Chapter 4

Defining Functions

Connor Baker

Compiled on December 21, 2019 at 9:04pm

4.1 New From Old

Function composition is an extremely powerful tool and should not be overlooked. If it works for Unix, it'll probably work for you.

4.2 Conditional Expressions

Conditional expressions in Haskell look a bit different than those in bog-standard imperative languages. Consider an implementation of the `signum` function for integers:

```
signum :: Int -> Int
signum n = if n < 0 then -1
           else if n == 0 then 0
           else 1
```

We can see from this example that conditional expressions may be nested.

> Note that unlike in some programming languages, conditional expressions in Haskell must always have an else branch, which avoids the well-known dangling else problem.
>
> Excerpt From: Graham Hutton. "Programming in Haskell" (2nd ed.).

4.3 Guarded Equations

Guarded equations are a sequence of logical expressions evaluated in order (think of it like **`switch`** statement meets **`if`**):

```
signum :: Int -> Int
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

The guard symbol | is read as such that.

> … the guard otherwise is defined in the standard prelude simply by `otherwise = True`. Ending a sequence of guards with otherwise is not necessary, but provides a convenient way of handling all other cases..
>
> Excerpt From: Graham Hutton. "Programming in Haskell" (2nd ed.).

4.4 Pattern Matching

Functions can also be defined via pattern matching, which is a list of expressions. As the program executes, the first pattern to match is chosen. As an example, consider logical conjunction:

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

We can simplify this by using the wildcard pattern _ which matches any value (it serves as a sort of else):

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False
```

> This version also has the benefit that, under lazy evaluation as discussed in chapter 15, if the first argument is False, then the result False is returned without the need to evaluate the second argument.
>
> Excerpt From: Graham Hutton. “Programming in Haskell (2nd ed.).

Tuple patterns

> A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

Pattern matching works exactly how you’d expect (component-wise, if you had forgotten what you were expecting):

```
fst :: (a,b) -> a
fst (x,_) = x
```

```
snd :: (a,b) -> b
snd (_,y) = y
```

List patterns

Lists are not primitives in Haskell.

> Lists are constructed one element at a time starting from the empty list `[]` using an operator `:` called cons that constructs a new list by prepending a new element to the start of an existing list…. the library functions `head` and `tail` that respectively select and remove the first element of a non-empty list are defined as follows:
>
> ```
> head :: [a] -> a
> head (x:_) = x
> ```
>
> ```
> tail :: [a] -> [a]
> tail (_:xs) = xs
> ```

> Note that cons patterns must be parenthesised, [sic] because function application has higher priority than all other operators in the language.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

4.5 Lambda expressions

> As an alternative to defining functions using equations, functions can also be constructed using lambda expressions, which comprise a pattern for each of the arguments, a body that specifies how the result can be calculated in terms of the arguments, but do not give a name for the function itself. In other words, lambda expressions are nameless functions.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

Haskell uses the symbol `\` to represent the Greek letter lambda, written as λ. Consider the lambda expression

```
\x -> x + x
```

which adds a number to itself. One might use the lambda expression in GHCi to double a number:

```
> (\x -> x + x) 2
4
```

These lambda expressions can define curried functions. Consider that we can rewrite

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

as

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

> … which makes precise that `add` is a function that takes an integer `x` and returns a function, which in turn takes another integer `y` and returns the result `x + y`. Moreover, rewriting the original definition in this manner also has the benefit that the type for the function and the manner in which it is defined now have the same syntactic form, namely `? -> (? -> ?)`.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

Lambda expressions are a natural way to define functions that return functions as part of how they work rather simply as a result of currying.

> … the library function const that returns a constant function that always produces a given value can be defined as follows:
>
> ```
> const :: a -> b -> a
> const x _ = x
> ```

> However, it is more appealing to define **`const`** in a way that makes explicit that it returns a function as its result, by including parentheses in the type and using a lambda expression in the definition itself:
>
> ```
> const :: a -> (b -> a)
> const x = _ -> x
> ```
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

Lastly, lambda expressions help get rid of the need to define small, single-use helper functions.

> … a function `odds` that returns the first `n` odd integers can be defined as follows:
>
> ```
> odds :: Int -> [Int]
> odds n = map f [0..n-1]
>        where f x = x*2 + 1
> ```
>
> (The library function map applies a function to all elements of a list.) However, because the locally defined function `f` is only referenced once, the definition for `odds` can be simplified by using a lambda expression:
>
> ```
> odds :: Int -> [Int]
> odds n = map (\x -> x*2 + 1) [0..n-1]
> ```
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

4.6 Operator Sections

Functions written between two arguments are called operators (formally known as an infix operator). Any function with two arguments can be converted into an operator by putting the function in-between back-ticks. The opposite is also possible: putting parenthesis around an operator turns it into a prefix operator.

> In general, if `#` is an operator, then expressions of the form `(#)`, `(x #)`, and `(# y)` for arguments `x` and `y` are called sections, whose meaning as functions can be formalised [sic] using lambda expressions as follows:
>
> `(#)= \x -> (\y -> x # y)`
>
> `(x #)= \y -> x # y`
>
> `(# y)= \x -> x # y`
>
> Sections have three primary applications. First of all, they can be used to construct a number of simple but useful functions in a particularly compact way, as shown in the following examples:
>
> - `(+)` is the addition function
>   - Example: `\x -> (\y -> x+y)`
> - `(1+)` is the successor function
>   - Example: `\y -> 1+y`

> - `(1/)` is the reciprocation function
>   - Example: `\y -> 1/y`
> - `(*2)` is the doubling function
>   - Example: `\x -> x*2`
> - `(/2)` is the halving function
>   - Example: `\x -> x/2`
>
> Secondly, sections are necessary when stating the type of operators, because an operator itself is not a valid expression in Haskell… Finally, sections are also necessary when using operators as arguments to other functions.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

4.7 Chapter Remarks

> The Greek letter λ used when defining nameless functions comes from the lambda calculus, the mathematical theory of functions upon which Haskell is founded.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).