List Map/Fold in Haskell

We have seen the following definitions of list map and fold:

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

fold z op [] = z
fold z op (x:xs) = x `op` fold z op xs
```

In Haskell, `map` in the Prelude is defined as above, but we have a number of variants of fold.

Folding left-and-right

Folding can be done in two ways.
The first is simply our `fold`, but with arguments reversed:

```
foldr op z [] = []
foldr op z (x:xs) = x `op` foldr op z xs
```

The second applies the operator (`f`) to the nil-replacement (`e`) and list-head (`x`) elements, and is *tail-recursive*.

```
foldl f e []      =  e
foldl f e (x:xs)  =  foldl f (f e x) xs
```

Why `foldl` and `foldr`?
See what they do in an example:

```
foldr (+) 0 [1,2,3,4,5]
 =  1 + (2 + (3 + (4 + (5 + 0))))
foldl (+) 0 [1,2,3,4,5]
 =  ((((0 + 1) + 2) + 3) + 4) + 5
```

`foldl` is not a "fold"!

It has a different type to `foldr`, which is a "proper" fold.

```
foldr :: (e -> t -> t) -> t -> [e] -> t
foldl :: (t -> e -> t) -> t -> [e] -> t
```

`foldl` is a "twisted" fold.
It exists because tail-recursion means it should be efficient, and also it makes some things easy to define as folds:

```
reverse = foldl (flip (:)) []

flip :: (a -> b -> c) -> b -> a -> c
flip op x y = y `op` x
```

Alternative definition:

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

List Catenation `(++)`

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

- Time to do `xs++ys` is:
  - proportional to length of `xs`.
    It walks along `xs` to the end and then links to `ys`
  - independent of the length of `ys`.
    List `ys` is not inspected, just linked in at the end.
- Syntactically, it is right associative:

  `xs ++ ys ++ zs  =  xs ++ (ys ++ zs)`

Concatenation of many lists: `concat`

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

- Why is `foldr` used here, rather than `foldl` ?
- Consider `concat [as,bs,cs,..,zs]` where ℓ is the length of the list of lists.
- With `foldl`:
  `(..((([]++as)++bs)++cs)..)++zs`
  Execution time is $O(\ell^2)$.
- With `foldr`:
  `as++(bs++(cs++(..(zs++[])..)))`
  Execution time is $O(\ell)$.

Strictness Analysis

- If a program requires everything to be evaluated, we have seen that strict evaluation is faster (no thunk overhead).
- In general, if an application `f a` always evaluates `a`, then it is more efficient to use strict-evaluation to reduce it.
- Compilers for lazy languages often perform *strictness analysis* to detect cases were such a optimisation is possible.
- However complete strictness analysis is undecidable.
- So Haskell allows programmers to annotate arguments as strict.
  `f $! a` applies `f` to `a`, but after forcing the evaluation of `a`.
- Here `$!` is an right-associative infix operator:
  `($!)  ::  ( a -> b) -> a -> b`

Strictness Example

From the Prelude:

- `foldr` (Lazy)

```
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- `foldl` (Lazy)

```
foldl _ z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- `foldl'` (Strict)

```
foldl' _ z [] = z
foldl' f z (x:xs) = foldl' f $! (f z x) xs
```

- Lets try them with arguments `(+) 0 [1,2,3]`

`foldr (+) 0 [1,2,3]`

```
foldr (+) 0 [1,2,3]
= (+) 1 (foldr (+) 0 [2,3])
= (+) 1 ((+) 2 (foldr (+) 0 [3]))
= (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
= (+) 1 ((+) 2 ((+) 3 0))
= (+) 1 ((+) 2 3)
= (+) 1 5
= 6
```

- Stack usage: $O(\mathit{length})$
- Heap usage: $O(\mathit{length})$

foldl (+) 0 [1,2,3]

```
foldl (+) 0 [1,2,3]
= foldl (+) ((+) 0 1) [2,3]
= foldl (+) ((+) ((+) 0 1) 2) [3]
= foldl (+) ((+) ((+) ((+) 0 1) 2) 3) []
= ((+) ((+) ((+) 0 1) 2) 3)
= ((+) ((+) 1 2) 3)
= ((+) 3 3)
= 6
```

- Stack usage: $O(1)$
- Heap usage: $O(\mathit{length})$

foldl' (+) 0 [1,2,3]

```
foldl' (+) 0 [1,2,3]
= foldl' (+) ((+) 0 1) [2,3]
= foldl' (+) 1 [2,3]
= foldl' (+) ((+) 1 2) [3]
= foldl' (+) 3 [3]
= foldl' (+) ((+) 3 3) []
= foldl' (+) 6 []
= 6
```

- Stack usage: $O(1)$
- Heap usage: $O(1)$

Stack Usage by Fold Variants

Stack is first-in-last-out (FILO) storage used to support function procedure calls

- `foldr` creates a new stack frame for every recursive call (as expected)
- `foldl` is tail-recursive—the recursive call is the last thing done in the function body. Any decent compiler can optimise this to a while-loop with no new stack frames being created.
- `foldl'` see `foldl`

Heap Usage by Fold Variants

Heap is memory that is allocated and freed in blocks, accessed by pointers.

- `foldr` accumulates a growing unevaluated addition on each recursive call, because it is lazy—this lives on the heap.
- `foldl` see `foldr`
- `foldl'` strictly evaluates the additions as it goes along, so there is no growing result expression, so the heap requirements stay constant.

`$!` has a lazy cousin (`$`)

- `$` (a.k.a. "apply") is the lazy (normal) version of `$!`
  - It is a right-associative infix operator

```
f $ a = f a  -- definition
f $ g $ h $ x  = f ( g ( h x ) )
```

  What is it good for ?
- Consider a complex nested function application:
  `f a ( g b ( h ( x k c d) ) )`
- We could use function composition:
  `(f a . g b . h . x k c) d`
  where `(f . g) x = f ( g x )`
- The `$` notation allows us to drop all brackets:
  `f a $ g b $ h $ x k c d`

Records in Haskell

A record is really just a standard data type with one constructor:

```
data Pair a b = Pair a b
```

To use a type like this you might provide some "accessor" functions:

```
first :: (Pair a b) -> a
first (Pair a _) = a

second :: (Pair a b) -> b
second (Pair _ b) = b
```

Records in Haskell

Defining the data type as a record type is just a syntactic convenience for creating those accessors, or "fields":

```
newtype Pair a b = Pair {
                          first :: a,
                          second :: b
                        }
```

Note that two different record declarations must have disjoint accessor names

We can either create values using the usual constructor, or using the named fields:

```
f = Pair 1 'a'

g = Pair {  second = 'b', first = 2 }
```

(notice the order doesn't matter when we use the fields).

Records in Haskell

We can look them up using the field names:

```
h :: (Pair Int b) -> Int
h p = (first p) + 1
```

There's also a convenient syntax for creating a new value based on an old one:

```
h = g { second = 'B' }


> h
Pair 2 'B'
```

Consider the following plan to treat Ints differently, by having a show instance that produces roman numerals, plus some code to do additions.

```
newtype Roman = Roman Int
```

We will want a way to get at the underlying Int inside a Roman:

```
unRoman (Roman i) = i
```

We can then define addition:

```
rAdd r1 r2 = Roman (unRoman r1 + unRoman r2)
```

We can now define our Show instance:

```
romanShow i
 | i < 4  =  replicate i 'I'
 | i == 4  =  "IV"
 | i < 9  = 'V' : replicate (i-5) 'I'
 | i == 9 = "IX"
 | i < 14  = 'X' : replicate (i-10) 'I'
 | otherwise = "my head hurts!"
```

newtypes using Records

From the Roman example we had:

```
newtype Roman = Roman Int
unRoman (Roman i) = i
```

Using the record idea, we can do this all in the newtype declaration (lets use R2/unR2 to avoid clashing with Roman/unRoman)

```
newtype R2 = R2 { unR2 :: Int }
```

So we can code show and addition as

```
instance Show R2 where show = romanShow . unR2
r2Add r1 r2 = R2 (unR2 r1 + unR2 r2)
```