Chapter 6

Functions upon Functions upon Functions

Now would be an appropriate moment to go back to Chapter 2 and look at the extra material and questions on types and typeclasses, if you have not yet done so.

Often we need to apply a function to every element of a list. For example, doubling each of the numbers in a list. We could do this with a simple recursive function, working over each element of the list:

```
doubleList :: Num a ⇒ [a] → [a]

doubleList []     = []                          no element to process
doubleList (x:xs) = (x * 2) : doubleList xs     process element, and the rest
```

For example,

```
        doubleList [1, 2, 4]
⟹      1 * 2 : doubleList [2, 4]
⟹      1 * 2 : 2 * 2 : doubleList [4]
⟹      1 * 2 : 2 * 2 : 4 * 2 : doubleList []
⟹      1 * 2 : 2 * 2 : 4 * 2 : []
⟹*     [2, 4, 8]
```

The result list does not need to have the same type as the argument list. For example, we can write a function which, given a list of numbers, returns the list containing a boolean for each: `True` if the number is even, `False` if it is odd.

```
evens :: Integral a ⇒ [a] → [Bool]

evens []     = []                               no element to process
evens (x:xs) = (x `rem` 2 == 0) : evens xs      process element, and the rest
```

For example,

```
        evens [1, 2, 4]
⟹      False : evens [2, 4]
⟹      False : True : evens [4]
⟹      False : True : True : evens []
⟹      False : True : True : []
⟹*     [False, True, True]
```

It would be tedious to write a similar function each time we wanted to apply a different operation to every element of a list – can we build one which works for any operation? We will use a function as an argument:

```
map' :: (a → b) → [a] → [b]

map' f []     = []                          no element to process
map' f (x:xs) = f x : map' f xs             process the element, and the rest
```

The `map'` function takes two arguments: a function which processes a single element, and a list. It returns a new list. We will discuss the type in a moment. For example, if we have a function `halve`:

```
halve :: Integral a ⇒ a → a

halve x = x `div` 2
```

Then we can use `map'` like this:

```
        map' halve [10, 20, 30]
⟹      10 `div` 2 : map' halve [20, 30]
⟹      10 `div` 2 : 20 `div` 2 : map' halve [30]
⟹      10 `div` 2 : 20 `div` 2 : 30 `div` 2 : map' halve []
⟹      10 `div` 2 : 20 `div` 2 : 30 `div` 2 : []
⟹*     [5, 10, 15]
```

Now, let us look at that type: $(a \to b) \to [a] \to [b]$. We can annotate the individual parts:

$$\overbrace{(a \to b)}^{\text{function f}} \to \overbrace{[a]}^{\text{argument list}} \to \overbrace{[b]}^{\text{result list}}$$

We have to put the function `f` in parentheses, otherwise it would look like `map'` had four arguments. This function can have any type $a \to b$. That is to say, it can have any argument and result types, and they do not have to be the same as each other – though they may be. The argument has type $[a]$ because each of its elements must be an appropriate argument for `f`. In the same way, the result list has type $[b]$ because each of its elements is a result from `f` (in our `halve` example, `a` and `b` were both numbers in typeclass **Integral**). We can rewrite our `evens` function to use `map'`:

```
isEven :: Integral a ⇒ a → Bool
evens :: Integral a ⇒ [a] → [Bool]

isEven x = x `rem` 2 == 0

evens l = map' isEven l
```

In this use of `map`, type `a` was **Integral** `a` ⇒ `a`, and type `b` was **Bool**. We can make `evens` still shorter: when we are just using a function once, we can define it directly, without naming it:

```
evens :: Integral a ⇒ [a] → [Bool]

evens l =
  map' (\x -> x `rem` 2 == 0) l
```

This is called an *anonymous function*. It is defined using `\`, a named argument, the `->` arrow and the function definition (body) itself. For example, we can write our halving function like this:

```
\x -> x `div` 2
```

and, thus, write:

```
        map' (\x -> x `div` 2) [10, 20, 30]
⟹*      [5, 10, 15]
```

We use anonymous functions when a function is only used in one place and is relatively short, to avoid defining it separately.

In the preceding chapter we wrote a sorting function and, in one of the questions, you were asked to change the function to use a different comparison operator so that the function would sort elements into reverse order. Now, we know how to write a version of the `mergeSort` function which uses any comparison function we give it. A comparison function would have type **Ord** `a` ⇒ `a` → `a` → **Bool**. That is, it takes two elements of the same type in typeclass **Ord**, and returns `True` if the first is "greater" than the second, for some definition of "greater" – or `False` otherwise.

So, let us alter our `merge` and `mergeSort` functions to take an extra argument – the comparison function. The result is shown in Figure 6.1. Now, if we make our own comparison operator:

```
greater :: Ord a ⇒ a → a → Bool

greater a b =
  a >= b
```

we can use it with our new version of the `mergeSort` function:

```
        mergeSort greater [5, 4, 6, 2, 1]
⟹*      [6, 5, 4, 2, 1]
```

```
merge :: (a → a → Bool) → [a] → [a] → [a]
mergeSort :: (a → a → Bool) → [a] → [a]

merge _ [] l = l
merge _ l [] = l
merge cmp (x:xs) (y:ys) =
  if cmp x y                                    use our comparison function
    then x : merge cmp xs (y : ys)              put x first – it is "smaller"
    else y : merge cmp (x : xs) ys              otherwise put y first

mergeSort _ [] = []
mergeSort _ [x] = [x]
mergeSort cmp l =
  merge cmp (mergeSort cmp left) (mergeSort cmp right)
    where left = take' (length' l `div` 2) l
          right = drop' (length' l `div` 2) l
```

Figure 6.1: Adding an extra argument to merge sort

In fact, we can ask Haskell to make such a function from an operator such as <= or + just by enclosing it in parentheses:

```
GHCi:
Prelude> :type (<=)
(<=) :: Ord a => a -> a -> a
Prelude> (<=) 4 5
True
```

So, for example:

```
        mergeSort (<=) [5, 4, 6, 2, 1]
⟹*      [1, 2, 4, 5, 6]
```

and

```
        mergeSort (>=) [5, 4, 6, 2, 1]
⟹*      [6, 5, 4, 2, 1]
```

Haskell provides a useful operator `.` to chain (or *compose*) functions together, so we may apply several functions to a value in order. For example, take the function `double`, we may write:

```
GHCi
Prelude> double x = x * 2
Prelude> (double . double) 6
24
```

The effect is to quadruple the number. We can write the function directly if we like:

```
GHCi
Prelude> quadruple x = (double . double) x
Prelude> quadruple 6
24
```

But, of course, we can drop the argument:

```
GHCi
Prelude> quadruple = double . double
Prelude> quadruple 6
24
```

This can lead to cleaner, easier to read programs, once you are used to it. See how we can find two ways to quadruple each element in a list using `double` and `map`:

```
GHCi:
Prelude> (map' double . map' double) [6, 4]
[24,16]
Prelude> map' (double . double) [6, 4]
[24,16]
```

The `.` or function composition operator has type $(b \to c) \to (a \to b) \to a \to c$. That is to say we give it a two functions which, when used in order, take something of type `a` to something of type `c` via something of type `b`, and something of type `a` and we get something of type `c` out. In our examples, `a`, `b`, and `c` were all numbers. Since the function composition operator is associative, that is `f . (g . h)` is equal to `(f . g) . h`, we do not need the parentheses, and may write simply `f . g . h`, for example `times8 = double . double . double`. We may define the `.` operator ourselves:

```
GHCi:
Prelude> f . g = \x -> f (g x)
```

We can, in fact, write it like this too:

```
GHCi:
Prelude> (f . g) x = f (g x)
```

See how we define an operator just like a function, since in reality it is one. As another example, consider defining `+++` to mean vector addition:

```
GHCi:
Prelude> (a, b) +++ (c, d) = (a + c, b + d)
```

The techniques we have seen in this chapter are forms of *program reuse*, which is fundamental to writing manageable large programs.

Questions

1. Write a simple recursive function `calm` to replace exclamation marks in a string with periods. For example `calm "Help! Fire!"` should evaluate to `"Help. Fire."`. Now rewrite your function to use `map'` instead of recursion. What are the types of your functions?
2. Write a function `clip` which, given a number, clips it to the range $1 \ldots 10$ so that numbers bigger than 10 round down to 10, and those smaller than 1 round up to 1. Write another function `clipList` which uses this first function together with `map'` to apply this clipping to a whole list of numbers.
3. Express your function `clipList` again, this time using an anonymous function instead of `clip`.
4. Write a function `apply` which, given another function, a number of times to apply it, and an initial argument for the function, will return the cumulative effect of repeatedly applying the function. For instance, `apply f 6 4` should return `f (f (f (f (f (f 4)))))`. What is the type of your function?
5. Modify the insertion sort function from the preceding chapter to take a comparison function, in the same way that we modified merge sort in this chapter. What is its type?
6. Write a function `filter'` which takes a function of type `a` $\rightarrow$ **Bool** and an [a] and returns a list of just those elements of the argument list for which the given function returns `True`.
7. Write the function `all'` which, given a function of type `a` $\rightarrow$ **Bool** and an argument list of type [**a**] evaluates to `True` if and only if the function returns `True` for every element of the list. Give examples of its use.
8. Write a function `mapl` which maps a function of type `a` $\rightarrow$ `b` over a list of type [[a]] to produce a list of type [[b]].
9. Use the function composition operator `.` together with any other functions you like, to build a function which, given a list, returns the reverse-sorted list of all its members which are divisible by fifteen.