Programming in Haskell: Lecture 11

S P Suresh

September 12, 2019

Combining elements

- Sum all numbers in a list

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

Combining elements

- Sum all numbers in a list

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

- Multiply all numbers in a list

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

Combining elements

- Sum all numbers in a list

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

- Multiply all numbers in a list

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

- What is the common pattern?

Combining elements

- Combining elements using `v` and `f`

```
combine :: (Int -> Int -> Int) -> Int -> [Int] -> Int
combine f v []     = v
combine f v (x:xs) = f x (combine f v xs)
```

Combining elements

- Combining elements using `v` and `f`

```
combine :: (Int -> Int -> Int) -> Int -> [Int] -> Int
combine f v []     = v
combine f v (x:xs) = f x (combine f v xs)
```

- Sum and product can be expressed as:

```
sum     = combine (+) 0
product = combine (*) 1
```

foldr

- Built-in combine is called **foldr** (fold right)

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


    foldr f v [x1, x2, x3]
  = f x1 (foldr f v [x2, x3])
  = f x1 (f x2 (foldr f v [x3]))
  = f x1 (f x2 (f x3 (foldr f v [])))
  = f x1 (f x2 (f x3 v))
```

foldr

- `foldr` replaces `[]` by `v` and `:` by `` `f` `` in the list:

```
xs
    = x1  :  (x2  :  (x3  :  (...  :  xn-1  :  (xn  :  []))))
foldr f v xs
    = x1 `f` (x2 `f` (x3 `f` (... `f` xn-1 `f` (xn `f` v ))))
```

`foldr` *examples*

- Sum and product

```
sum     = foldr (+) 0
product = foldr (*) 1
xs
    = x1 : (x2 : (x3 : (... : xn-1 : (xn : []))))
sum xs
    = x1 + (x2 + (x3 + (... + xn-1 + (xn + 0 ))))
product xs
    = x1 * (x2 * (x3 * (... * xn-1 * (xn * 1 ))))
```

`foldr` *and anonymous functions*

- Can express `length` in terms of `foldr`

```
length = foldr f 0
    where
        f x n = n+1
```

`foldr` *and anonymous functions*

- Can express `length` in terms of `foldr`

```
length = foldr f 0
    where
        f x n = n+1
```

- Not always convenient to name such functions

`foldr` *and anonymous functions*

- Can express `length` in terms of `foldr`

```
length = foldr f 0
    where
        f x n = n+1
```

- Not always convenient to name such functions
- Impedes readability sometimes

`foldr` *and anonymous functions*

- Can express `length` in terms of `foldr`

```
length = foldr f 0
    where
        f x n = n+1
```

- Not always convenient to name such functions
- Impedes readability sometimes
- Anonymous functions:

```
length = foldr (\x n -> n+1) 0
```

Anonymous functions

- Anonymous functions are described using **lambdas**

Anonymous functions

- Anonymous functions are described using **lambdas**
- `\x n -> n+1` is an **unnamed** function of two arguments that increments its second argument by `1`

Anonymous functions

- Anonymous functions are described using **lambdas**
- `\x n -> n+1` is an **unnamed** function of two arguments that increments its second argument by `1`
- Can also give it a name!

Anonymous functions

- Anonymous functions are described using **lambdas**
- `\x n -> n+1` is an **unnamed** function of two arguments that increments its second argument by `1`
- Can also give it a name!
- The two definitions of `f` below are equivalent:

```
f = \x n -> n+1
f x n = n+1
```

Anonymous functions

- Anonymous functions are described using **lambdas**
- `\x n -> n+1` is an **unnamed** function of two arguments that increments its second argument by `1`
- Can also give it a name!
- The two definitions of `f` below are equivalent:

```
f = \x n -> n+1
f x n = n+1
```

- Anonymous functions are very convenient to use with higher order functions

More `foldr` examples

- `foldr (:) []` is equivalent to the identity function on lists

More `foldr` examples

- `foldr (:) []` is equivalent to the identity function on lists
- `f = foldr (\x l -> l++[x]) []`

```
  f [x1, x2, x3]
= (f [x2, x3])                ++ [x1]
= ((f [x3])         ++ [x2]) ++ [x1]
= (((f []) ++ [x3]) ++ [x2]) ++ [x1]
= (([]     ++ [x3]) ++ [x2]) ++ [x1]
= [x3, x2, x1]
```

More `foldr` examples

- `foldr (:) []` is equivalent to the identity function on lists
- `f = foldr (\x l -> l++[x]) []`

```
      f [x1, x2, x3]
    = (f [x2, x3])                 ++ [x1]
    = ((f [x3])          ++ [x2]) ++ [x1]
    = (((f []) ++ [x3]) ++ [x2]) ++ [x1]
    = (([]     ++ [x3]) ++ [x2]) ++ [x1]
    = [x3, x2, x1]
```

- `f` is just `reverse`, but takes time proportional to n^2

More `foldr` examples

- `foldr (:) []` is equivalent to the identity function on lists
- `f = foldr (\x l -> l++[x]) []`

```
  f [x1, x2, x3]
= (f [x2, x3])                 ++ [x1]
= ((f [x3])          ++ [x2]) ++ [x1]
= (((f []) ++ [x3]) ++ [x2]) ++ [x1]
= (([]     ++ [x3]) ++ [x2]) ++ [x1]
= [x3, x2, x1]
```

- `f` is just `reverse`, but takes time proportional to n^2
- `concat` is just `foldr (++) []`

`foldr1`

- Sometimes there is no natural value to assign to the empty list

`foldr1`

- Sometimes there is no natural value to assign to the empty list
- For example, finding the maximum value in a list

`foldr1`

- Sometimes there is no natural value to assign to the empty list
- For example, finding the maximum value in a list
- Maximum is undefined for empty list

`foldr1`

- Sometimes there is no natural value to assign to the empty list
- For example, finding the maximum value in a list
- Maximum is undefined for empty list
- We use `foldr1` in such cases

`foldr1`

- Sometimes there is no natural value to assign to the empty list
- For example, finding the maximum value in a list
- Maximum is undefined for empty list
- We use `foldr1` in such cases
- Uses the last element as initial value

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)

maximum = foldr1 max
```

Folding from the left

- Sometimes it is useful to fold from the left

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

foldl f v [x1,x2,...,xn-1,xn]
    = (((v `f` x1) `f` x2) ... xn-1) `f` xn
```

Folding from the left

- Sometimes it is useful to fold from the left

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

foldl f v [x1,x2,...,xn-1,xn]
    = (((v `f` x1) `f` x2) ... xn-1) `f` xn
```

- Translate a string of digits to a number

```
strToNum :: String -> Int
strToNum = foldl (\n c -> 10*n + digitToInt c) 0
```

Folding from the left

- Let `g n c = 10*n + digitToInt c`

- Let `g n c = 10*n + digitToInt c`
- Here is how `strToNum = foldl g 0` works

```
  strToNum                          "234"
= foldl g   0                       "234"
= foldl g   (g 0 '2')                "34"
= foldl g   (g (g 0 '2') '3')         "4"
= foldl g   (g (g (g 0 '2') '3') '4')  ""
=           g (g (g 0 '2') '3') '4'
=           10 * (g (g 0 '2') '3') + 4
=           10 * (10 * (g 0 '2') + 3) + 4
=           10 * (10 * (10*0 + 2) + 3) + 4
=           234
```

foldr

- Fold from the right using function `f` and initial value `v`

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

  foldr f v [x1, x2, x3, ..., xn]
= f x1 (foldr f v [x2, x3, ..., 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) ...)))
```

foldr

- Fold from the right using function `(+)` and initial value `0`

```
  foldr (+) 0 [1..100]
= (+) 1 (foldr (+) 0 [2..100])
= (+) 1 ((+) 2 (foldr (+) 0 [3..100]))
= (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [4..100])))
= ...
= (+) 1 ((+) 2 ((+) 3 (... ((+) 100
                          (foldr (+) 0 [])) ...)))
= (+) 1 ((+) 2 ((+) 3 (... ((+) 100 0) ...)))
= ...
=  5050
```

`foldr`

- Fold from the right using function `f` and initial value `v`

```
  foldr f v [x1, x2, x3, ..., xn]
= ...
= f x1 (f x2 (f x3 (... (f xn v) ...)))
```

`foldr`

- Fold from the right using function `f` and initial value `v`

```
  foldr f v [x1, x2, x3, ..., xn]
= ...
= f x1 (f x2 (f x3 (... (f xn v) ...)))
```

- If `f` needs both inputs, it will be applied only at the end

`foldr`

- Fold from the right using function `f` and initial value `v`

```
  foldr f v [x1, x2, x3, ..., 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 huge expressions around

foldl

- Fold from the left using function `f` and initial value `v`

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

  foldl f    v    [x1, x2, x3, ..., xn]
= foldl f    (f v x1)    [x2, x3, ..., 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
```

foldl

- Fold from the left using function `(+)` and initial value `0`

```
  foldl (+)    0    [1..100]
= foldl (+)    ((+) 0 1)    [2..100]
= foldl (+)    ((+) ((+) 0 1) 2)    [3..100]
= foldl (+)    ((+) ((+) ((+) 0 1) 2) 3)    [4..100]
= ...
= foldl (+)    ((+) (... ((+) ((+) ((+) 0 1) 2) 3)
                    ...) 100)
               []
= (+) (... ((+) ((+) ((+) 0 1) 2) 3) ...) 100
```

foldl

- Fold from the left using function `f` and initial value `v`

```
  foldl f v [x1, x2, x3, ..., xn]
= ...
= f (... (f (f (f v x1) x2) x3) ...) xn
```

`foldl`

- Fold from the left using function `f` and initial value `v`

```
      foldl f v [x1, x2, x3, ..., xn]
    = ...
    = f (... (f (f (f v x1) x2) x3) ...) xn
```

- Same problem as with **`foldr`**

`foldl`

- Fold from the left using function `f` and initial value `v`

```
    foldl f v [x1, x2, x3, ..., xn]
  = ...
  = f (... (f (f (f v x1) x2) x3) ...) xn
```

- Same problem as with **`foldr`**
- Need space to carry huge expressions around

`foldl'`

- Defined in `Data.List`

foldl'

- Defined in Data.List
- Eager version of **foldl**

```haskell
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' f v []     = v
foldl' f v (x:xs) = y `seq` foldl' f y xs
    where y = f v x
```

`foldl'`

- Defined in `Data.List`
- Eager version of `foldl`

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

- `seq :: a -> b -> b`

`foldl'`

- Defined in `Data.List`
- Eager version of `foldl`

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

- `seq :: a -> b -> b`
- Evaluates the first argument first and then returns the second argument

`foldl'`

- Defined in `Data.List`
- Eager version of **`foldl`**

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

- `seq :: a -> b -> b`
- Evaluates the first argument first and then returns the second argument
- Useful when first argument is used in second argument

`foldl'`

- Defined in `Data.List`
- Eager version of `foldl`

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

- `seq :: a -> b -> b`
- Evaluates the first argument first and then returns the second argument
- Useful when first argument is used in second argument
- Forces the values in `foldl'` to be evaluated as early as possible

foldl'

- Computing with `foldl'`:

```
  foldl' f v  [x1,x2,x3,...,xn]
= foldl' f y1    [x2,x3,...,xn]     -- y1 = f v x1
= foldl' f y2       [x3,...,xn]     -- y2 = f y1 x2
= foldl' f y3       [x4,...,xn]     -- y3 = f y2 x3
= ...
= foldl' f yn                []     -- yn = f yn-1 xn
= yn
```

foldl'

- foldl' (+) 0:

```
  foldl' (+) 0  [1..100]
= foldl' (+) 1  [2..100]     -- 1 = (+) 0 1
= foldl' (+) 3  [3..100]     -- 3 = (+) 1 2
= foldl' (+) 6  [4..100]     -- 6 = (+) 3 3
= ...
= foldl' (+) 5050      []    -- 5050 = (+) 4950 100
= 5050
```

`foldr` *on infinite lists*

- `foldr` can be made to work on infinite lists

`foldr` *on infinite lists*

- `foldr` can be made to work on infinite lists
- If `f` does not require the second argument, the fold can terminate

`foldr` *on infinite lists*

- `foldr` can be made to work on infinite lists
- If `f` does not require the second argument, the fold can terminate
- A complicated `head`:

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

`foldr` *on infinite lists*

- `foldr` can be made to work on infinite lists
- If `f` does not require the second argument, the fold can terminate
- A complicated `head`:

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

- Does not work with left folds:

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

Simulating `foldl` *using* `foldr`

- Let `step x g a = g (f a x)`

Simulating `foldl` *using* `foldr`

- Let `step x g a = g (f a x)`
- **Claim**: For all `g`, `xs` and `e`, `foldr step g xs e = g (foldl f e xs)`

Simulating `foldl` *using* `foldr`

- Let `step x g a = g (f a x)`
- **Claim**: For all `g`, `xs` and `e`, `foldr step g xs e = g (foldl f e xs)`
- **Proof**: By induction on `length xs`

```
  foldr step g [] e = g e = g (foldl f e [])


  foldr step g (x:xs) e
= step x (foldr step g xs) e
= foldr step g xs (f e x)
= g (foldl f (f e x) xs)
          -- (ind. hyp. applied on g, xs and (f e x))
= g (foldl f e (x:xs))
```

Simulating `foldr` *using* `foldl`

- Let `step' g x a = g (f x a)`

Simulating **`foldr`** *using* **`foldl`**

- Let `step' g x a = g (f x a)`
- Claim: For all `g`, `xs` and `e`, `foldl step' g xs e = g (foldr f e xs)`

Simulating `foldr` *using* `foldl`

- Let `step' g x a = g (f x a)`
- **Claim**: For all `g`, `xs` and `e`, `foldl step' g xs e = g (foldr f e xs)`
- **Proof**: By induction on `length xs`

```
  foldl step' g [] e = g e = g (foldr f e [])


  foldl step' g (x:xs) e
= foldl step' (step' g x) xs e
= step' g x (foldr f e xs)
      -- (ind. hyp. applied on step' g x, xs and e)
= g (f x (foldr f e xs))
= g (foldr f e (x:xs))
```

Some useful functions

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

Some useful functions

- `flip :: (a -> b -> c) -> b -> a -> c`
- `flip f` behaves like `f`, but accepts the arguments in reverse order

Some useful functions

- `flip :: (a -> b -> c) -> b -> a -> c`
- `flip f` behaves like `f`, but accepts the arguments in reverse order
- `flip (:) [1..10] 0 = [0..10]`

Some useful functions

- **`flip`** `:: (a -> b -> c) -> b -> a -> c`
- **`flip`** `f` behaves like `f`, but accepts the arguments in reverse order
- **`flip`** `(:) [1..10] 0 = [0..10]`
- **`foldr`** `f v l` can be changed to **`foldl`** `(`**`flip`** `f) v l`

Some useful functions

- `flip :: (a -> b -> c) -> b -> a -> c`
- `flip f` behaves like `f`, but accepts the arguments in reverse order
- `flip (:) [1..10] 0 = [0..10]`
- `foldr f v l` can be changed to `foldl (flip f) v l`
- Other useful functions

```
const :: a -> b -> a
const x y = x
($) :: (a -> b) -> a -> b
($) f x = f x
($!) :: (a -> b) -> a -> b
($!) f x = x `seq` f x      -- Eager version
```

`foldl` *using* `foldr`, *again*

- For finite lists:

```
foldl f = flip (foldr step id)
    where step x g a = g (f a x)

  flip (foldr step id) e xs
= foldr step id xs e
= id (foldl f e xs)
= foldl f e xs
```

`foldr` *using* `foldl`*, again*

- For finite lists:

```
foldr f = flip (foldl step' id)
    where step' g x a = g (f x a)

  flip (foldl step' id) e xs
= foldl step' id xs e
= id (foldr f e xs)
= foldr f e xs
```