Programming In Haskell Chapter 10

CS 1JC3

Generalizing Patterns of Computation

- We've already seen one aspect of generality through polymorphism, i.e consider the functions

```
length :: [a] -> Int
(++)   :: [a] -> [a] -> [a]
take   :: Int -> [a] -> [a]
```

- All of the above operate on all lists regardless of their type

Generalizing Patterns of Computation

Another way of generalizing is by creating functions that encapsulate different often performed patterns of computation.

As an example, two very common patterns of computation are

- Transform every element in a list some way (known as mapping)
- Combine the elements of a list using some operator (known as folding)

The Map Function

The Prelude function map applies a function to every element of a list

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

For Example:

```
map (1+) [1,3,5,7]
   = [2,4,6,8]
```

The Map Function

- The map function can be defined in a particularly simple manner using a list comprehension

The Map Function

- The map function can be defined in a particularly simple manner using a list comprehension

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

- Alternatively, for the purpose of proofs, the map function can also be defined using recursion

The Map Function

- The map function can be defined in a particularly simple manner using a list comprehension

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

- Alternatively, for the purpose of proofs, the map function can also be defined using recursion

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

The Filter Function

The Prelude function filter selects every element from a list that satisfies a given boolean function

```
filter :: (a -> Bool) -> [a] -> [a]
```

For Example:

```
filter even [1..10]
  = [2,4,6,8,10]
```

The Filter Function

- filter can be defined using a list comprehension

The Filter Function

- filter can be defined using a list comprehension

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

- Alternatively, it can be defined using recursion

The Filter Function

- filter can be defined using a list comprehension

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

- Alternatively, it can be defined using recursion

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

The Zip Function

The Prelude function zip is a useful tool for working on lists. It takes two lists and returns a list of corresponding pairs

```
zip    :: [a] -> [b] -> [(a,b)]
```

For example:

```
zip ['a','b','c'] [1,2,3,4]
        [('a',1),('b',2),('c',3)]
```

The Zip Function

- Using zip we can define a function that returns a list of all pairs of adjacent elements from a list:

```
pairs    :: [a] -> [(a,a)]
pairs xs  = zip xs (tail xs)
```

- For example:

```
pairs [1,2,3,4]
   [(1,2),(2,3),(3,4)]
```

Using pairs we can define a function that decides if the elements in a list are sorted

```
pairs       :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

Sorted

Using pairs we can define a function that decides if the elements in a list are sorted

```
pairs       :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

```
sorted      :: (Ord a) => [a] -> Bool
sorted xs = and [ x <= y | (x,y) <- pairs xs]
```

Positions

Using zip we can also define a function that returns the list of all positions of a value in a list

```
positions    :: (Eq a) => a -> [a] -> [Int]
positions x xs = [i | (x2,i) <- zip xs [0..n], x == x2]
           where
              n = length xs - 1
```

For example:

```
positions 0 [1,0,0,1,0,1,1,0]
     [1,2,4,7]
```

The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion

```
f []     = v
f (x:xs) = x (X) f xs
```

Note: (X) represents some infix function and v is usually the identity of that function (The final value the function takes after it has been applied to every element in the list)

The Foldr Function

For Example:

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

product []     = 1                   -- v   = 1
product (x:xs) = x * product xs      -- (X) = *

and []       = True                  -- v   = True
and (x:xs)   = x && and xs           -- (X) = &&
```

The Foldr Function

The high order library function foldr encapsulates this simple pattern of recursion, with the function (X) and value v as arguments.

For Example:

```
sum     = foldr (+) 0

product = foldr (*) 1

or      = foldr (||) False

and     = foldr (&&) True
```

The Foldr Function

foldr itself can be defined using recursion

The Foldr Function

foldr itself can be defined using recursion

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

Evaluating Foldr

It helps to think of foldr as being evaluateed in the following manner

```
  sum [1,2,3]
=
  foldr (+) 0 [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  1+(2+(3+0)) -- Replace each (:) with (+) and [] with 0
=
  6
```

Other Foldr Examples

Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.
Recall that the length function can be defined in the following manner

```
length        :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

We can also define length using foldr and lambda expressions

Other Foldr Examples

Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.
Recall that the length function can be defined in the following manner

```
length          :: [a] -> Int
length []       = 0
length (x:xs)   = 1 + length xs
```

We can also define length using foldr and lambda expressions

```
length = foldr (\_ n -> 1+n) 0
```

Other Foldr Examples

- Now recall the reverse function that can be defined as

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

- Using foldr we can define reverse as

Other Foldr Examples

- Now recall the reverse function that can be defined as

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

- Using foldr we can define reverse as

```
reverse = foldr (\x xs -> xs ++ [x]) []
```

- Consider the following data type for Binary Trees

```
data BinTree a = Node (BinTree a) (BinTree a) a
               | Leaf a
        deriving (Show,Eq)
```

Generalizing Patterns of Computation: Beyond Lists

- Consider the following data type for Binary Trees

```
data BinTree a = Node (BinTree a) (BinTree a) a
               | Leaf a
         deriving (Show,Eq)
```

- Question: Can we apply the same patterns of computation on Lists to Trees?

Generalizing Patterns of Computation: Beyond Lists

- We'll start off with defining mapping over BinTree, which will have type

  ```
  treeMap :: (a -> b) -> BinTree a -> BinTree b
  ```

- The implementation should not change the structure of BinTree, and should transform every element

Generalizing Patterns of Computation: Beyond Lists

- We'll start off with defining mapping over BinTree, which will have type

  ```
  treeMap :: (a -> b) -> BinTree a -> BinTree b
  ```

- The implementation should not change the structure of BinTree, and should transform every element

  ```
  treeMap f (Node t1 t2 x) =
          Node (treeMap f t1) (treeMap f t2) (f x)
  treeMap f (Leaf x) = Leaf (f x)
  ```

Generalizing Patterns of Computation: Beyond Lists

- Next up, we define folding over BinTree, which will have type

```
treeFold :: (a -> b -> b) -> b -> BinTree a -> b
```

- The implementation should still use a binary op as the type insists, although it is tricky to do so

Generalizing Patterns of Computation: Beyond Lists

- Next up, we define folding over BinTree, which will have type

```
treeFold :: (a -> b -> b) -> b -> BinTree a -> b
```

- The implementation should still use a binary op as the type insists, although it is tricky to do so

```
treeFold op u (Node t1 t2 x) = let
    u'  = x `op` u''
    u'' = treeFold op u t1
  in treeFold op u' t2

treeFold op u (Leaf x) = x `op` u
```

Exercise 1

Express the comprehension

```
[f x | x <- xs, p x]
```

using the functions map and filter

Exercise 1

Express the comprehension

```
[f x | x <- xs, p x]
```

using the functions map and filter

Solution

```
map f (filter p xs)
```

For example:

```
[1+x | x <- [1..8], even x]    = [3,5,7,9]

map (1+) (filter even [1..8]) = [3,5,7,9]
```

Exercise 2

Redefine map f and filter p using foldr

Exercise 2

Redefine map f and filter p using foldr

Solution

```
map    :: (a -> b) -> [a] -> [b]
map f = foldr (\x ys -> (f x) : ys ) []

filter    :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\x xs -> if p x then x:xs else xs) []
```

Exercise 3

foldr takes a function and folds to the right. For example:

```
foldr (/) 1 [18,27,3]
      = 18 / (27 / (3 / 1))
```

Define a function foldl that folds to the left. For example

```
foldl (/) 18 [1,2,9]
       = (((18 / 9) / 2) / 1)
```

Solution 3

```haskell
foldl  :: (a -> b -> a) -> a -> [b] -> a
foldl f v []     = v
foldl f v (x:xs) = f (foldl f v xs) x
```

Exercise 4

Define a new fold called foldt that works in the following manner:

```
foldt (+) 0 [1..5]
  =  (1+2) + ((3+4) + (5+0))

foldt (/) 1 [1..6]
  = (1 / 2) / ((3 / 4) / ((5 / 6) / 1))
```

Note: foldt associates to the right like foldr

Solution 4

```
foldt   :: (a -> a -> a) -> a -> [a] -> a
foldt f v []           = v
foldt f v [x]          = f x v
foldt f v (x1:x2:xs) = f (f x1 x2) (foldt f v xs)
```

Exercise 5

Recall the function merge we used in the creation of the merge sort.

```
merge (Ord a) => [a] -> [a] -> [a]
merge  xs [] = xs
merge  [] ys = ys
merge (x:xs) (y:ys)
        | x <= y     = x : merge xs (y:ys)
        | otherwise = y : merge (x:xs) ys
```

Use the foldt function and merge to define mergesort (Remember, mergesort breaks up a list into individual elements and recursively merges them together)

Solution 5

```
mergesort    :: (Ord a) => [a] -> [a] -> [a]
merge  xs [] = xs
merge  [] ys = ys
merge (x:xs) (y:ys)
        | x <= y     = x : merge xs (y:ys)
        | otherwise = y : merge (x:xs) ys

mergesort    :: (Ord a) => [a] -> [a]
mergesort xs = foldt merge [] [[x] | x <- xs]
```

Exercise 6

Define a folding function for the following Tree type

```
data Tree a = TNode [Tree a] a
  deriving (Show,Eq)
```

Hint: use the list foldr to recursively solve and combine the list of Trees

Exercise 6

Define a folding function for the following Tree type

```
data Tree a = TNode [Tree a] a
  deriving (Show,Eq)
```

Hint: use the list foldr to recursively solve and combine the list of Trees

Solution

```
treeFold2 :: (a -> b -> b) -> b -> Tree a -> b
treeFold2 op u (TNode ts x) = let
    u' = foldr (\t u'' -> treeFold2 op u'' t) u ts
  in x `op` u'
```