Programming in Haskell

S P Suresh

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

Lecture 9

September 6, 2017

Translating list comprehensions

- List comprehension can be rewritten using `map`, `filter` and `concat`
- A list comprehension has the form
  - `[e | q1, q2, ..., qN]`
- where each `qj` is either
  - a boolean condition `b` or
  - a generator `p <- l`, where `p` is a pattern and `l` is a list valued expression

Translating ...

- A boolean condition acts as a filter.

- `[e | b,Q] = if b then [e | Q] else []`

- Depends only on generators/qualifiers to its left

Translating ...

- Generator `p <- l` produces a list of candidates
- Naive translation
  - `[e | p <- l, Q] = map f l`
    `where`
    `f p = [e | Q]`
    `f _ = []`

Translating ...

```
[n*n | n <- [1..7], mod n 2 == 0]
```

```
⇛ map f [1..7]
   where
   f n = [ n*n | mod n 2 == 0]

⇛ map f [1..7]
   where
   f n = if (mod n 2 == 0) then [n*n] else []

⇛ [[],[4],[],[16],[],[36],[]]
```

Translating ...

- Need an extra `concat` when translating `p <- l`
- Correct translation
  - ```
    [e | p <- l, Q] = concat $ map f l
      where
      f p = [e | Q]
      f _ = []
    ```
    ```

Translating ...

```
[n*n | n <- [1..7], mod n 2 == 0]

⇛ concat $ map f [1..7]
    where
    f n = [ n*n | mod n 2 == 0]

⇛ concat $ map f [1..7]
    where
    f n = if (mod n 2 == 0) then [n*n] else []

⇛ concat [[],[4],[],[16],[],[36],[]]

⇛ [4,16,36]
```

The Sieve of Eratosthenes

- Start with the (infinite) list *[2,3,4,...]*
- Enumerate the left most element as next prime
- Remove all its multiples from the list
- Repeat the above with this list

The Sieve of Eratosthenes

In Haskell,

```
primes = sieve [2..]
  where
  sieve (x:xs) =
     x:(sieve [y | y <- xs, y `mod` x /= 0])
```

The Sieve of Eratosthenes

```
primes ⇛ sieve [2..]

⇛ 2:(sieve [ y | y <- [3..] , y `mod` 2 /= 0])

⇛ 2:(sieve (3:[y | y <- [4..], y `mod` 2 /= 0])

⇛ 2:(3:(sieve [z |
    z <- [y | y <- [4..], y `mod` 2 /= 0] |
                                  z `mod` 3 /= 0])

⇛ 2:(3:(5:(sieve [w |
    w <- [z |
       z <- [y | y <- [4..], y `mod` 2 /= 0] |
                              z `mod` 3 /= 0] |
                                  w `mod` 5 /= 0])

⇛ ...
```

Summary

- List comprehension is a succinct, readable notation for combining map and filter
- Can translate list comprehension in terms of `concat`, `map`, `filter`

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) = f x (combine f v xs)
  ```

- We can then write

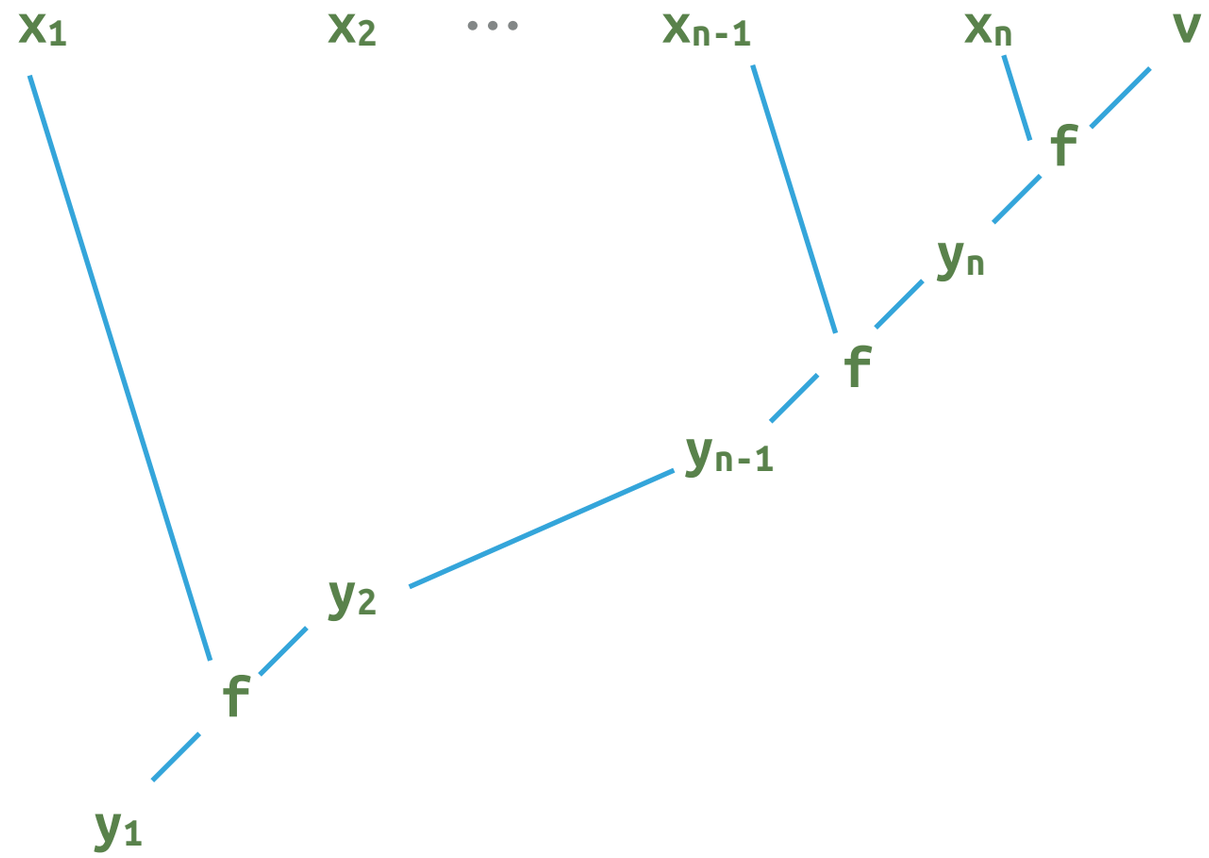
- ```
  sumlist l  = combine (+) 0 l
  ```

- ```
  multlist l = combine (*) 1 l
  ```
  ```

foldr

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

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


  ```

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` ⇛ `foldr f 0 l`
- `mylength = foldr (\_ y -> y+1) 0`

Examples ...

- Recall

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

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`

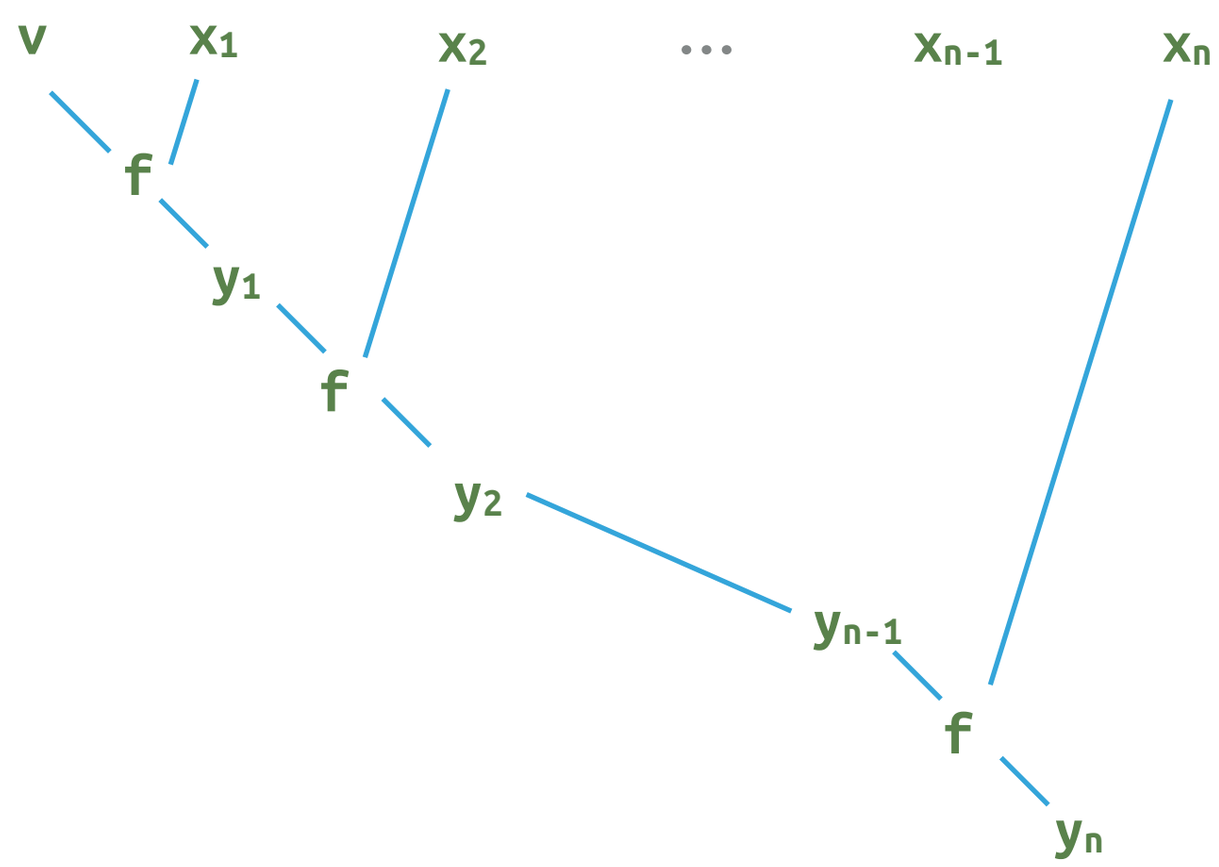
foldr1

- 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
  ```

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`