Chapter 7

Higher-Order Functions

Connor Baker

Compiled on December 21, 2019 at 9:04pm

7.1 Basic Concepts

Functions with multiple arguments are defined with currying (which is part of why we have right-associative groupings for parameters). As an example, we can rewrite

```
add :: Int -> Int -> Int
add xy = x + y
```

as

```
add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)
```

which makes it clear that add is a function that takes an integer and returns another function which takes an integer and maps to the sum `x + y`.

As an example, a function which takes as arguments a function and a value and returns the result of the application of the function to the value could be written as

```
applyMe :: (a -> a) -> a -> a
applyMe f x = f x
```

As a result of using curried functions to define multi-argument functions, we can use partial application. Consider the following function `succ` which yields the successor of a natural number:

```
succ :: Int -> Int
succ = applyMe (+1)
```

As a definition, a function which takes multiple arguments or maps to a function (that is to say that it *returns* a function) is called a *higher-order function*. We have a small conflict as *curried* functions return functions, and so within the context of Haskell we take higher-order functions to mean functions which takes arguments that are functions.

7.2 Processing Lists

One of the more commonly used functions in Haskell's standard library, `map` takes a function as an argument and applies it to every element of a list. It can be defined as

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

The `map` function is described a *polymorphic* because it can be applied to lists of any type.

As an aside, with the previous chapter's focus on recursion behind us, we could re-write map as a recursive function if we so chose to:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

The `map` function can also be applied to itself to operate on a list of lists. As an example, if we had a list of lists of integers and we wished to double every value, we could define `deepDouble` to do so:

```
deepDouble :: [[Int]] -> [[Int]
deepDouble = map (map (*2))
```

The `filter` function is another useful higher-order function included in Haskell's standard library:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

The function `filter` creates a list of all the element from the list passed in as an argument which satisfy the predicate `p`.

As an example, if we wanted to remove all the of the spaces in a `String` we could do

```
> filter (/= ' ') "a toddler wrote me"
"atoddlerwroteme"
```

As another exercise in recursion, we can write `filter` as a recursive function:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Typically, `map` and `filter` are used in conjunction: `filter` is applied to a list to remove undesirable elements, and then `map` is invoked on the result.

There are several other higher-order functions present in Prelude which are worth mentioning:

```
> all even [2,4,6,8]
True
```

```
> any odd [2,4,6,8]
False
```

```
> takeWhile even [2,4,5,6,8]
[2,4]
```

```
> dropWhile even [2,4,5,6,8]
[5,6,8]
```

7.3 The foldr Function

As you probably noticed from the examples above, most functions that operate on lists can be defined using a simple recursive pattern:

```
f []     = v
f (x:xs) = x ? f xs
```

where `v` is some value, and `?` is some operator.

For practice, we redefine the `sum`, `product`, `or`, and `and` functions to operate on lists:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

```
product [] = 1
product (x:xs) = x * product xs
```

```
or [] = False
or (x:xs) = x || or xs
```

```
and [] = True
and (x:xs) = x && and xs
```

Given how similar the definition of all of these functions are, you would think that we can make a function that applies some sort of operation recursively… and you'd be right! Enter: `foldr`.

The higher-order library function `foldr` works using the simple recursive pattern above, taking as arguments `?` and `v`. Redefining the functions above, we can write each of them in a single line (if we omit the type annotation, which is inferred by the compiler anyways):

```
sum = foldr (+) 0
```

```
product = foldr (*) 1
```

```
or = foldr (||) False
```

```
and = foldr (&&) True
```

Note: Remember that operators need to be parenthesized when used as arguments.

Probably unsurprisingly, the `foldr` function can also be defined using recursion (though the type annotation might be surprising):

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Let's break down the type of `foldr` (I myself didn't understand it until I read https://stackoverflow.com/a/11425454, which also helps explain how the `length` function could be rewritten in terms of `foldr`).

We can see that there are three arguments: a function, the starting value of an accumulator, and a list. The function argument takes an element of a list (the `a`), an accumulator (that's the first `b`), and returns the accumulator. After execution, `foldr` returns the result of the accumulator.

As an example of how `foldr` is applied, consider the following:

```
foldr (+) 0 [1,2,3]
```

is equivalent to

```
foldr (+) 0 1:(2:(3:[]))
```

which, after the application of `foldr`, turns into the expression

```
1+(2+(3+0))
```

As an example, let's re-write the library functions `length` and `reverse` using `foldr`.

We know that `length` is defined as

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

We start by constructing the first argument of `foldr`. Just like the definition above, we shouldn't care what elements of the list we're dealing with. We know that our accumulator will be a number, and that the accumulator should increase by one after each element is processed. As such, we end up with the lambda below:

```
(\_ n -> 1 + n)
```

Keep in mind that the `\` only indicates that we're declaring a lambda function, not that `_` is the only argument of the lambda function.

The second argument of `foldr` is the value that the accumulator starts at, which will be zero, per the definition of `length` (in the case that the list is empty, `foldr` has no elements to operate on so it returns the initial value of the accumulator). All together, we have

```
length :: [a] -> Int
length = foldr (\_ n -> 1 + n) 0
```

We don't need to write the list `xs` on either side since we can take advantage of partial application (alternatively, we can say that `length` is a mapping *defined by* `foldr (\_ n -> 1 + n)0`).

For a more thorough explanation, consult https://stackoverflow.com/a/11425454.

We know that the definition of reverse is

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

To better get a sense of how we might re-write it with `foldr`, let's work through an example. If we have the list

```
1 : (2 : (3 : []))
```

and we invoke reverse, we have

```
(([] ++ [3]) ++ [2]) ++ [1]
```

We can see that the result is equivalent to concatenating each element to the end of a list:

```
3:(2:(1:[]))
```

Beginning again by constructing the function argument of `fodlr`, we have

```
(\x y -> y ++ [x])
```

Note: We must use `[x]` because `x` is not a list, and `(++)` is only defined to work on lists.

The default value of the accumulator in this case is the empty set, so our definition of `reverse` in terms of `foldr` is

```
reverse :: [a] -> [a]
reverse = foldr (\x y -> y ++ [x]) []
```

The function is called `foldr` because it is assumed that the function passed as an argument is right-associative. In general, the behavior of `foldr` is summarized as

```
foldr (?) v [x0,x1,...,xn] = x0 ? (x1 ? (... (xn ? v) ...))
```

7.4 The foldl Function

Just like we have a `foldr`, we have a function that handles left-associative operators: `foldl`. Instead of the simple recursive pattern we identified in the previous section, consider the following pattern:

```
f v [] = v
f v (x:xs) = f (v ? x) xs
```

Just as we redefined the `sum`, `product`, `or`, `and`, `length`, and `reverse` functions in terms of `foldr`, we can do the same for them in terms of `foldl`:

```
sum = foldl (+) 0
```

```
product = foldl (*) 1
```

```
or = foldl (||) False
```

```
and = foldl (&&) True
```

```
length = foldl (\n _ -> 1 + n) 0
```

```
reverse = foldl (\x y -> y:x) []
```

Note: The first four functions didn't have to change anything other than the type of fold because those operators are associative.

Note: The choice of which function to use when both work for a given operator is driven by a need for efficiency, which requires consideration of the evaluation mechanism Haskell uses.

We can define `foldl` using recursion, just like we did for `foldr`

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []     = v
foldl f v (x:xs) = foldl f (f v x) xs
```

The function `foldl` can be summarized as

```
foldl ? v [x0,x1,...,xn] =  (... ((v ? xn) ? x1) ...) ? x0
```

7.5 The Composition operator

The higher-order library operator `.` returns the composition of two functions as a single function. It can be defined as

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

We could also have written the definition as `(f . g)x = f (g x)`, but the definition above uses a lambda to make clear the fact that `(.)` returns a function.

One use of composition is to reduce the number of parenthesis in your code. We could write

```
sumCubeOfOdd = sum (map (^3) (filter odd))
```

as

```
sumCubeOfOdd = sum . map (^3) . filter odd
```

Composition has an identity:

```
id :: a -> a
id = \x -> x
```

The identity is a good place to start when trying to work with a sequence of compositions. Consider the composition of a list of functions:

```
compose :: [a -> a] -> (a -> a)
compose = foldr (.) id
```

7.6 Binary String Transmitter

The rest of the chapter consists of two longer programming exercises. The first is about simulating the transmission of a string of characters as a list of binary digits.

Binary Numbers

Omitted as I assume that my readers are familiar with the topic. If not, consult some other sources; I like Wolfram Mathworld for all topics related to mathematics (here's their entry on Binary Numbers).

The one take away from this section was that we can express a number as a sum of its digits to powers of the index of the digit. Generalizing this concept we have the concept of the β-expansion (or radix-point-expansion). It states that a number n with i digits $n_{i-1} \dots n_0$ in base β can be written as

$$n = n_{i-1} \times \beta^{i-1} + \cdots + n_0 \times \beta^0$$

If you prefer summation notation, it would be

$$n = \sum_{k=0}^{i-1} n_k \times \beta^k$$

So if we had a number 101.1001_φ, where φ is the golden ratio (base golden ratio, also called Phinary, has some interesting properties), can be written as

$$1 \times \varphi^2 + 0 \times \varphi^2 + 1 \times \varphi^0 + 1 \times \varphi^{-1} + 0 \times \varphi^{-2} + 0 \times \varphi^{-3} + 1 \times \varphi^{-4}$$

Base Conversion

Since we'll be dealing with characters, it's prudent to add `import Data.Char` to our workspace.

To make the purpose of our functions more clear, we'll define a new *type* `Bit`, which will be a synonym for integers:

```
type Bit = Int
```

A binary number is no different from any other type of number, and it is helpful to be able to convert them to decimal notation, so we define a function to do so

```
bin2Int :: [Bit] -> Int
bin2Int bits = sum [w*b | (w,b) <- zip weights bits]
where weights = iterate (*2) 1
```

The higher-order library function `iterate` creates an infinite list by repeatedly applying a function to an initial value. Visually,

```
iterate f x = [x, f x, (f . f) x, (f . f . f) x]
```

Note: Our function assumes that the binary input is Little-Endian; that is to say that the least significant bit (LSB) is at the lowest index. So in effect, our list is a "normal" binary number in reverse!

Let's instead try to think of a more succinct. To do that, let's trace what our ideal version of the function would do, for some four bit input `[a,b,c,d]`.

```
 > [a,b,c,d]
-> (1 * a) + (2 * b) + (4 * c) + (8 * d)
```

It looks like high-school algebra, so let's borrow a technique to manipulate it: factoring!

```
-> (1 * a) + (2 * b) + (4 * c) + (8 * d)
-> a + 2 * (b + 2 * (c + 2 * (d + 2 * 0)))
```

Well this looks familiar! (If it doesn't go peek back at the summarized form of `foldr`.) Each element is being added to twice the second element, except for the last element.

```
bin2Int :: [Bit] -> Int
bin2Int = foldr (\x y -> x + 2*y) 0
```

Let's also create a function that converts a non-negative integer to a binary sequence.

```
int2Bin :: Int -> [Bit]
int2Bin 0 = []
int2Bin n = mod n 2 : int2Bin (div n 2)
```

For the purpose of this problem, we restrict all of our numbers to a constant length. We use the function

```
make8 :: [Bit] -> [Bit]
make8 bits = take 8 (bits ++ repeat 0)
```

to truncate or pad our input (with zeros) to the correct length.

Note: Times like these are where the beauty of lazy evaluation comes in to play; lazy evaluation ensures that, even though repeat generates an infinite list, if bits is already eight elements or more, the list is not created.

Transmission

Having set up all the of the prerequisite functions, we can make a one-liner that takes a string of characters, converts each into a Unicode number, converts each number into binary, and concatenate the result. We'll name the function `encode`, because we're real, and we call it like we see it.

```
encode :: String -> [Bit]
encode = concat . map (make8 . int2Bin . ord)
```

Note: If you've forgotten, `Data.Char`'s `ord` takes a character and returns the ASCII value.

To decode the result of `encode`, we need to split the list every eight elements, turn the binary number into a decimal value, and turn that decimal value into a character.

```
chop8 :: [Bit] -> [[Bit]]
chop8 [] = []
chop8 bits = take 8 bits : chop8 (drop 8 bits)
```

```
decode :: [Bit] -> String
```

```
decode = map (chr . bin2Int) . chop8
```

Lastly, let's define a function that simulates transmitting a string of characters as a list of binary bits. To make things easier (and avoid digressing into a post about information theory), we'll assume that the transmission channel is perfect and implement it using the identity function:

```
channel :: [Bit] -> [Bit]
channel = id
```

```
transmit :: String -> String
transmit = decode . channel . encode
```

7.7 Voting Algorithms

The second programming exercise looks at two algorithms for deciding the winner of an election: the first algorithm is the easy-to-implement *first past the post*; the second is the more refined *alternative vote* system.

First Past the Post

In this system, each person gets one vote, and the winner of the election is the candidate with the largest number of votes.

To implement this method, we'll need a function that counts the number of appearances of a value in a list, with the caveat that type is comparable.

```
count :: Eq a => a -> [a] -> Int
count x = length . filter (== x)
```

We can also create a function which gives us a list of all the candidates that got any votes:

```
candidates :: Eq a => [a] -> [a]
candidates [] = []
candidates (x:xs) = x : filter (/= x) (candidates xs)
```

With these two functions, we can create a function which gives us the results of the vote:

```
results :: Eq a -> [a] -> [(Int,a)]
results votes = [(count v votes, v) | v <- candidates votes]
```

If we want to sort the results, we can use the `sort` function provided by `Data.List`:

```
sortedResults :: Ord a -> [a] -> [(Int,a)]
sortedResults votes = sort [(count v votes, v) | v <- candidates votes]
```

We can find the winner by taking the second element of the last tuple:

```
winner :: Ord a => [a] -> a
winner = snd . last . result
```

Alternative Vote

In the alternative vote system, each person can vote for as many candidates as they want, but they must order them based on preference. (So their most desired candidate is first, and their least desired candidate is last.)

The winner is found by the following process:

1. Remove all the empty ballots
2. Remove the candidate that was chosen the fewest number of times as the first choice (this causes candidates on the ballot to shift left to fill the gap)
3. Repeat the process

Creating a function that removes empty ballots isn't too difficult

```
rmEmpty :: Eq a => [[a]] -> [[a]]
rmEmpty = filter (/= [])
```

Nor is creating a function that removes a given candidate from all the ballots

```
elimCandidate :: Eq a => a -> [[a]] -> [[a]]
elimCandidate x = map (filter (/= x))
```

Because we defined all our functions in a general manner, we can reuse the `sortedResults` function from the previous section to help us with ranking candidates in each ballot

```
rank :: Ord a => [[a]] -> [a]
rank = map snd . sortedResults . map head
```

To implement the last step of the process which finds a winner, we can implement a recursive function which applies the first two steps.

```
winnerAlt :: Ord a => [[a]] -> a
winnerAlt ballots = case rank (rmEmpty ballots) of
    [c]    -> c
    (c:cs) -> winnerAlt (elimCandidate c ballots)
```

*Note: The **`case`** mechanism allows us to perform pattern matching in the body of a definition.*

7.8 Chapter Remarks

There are several real-world applications of higher-order functions that crop up in common environments:

- Digital production of music
- Financial contracts
- Digital graphics
- Hardware descriptions
- Logic programs