Haskell Unit 6: `fold` functions

Antoni Diller

26 July 2011

Introduction

More information can be found in sections 4.5 and 4.6 of Bird, *Introduction to Functional Programming using Haskell* (1998).

The higher-order function `foldr`

The higher-order function `foldr` can be defined like this:

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

Intuitively, what `foldr` does can be shown like this, where `#` is a binary infix operator:

```
foldr (#) u [x1, x2, ..., xn] = x1 # (x2 # (...(xn # u)...))
```

The function `foldr` has many uses. Some of these are as follows:

```
and, or :: [Bool] -> Bool
and = foldr (&&) True
or  = foldr (||) False

sum, product :: [Int} -> Int
sum     = foldr (+) 0
product = foldr (*) 1

concat :: [[a]] -> [a]
concat = foldr (++) []
```

The higher-order functions `map` and `filter` can be defined using `foldr`

```
map f = foldr ( (:) . f ) []

filter pred = foldr ( (++) . sel ) []
              where sel x | pred x    = [x]
                          | otherwise = []
```

`fold-map` fusion

`fold-map` fusion is best illustrated by means of an example. Problem: Define a function `evenList` which, when applied to a list of integers, returns `True` if they are all even and `False` otherwise. Solution:

```
evenList = and . map even
         = foldr (&&) True . map even
         = foldr ( (&&) . even ) True
```

by `fold-map` fusion. In general,

```
foldr op u . map f = foldr ( op . f ) u
```

The higher-order function `foldl`

The higher-order function `foldl` can be defined like this:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl op u []     = u
foldl op u (x:xs) = foldl op (op u x) xs
```

Intuitively, what `foldl` does can be shown like this, where `#` is a binary infix operator:

```
foldl (#) u [x1, x2, ..., xn] = (...((u # x1) # x2) # ...) # xn
```

The function `foldl` has many uses. Some of these are as follows:

```
and, or :: [Bool] -> Bool
and = foldl (&&) True
or  = foldl (||) False

sum, product :: [Int] -> Int
sum     = foldl (+) 0
product = foldl (*) 1

concat :: [[a]] -> [a]
concat = foldl (++) []
```

The first duality theorem

```
foldl (#) u xs = foldr (#) u xs
```

if `#` is associative and `u` is a unit for `#`.

The function `reverse`

There are some applications where `foldl` is preferable to `foldr`. One of these is in defining `reverse`:

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
```

where `flip` is a predfined Haskell function

```
flip op x y = op y x
```

This is better than the obvious definition:

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

Folding non-empty lists

The function `foldr1` can be defined like this:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 op [x] = x
foldr1 op (x:xs) = op x (foldr1 op xs)
```

Intuitively, what `foldr1` does can be shown like this, where `#` is a binary infix operator:

```
foldr1 (#) [x1, x2, ..., xn] = x1 # (x2 # (...(x(n-1) # xn)...))
```

Using `foldr1` it is easy to define a function that finds the maximum element of a list:

```
maxlist = foldr1 max
```

The function `foldl1` can be defined in terms of `foldl` like this:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 op (x:xs) = foldl op x xs
```

Intuitively, what `foldl1` does can be shown like this, where `#` is a binary infix operator:

```
  foldl1 (#) [x1, x2, ..., xn]
= (...((x1 # x2) # x3)... # x(n-1)) # xn>
```

The higher-order scan functions

The function scanr applies foldr to every tail segment of a list. For example,

```
scanr (#) e [x, y, z] = [x # (y # (z # e)), y # (z # e), z # e, e]
```

The function scanl appllies foldl to every initial segment of a list. For example,

```
scanl (#) e [x, y, z] = [e, e # x, (e # x) # y, ((e # x) # y) # z]
```

The infinite list of factorials can be defined as follows:

```
   scanl (*) 1 [2 .. ]
= [1, 1 * 2, (1 * 2) * 3, ((1 * 2) * 3) * 4, ...]

  factorial i
= (scanl (*) 1 [ 2 .. ] ) !! (i - 1)
```