Folds in Haskell

Principles of Programming Languages

Folds in Haskell

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

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

f maps the empty list to some value v, and any non-empty list to some function ⊕ applied to its head and f of its tail.

Recursion to Folds

- When we deal with recursion, we noticed a theme throughout many of the recursive functions that operated on lists:
  - we had an edge case for the empty list
  - we introduced the x: xs pattern and then we did some activities that involved a single element and the rest of the list
- It turns out this is a very common pattern, so a couple of very useful functions were introduced to encapsulate it. These functions are called folds.
- That is, a fold takes:
  - a binary function
  - a starting value, a.k.a. the accumulator
  - a list to fold up

Folding

- Folding is a general name for a family of related recursive patterns.
- The essential idea of folding is to take a list and reduce it to, for instance, a single number.
- For example, to sum the list [1,2,3,4], we can evaluate it as 1 + (2 + (3 + 4)). Folding generalizes this pattern to work with lists of any type and any folding function that makes sense.

Working of folds

1. The binary function is called with the accumulator and the first element of the list (or the last element, depending on whether we fold from the left or from the right), and produces a new accumulator
2. Then, the binary function is called again with the new accumulator and the now new first (or last) element, and so on
3. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to

Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that.

Understanding through example

- Consider the sum and product functions

```
sum' :: [Int] -> Int              -
sum' []     = 0
sum' (x:xs) = x + sum' xs

product' :: [Integer] -> Integer -
product' []     = 1
product' (x:xs) = x * product' xs
```

  - Both sum' and product' apply arithmetic operations to integers.
- Similarly, consider a function concat that concatenates a list of lists of some type into a list of that type with the order of the input lists and their elements preserved.

```
concat' :: [[a]] -> [a]   -- concat in Prelude
concat' []       =  []
concat' (xs:xss) =  xs ++ concat' xss
```

Understanding through example

- Observe the working:-

```
sum' [1,2,3]           = (1 + (2 + (3 + 0)))
product' [1,2,3]       = (1 * (2 * (3 * 1)))
concat' ["1","2","3"] = ("1" ++ ("2" ++ ("3" ++ "")))
```

  - All take a list.
  - All insert a binary operator between all the consecutive elements of the list in order to reduce the list to a single value.
  - All group the operations from the right to the left.
  - Each function returns some value for an empty list.
  - All return a value of the same element type as the input list.

Understanding through example

- We can abstract the pattern of computation common to sum', product', and concat’ as the function foldr (pronounced "fold right") found in the Prelude. (Here we use foldrX to avoid the name conflict.)

```
foldrX :: (a -> b -> b) -> b -> [a] -> b   -- foldr in Prelude
foldrX f z []     = z
foldrX f z (x:xs) = f x (foldrX f z xs)
```

- Function foldr:
  - uses two type parameters a and b -- one for the type of elements in the list and one for the type of the result
  - passes in the general binary operation f (with type a -> b -> b) that combines (i.e., folds) the list elements
  - passes in the "seed" element z (of type b) to be returned for empty lists

foldr

- The foldr function "folds" the list elements (of type a) into a value (of type b) by "inserting" operation f between the elements, with value z "appended" as the rightmost element.
- Often the seed value z is the right identity element for the operation, but foldr may be useful in some circumstances where it is not (or perhaps even if there is no right identity).
- For example, foldr f z [1,2,3] expands to f 1 (f 2 (f 3 z)), or, using an infix style: `` 1 `f` (2 `f` (3 `f` z)) ``
- Function foldr does not depend upon f being associative or having either a right or left identity.

foldr

- In Haskell, foldr is called a fold operation. Other languages sometimes call this a reduce or insert operation.
- We can specialize foldr to restate the definitions for sum', product', and concat’.

```
sum2 :: [Int] -> Int
sum2 xs = foldrX (+) 0 xs

product2 :: [Int] -> Int
product2 xs = foldrX (*) 1 xs

concat2:: [[a]] -> [a]
concat2 xss = foldrX (++) [] xss
```

Example1

```
sum [1,2,3]
=
foldr (+) 0 [1,2,3]
=
foldr (+) 0 (1:(2:(3:[])))
=
1+(2+(3+0))
=
6
```

Replace each (:)
by (+) and [] by 0.

Example2

```
  product [1,2,3]
=
  foldr (*) 1 [1,2,3]
=
  foldr (*) 1 (1:(2:(3:[])))
=
  1*(2*(3*1))
=
  6
```

Replace each (:) by (*) and [] by 1.

Other Folder 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 the length function:

```
length          :: [a] → Int
length []        = 0
length (_:xs) = 1 + length xs
```

```
length2 :: [a] -> Int   -- length
length2 xs   = foldr len  0  xs
    where len _ acc = acc + 1
```

Understanding foldl through foldr

- We designed function foldr as a backward linear recursive function with the signature:

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

- Example:

```
foldr f z [1,2,3] ==  f 1 (f 2 (f 3 z))
                  ==  1 `f` (2 `f` (3 `f` z))
```

- Consider a function foldl (pronounced "fold left") such that:

```
foldl f z [1,2,3]  ==  f (f (f z 1) 2) 3
                   ==  ((z `f` 1) `f` 2) `f` 3`
```

Understanding foldl through foldr

- foldl function folds from the left. It offers us the opportunity to use parameter z as an accumulating parameter in a tail recursive implementation.
- This is shown below as foldlX, which is similar to foldl in the Prelude.

```
foldlX :: (a -> b -> a) -> a -> [b] -> a
foldlX f z []     = z
foldlX f z (x:xs) = foldlX f (f z x) xs
```

- In the recursive call of foldlX the "seed value" argument is used as an accumulating parameter.

Left fold

- foldl function, also called the left fold.
  - it folds the list up from the left side
  - the binary function is applied to the starting accumulator and the head of the list
  - that produces a new accumulator value and the binary function is called with that value and the next element of the list etc.

- Let's implement sum using a fold instead of explicit recursion.

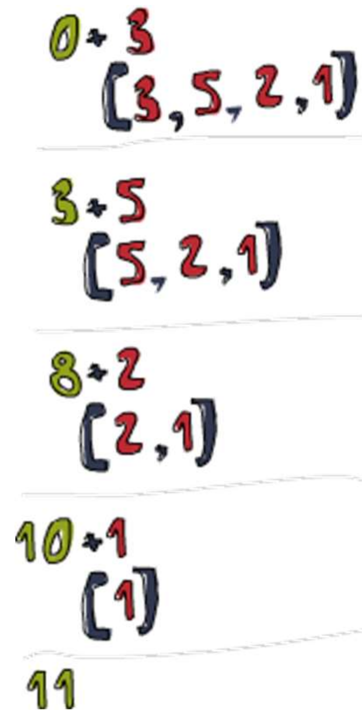
```haskell
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

Left fold

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

- Understanding with example:

```
ghci> sum' [3,5,2,1]
11
```



1. \acc x -> acc + x is the binary function.
2. **0** is the starting value and xs is the list to be folded up.
3. Now first, 0 is used as the acc parameter to the binary function and 3 is used as the x (or the current element) parameter. 0 + 3 produces a 3 and it becomes the new accumulator value.
4. Next up, 3 is used as the accumulator value and 5 as the current element and 8 becomes the new accumulator value.
5. Moving forward, 8 is the accumulator value, 2 is the current element, the new accumulator value is 10.
6. Finally, that 10 is used as the accumulator value and 1 as the current element, producing an 11.

Left fold

- If we consider that functions are curried, then the curried version of the same can be written as

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

The lambda function (\acc x -> acc + x) is the same as (+).
We can omit the xs as the parameter because calling foldl (+) 0 will return a function that takes a list.

- Generally, if you have a function like foo a = bar b a, you can rewrite it as foo = bar b, because of currying.

Why Foldr

- Some recursive functions on lists, such as sum, are simpler to define using foldr.
- Properties of functions defined using foldr can be proved using algebraic properties of foldr.
- Advanced program optimizations can be simpler if foldr is used in place of explicit recursion.

foldl1 and foldr1

- The foldl1 and foldr1 functions work much like foldl and foldr, only you don't need to provide them with an explicit starting value.
- They assume the first (or last) element of the list to be the starting value and then start the fold with the element next to it.
- The sum function can be implemented like **sum = foldl1 (+)**. Because they depend on the lists, they fold up having at least one element, they cause runtime errors if called with empty lists.
- foldl and foldr, on the other hand, work fine with empty lists.

Examples on folds

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: (Num a) => [a] -> a
product' = foldr1 (*)

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

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

Next – Lamda in Haskell