Haskell Tutorial: Higher Order Functions

October 11, 2019

[2]:
```
:opt no-lint
```

1 Higher Order Functions

Functions are first class citizens in Haskell. They can be passed as arguments, they can be returned as results, and they can be constructed from other functions (e.g., via function composition or currying). In the following, we will look at some of the higher order functions that are used very frequently.

1.1 Mapping with `map`

The perhaps most basic function that takes a function as an argument is `map`, which applies a function given as the first argument to all elements of a list given as the second argument.

[3]:
```
map even [1..10]
map (*2) [1..10]
map (\x -> x*x) [1..10]
```

```
[False,True,False,True,False,True,False,True,False,True]
```

```
[2,4,6,8,10,12,14,16,18,20]
```

```
[1,4,9,16,25,36,49,64,81,100]
```

We could implement our own version of `map` in the following way:

[4]:
```
map' :: (a -> b) -> [a] -> [b]
map' f [] = []
map' f (x:xs) = (f x) : (map f xs)

map' even [1..10]
map' (*2) [1..10]
map' (\x -> x*x) [1..10]
```

```
[False,True,False,True,False,True,False,True,False,True]
```

```
[2,4,6,8,10,12,14,16,18,20]
```

```
[1,4,9,16,25,36,49,64,81,100]
```

1.2 Filtering with `filter`

A filter is a function that takes a predicate and a list and returns the list of elements that satisfy the predicate. A predicate is a function returning a boolean value indicating whether an element should pass the filter or not.

[5]:
```
filter even [1..10]
filter (<6) [1..10]
```

```
[2,4,6,8,10]
```

```
[1,2,3,4,5]
```

We could implement our own version of `filter` in the following way:

[6]:
```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p [] = []
filter' p (x:xs)
    | p x       = x : filter' p xs
    | otherwise =     filter' p xs

filter' even [1..10]
filter' (<6) [1..10]
```

```
[2,4,6,8,10]
```

```
[1,2,3,4,5]
```

1.3 Zipping with `zip` and `zipWith`

Sometimes values of two lists have to be zipped together by building a list of tuples where the first tuple contains the first list elements, the second tuple the second list elements and so forth. This is what the `zip` function does. The `zipWith` function generalizes this idea by allowing you to provide a function that should be applied to the matching list elements. Note that the zipping stops when one of the list runs out of elements.

[7]:
```
zip [1..] "hello"
zipWith (\ a b -> (a, b)) [1..] "hello"
```

```
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
```

```
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
```

The `zip` and `zipWith` functions can be easily implemented.

[8]:
```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' f _ [] = []
zipWith' f [] _ = []
zipWith' f (x:xs) (y:ys) = (f x y) : zipWith' f xs ys

zip' = zipWith' (\ a b -> (a, b))

zip' [1..] "hello"
zipWith' (\ a b -> (a, b)) [1..] "hello"
```

```
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
```

```
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
```

With the `zipWith` function, we can implement a function generating the Fibonacci sequence in a rather efficient way. Apparently, Haskell's lazyness allows us to process lists while they are being constructed.

[9]:
```
fib = 0 : 1 : zipWith (+) fib (tail fib)

take 20 fib
```

```
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181]
```

1.4 Reducing with `foldl` and `foldr`

Sometimes it is necessary to reduce the elements of a list to a single value. The fold functions are often an effective means to do that. An example is the calculation of the sum of all elements in a list of numbers.

[10]:
```
 foldl (+) 0 [1..3]
```

```
6
```

The above call to `foldl` does a left associative addition of all list elements: $(((0+1)+2)+3=6$. The first parameter of `foldl` is a step function and the second argument initializes an accumulator. Subsequently, the step function is called with the accumulator and the next list element until all list elements have been processed.

[11]:
```
foldr (+) 0 [1..3]
```

```
6
```

The above call to foldr does a right associative addition of all list elements: $0 + (1 + (2 + 3)) = 6$. Since the addition is associative, both folds produce the same result. (An operation is associative if within an expression containing two or more occurrences of the same associative operator, the order in which the operations are performed does not matter as long as the sequence of the operands is not changed.)

For operations that are not associative, the difference between foldl and foldr matters. Lets look at substraction, which is not associative:

[12]:
```
foldl (-) 0 [1..3]
foldr (-) 0 [1..3]
```

```
-6
```

```
2
```

The left fold calculates $(((0 - 1) - 2) - 3 = -6$. The right fold, however, calculates $0 - (1 - (2 - 3)) = 2$.

The foldl and foldr functions require that a starting value is provided. In some situations, it is possible to use the first (or last) element of a list as a starting value. In such situations, the foldl1 and foldr1 functions can be used if the list is guaranteed to have at least one element. (If the list has only a single element, then foldl1 and foldr1 return that element.)

It is relatively easy to implement the fold functions:

[13]:
```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' f z []     = z
foldr' f z (x:xs) = f x (foldr' f z xs)

foldr' (-) 0 [1..3]
```

```
2
```

[14]:
```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f z []     = z
foldl' f z (x:xs) = foldl' f (f z x) xs

foldl' (+) 0 [1..3]
```

```
6
```

The fold functions can be used to implement many standard utility functions. Here are some examples:

[15]:
```
sum' :: Num a => [a] -> a
sum' = foldl (+) 0
sum'' = foldl1 (+)

sum' [1..5]
sum'' [1..5]
sum' []
```

```
15
```

```
15
```

```
0
```

[16]:
```
product' :: Num a => [a] -> a
product' = foldl (*) 1
product'' = foldl1 (*)

product' [1..5]
product'' [1..5]
product' []
```

```
120
```

```
120
```

```
1
```

[17]:
```
maximum' :: Ord a => [a] -> a
maximum' = foldl1 (\x acc -> if x > acc then x else acc)

maximum' [1,3,5,2,3,4]
maximum' "hello world"
```

```
5
```

```
'w'
```

1.5 Example: Sorting using quicksort

The basic idea of the quicksort algorithm is to sort a list by picking an element and and then sorting all elements smaller than the chosen element and sorting all elements larger than the chosen element and combining the results. Obviously, the lists to be sorted get smaller and smaller until we reach the trival case where the list to be sorted is an empty list. This can be translated directly into Haskell. (The typeclass `Ord` is for totally ordered datatypes. The `compare` function compares two values and returns whether the first is less than (LT), greater than (GT) or equal (EQ) to the second.)

[18]:
```
quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = quickSort less ++ [x] ++ quickSort greater
    where
        less = filter (< x) xs
        greater  = filter (>= x) xs
```

[19]:
```
quickSort [1..3]
quickSort [1,4,2,3,1,2,3]
quickSort "Hello World"
```

```
[1,2,3]
```

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

```
" HWdellloor"
```

Note that this implementation of quicksort can be improved since it does multiple passes over the to be sorted list in each iteration and it does not perform an in place sort, i.e., it creates new temporary lists. You will learn more about sorting algorithms and quicksort in the Algorithms and Data Structures module.

The quickSort function is nice but kind of limited since it always sorts the data according to the native order of the elements. We can make our sorting function more powerful by adding an extra argument that provides a comparison function.

[20]:
```
quickSort' :: (Ord a) => (a -> a -> Ordering) -> [a] -> [a]
quickSort' _ [] = []
quickSort' c (x:xs) = quickSort' c less ++ [x] ++ quickSort' c greater
    where
        less = filter (\ y -> c y x == LT) xs
        greater = filter (\ y -> c y x /= LT) xs
```

[21]:
```
quickSort' compare [1,4,2,3,9,8,7,6,5]
quickSort' (flip compare) [1,4,2,3,9,8,7,6,5]
```

```
[1,2,3,4,5,6,7,8,9]
```

```
[9,8,7,6,5,4,3,2,1]
```

The flip function flips the arguments of a function, which causes the comparison to be in the diferrent direction.

Note that we can get back a quicksort function using the native sort order out of the more generic quicksort function via currying.

[22]:
```
quickSortNative = quickSort' compare

quickSortNative [1,4,2,3,9,8,7,6,5]
```

```
[1,2,3,4,5,6,7,8,9]
```