Haskell Answers 6: *foldr* and *foldl*

Antoni Diller

4 August 2011

(1) Using the higher-order function *foldr* define a function *sumsq* which takes an integer n as its argument and returns the sum of the squares of the first n integers. That is to say,

$$sumsq\ n = 1^2 + 2^2 + 3^2 + \ldots + n^2.$$

Do not use the function *map*.

```
sumsq1 :: Integral a => a -> a
sumsq1 n = foldr op 0 [1..n]
           where op :: Num a => a -> a -> a
                 op x y = x*x + y
```

(2) Define *length*, which returns the number of elements in a list, using *foldr*. Redefine it using *foldl*.

```
lengthr, lengthl :: [Int] -> Int
lengthr = foldr (\x y -> 1 + y) 0
lengthl = foldl (\x y -> x + 1) 0
```

(3) Define *minlist*, which returns the smallest integer in a non-empty list of integers, using *foldr1*. Redefine it using *foldl1*.

```
minlistr, minlistl :: [Int] -> Int
minlistr = foldr1 min
minlistl = foldl1 min
```

(4) Define *reverse*, which reverses a list, using *foldr*.

(5) Using *foldr*, define a function *remove* which takes two strings as its arguments and removes every letter from the second list that occurs in the first list. For example, `remove "first" "second" = "econd"`.

(6) Define *filter* using *foldr*. Define *filter* again using *foldl*.

```
onefilter :: (a -> Bool) -> a -> [a] -> [a]
onefilter pred x xs
  | pred x    = [x] ++ xs
  | otherwise = xs

filterr :: (a -> Bool) -> [a] -> [a]
filterr pred ys = foldr (onefilter pred) [] ys
```

(7) The function *remdups* removes adjacent duplicates from a list. For example,

$$\textit{remdups}\ [1,2,2,3,3,3,1,1] = [1,2,3,1].$$

Define *remdups* using *foldr*. Give another definition using *foldl*.

```
remdups :: Eq a => [a] -> [a]
remdups [] = []
remdups [x] = [x]
remdups (x1:x2:xs)
  | x1 == x2  = remdups (x2:xs)
  | otherwise = x1 : remdups (x2:xs)

joinr :: Eq a => a -> [a] -> [a]
joinr x [] = [x]
joinr x xs
  | x == head xs = xs
  | otherwise    = [x] ++ xs

remdupsr :: Eq a => [a] -> [a]
remdupsr [] = []
remdupsr (y:ys) = foldr joinr [y] ys

joinl :: Eq a => [a] -> a -> [a]
joinl [] x = [x]
joinl xs x
  | last xs == x = xs
  | otherwise    = xs ++ [x]

remdupsl :: Eq a => [a] -> [a]
remdupsl ys = foldl joinl [] ys
```

(8) The function *inits* returns the list of all initial segments of a list. Thus, `inits`

"ate" = [[], "a", "at", "ate"]. Define *inits* using *foldr*.

```
inits = foldr f [[]]
        where f x xss = [] : map (x:) xss
```

(9) Using *foldl* define *approxe n* such that

$$approxe\ n = \sum_{i=0}^{i=n} \frac{1}{i!}.$$

For example,

$$\begin{aligned} approxe\ 4 &= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!}, \\ &= 1 + 1 + 0.5 + 0.\dot{3} + 0.25, \\ &= 3.08\dot{3}, \end{aligned}$$

where $0.\dot{3}$ means 'point 3 recurring'.

```
approxe n = foldl op 1 [1..n]
            where op x y = x + (1/fact y)
                  fact i = product [1..i]
```

(10) Using *scanl* define a function *sae* (successive approximations to e) such that

$$sae\ n = \left[\sum_{i=0}^{i=1} \frac{1}{i!}, \sum_{i=0}^{i=2} \frac{1}{i!}, \sum_{i=0}^{i=3} \frac{1}{i!}, \ldots, \sum_{i=0}^{i=n} \frac{1}{i!}\right].$$

```
sae n = scanl op 1 [1..n-1]
        where op x y = x + (1/fact y)
              fact i = product [1..i]
```

(11) Define *iterate* using *scanl*.

```
iterate1 f x = scanl app x fs
               where app x f = f x
                     fs      = f:fs
```

(12) Define *shift*, which sticks the first element of a list at the end. Thus, `shift [1, 2, 3] = [2, 3, 1]` and `shift "eat" = "ate"`. Unsing *foldl* and *shift* define *rotate*, which produces all the rotations of a list. Thus, `rotate [1, 2, 3] = [[1, 2, 3], [2, 3, 1], [3, 1, 2]]`.

```
shift []     = []
shift (x:xs) = xs ++ [x]
its f = f : map (f.) (its f)
tate xs = foldl op xs (take ((length xs) - 1) (its shift))
          where op ys f = ys ++ (f (take (length xs) ys))
sips i [] = []
sips i xs = [take i xs] ++ sips i (drop i xs)
rotate xs = sips (length xs) (tate xs)
```

(13) The function *add* can be defined in terms of

```
succ i = i + 1
pred i = i - 1
```

by the equations

```
add i 0 = i
add i j = succ (add i (pred j))
```

(a) Give a similar definition of *mult* which uses only *add* and *pred*. Give a definition of *exp* which uses only *mult* and *pred*. What is the next function in this sequence?

(b) The *fold* function on integers *foldi* can be defined as follows:

```
foldi :: (a -> a) -> a -> Int -> a
foldi f q 0 = q
foldi f q i = f (foldi f q (pred i))
```

Define the functions *add*, *mult* and *exp* in terms of *foldi*.

(c) Define the functions *fact* (factorial) and *fib* (Fibonacci numbers) using the function *foldi*.