Cambridge University Press
978-1-107-08720-0 - Thinking Functionally with Haskell
Richard Bird
Excerpt
More information

Chapter 1

What is functional programming?

In a nutshell:

- Functional programming is a method of program construction that emphasises functions and their application rather than commands and their execution.
- Functional programming uses simple mathematical notation that allows problems to be described clearly and concisely.
- Functional programming has a simple mathematical basis that supports equational reasoning about the properties of programs.

Our aim in this book is to illustrate these three key points, using a specific functional language called Haskell.

1.1 Functions and types

We will use the Haskell notation

```
f :: X -> Y
```

to assert that `f` is a function taking arguments of type `X` and returning results of type `Y`. For example,

```
sin     :: Float -> Float
age     :: Person -> Int
add     :: (Integer,Integer) -> Integer
logBase :: Float -> (Float -> Float)
```

`Float` is the type of floating-point numbers, things like 3.14159, and `Int` is the type of limited-precision integers, integers n that lie in a restricted range such as

$-2^{29} \leq n < 2^{29}$. The restriction is lifted with the type `Integer`, which is the type of unlimited-precision integers. As we will see in Chapter 3, numbers in Haskell come in many flavours.

In mathematics one usually writes $f(x)$ to denote the application of the function f to the argument x. But we also write, for example, $\sin\theta$ rather than $\sin(\theta)$. In Haskell we can always write `f x` for the application of `f` to the argument `x`. The operation of application can be denoted using a space. If there are no parentheses the space is necessary to avoid confusion with multi-letter names: `latex` is a name but `late x` denotes the application of a function `late` to an argument `x`.

As examples, `sin 3.14` or `sin (3.14)` or `sin(3.14)` are three legitimate ways of writing the application of the function `sin` to the argument `3.14`.

Similarly, `logBase 2 10` or `(logBase 2) 10` or `(logBase 2)(10)` are all legitimate ways of writing the logarithm to base 2 of the number 10. But the expression `logBase (2 10)` is incorrect. Parentheses are needed in writing `add (3,4)` for the sum of 3 and 4 because the argument of `add` is declared above as a pair of integers and pairs are expressed with parentheses and commas.

Look again at the type of `logBase`. It takes a floating point number as argument, and returns a function as result. At first sight that might seem strange, but at second sight it shouldn't: the mathematical functions $\log_2$ and $\log_e$ are exactly what is provided by `logBase 2` and `logBase e`.

In mathematics one can encounter expressions like $\log\sin x$. To the mathematician that means $\log(\sin x)$, since the alternative $(\log\sin)\ x$ doesn't make sense. But in Haskell one has to say what one means, and one has to write `log (sin x)` because `log sin x` is read by Haskell as `(log sin) x`. Functional application in Haskell *associates* to the left in expressions and also has the highest *binding power*. (By the way, `log` is the Haskell abbreviation for `logBase e`.)

Here is another example. In trigonometry one can write

$$\sin 2\theta = 2\sin\theta\cos\theta.$$

In Haskell one has to write

```
sin (2*theta) = 2 * sin theta * cos theta
```

Not only do we have to make the multiplications explicit, we also have to put in parentheses to say exactly what we mean. We could have added a couple more and written

```
sin (2*theta) = 2 * (sin theta) * (cos theta)
```

but the additional parentheses are not necessary because functional application binds tighter than multiplication.

1.2 Functional composition

Suppose `f :: Y -> Z` and `g :: X -> Y` are two given functions. We can combine them into a new function

```
f . g :: X -> Z
```

that first applies `g` to an argument of type `X`, giving a result of type `Y`, and then applies `f` to this result, giving a final result of type `Z`. We always say that functions take *arguments* and return *results*. In fact we have

```
(f . g) x = f (g x)
```

The order of composition is from right to left because we write functions to the left of the arguments to which they are applied. In English we write ‘green pig’ and interpret adjectives such as ‘green’ as functions taking noun phrases to noun phrases. Of course, in French . . .

1.3 Example: common words

Let us illustrate the importance of functional composition by solving a problem. What are the 100 most common words in *War and Peace*? What are the 50 most common words in *Love’s Labours Lost*? We will write a functional program to find out. Well, perhaps we are not yet ready for a complete program, but we can construct enough of one to capture the essential spirit of functional programming.

What is given? Answer: a *text*, which is a list of characters, containing visible characters like `'B'` and `','`, and blank characters like spaces and newlines (`' '` and `'\n'`). Note that individual characters are denoted using single quotes. Thus `'f'` is a character, while `f` is a name. The Haskell type `Char` is the type of characters, and the type of lists whose elements are of type `Char` is denoted by `[Char]`. This notation is not special to characters, so `[Int]` denotes a list of integers, and `[Float -> Float]` a list of functions.

What is wanted as output? Answer: something like

```
the: 154
of: 50
```

```
a: 18
and: 12
in: 11
```

This display is also a list of characters, in fact it is the list

```
"  the: 154\n  of: 50\n  a: 18\n  and: 12\n  in: 11\n"
```

Lists of characters are denoted using double quotes. More on this in the exercises.

So we want to design a function, `commonWords` say, with type

```
commonWords :: Int -> [Char] -> [Char]
```

The function `commonWords n` takes a list of characters and returns a list of the `n` most common words in the list as a *string* (another name for a list of characters) in the form described above. The type of `commonWords` is written without parentheses, though we can put them in:

```
commonWords :: Int -> ([Char] -> [Char])
```

Whenever two `->` signs are adjacent in a type, the order of association is from right to left, exactly the opposite convention of functional application. So `A -> B -> C` means `A -> (B -> C)`. If you want to describe the type `(A -> B) -> C` you have to put in the parentheses. More on this in the next chapter.

Having understood precisely what is given and what is wanted, different people come up with different ways of solving the problem, and express different worries about various parts of the problem. For example, what is a 'word' and how do you convert a list of characters into a list of words? Are the words `"Hello"`, `"hello"` and `"Hello!"` distinct words or the same word? How do you count words? Do you count all the words or just the most common ones? And so on. Some find these details daunting and overwhelming. Most seem to agree that at some intermediate point in the computation we have to come up with a list of words and their frequencies, but how do we get from there to the final destination? Do we go through the list `n` times, extracting the word with the next highest frequency at each pass, or is there something better?

Let's start with what a word is, and just assert that a word is a maximal sequence of characters not containing spaces or newline characters. That allows words like `"Hello!"`, or `"3*4"` or `"Thelma&Louise"` but never mind. In a text a word is identified by being surrounded by blank characters, so `"Thelma and Louise"` contains three words.

We are not going to worry about how to split a text up into a list of its component words. Instead we just assume the existence of a function

```
words :: [Char] -> [[Char]]
```

that does the job. Types like `[[Char]]` can be difficult to comprehend, but in Haskell we can always introduce *type synonyms*:

```
type Text = [Char]
type Word = [Char]
```

So now we have `words :: Text -> [Word]`, which is much easier on the brain. Of course, a text is different from a word in that the former can contain blank characters and the latter cannot, but type synonyms in Haskell do not support such subtle distinctions. In fact, `words` is a library function in Haskell, so we don't have to define it ourselves.

There is still the issue of whether `"The"` and `"the"` denote the same or different words. They really should be the same word, and one way of achieving this is to convert all the letters in the text to lowercase, leaving everything else unchanged. To this end, we need a function `toLower :: Char -> Char` that converts uppercase letters to lowercase and leaves everything else unchanged. In order to apply this function to every character in the text we need a general function

```
map :: (a -> b) -> [a] -> [b]
```

such that `map f` applied to a list applies `f` to every element of the list. So, converting everything to lowercase is done by the function

```
map toLower :: Text -> Text
```

Good. At this point we have `words . map toLower` as the function which converts a text into a list of words in lowercase. The next task is to count the number of occurrences of each word. We could go through the list of words, checking to see whether the next word is new or has been seen before, and either starting a new count for a new word or incrementing the count for an existing word. But there is a conceptually simpler method, namely to *sort* the list of words into alphabetical order, thereby bringing all duplicated words together in the list. Humans would not do it this way, but the idea of sorting a list to make information available is probably the single most important algorithmic idea in computing. So, let us assume the existence of a function

```
sortWords :: [Word] -> [Word]
```

that sorts the list of words into alphabetical order. For example,

```
sortWords ["to","be","or","not","to","be"]
        = ["be","be","not","or","to","to"]
```

Now we want to count the runs of adjacent occurrences of each word in the sorted list. Suppose we have a function

```
countRuns :: [Word] -> [(Int,Word)]
```

that counts the words. For example,

```
countRuns ["be","be","not","or","to","to"]
  = [(2,"be"),(1,"not"),(1,"or"),(2,"to")]
```

The result is a list of words and their counts in alphabetical order of the words.

Now comes the key idea: we want the information in the list to be ordered not by word, but by decreasing order of count. Rather than thinking of something more clever, we see that this is just another version of sorting. As we said above, sorting is a *really* useful method in programming. So suppose we have a function

```
sortRuns :: [(Int,Word)] -> [(Int,Word)]
```

that sorts the list of runs into descending order of count (the first component of each element). For example,

```
sortRuns [(2,"be"),(1,"not"),(1,"or"),(2,"to")]
       = [(2,"be"),(2,"to"),(1,"not"),(1,"or")]
```

The next step is simply to take the first `n` elements of the result. For this we need a function

```
take :: Int -> [a] -> [a]
```

so that `take n` takes the first `n` elements of a list of things. As far as `take` is concerned it doesn't matter what a 'thing' is, which is why there is an `a` in the type signature rather than `(Int,Word)`. We will explain this idea in the next chapter.

The final steps are just tidying up. We first need to convert each element into a string so that, for example, `(2,"be")` is replaced by `"be  2\n"`. Call this function

```
showRun :: (Int,Word) -> String
```

The type `String` is a predeclared Haskell type synonym for `[Char]`. That means

```
map showRun :: [(Int,Word)] -> [String]
```

is a function that converts a list of runs into a list of strings.

The final step is to use a function

```
concat :: [[a]] -> [a]
```

that concatenates a list of lists of things together. Again, it doesn't matter what the 'thing' is as far as concatenation is concerned, which is why there is an `a` in the type signature.

Now we can define

```
commonWords :: Int -> Text -> String
commonWords n = concat . map showRun . take n .
                sortRuns . countRuns . sortWords .
                words . map toLower
```

The definition of `commonWords` is given as a pipeline of eight component functions glued together by functional composition. Not every problem can be decomposed into component tasks in quite such a straightforward manner, but when it can, the resulting program is simple, attractive and effective.

Notice how the process of decomposing the problem was governed by the declared types of the subsidiary functions. Lesson Two (Lesson One being the importance of functional composition) is that deciding on the type of a function is the very first step in finding a suitable definition of the function.

We said above that we were going to write a *program* for the common words problem. What we actually did was to write a functional definition of `commonWords`, using subsidiary definitions that we either can construct ourselves or else import from a suitable Haskell library. A list of definitions is called a *script*, so what we constructed was a script. The order in which the functions are presented in a script is not important. We could place the definition of `commonWords` first, and then define the subsidiary functions, or else define all these functions first, and end up with the definition of the main function of interest. In other words we can tell the story of the script in any order we choose. We will see how to compute with scripts later on.

1.4 Example: numbers into words

Here is another example, one for which we will provide a complete solution. The example demonstrates another fundamental aspect of problem solving, namely that a good way to solve a tricky problem is to first simplify the problem and then see how to solve the simpler problem.

Sometimes we need to write numbers as words. For instance

```
convert 308000 = "three hundred and eight thousand"
convert 369027 = "three hundred and sixty-nine thousand and
                  twenty-seven"
convert 369401 = "three hundred and sixty-nine thousand
                  four hundred and one"
```

Our aim is to design a function

```
convert :: Int -> String
```

that, given a nonnegative number less than one million, returns a string that represents the number in words. As we said above, `String` is a predeclared type synonym in Haskell for `[Char]`.

We will need the names of the component numbers. One way is to give these as three lists of strings:

```
> units, teens, tens :: [String]
> units = ["zero","one","two","three","four","five",
>          "six","seven","eight","nine"]
> teens = ["ten","eleven","twelve","thirteen","fourteen",
>          "fifteen","sixteen","seventeen","eighteen",
>          "nineteen"]
> tens  = ["twenty","thirty","forty","fifty","sixty",
>          "seventy","eighty","ninety"]
```

Oh, what is the > character doing at the beginning of each line above? The answer is that, in a script, it indicates a line of Haskell code, not a line of comment. In Haskell, a file ending with the suffix `.lhs` is called a *Literate Haskell Script* and the convention is that every line in such a script is interpreted as a comment unless it begins with a > sign, when it is interpreted as a line of program. Program lines are not allowed next to comments, so there has to be at least one blank line separating the two. In fact, the whole chapter you are now reading forms a legitimate `.lhs` file, one that can be loaded into a Haskell system and interacted with. We won't carry on with this convention in subsequent chapters (apart from anything else, it would force us to use different names for each version of a function that we may want to define) but the present chapter does illustrate *literate* programming in which we can present and discuss the definitions of functions in any order we wish.

Returning to the task in hand, a good way to tackle tricky problems is to solve a simpler problem first. The simplest version of our problem is when the given number n contains only one digit, so $0 \leq n < 10$. Let `convert1` deal with this version. We can immediately define

```
> convert1 :: Int -> String
> convert1 n = units!!n
```

This definition uses the list-indexing operation `(!!)`. Given a list `xs` and an index `n`, the expression `xs!!n` returns the element of `xs` at position `n`, counting from 0. In particular, `units!!0 = "zero"`. And, yes, `units!!10` is undefined because `units` contains just ten elements, indexed from 0 to 9. In general, the functions we define in a script are *partial* functions that may not return well-defined results for each argument.

The next simplest version of the problem is when the number n has up to two digits, so $0 \leq n < 100$. Let `convert2` deal with this case. We will need to know what the digits are, so we first define

```
> digits2 :: Int -> (Int,Int)
> digits2 n = (div n 10, mod n 10)
```

The number `div n k` is the whole number of times `k` divides into `n`, and `mod n k` is the remainder. We can also write

```
    digits2 n = (n `div` 10, n `mod` 10)
```

The operators `` `div` `` and `` `mod` `` are infix versions of `div` and `mod`, that is, they come between their two arguments rather than before them. This device is useful for improving readability. For instance a mathematician would write $x \operatorname{div} y$ and $x \bmod y$ for these expressions. Note that the back-quote symbol `` ` `` is different from the single quote symbol `'` used for describing individual characters.

Now we can define

```
> convert2 :: Int -> String
> convert2 = combine2 . digits2
```

The definition of `combine2` uses the Haskell syntax for *guarded equations*:

```
> combine2 :: (Int,Int) -> String
> combine2 (t,u)
>    | t==0         = units!!u
>    | t==1         = teens!!u
>    | 2<=t && u==0 = tens!!(t-2)
>    | 2<=t && u/=0 = tens!!(t-2) ++ "-" ++ units!!u
```

To understand this code you need to know that the Haskell symbols for equality and comparison tests are as follows:

Cambridge University Press
978-1-107-08720-0 - Thinking Functionally with Haskell
Richard Bird
Excerpt
More information


```
==   (equals to)
/=   (not equals to)
<=   (less than or equal to)
```

These functions have well-defined types that we will give later on.

You also need to know that the conjunction of two tests is denoted by `&&`. Thus `a && b` returns the boolean value `True` if both `a` and `b` do, and `False` otherwise. In fact

```
(&&) :: Bool -> Bool -> Bool
```

The type `Bool` will be described in more detail in the following chapter.

Finally, `(++)` denotes the operation of concatenating two lists. It doesn't matter what the type of the list elements is, so

```
(++) :: [a] -> [a] -> [a]
```

For example, in the equation

```
[sin,cos] ++ [tan] = [sin,cos,tan]
```

we are concatenating two lists of functions (each of type `Float -> Float`), while in

```
"sin cos" ++ " tan" = "sin cos tan"
```

we are concatenating two lists of characters.

The definition of `combine2` is arrived at by carefully considering all the possible cases that can arise. A little reflection shows that there are three main cases, namely when the tens part `t` is 0, 1 or greater than 1. In the first two cases we can give the answer immediately, but the third case has to be divided into two subcases, namely when the units part `u` is 0 or not 0. The order in which we write the cases, that is, the order of the individual guarded equations, is unimportant as the guards are disjoint from one another (that is, no two guards can be true) and together they cover all cases.

We could also have written

```
combine2 :: (Int,Int) -> String
combine2 (t,u)
  | t==0      = units!!u
  | t==1      = teens!!u
  | u==0      = tens!!(t-2)
  | otherwise = tens!!(t-2) ++ "-" ++ units!!u
```