Programming in Haskell

S P Suresh

http://www.cmi.ac.in/~spsuresh

Lecture 10

September 11, 2017

Combining elements

- ```
  sumlist :: [Int] -> Int
  sumlist [] = 0
  sumlist (x:xs) = x + (sumlist xs)
  ```
- ```
  multlist :: [Int] -> Int
  multlist [] = 1
  multlist (x:xs) = x * (multlist xs)
  ```
- What is the common pattern?

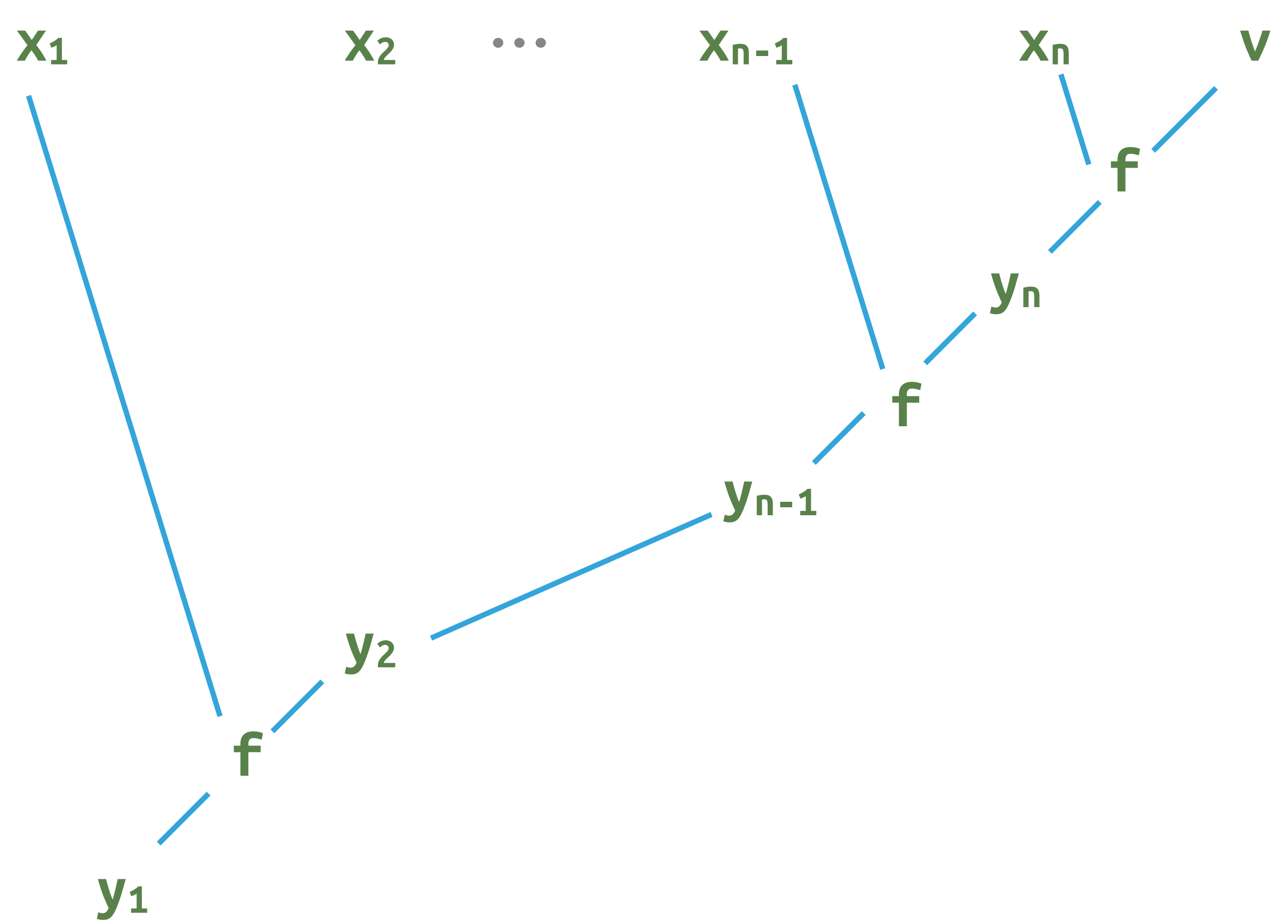
Combining elements ...

- ```
  combine f v [] = v
  combine f v (x:xs) = x `f` (combine f v xs)
  ```
- We can then write
- `sumlist l  = combine (+) 0 l`
- `multlist l = combine (*) 1 l`
  ```

foldr

- The built-in version of combine is called **`foldr`**

- ```
  foldr f v [] = v
  foldr f v (x:xs) = x `f` (foldr f v xs)
  ```


  ```

foldr

- The built-in version of combine is called **`foldr`**
- ```
  foldr f v [] = v
  foldr f v (x:xs) = x `f` (foldr f v xs)
  ```
- `sumlist [1,2,3] = 1 + (2 + (3 + 0))`
- foldr f v [x_1, x_2, x_3] = x_1 \`f\` (x_2 \`f\` (x_3 \`f\` v))
- foldr f v x_1:(x_2:(x_3:[])) = x_1 \`f\` (x_2 \`f\` (x_3 \`f\` v))
  - Replace `[]` by `v`, and replace `:` by `` `f` ``
  ```

Examples

- `sumlist l = foldr (+) 0 l`
- `multlist l = foldr (*) 1 l`
- ```
  mylength :: [Int] -> Int
  mylength l =  foldr f 0 l
    where
    f x y = y+1
  ```
- Note: can simply write `mylength = foldr f 0`
  - Outermost reduction: `mylength l` $\Rrightarrow$ `foldr f 0 l`
- `mylength = foldr (\_ y -> y+1) 0`
  ```

Aside: Anonymous functions

- Usual practice with functions
  - Define functions – giving it a name
  - Use them elsewhere
- Sometimes it breaks the flow to follow this pattern
- Unnamed functions

Aside: Anonymous functions

- Example:

```
foldr f 0 [1..]
    where f x y = x
```

- Easier to say this:

```
foldr (\x y -> x) 0 [1..]
```

- We are specifying the function we want to use without naming it

- `\x y -> x` is a function that takes two inputs and returns the first input

More foldr examples …

- Recall
- `appendright x l = l ++ [x]`
- `foldr appendright [] = ??`
- `foldr appendright [] = reverse`

More foldr examples …

- What is `foldr (++) []` ?
- Dissolves one level of brackets
  - Flattens a list of lists into a single list
- The built-in function `concat`

foldr

- ```
  foldr f v [] = v
  foldr f v (x:xs) = f x (foldr f v xs)
  ```
- What is the type of `foldr`?
  - `foldr :: (a -> b -> b) -> b -> [a] -> b`
  ```

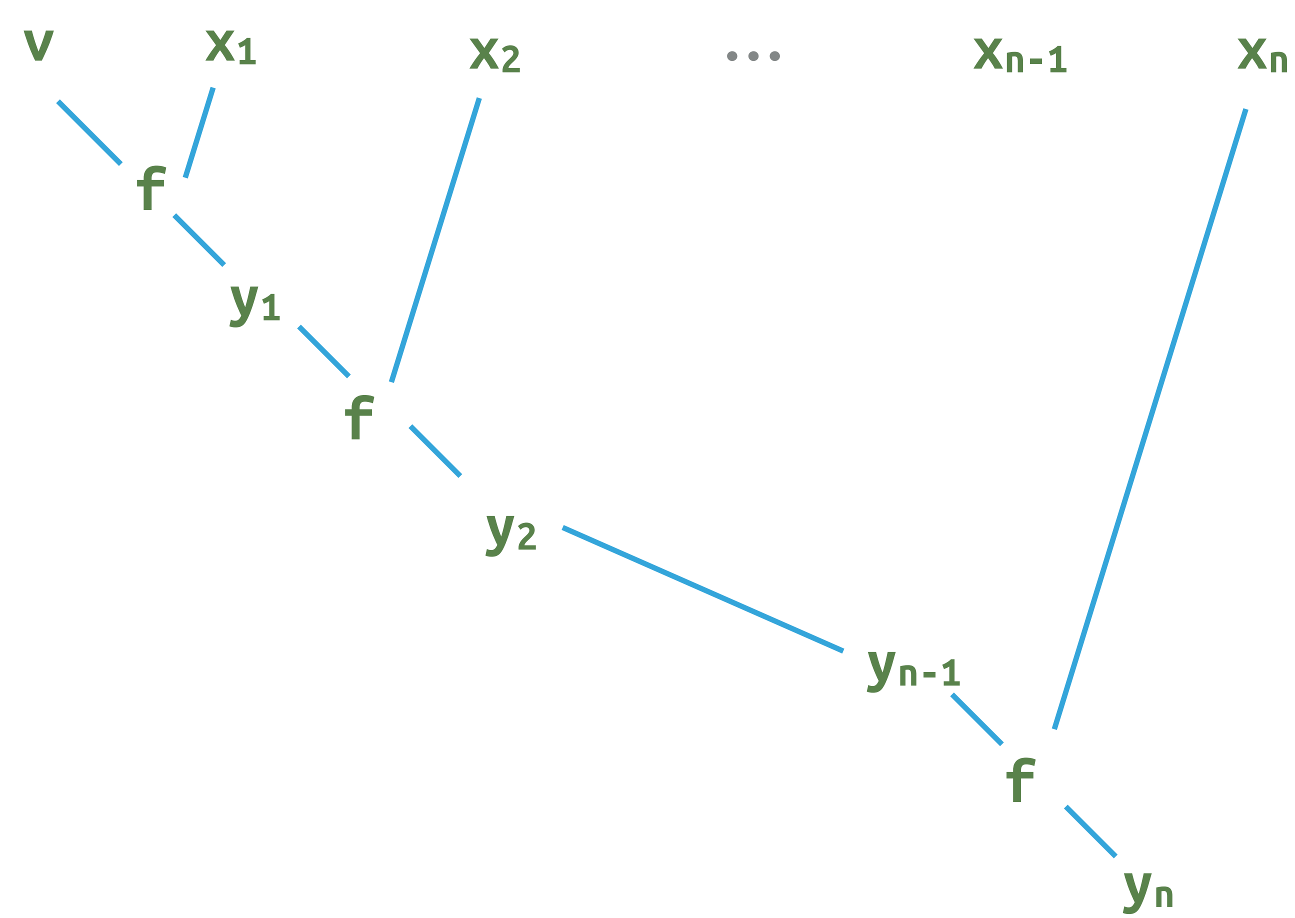
foldr1

- Sometimes there is no natural value to assign to the empty list
- Finding the maximum value in the list
  - Maximum is undefined for empty list
- ```
  foldr1 f [x] = x
  foldr1 f (x:xs) = f x (foldr1 f xs)
  ```
- `maxlist = foldr1 max`
  ```

Folding from the left

- Sometimes useful to fold left to right

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


  ```

Example

- Translate a string of digits to an integer
- `strtonum "234" = 234`
- Convert a character into the corresponding digit:
-
```
chartonum :: Char -> Int
chartonum c
    | ('0' <= c) && (c <= '9')
                   = (ord c) - (ord '0')
```

Example ...

- Process the digits left to right
- Multiply current sum by 10 and add next digit
- ```
  nextdigit :: Int -> Char -> Int
  nextdigit i c = 10*i + (chartonum c)
  ```
- ```
  strtonum = foldl nextdigit 0
  ```

Computations with foldr

- `foldr f v [x1, x2,..., xn]`
- ⇛ `f x1 (foldr f v [x2,...,xn])`
- ⇛ `f x1 (f x2 (foldr f v [x3,...,xn]))`
- ⇛ `f x1 (f x2 (f x3 (foldr f v [x4,...,xn])))`
- ⇛ `...`
- ⇛ `f x1 (f x2 (f x3 (...(f xn (foldr f v []))...)))`
- ⇛ `f x1 (f x2 (f x3 (... (f xn v)...)))`

Computations with foldr

- `foldr (+) 0 [1..100]`
- ⇛ `1 + (foldr (+) 0 [2..100])`
- ⇛ `1 + (2 + (foldr (+) 0 [3..100]))`
- ⇛ `...`
- ⇛ `1 + (2 + (... ((+) 100 (foldr (+) 0 []))...))`
- ⇛ `1 + (2 + (... (100 + 0)...))`
- ⇛ `...`
- ⇛ `5050`

Computations with foldr

- `foldr f v [x1, x2,..., xn]`
- ⇛ `f x1 (foldr f v [x2,...,xn])`
- ⇛ `...`
- ⇛ `f x1 (f x2 (f x3 (... (f xn v)...)))`
- If `f` needs both inputs, it will be applied only at the end
- Need space to carry around huge expressions

Computations with foldl

- `foldl f v [x1, x2,..., xn]`
- ⇛ `foldl f (f v x1) [x2,...,xn]`
- ⇛ `foldl f (f (f v x1) x2) [x3,...,xn]`
- ⇛ `foldl f (f (f (f v x1) x2) x3) [x4,...,xn]`
- ⇛ `...`
- ⇛ `foldl f (f ...(f (f (f v x1) x2) x3))... xn) []`
- ⇛ `f ...(f (f (f v x1) x2) x3))... xn`

Computations with foldl

- `foldl (+) 0 [1..100]`
- ⇛ `foldl (+) (0 + 1) [2..100]`
- ⇛ `foldl (+) ((0 + 1) + 2) [3..100]`
- ⇛ `...`
- ⇛ `foldl (+) ((...(0 + 1) + 2)...) + 100) []`
- ⇛ `((...(0 + 1) + 2)...) + 100)`
- ⇛ `...`
- ⇛ `5050`

Computations with foldl

- `foldl f v [x1, x2,..., xn]`
- ⇛ `foldl f (f v x1) [x2,...,xn]`
- ⇛ `...`
- ⇛ `f ...(f (f (f v x1) x2) x3))... xn`
- Same problem as with `foldr`
- Huge expression carried around till the end

Computations with foldl'

- `foldl' f a [x1, x2,..., xn]`
- ⇛ `foldl' f y1 [x2,...,xn]` — `y1 = f a x1`
- ⇛ `foldl' f y2 [x3,...,xn]` — `y2 = f y1 x2`
- ⇛ `foldl' f y3 [x4,...,xn]` — `y3 = f y2 x3`
- ⇛ `...`
- ⇛ `foldl' f yn []` — `yn = f y(n-1) xn`
- ⇛ `yn`
- **Eager evaluation**

Computations with foldl'

- `foldl' (+) 0 [1..100]`
- ⇛ `foldl' (+) 1 [2..100]`
- ⇛ `foldl' (+) 3 [3..100]`
- ⇛ `...`
- ⇛ `foldl' 5050 []`
- ⇛ `5050`

Computations with foldl'

- `foldl'` defined in `Data.List`
- ```
  foldl' f a [] = a
  foldl' f a (x:xs) = y `seq` foldl' f y xs
                      where y = f a x
  ```
- The `seq` function takes two arguments, evaluates the first, and returns the value of the second
- `seq :: a -> b -> b`
- Forces the values in `foldl'` to computed as early as possible
  ```

foldr on infinite lists

- `foldr` works on infinite lists sometimes when `foldl` or `foldl'` does not

- ```
  foldr (\x y -> x) 0 [1..]
  ⇛ (\x y -> x) 1 (foldr (\x y -> x) 0 [2..])
  ⇛ 1
  ```

- ```
  foldl' (\x y -> x) 0 [1..]
  ⇛ foldl' (\x y -> x) 0 [2..]
  ⇛ foldl' (\x y -> x) 0 [3..]
  ⇛ foldl' (\x y -> x) 0 [4..]
   ⇛ ...
  ```

foldl using foldr

- Let `step x g = \a -> g (f a x)`
- **Claim**: For all expressions `e`,

  `foldr step id xs e = foldl f e xs`
- **Proof**: By induction on length of `xs`
  - `(foldr step id []) e = id e = e = foldl f e []`
  - `(foldr step id (x:xs)) e`

    ⇛ `(step x (foldr step id xs)) e`

    ⇛ `(\a -> (foldr step id xs) (f a x)) e`

    ⇛ `(\a -> foldl f (f a x) xs) e` – By induction hypothesis

    ⇛ `foldl f (f e x) xs = foldl f e (x:xs)`

Useful functions

- `flip :: (a -> b -> c) -> b -> a -> c`
- If we have a definition `foldr f a l` and want to change it to `foldl`, we do `foldl (flip f) a l`
- `const :: a -> b -> a`
- `const x y = x`
- `foldr const 0 [1..] = 1`
- ```
  ($) :: (a -> b) -> a -> b
  ($) f x = f x
  ```
- `($!) :: (a -> b) -> a -> b` – This is not the official definition
  `($!) f x = x `seq` f x` – Only conveys the intended behaviour
  ```

takeWhile

- `take n l` returns `n` element prefix of list `l`
- Instead, use a property to determine the prefix
- `takeWhile :: (a -> Bool) -> [a] -> [a]`
- `takeWhile (> 7) [8,1,9,10] = [8]`
- `takeWhile (< 10) [8,1,9,10]= [8,1,9]`

Example: position

- **position c s** : first position in **s** where **c** occurs

```
position :: Char -> String -> Int
position c "" = 0
position c (d:ds)
   | c == d    = 0
   | otherwise = 1 + (position c ds)
```

- Using **takeWhile**
- **position c s = length (takeWhile (/= c) s)**
- Symmetric function **dropWhile**