SET07106 - Exercise sheet 6
Advanced topics

There is nothing in this practical sheet that you cannot achieve using the techniques you have seen so far. However, because of the repeated use of certain patterns and functions, there are mechanisms built in to Haskell.

Wildcards

Wild cards are fairly straightforward. You will have noticed that sometimes the arguments you pass to a function (given on the left-hand side of a clause) never get used on the right-hand side of the clause. For example, consider the following two clauses of the `myOr` function:

```
myOr True False = True
myOr True True = True
```

or, the function `myFirst`:

```
myFirst (x,y) = x
```

Note that, in `myOr`, the second argument is irrelevant: the output depends only on the first argument. For `myFirst`, similarly, the second argument (`y`) is never used in the definition.

In such cases, we can use wildcards. These are instructions in Haskell which match *any* input. So, we could rewrite the two clauses above as:

```
myOr True _ = True
myFirst (x,_) = x
```

Why would we do this? It is really a form of documentation: we are letting whoever reads our code know that the particular arguments are never used in that clause. Of course, if you still include the redundant arguments if you wish.

As an example, then, we could write `myOr` using only two clauses:

```
myOr ::  Bool -> Bool -> Bool
myOr True _ = True
myOr False x = x
```

Exercise 1 *Write the function `myAnd`. It is an encoding of the $\wedge$ connective, but only uses two clauses instead of four.*

Exercise 2 *Write a function `mySnd`. It is an encoding of `snd` using only one clause.*

Exercise 3 *(Harder) Write a function called `ignoreButN`. This function takes an integer input and three inputs of type `a`, and returns a value of type `Maybe a`. The behaviour is as follows: if the first argument is 1, the function returns just the second argument; if the first argument is 2, the function returns just the third argument; if the first argument is 3, the function returns just the fourth argument. Otherwise, it returns nothing.*

Anonymous functions

Anonymous functions are more complicated. They are used, usually, in association with `map`. Suppose the function we want to map over a list does not exist. Then, we could write it and give it a name. However, it may be that this function would never get used again. It seems like a wasted effort to name a function that only ever gets used inside a function and never again. Moreover, any comment in our code against this function would only say "gets used by `foo`" anyway. A neater, more sophisticated approach, is to create a helper function that only `foo` (for example) can see.

Consider the problem of transforming every integer, x, in a list into $x^2 + 4x + 1$. Obviously we could write a function that (of type `Int -> Int`) that could perform this transformation, and then pass it to `map`. The function is fairly specific, however, and so would probably never be reused throughout the code. So, we examine two approaches to doing this in a more compact fashion. We will first use the `where` keyword:

```
strange ::  [Int] -> [Int]
strange xs = map calculate xs
  where calculate x = x^2 + 4*x + 1
```

This notation is fairly easy to understand. We just write the function, which we have given a name that only `strange` can see, underneath.

The second is even more compact, and uses anonymous function syntax:

```
strange2 ::  [Int] -> [Int]
strange2 xs = map (\x -> x^2 + 4*x + 1) xs
```

This notation is harder to understand, so we break it down:

```
\x -> x^2 + 4*x + 1
```

This notation means "take x and replace it with $x^2 + 4x + 1$". The notation is derived from something called $\lambda-$calculus, which is far beyond the scope of this module, but would present the above function as $\lambda x.(x^2 + 4x + 1)$.

Exercise 4 *Write a function `absPlusOne`. This takes a list of type `[Int]` and returns a list of type `[Int]`. The function takes the absolute value of the input, and adds one to it. Do it using `where`, and then again using an anonymous function. The function `abs` is inbuilt into Haskell.*

Exercise 5 *(Harder) Write a function `addPairs`. This takes a list of type `[(Int,Int)]` and returns a list of type `[Int]`. The function adds together the elements of each pair and returns them. Do it using `where`, and then again using an anonymous function.*

Using folds

Folds are one of the main tools in a functional programmers tool-kits. They come in two flavours: left and right folds. They take a binary function (i.e. one with two inputs), a starting value, and a list, and return a value. The starting value is used like an accumulator: we keep updating this accumulator using the next value in the list and the function. For example, we can add up the first ten numbers using:

```
ex2> foldl (+) 0 [1..10]
55
```

The accumulator starts as 0, and we take the first element in the list and add it to the accumulator, giving 1, which becomes the new accumulator. Next, we add the accumulator and the next element in the list (now 2), giving the new accumulator of 3, and so on. If instead we wanted to multiply all of the numbers together, we would change the function `+` to `*`:

```
ex2> foldl (*) 1 [1..10]
3628800
```

Note we changed the accumulator to 1 as well, since otherwise we would multiply by 0 and always have an accumulator of 0!

Exercise 6 *The function `max` takes two numbers and returns the bigger of the two. Using `foldl` and an appropriate starting value, determine the maximum value in the list `[x | x<-[1..10], x*x < 40]`. In other words, determine the biggest whole number whose square is less than 40.*

We use folds to determine whether universal or existential statements are correct. Suppose we have a list of type `[Bool]`, and we want to check they are all `True`. Then, we could do start with a value of `True`, and keep taking the conjunction of this value with the next element in the list. If they are all `True`, then the final value that the fold spits out will be `True`. If any one of them is `False`, then the accumulator will get "stuck" with a value of `False` until the end.

```
forAllFoldL ::  (a -> Bool) -> [a] -> Bool
forAllFoldL f xs = foldl (&&) True (map f xs)
```

Similarly, if we want to check whether at least one of a list is `True`, then we start with a value of `False` and then fold using disjunction. If any one of the values is `True`, then the accumulator will get "stuck" with a value of `True` until the end.

```
thereExistsFoldL ::  (a -> Bool) -> [a] -> Bool
thereExistsFoldL f xs = foldl (||) False (map f xs)
```

This is exceptionally neat notation, and when you can understand it is very easy to read as well.

What is the difference between left and right folds? It is down to what gets evaluated first. Roughly speaking, left folds "pass control" to the `foldl` function before the function `f`, whereas right folds do the opposite. The upshot of this difference is that right folds can operate on infinite lists, whereas left folds cannot. Why? Consider disjunction. If the first argument is `True`, then we never need to bother checking the second argument, because the disjunction as a whole will be `True`. Similarly, if the first argument to a conjunction is `False`, then we never need to bother checking the second argument, as the result will be `False`. So, when using `foldr` and `(||)`, even on an infinite list of Booleans, if it ever comes across a `True`, then the computation stops and returns `True`. Similarly, if using `foldr` and (&&), should the computation ever come across a `False`, then computation stops and returns `False`. For example, consider the following statement about whether there exists a number bigger than 20. The notation `[1..]` means the list starting at 1 and continuing forever:

```
ex2> foldr (||) False (map (>20) [1..])
True
ex2> foldl (||) False (map (>20) [1..])
```

The second line will fail (i.e. keep running for ever. Remember that `Ctrl+c` can force a process to stop), because it is trying to write the list *before* working out any of the disjunctions. The first succeeds because it works out the disjunction and *then* finds the next value from the list.

With that in mind we can use `foldr` to show that universal statements are `False` over infinite lists, and existential statements are `True`, but the converses of both will still run forever.

Exercise 7 *Determine whether every prime number is less than* 10000*. Determine whether there exists a prime number greater than* 20000*.*

Exercise 8 *(Harder) Using folds, write a function `howMany`, which takes an input of type `a` and a list of type `[a]`, and returns a value of type `Int`. `howMany x ys` should return the number of values which are equal to `x` in `ys`.*

Exercise 9 *(Even harder) Write the function `howMany2`, which is like `howMany` but uses anonymous functions and folds. It may help you to know that the notation `\x y -> x + y` is an example of an anonymous function with two inputs.*

Using sorting

Just because lists are ordered (i.e. the elements appear in a particular order), does not necessarily mean that the elements as a whole are orderable! With integers and characters, we have a "natural" order, but with other values (e.g. pairs of integers, strings) there is no obvious ordering.

Where we do have an order for a type, however, we can sort a list of values of that type. We use a function called `quickSort`:

```
quickSort ::  (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = (lessThan xs) ++ [x] ++ (greaterThan xs)
  where lessThan xs = quickSort (filter (<x) xs)
         greaterThan xs = quickSort (filter (>=x) xs)
```

This uses two functions which are visible only to `quickSort`, called `lessThan` and `greaterThan`. The former takes a list and returns a sorted list of values, all of which are less than `x`. The latter does a similar thing, returning a sorted list of values, all of which are greater than or equal to `x`. The `where` keyword is what tells us that these functions are only available to the function `quickSort`: they define some short-cuts to make our code more readable.

How does this work? First, the type signature says it will only work on orderable types. The empty list is already sorted. Finally, we take the head of a non-empty list, and work out all the values smaller than it, and all the values at least as big as it, and sort those two lists, then add them all back together in the correct order.

```
ex3> quickSort [4,3,2,1]
[1,2,3,4]
ex3> quickSort ''Duh-duh-DUH''
''--DDHUdhhuu''
```

That is all well and good, but we still have duplicate entries in our sorted list.

Exercise 10 *Write a function which both sorts a list and removes duplicate entries. Call it `quickSort2`. (Hint: whilst there are many ways to do this, the easiest is copying the definition of `quickSort`, changing all references to `quickSort` to `quickSort2`, then deleting **exactly one symbol**.)*

We can now define set equality in yet another way: we take two lists, sort them and remove duplicates, and see if the resulting lists are the same:

```
setEquality2 ::  (Ord a, Eq a) => [a] -> [a] -> Bool
setEquality2 xs ys = quickSort2 xs == quickSort2 ys
```