Higher-order Functions
Informatics 1 – Introduction to Computation
Functional Programming Tutorial 5

Cooper, Heijltjes, Melkonian, Scott, Vlassi-Pandi, Wadler, Yallop

Week 6
due 4pm Wednesday 28 October 2020
tutorials on Friday 20 October 2020

Attendance at tutorials is obligatory; please send email to lambrose@ed.ac.uk if you cannot join your assigned tutorial.

Good Scholarly Practice: Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page

http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct.

This also has links to the relevant University pages. Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

1   Higher-order Functions

Haskell functions are *values*, which may be processed in the same way as other data such as numbers, tuples or lists. In this tutorial we'll use a number of *higher-order functions*, which take other functions as arguments, to write succinct definitions for the sort of list-processing tasks that you've previously coded explicitly using recursion or comprehensions.

The first part of the tutorial deals with three higher-order functions, `map`, `filter`, and `fold`. For each of these you will be asked to write several functions. The second part deals with `fold` in some more detail, and will ask you to write functions using both `map` and `filter` at the same time.

1.1   Map

Transforming every list element by a particular function is a common need when processing lists—for example, we may want to

- add one to each element of a list of numbers,
- extract the first element of every pair in a list,
- convert every character in a string to uppercase, or
- add a grey background to every picture in a list of pictures.

The `map` function captures this pattern, allowing us to avoid the repetitious code that results from writing a recursive function for each case.

$$x_1 \quad : \quad x_2 \quad : \quad \cdots \quad : \quad x_n \quad : \quad \texttt{[]}$$

$$f(x_1) \quad : \quad f(x_2) \quad : \quad \cdots \quad : \quad f(x_n) \quad : \quad \texttt{[]}$$
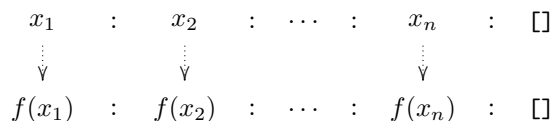
Figure 1: The `map` function

Consider a function `g` defined in terms of an imaginary function `f` as follows:

```
g []     = []
g (x:xs) = f x : g xs
```

The function `g` can be written with recursion (as above), or with a comprehension, or with `map`: all three definitions are equivalent.

```
g xs  =  [ f x | x <- xs ]
g xs  =  map f xs
```

Below right is the definition of `map`. Note the similarity to the recursive definition of `g` (below left). As compared with `g`, `map` takes one additional argument: the function `f` that we want to apply to each element.

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

Given `map` and a function that operates on a single element, we can easily write a function that operates on a list. For instance, the function that extracts the first element of every pair can be defined as follows (using `fst :: (a,b) -> a`):

2

```
fsts :: [(a,b)] -> [a]
fsts pairs = map fst pairs
```

Exercise 1

Using suitable library functions write definitions for the following:

(a) A function `doubles :: [Int] -> [Int]` that doubles every item in a list by using the `map` function.

(b) A function `penceToPounds :: [Int] -> [Float]` that turns prices given in pence into the same price in pounds by using the `map` function.

(c) A function `uppers :: String -> String` that converts a string to uppercase using the `map` function.

(d) A function `uppersComp` that behaves the same as `uppers` but uses a *list comprehension*. Use QuickCheck to check the two are equivalent.

1.2 Filter

Removing elements from a list is another common need. For example, we might want to remove non-alphabetic characters from a string, or negative integers from a list. This pattern is captured by the `filter` function.

Consider a function `g` defined in terms of an imaginary predicate `p` as follows (a predicate is just a function into a `Bool` value):

```
g []      = []
g (x:xs) | p x        = x : g xs
         | otherwise = g xs
```

The function `g` can be written with recursion (as above), or with a comprehension, or with `filter`: all three definitions are equivalent.

```
g xs  =  [ x | x <- xs, p x ]
g xs  =  filter p xs
```

For instance, we can write a function `evens :: [Int] -> [Int]`, which removes all odd numbers from a list using `filter` and the standard function `even :: Int -> Int`:

```
evens list = filter even list
```

This is equivalent to:

```
evens list = [x | x <- list, even x]
```

Below right is the definition of `filter`. Note the similarity to the way `g` is defined (below left). As compared with `g`, `filter` takes one additional argument: the predicate that we use to test each element.

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

Exercise 2

Using standard library functions, write definitions for the following:

(a) A function `alphas :: String -> String` that removes all non-alphabetic characters from a string by using the `filter` function.

(b) A function `above :: Int -> [Int] -> [Int]` that removes all numbers less than or equal to a given number by using the `filter` function.

(c) A function `unequals :: [(Int,Int)] -> [(Int,Int)]` that removes all pairs `(x,y)` where `x == y` by using the `filter` function.

(d) A function `rmChar :: Char -> String -> String` that removes all occurrences of a character from a string.

(e) A function `rmCharComp` that behaves the same as `rmChar` but uses a *list comprehension*. Use QuickCheck to check the two are equivalent.

1.3 Comprehensions, map and filter

As we have seen, list comprehensions process a list using transformations similar to `map` and `filter`.

```
[f x | x <- xs, p x]
is equivalent to
map f (filter p xs)
```

Exercise 3

Define functions that are equivalent to the following using `map` and `filter`. Use QuickCheck to verify your answers.

(a) `largeDoubles xs = [2 * x | x <- xs, x > 3]`

(b) `reverseEvens str = [reverse s | s <- strs, even (length s)]`

1.4 Fold

The `map` and `filter` functions act on elements individually; they never combine one element with another.

Sometimes we want to combine elements using some operation. For example, the `sum` function can be written like this:

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

Here we're essentially just combining the elements of the list using the `+` operation. Another example is `reverse`, which reverses a list:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

This function is just combining the elements of the list, one by one, by appending them onto the end of the reversed list. This time the "combining" function is a little harder to see. It might be easier if we wrote it this way:

```
reverse []     = []
reverse (x:xs) = x `snoc` reverse xs

snoc x xs = xs ++ [x]
```

Now you can see that `` `snoc` `` plays the same role as `+` played in the example of `sum`.

These examples (and many more) follow a pattern: we break down a list into its head (`x`) and tail (`xs`), recurse on `xs`, and then apply some function to `x` and the modified `xs`. The only things we need to specify are the function (such as (`+`) or `snoc`) and the *initial value* (such as `0` in the case of `sum` and `[]` in the case of `reverse`).

This pattern is called "a fold" and is implemented in Haskell via the function `foldr`.

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

The function `g` can be written with recursion (as above) or by using a fold: both definitions are equivalent.

```
 g xs = foldr f u xs
```

One way to visualize the action of `foldr` is shown in Figure 2. Given a function `f :: a -> b -> b`, an initial value `u :: b` (sometimes called the "unit"), and a list $[x_1, x_2, ..., x_n]$ of type `[a]`, the `foldr` function returns the value that results from replacing every `:` (cons) in `list` with `f` and replacing the terminating `[]` (nil) with `u`.
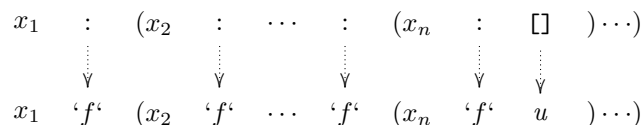
$$
\begin{array}{ccccccccccc}
x_1 & : & (x_2 & : & \cdots & : & (x_n & : & [] &) \cdots) \\
 & & & & & & & & & \\
x_1 & \text{`}f\text{`} & (x_2 & \text{`}f\text{`} & \cdots & \text{`}f\text{`} & (x_n & \text{`}f\text{`} & u &) \cdots)
\end{array}
$$

Figure 2: The `foldr` function

For example, we can define `sum :: [Int] -> Int` as follows, using (`+`) as the function and `0` as the initial value (unit):

```
 sum :: [Int] -> Int
 sum ns = foldr (+) 0 ns
```

(**Note:** to treat an infix operator like `+` as a function name, we need to wrap it in parentheses.)
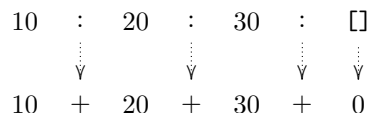
$$
\begin{array}{ccccccc}
10 & : & 20 & : & 30 & : & [] \\
 & & & & & & \\
10 & + & 20 & + & 30 & + & 0
\end{array}
$$

Figure 3: Illustration of `foldr (+) 0 [10,20,30]`

Exercise 4

> We will practice the use of `foldr` by writing several functions first with recursion, and then using `foldr`. You can use other standard library functions as well. For each pair of functions that you write, test them against each other using QuickCheck.
>
> (a) Write a recursive function `andRec :: [Bool] -> Bool` that returns true if every element in a list of booleans is true. Write an equivalent function `andFold` using `foldr`, and test the two with QuickCheck.

(b) Write a recursive function `concatRec :: [[a]] -> [a]` that concatenates a list of lists into a single list. Write an equivalent function `concatFold` using `foldr`, and test the two with QuickCheck.

(c) Write a recursive function `rmCharsRec :: String -> String -> String` that removes all characters in the first string from the second string, using your function `rmChar` from exercise 2d.

```
*Main> rmCharsRec ['a'..'l'] "football"
"oot"
```

Write an equivalent function `rmCharsFold` using `foldr`, and test the two with QuickCheck.

2 Optional Material: Matrices

Please note that optional exercices **do** contribute to the final mark. If you don't do the optional work and get the rest mostly right you will get a mark of 3/4. To get a mark of 4/4, you must get almost all of the tutorial right, including the optional questions.

2.1 Matrix manipulation

Next, we will look at matrix addition and multiplication. As matrices we will use lists of lists of `Rational`s; for example:

$$\begin{pmatrix} 1 & 4 & 9 \\ 2 & 5 & 7 \end{pmatrix} \quad \text{is represented as} \quad \begin{matrix} \texttt{[[1,4,9],} \\ \texttt{[2,5,7]]} \end{matrix}$$

The declaration below, which you can find in your `Tutorial5.hs`, makes the type `Matrix` a shorthand for the type `[[Rational]]`.

```
type Matrix = [[Rational]]
```

Our first task is to write a test to show whether a list of lists of `Rational`s is a matrix. This test should verify two things: (1) that the lists of `Rational`s are all of equal length, and (2) that there is at least one row and one column in the list of lists.

Exercise 5

(a) Write a function `uniform :: [Int] -> Bool` that tests whether the integers in a list are all equal. You can use the library function `all`, which tests whether all the elements of a list satisfy a predicate; check the type to see how it is used. If you want, you can try to define `all` in terms of `foldr` and `map`.

(b) Using your function `uniform` write a function `valid :: Matrix -> Bool` that tests whether a list of lists of `Rational`s is a matrix (it should test the properties (1) and (2) specified above).

A useful higher-order function is `zipWith`. It is a lot like the function `zip` that you have seen, which takes two lists and combines the elements in a list of pairs. The difference is that instead of combining elements as a pair, you can give `zipWith` a specific function to combine each two elements. The definition is as follows (Figure 4 gives an illustration):

```
zipWith f []      _       = []
zipWith f _       []      = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

$$x_1 \quad : \quad (x_2 \quad : \quad \cdots \quad : \quad (x_n \quad : \quad \texttt{[]} \quad)\ldots)$$

$$y_1 \quad : \quad (y_2 \quad : \quad \cdots \quad : \quad (y_n \quad : \quad \texttt{[]} \quad)\ldots)$$

$$f(x_1)(y_1) \quad : \quad (f(x_2)(y_2) \quad : \quad \cdots \quad : \quad (f(x_n)(y_n) \quad : \quad \texttt{[]} \quad)\ldots)$$
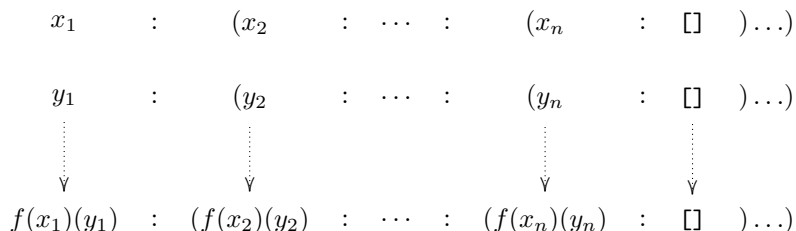
Figure 4: Illustration of `zipWith` for lists of equal length.

Another useful function for working with pairs is `uncurry`, which turns a function that takes two arguments into a function that operates on a pair.

2.1.1 Matrix Addition

Adding two matrices of equal size is done by pairwise adding the elements that are in the same position, i.e. in the same column and row, to form the new element at that position. For example:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{pmatrix} = \begin{pmatrix} 11 & 22 & 33 \\ 44 & 55 & 66 \end{pmatrix}$$

We will use `zipWith` to implement matrix addition.

Exercise 6

> Write a function `plusM` that adds two matrices. Return an error if the input is not suitable. It might be helpful to define a helper function `plusRow` that adds two rows of a matrix.

2.1.2 Matrix Multiplication

For matrix multiplication we need what is called the *dot product* or *inner product* of two vectors:

$$(a_1, a_2, \ldots, a_n) \cdot (b_1, b_2, \ldots, b_n) = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

Matrix multiplication is then defined as follows: two matrices with dimensions (n, m) and (m, p) are multiplied to form a matrix of dimension (n, p) in which the element in row i, column j is the dot product of row i in the first matrix and column j in the second. For example:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix} = \begin{pmatrix} 58 & 64 \\ 139 & 154 \end{pmatrix}$$

For more information see http://en.wikipedia.org/wiki/Matrix_multiplication.

Exercise 7

> Define a function `timesM` to perform matrix multiplication. Return an error if the input is not suitable. It might be helpful to define a helper function `dot` for the dot product of two vectors (lists). The function should then take the dot product of the single row with every column of the matrix, and return the values as a list. To make the columns of a matrix readily available you can use the library function `transpose`.

Exercise 8

> How to define `zipWith`.
>
> (a) Look up the definition of `uncurry`. What is returned by the following expression?
>
> ```
> Main> uncurry (+) (10,8)
> ```
>
> (b) Show how to define `zipWith` using `zip` and a list comprehension.
>
> (c) Show how to define `zipWith` using `zip` and the higher-order functions `map` and `uncurry`, instead of the list comprehension.

3   (Really Optional) Challenge

> Please note that challenges are entirely optional; you can receive full marks without attempting the challenge.

For a real challenge, you can try to compute the inverse of a matrix. There are a few steps involved in this process; you may find helpful to look at the type signatures of some helper functions given in the `Tutorial5.hs` file to construct your answer.

Exercise 9

    (a) You will need a function to find the *determinant* of a matrix. This will tell you if it has an inverse.

    (b) You will need a function to do the actual inversion.

    (c) Finally, you should implement some appropriate QuickCheck tests for your function.

There are several different algorithms available to compute the determinant and the inverse of a matrix. Good places to start looking are:

http://mathworld.wolfram.com/MatrixInverse.html

http://en.wikipedia.org/wiki/Invertible_matrix