overestimating humans' artisanal creative capacity.

It is precisely in designing very large and robust software systems that we would clearly benefit from true engineering. Humanity has been building bridges and using chemical reactions, solving engineering problems by trial, error, and adherence to tradition, long before mechanical or chemical engineering disciplines were developed, based on scientific theory. But now, using the available theory, we are able to create unimaginably more complicated structures and devices.

For building large and reliable software, such as new mobile or embedded operating systems, or distributed peer-to-peer trust archi-

tectures, we will most likely need the sharp qualitative increase in productivity and reliability that can only come from transforming artisanal programming into a proper engineering discipline. Functional programming is a first step in that direction.

17 Essay: Towards functional data engineering with Scala

Published on September 29, 2017

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, Akka, or Kafka? Why is Scala in such demand within the big data world?

There are reasons to believe that the choice of Scala was quite far from pure happenstance.

17.1 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, etc., — 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. 17.1).

Figure 17.1: A nonsensical calculation arises when mixing incompatible data types.

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. 17.2).

17.2 Functional programming is math

The functional programming paradigm is based on very 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 literally 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 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 (OCaml, Haskell, Scala, F#, Swift, Rust) 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.

17.3 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 pow-

Figure 17.2: The Pyongyang method of error-free programming.

erful 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, or Akka Streams. While these abstractions are certainly implementable in Java or Python, true declarative and type-safe usage is possible only 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 type classes and higher-rank generic collections.

It remains to see why Scala became the *lingua franca* of big data and not, say, Haskell.

17.4 Scala is Java on math

The recently invented general-purpose functional programming languages can be grouped into "industrial" (F#, Scala, Swift) and "academic" (OCaml, Haskell).

The "academic" languages are clean-room implementations of well-researched mathematical principles of programming language design (the CH correspondence being one such principle). These languages are unencumbered by requirements of compatibility with any existing platform or libraries. Because of this, the "academic" languages are perfect playgrounds for taking various mathematical ideas to their logical conclusion. At the same time, software practitioners struggle to adopt these languages due to a steep learning curve, a lack of enterprise-grade libraries and tool support, and immature package management.

The languages from the "industrial" group are based on existing and mature software ecosystems: F# on .NET, Scala on JVM, and Swift on Apple's Cocoa/iOS platform. One of the important design requirements for these languages was 100% binary compatibility with their "parent" platforms. 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 100% 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, all these 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 do not support OCaml's parameterized modules and some other features. Of all mentioned languages, Scala and Haskell are the

only ones that directly support type classes 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.

17.5 Conclusion

The decisive advantages of Scala over other contenders (such as OCaml, Haskell, F#, Swift, or Rust) are

1. functional programming, functional collections in the standard library;
2. a highly sophisticated type system, with support for type classes 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.

A Notations

F^A means a type constructor F with a type argument A. In Scala, this is `F[A]`.

$x^{\underline{A}}$ means a variable x that has type A. The subscript is underlined to show that it is not a type argument. In Scala, this is `x : A`.

$A + B$ means the disjunction of types A and B. In Scala, this is the type expression `Either[A, B]`.

$A \times B$ means the product of types A and B. In Scala, this is the tuple `(A,B)`.

$A \Rightarrow B$ means a function type from A to B. In Scala, this is the function type `A => B`.

$\equiv$ means "equal by definition". Examples:

- $f \equiv x^{\underline{\text{Int}}} \Rightarrow x + 1$ is a definition of the function f. In Scala, this is `val f = (x: Int) => x + 1`.
- $F^A \equiv 1 + A$ 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 the equivalence allows nested tuples to be reordered whenever needed, we can write $(a \times b) \times c \cong a \times (b \times c)$.

fmap_F means the standard function fmap pertaining to the functor F. Since each functor F will have a different definition of fmap, the subscript "F" is not a type parameter of fmap. (The function fmap has two type parameters, and its type signature is $\text{fmap}_F^{A,B} : (A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$.) Similarly, a monad's standard function pure_F has the subscript denoting its monad F.

B Glossary of terms

I chose certain terms in this book to be different from the ones currently used in the functional programming community. My proposed terminology is designed to help readers understand the concepts behind the terms.

Nameless function An expression of function type, representing a function. For example, `x ⇒ x * 2`. Also known as anonymous function, lambda-function, lambda-expression, function literal.

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.

Disjunction 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, co-product type, variant type (in OCaml). It is shorter to say "sum type" but the English word "sum" is much more ambiguous to the ear than "disjunction".

Polynomial functor A type built out of disjunctions (sums) and products of type parameters and constant types. For example, in Scala, `type F[A] = Either[(Int, A), A]`. Also known as algebraic data type (ADT). A polynomial type constructor is always a functor in any of its type parameters, hence the shorter name "polynomial functor" instead of "polynomial type constructor".

Exponential-polynomial type A type built out of disjunctions (sums), products, and function types. For brevity, I also say "exp-poly". 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. Disjunctions are denoted by +, products by ×, and function types by ⇒. 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]), Int ⇒ A]
```

is denoted in the short type notation as

$$F^A \equiv A \times (A \Rightarrow 1 + \text{Int}) + (\text{Int} \Rightarrow A) \quad .$$

B.1 On the current misuse of the term "algebra"

In this book, I do not use the terms "algebra", "algebraic" because they 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 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 definition of "algebra" is not 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. There is no direct connection between this "algebra" and Definition 0, except when the functor F is defined by $F^A \equiv A \times A$,

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). I prefer to call such functions "F-algebras", emphasizing that they depend on and characterize the functor F.

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], String]`, or $F^A = 1 + A + \text{String}$ 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". It does not actually seem to admit *any* reasonably defined binary or unary operations at all. 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 laws. But these laws do not form an "algebra" in the mathematical sense (there are no binary operations on them). Neither do they form an algebra in the sense of Definition 1. The laws of `map` or `flatMap` are in no way related to "algebraic data types". So here the word "algebra" is used in a way that is unrelated to the three previous definitions. It does not seem to add any value to say "algebra" when talking about equational laws.

Definition 4. In the "final tagless" encoding of a free construction, the term "algebra" refers to the type constructor parameter 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, nor a function of type $F^A \Rightarrow A$ is presumed to exist. Definition 2 seems to be the most

closely related meaning, since F is often a polynomial type in practical usage. However, it is somewhat inconsistent to call the same entity an "algebraic type" and an "algebra". Definition 3 does not apply since it is not assumed that any laws hold about F.

I propose to 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 propose to talk about "F-algebras"; "polynomial types" or "polynomial functors" or "exponential-polynomial functors" etc.; "equational laws"; and "type constructor parameter" F.

C Scala syntax and code conventions

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 }
```

The type 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, the 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 Typed lambda-calculus

E Intuitionistic logic

F Category theory

G GNU Free Documentation License

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.

G.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.

G.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 G.0.2.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.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.

G.0.3 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections G.0.1 and G.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 G.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 G.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 G.0.3) to Preserve its Title (section G.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

Index