Recollecting Haskell, Part V

Higher Types

CIS 352/Spring 2018

Programming Languages

January 29, 2019

Watch out for the arrows

A start on higher types: Mapping, 1

Mapping via list comprehension

```
doubleAll :: [Int] -> [Int]
doubleAll lst = [ 2*x | x <- lst ]

addPairs :: [(Int,Int)] -> [[Int]]
addPairs mns = [[m+n] | (m,n) <- mns ]

multAll :: Int -> [Int] -> [Int]
multAll x ys = [ x*y | y <- ys ]
```

More generally for any function `f :: a -> b`, we can define a function

```
apply_f :: [a] -> [b]
apply_f xs = [f x | x <- xs]
```

A start on higher types: Mapping, 2

Mapping via structural recursion over lists

```
doubleAll' :: [Int] -> [Int]
doubleAll' [] = []
doubleAll' (x:xs) = (2*x):doubleAll xs

addPairs' :: [(Int,Int)] -> [[Int]]
addPairs' [] = []
addPairs' ((m,n):mns) = [m+n]:addPairs mns

multAll' :: Int -> [Int] -> [Int]
multAll' x [] = []
multAll' x (y:ys) = (x*y):(multAll' x ys)
```

More generally for any function `f :: a -> b`, we can define a function

```
apply_f' :: [a] -> [b]
apply_f' []     = []
apply_f' (x:xs) = (f x):apply_f' xs
```

A start on higher types: Mapping, 3

Mapping via `map`

Let us define a *generic* function to do mapping:

```
map :: (a -> b) -> [a] -> [b]
map f lst = [ f x | x <- lst ]
```

—or—

```
map’ :: (a -> b) -> [a] -> [b]
map’ f []     = []
map’ f (x:xs) = (f x):map’ f xs
```

map is higher order, it accepts a function as an argument. E.g.,

```
map fst    [(1,False), (3,True), (-5,False), (34,False)] ⇝ [1,3,-5,34]
map length [[1,5,6], [3,5], [], [3..10]]                 ⇝ [3,2,0,8]
map sum    [[1,5,6], [3,5], [], [3..10]]                 ⇝ [12,8,0,52]
```

A start on higher types: Filtering, 1

Filtering elements from a list via list comprehensions

```
lessThan10 :: [Int] -> [Int]
lessThan10 xs = [ x | x <- xs, x<10 ]

offDiagonal :: [(Int,Int)] -> [(Int,Int)]
offDiagonal mns = [(m,n) | (m,n) <- mns , m/=n]
```

A start on higher types: Filtering, 2

Here is a generic way of doing filtering:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p lst = [x | x <- lst, p x]
```

—or—

```
filter’ :: (a -> Bool) -> [a] -> [a]
filter’ p [] = []
filter’ p (x:xs) | p x       = x:(filter’ p xs)
                 | otherwise = filter’ p xs
```

So

```
isOffDiag :: (Int,Int) -> Bool
isOffDiag (m,n) = (m/=n)

filter isOffDiag [(3,4),(5,5),(10,-2),(99,99)]  ⇝ [(3,4),(10,-2)]
filter isDigit "a37bZ9?"                        ⇝  "379?"
filter not [True,False,False,True]              ⇝ [False,False]
```

Functions as First-Class Values

In functional languages (generally), functions are *first-class values*, i.e. are treated just like any other value.

So functions can be

- passed as arguments to functions
- returned as results from functions
- bound to variables
- expressed without being given a name (λ-expressions)
- elements of list (and other data structures)
- ...

A function that

(i) accepts functions as arguments *or*

(ii) returns a function as a value *or*

(iii) both (i) and (ii)

is higher order. *E.g., map and filter.*

Higher-type goodies, 1

dropWhile, takeWhile

```
dropWhile, takeWhile
      ::  (a -> Bool) -> [a] -> [a]
```

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

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

Q: What is (<10) doing?

Q: What is "." doing??

For example:

```
takeWhile (<10) [0,3..20]                  ⇝  [0,3,6,9]
dropWhile (<10) [0,3..20]                  ⇝  [12,15,18]
dropWhile isSpace "     hi there   "       ⇝  "hi there   "
takeWhile (not . isSpace) "hi there "      ⇝  "hi"
dropWhile (not . isSpace) "hi there "      ⇝  " there   "
```

Digression: Sections and the composition operator

Sections

`10 + 3`	≡	`(+) 10 3`	≡	`(10 +) 3`	≡	`(+ 3) 10`
`10 == 3`	≡	`(==) 10 3`	≡	`(10 ==) 3`	≡	`(== 3) 10`
``10 `div` 3``	≡	`div 10 3`	≡	``(10 `div`) 3``	≡	``(`div` 3) 10``

```
(.)  ::  (b -> c) -> (a -> b) -> a -> c
```

```
(f . g) x = f (g x)
```

Example: Define a function `trim` that deletes leading and trailing white space from a string

```
trimFront str = dropWhile isSpace str
trim str = reverse (trimFront (reverse (trimFront str)))
-- or better yet
trim'    = reverse . trimFront . reverse . trimFront
```

Higher-type goodies, 2

```
span :: (a -> Bool) -> [a] -> ([a],[a])
span p []          = ([],[])
span p xs@(x:xs')
    | p x          =  (x:ys,zs)
    | otherwise =  ([],xs)
    where (ys,zs) = span p xs'
```

For example:

```
span (<10) [0,3..20]                ⇝   ([0,3,6,9],[12,15,18])
span isSpace "     hi there   "     ⇝   ("     ","hi there   ")
```

Q: What is the `@` doing in "`span p xs@(x:xs')`"?

Higher-type goodies, 3

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
zipWith' _ [] _           = []
zipWith' _ _ []           = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

For example:

```
sum $ zipWith (*) [2, 5, 3] [1.75, 3.45, 0.25]
  ⇝ sum [3.5, 17.25, 0.75]
  ⇝ 21.50

zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
  ⇝ [153.0,61.5,31.0,15.75,6.6]
```

Q: What is the "$" doing??

Q: What is the `(\a b -> (a * 30 + 3) / b)` doing?

Digression: The application operator

```
($) ::  (a -> b) -> a -> b
f $ x = f x      -- $ has low, right-associative binding precedence
```

So

```
sum $ filter (> 10) $ map (*2) [2..10]
```

$$\equiv$$

```
sum (filter (> 10) (map (*2) [2..10]))
```

Digression: λ-expressions

The following definitions are equivalent

```
munge, munge' :: Int -> Int
munge x = 3*x+1
munge'  = \x -> 3*x+1
```

So the following expressions are equivalent

```
map munge [2..8]
map munge' [2..8]
map (\x -> 3*x+1) [2..8]
```

So, `(\x -> 3*x+1)` defines a “nameless” function.

We can use `(\    ->    )` to return functional results. E.g.,

```
addNum :: Int -> (Int->Int)
addNum n = \x->(x+n)
```

Higher-types, structural recursion on lists, 1

Consider some structural recursion on lists:

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

concat' []       = []
concat' (xs:xss) = xs ++ concat' xss      -- = (++) xs (concat' xxs)

unzip' [] = ([],[])                       -- = f (x,y) (unzip' xys)
unzip' ((x,y):xys) = (x:xs,y:ys)          --    where f (a,b) (as,bs)
    where (xs,ys) = unzip' xys            --              = (a:as,b:bs)
```

These all have the general form:

```
someFun []     = z
someFun (x:xs) = f x (someFun xs)
```

So we can encapsulate this by:

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

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

Higher-types, structural recursion on lists, 2

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

Original

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

```
concat' []       = []
concat' (xs:xss)
    = xs ++ concat' xss
```

```
unzip' [] = ([],[])
unzip' ((x,y):xys) = (x:xs,y:ys)
    where (xs,ys) = unzip' xys
```

As a foldr

```
sum'' xs = foldr (+) 0 xs
```

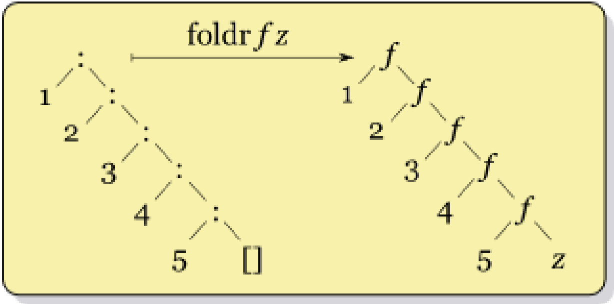
```
concat'' xss = foldr (++) [] xss
```

```
unzip'' xys = foldr f ([],[]) xys
  where
    f (x,y) (xs,ys) = (x:xs,y:ys)
```

Higher-types, structural recursion on lists, 3

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

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




```
foldr f z [x1, x2, ..., xn]
                   == x1 `f` (x2 `f` ... (xn `f` z)...)
```

Higher-types, structural recursion on lists, 4

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

Foldr's cousins, 1: `foldl`

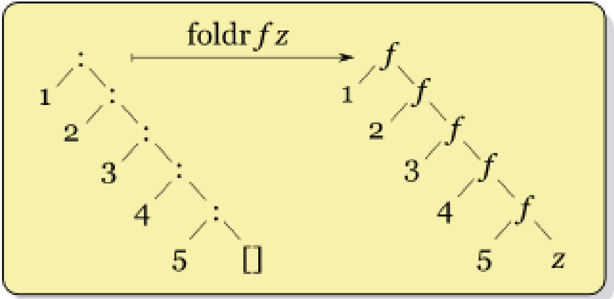
```
foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f z []     = z
  foldr f z (x:xs) = f x (foldr f z xs)
```

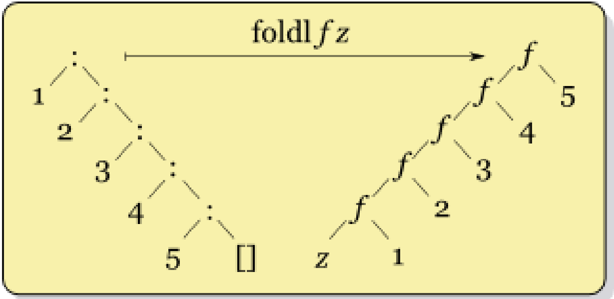
```
foldr f z [x1, x2, ..., xn]
                == x1 `f` (x2 `f` ... (xn `f` z)...)
```

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

```
foldl f z [x1, x2, ..., xn]
                == (...((z `f` x1) `f` x2) `f`...) `f` xn
```

foldr vs. foldl





Puzzles

Puzzle 1

```
foldr (:) [] [1,2,3] = ??
```

Puzzle 2

```
foldl (flip (:)) [] [1,2,3] = ??
```

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

Puzzles

Puzzle 1

```
foldr (:) [] [1,2,3] = ??
```

Puzzle 2

```
foldl (flip (:)) [] [1,2,3] = ??
```

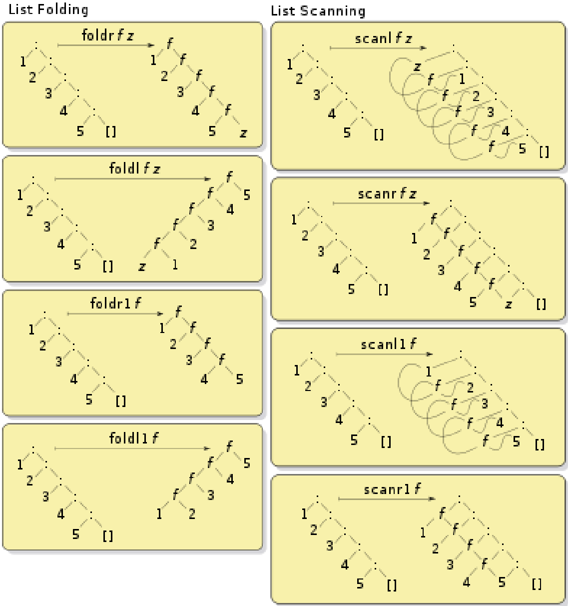
```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Pro Tip: It is almost always better to use `foldl'` than `foldl`. See

https://wiki.haskell.org/Foldr_Foldl_Foldl%27

for the gory details.

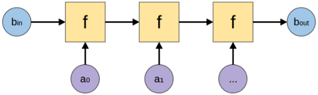
Foldr's cousins, 2



For folds: look here
For scans: look here

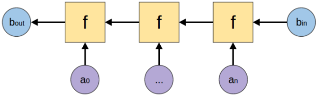
Foldr's cousins, 3

foldl



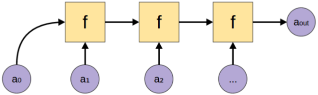

```
foldl :: (b -> a -> b)
            -> b -> [a] -> b
```

foldr



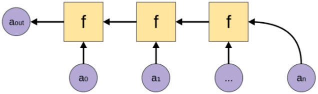

```
foldr :: (a -> b -> b)
            -> b -> [a] -> b
```

foldl1



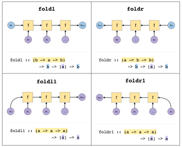

```
foldl1 :: (a -> a -> a)
               -> [a] -> a
```

foldr1




```
foldr1 :: (a -> a -> a)
               -> [a] -> a
```

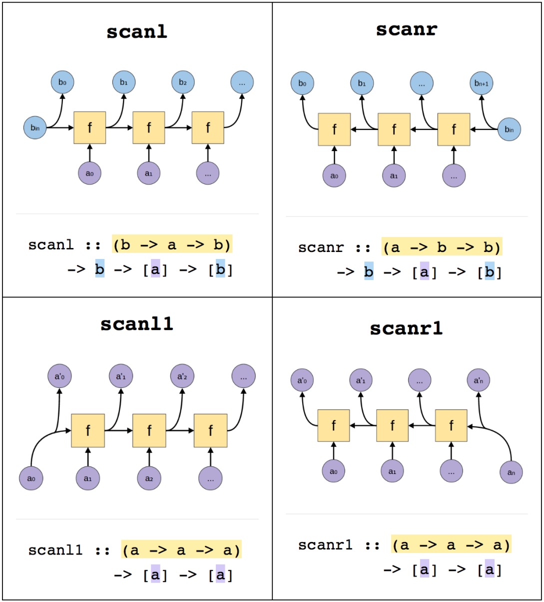
Foldr's cousins, 3



Try:

- `foldr1 max [1,4,8,4,9,4]`
- `foldl1 max [1,4,8,4,9,4]`
- `scanr1 max [1,4,8,4,9,4]`
- `scanl1 max [1,4,8,4,9,4]`

Foldr’s cousin’s: alternatively, 2

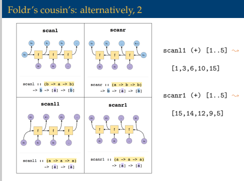


scanl1 (+) [1..5] ⇝

[1,3,6,10,15]

scanr1 (+) [1..5] ⇝

[15,14,12,9,5]



For folds:

http://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#g:11

For scans:

http://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#g:16

Class Exercises

1. Use `foldr` to define $n \mapsto 1^2 + 2^2 + 3^2 + \cdots + n^2$.
2. Use `foldr` and `foldl` to define `length`.
3. Use `foldr` and `foldl` to define `and` and `or`.
4. Use `foldr` or `foldl` to define `reverse`.
5. Use `scanr` or `scanl` to define $n \mapsto [1!, 2!, 3!, \ldots, n!]$.

Class Exercises

1. Use `foldr` to define $n \mapsto 1^2 + 2^2 + 3^2 + \cdots + n^2$.
2. Use `foldr` and `foldl` to define `length`.
3. Use `foldr` and `foldl` to define `and` and `or`.
4. Use `foldr` or `foldl` to define `reverse`.
5. Use `scanr` or `scanl` to define $n \mapsto [1!, 2!, 3!, \ldots, n!]$.

```
sumSq n = foldr (\ x r -> x*x+r) 0 [1..n]

length1 xs = foldr (\ x r -> 1+r) 0 xs
length2 xs = foldl (\ r x -> 1+r) 0 xs

and1 bs = foldr (\ x r -> x && r) True bs
and2 bs = foldl (\ r x -> x && r) True bs

or1 bs = foldr (\ x r -> x || r) False bs
or2 bs = foldl (\ r x -> x || r) False bs

reverse' xs = foldl (flip(:)) [] xs

facts n = scanl (*) 1 [2..n]
```

Aside: Structural Recursions on Natural Numbers, 1

We can introduce a “natural number data type” by:

```
data Nat = Zero | Succ Nat
```

where `Zero` stands for 0 and `Succ` stands for the function $x \mapsto x + 1$.
A structural recursion over `Nat`’s is a function of the form:

```
fun :: Nat -> a
fun Zero     = z
fun (Succ n) = f (fun n)
```

where `z::a` and `f::a -> a`. So if you expand things out, you see that

$$\texttt{fun}\ \underbrace{(\mathit{Succ}\ (\mathit{Succ}\ (\quad \ldots\ \mathit{Zero})))}_{\text{k many Succ's}} = \underbrace{(f\ (f\ (\quad \ldots\ z)))}_{\text{k many f's}}$$

We can define a fold for `Nat`’s by:

```
foldn :: (a->a) -> a -> Nat -> a
foldn f z Zero     = z
foldn f z (Succ n) = f (foldn f z n)
```

Aside: Structural Recursions on Natural Numbers, 2

Using

```
data Nat = Zero | Succ Nat

foldn :: (a->a) -> a -> Nat -> a
foldn f z Zero     = z
foldn f z (Succ n) = f (foldn f z n)
```

we can bootstrap arithmetic by:

```
add m n   = foldn Succ n m
times m n = foldn (`add` n) Zero m
```

etc.

Functions and types

In Haskell every function

- takes exactly one argument and
- returns exactly one value.

For example: `f ::` $\underbrace{\texttt{Int}}_{\text{arg type}}$ `->` $\underbrace{\texttt{Bool}}_{\text{result type}}$ In general: `g ::` $\underbrace{\texttt{t1}}_{\text{arg type}}$ `->` $\underbrace{\texttt{t2}}_{\text{result type}}$

Examples:

- `(Int -> Bool) -> Char`
- `Int -> (Bool -> Char)` $\equiv$ `Int -> Bool -> Char`

`->` associates to the right

$$t_1 \to t_2 \to \cdots \to t_n \to t \quad \equiv \quad t_1 \to (t_2 \to \ldots (t_n \to t) \ldots)$$

Associations

Convention: `->` associates to the right

$$t_1 \to t_2 \to t_3 \to \cdots \to t_n \to t \equiv t_1 \to (t_2 \to (t_3 \to (\ldots (t_n \to t) \ldots)))$$

Convention: application associates to the left

$$f\, x_1\, x_2\, x_3\, \ldots\, x_n \quad \equiv \quad (\ldots (((f\, x_1)\, x_2)\, x_3) \ldots x_n)$$

WHY?

Suppose

```
f  :: t1 -> t2 -> t3 -> t
e1 :: t1
e2 :: t2
e3 :: t3
```

Then

```
f e1          :: t2 -> t3 -> t
f e1 e2       :: t3 -> t
f e1 e2 e3 :: t
```

Currying and Uncurrying

Consider

```
comp1 :: Int -> Int -> Bool
comp1 x y = (x<y)

comp2 :: (Int,Int) -> Bool
comp2 (x,y) = (x<y)
```

Every	`f :: t1 -> t2 -> ...  -> tn -> t`
has a corresponding	`f' :: (t1,t2,...,tn) -> t`
and vise versa.	

In fact

```
curry2 :: ((a,b)->c) -> a -> b -> c
curry2 g = \ x y -> g(x,y)

uncurry2 :: (a->b->c) -> (a,b) -> c
uncurry2 f = \ (x,y) -> f x y
```

Mathematically: This is just a fancier version of:

$$(c^b)^a = c^{a \times b}$$

from High School math.

Was that so bad?