Higher-order Functions

Informatics 1 – Introduction to Computation
Functional Programming Tutorial 4

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

Week 5
due 4pm Tuesday 18 October 2022
tutorials on Thursday 20 and Friday 21 October 2022

You will not receive credit for your coursework unless you attend the corresponding tutorial session. Please email kendal.reid@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 Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.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.

$$
\begin{array}{ccccccccc}
x_1 & : & x_2 & : & \cdots & : & x_n & : & \texttt{[]} \\
\downarrow & & \downarrow & & & & \downarrow & & \\
f(x_1) & : & f(x_2) & : & \cdots & : & f(x_n) & : & \texttt{[]}
\end{array}
$$

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.

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

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

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

And the function that multiplies every element in a list by a given number can be written as follows.

```
scales :: Int -> [Int] -> [Int]
scales c xs = map sc xs
  where
  sc x = c * x
```

Exercise 1

Consider a function `doubles :: [Int] -> [Int]` that doubles every item in a list.

```
doubles [3, 1, 4, 2, 3] == [6, 2, 8, 4, 6]
```

(a) Write a version `doublesComp` of the above function using *list comprehension*.

(b) Write a version `doublesRec` of the above function using *recursion*.

(c) Write a version `doublesHO` of the above function using the *higher-order* function `map`.

(d) Write a function `prop_doubles` to check that the three functions above 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

Consider a function `aboves :: Int -> [Int] -> [Int]` that takes a number and a list and returns a list of all numbers in the list that are greater than the given number.

```
aboves 2 [3, 1, 4, 2, 3] == [3, 4, 3]
```

(a) Write a version `abovesComp` of the above function using *list comprehension*.

(b) Write a version `abovesRec` of the above function using *recursion*.

(c) Write a version `abovesHO` of the above function using the *higher-order* function `filter`.

(d) Write a function `prop_aboves` to check that the three functions above are equivalent.

1.3 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. Recall that an infix operator is just another way to write a function of two elements: `x + y` is equivalent to `(+) x y`.

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

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

Now you can see that `snoc` plays the same role as `(+)` played in the example of `sum`. Again, an infix operator is just another way to write a function of two elements: ``x `snoc` y`` is equivalent to `snoc x y`.

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 captured by the `foldr` function (which is short for *fold right*; the lectures and textbook also discuss *fold left*).

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

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

Again, recall that `` x `f` y `` and `f x y` are equivalent.

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 ??. 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`.

$$\begin{array}{cccccccccc}
x_1 & : & (x_2 & : & \cdots & : & (x_n & : & \texttt{[]} &)\cdots) \\
 & \downarrow & & \downarrow & & \downarrow & & \downarrow & \downarrow & \\
x_1 & `f` & (x_2 & `f` & \cdots & `f` & (x_n & `f` & u &)\cdots)
\end{array}$$

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 that to treat an infix operator like `+` as a function name, we need to wrap it in parentheses.

$$\begin{array}{ccccccc}
10 & : & 20 & : & 30 & : & \texttt{[]} \\
 & \downarrow & & \downarrow & & \downarrow & \downarrow \\
10 & + & 20 & + & 30 & + & 0
\end{array}$$

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

Exercise 3

A list of booleans has even parity if it contains `True` an even number of times, and odd parity otherwise. The parity of a list can be computed using `xor`, defined by the following truth table.

xor	False	True
False	False	True
True	True	False

The parity can be computed by repeatedly applying `xor`.

```
parity [] == True
parity [True] == True `xor` parity [] == False
parity [True, True] == True `xor` parity [True] == True
parity [False, True, True] == False `xor` parity [True,True] == True
```

(a) Define `xor` according to the above truth table.

(b) Write a version `parityRec` of the above function using *recursion*.

(c) Write a version `parityHO` of the above function using the *higher-order* function `fold`.

(d) Write a function `prop_parity` to check that the two functions above are equivalent.

Exercise 4

Consider a function `allcaps :: String -> Bool` that takes a string and returns true if every alphabetic character in the string is upper case.

```
allcaps "" == True
allcaps "Hello!" == False
allcaps "HELLO!" == True
```

(a) Write a version `allcapsComp` of the above function using *list comprehension* and the library function `and`.

(b) Write a version `allcapsRec` of the above function using *recursion* and the library function `&&`.

(c) Write a version `allcapsHO` of the above function using the *higher-order* functions `map`, `filter`, and `foldr` and the library function `&&`.

(d) Write a function `prop_allcaps` to check that the three functions above are equivalent.

2 Optional Material: Matrices

Following the Common Marking Scheme, a student with good mastery of the material is expected to get 3/4 points. This section is for demonstrating exceptional mastery of the material. It is optional and worth 1/4 points.

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:

$$\begin{pmatrix} \frac{1}{2} & 4 & 9 \\ 2 & 5 & 7 \end{pmatrix} \quad \text{is represented as}$$

```
[[1%2, 4,  9],
 [2,   5,  7]]
```

where $m\%n$ is a `Rational` representing $\frac{m}{n}$. The declaration below, which you can find in your `Tutorial4.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 **??** gives an illustration):

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

$$\begin{array}{ccccccccccc}
x_1 & : & (x_2 & : & \cdots & : & (x_n & : & \texttt{[]} &)\ldots) \\
y_1 & : & (y_2 & : & \cdots & : & (y_n & : & \texttt{[]} &)\ldots) \\
\downarrow & & \downarrow & & & & \downarrow & & \downarrow & \\
f(x_1)(y_1) & : & (f(x_2)(y_2) & : & \cdots & : & (f(x_n)(y_n) & : & \texttt{[]} &)\ldots)
\end{array}$$

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 https://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`.

3 (Really Optional) Challenge

> Challenges are meant to be difficult. 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 `Tutorial4.hs` file to construct your answer.

Exercise 8

(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:

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

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