Programming in Haskell: Lecture 11

S P Suresh

September 12, 2019

Suresh

PRGH 2019: Lecture 11

September 12, 2019 1 / 27

• Sum all numbers in a list

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

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

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?

Suresh

• 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 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

• 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 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

• Can express length in terms of foldr

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

• Can express length in terms of foldr

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

• Not always convenient to name such 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

• 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 ($x n \rightarrow n+1$) 0

• Anonymous functions are described using lambdas

- 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 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 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:

 $f = \langle x n -> n+1$ f x n = n+1

- 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:

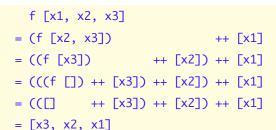
 $f = \langle x n -> n+1 \rangle$ f x n = n+1

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

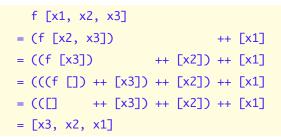
Suresh

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

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

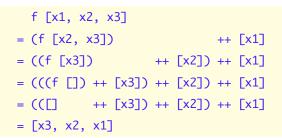


- foldr (:) [] is equivalent to the identity function on lists
- f = foldr (\x l -> l++[x]) []



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

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



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

Suresh

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

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

- 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

- 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

- 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
```

Sometimes it is useful to fold from the left

Sometimes it is useful to fold from the left

• Translate a string of digits to a number

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

Suresh

PRGH 2019: Lecture 11

• 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

Suresh

• Fold from the right using function f and initial value v

```
foldr :: (a \to b \to b) \to b \to [a] \to b
  foldr f \vee [7] = \vee
  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) ...)))
```

Suresh

PRGH 2019: Lecture 11

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

• 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) ...)))

• 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

• 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

• 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
```

PRGH 2019: Lecture 11

Г٦

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

foldl (+)	0	[1100]			
= foldl (+)	((+)	0 1)	[210	0]	
= foldl (+)	((+)	((+) 0 1) 2)	[310	0]
= foldl (+)	((+)	((+) ((+) 0 1)	2) 3)	[4100]
=					
= foldl (+)	((+)	(((+) ((+)	((+) 0	1) 2) 3)
) 100)					
= (+) (((+) ((+) ((+) 0 1) 2) 3)) 100					

• 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

• 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

• 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

• Defined in Data.List

- 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

- Defined in Data.List
- Eager version of foldl

• seq :: a -> b -> b

- Defined in Data.List
- Eager version of foldl

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

Suresh

- Defined in Data.List
- Eager version of foldl

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

PRGH 2019: Lecture 11

- Defined in Data.List
- Eager version of foldl

- 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

Suresh

PRGH 2019: Lecture 11

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' (+) 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 can be made to work 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 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 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 [])

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, fold1 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, fold1 step' g xs e = g (foldr f e xs)
- **Proof**: By induction on length xs

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

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

- 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]

- 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

- 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