# Programming in Haskell
# Aug-Nov 2015

## LECTURE 9

## SEPTEMBER 1, 2015

S P Suresh
Chennai Mathematical Institute

# 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**

# 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

# Computations with foldr

- `foldr f a [x1, x2,..., xn]`

  ⇛ `f x1 (foldr f a [x2,...,xn])`

  ⇛ `f x1 (f x2 (foldr f a [x3,...,xn]))`

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

  ⇛ `...`

  ⇛ `f x1 (f x2 (f x3 (...(f xn (foldr f a []))...)))`

  ⇛ `f x1 (f x2 (f x3 (... (f xn a)...)))`

# 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 a [x1, x2,..., xn]`

  ⇛ `f x1 (foldr f a [x2,...,xn])`

  ⇛ `...`

  ⇛ `f x1 (f x2 (f x3 (... (f xn a)...)))`

- 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 a [x1, x2,..., xn]`

  ⇛ `foldl f (f a x1) [x2,...,xn]`

  ⇛ `foldl f (f (f a x1) x2) [x3,...,xn]`

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

  ⇛ `...`

  ⇛ `foldl f (f ...(f (f (f a x1) x2) x3))... xn) []`

  ⇛ `f ...(f (f (f a 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 a [x1, x2,..., xn]`

  ⇛ `foldl f (f a x1) [x2,...,xn]`

  ⇛ `...`

  ⇛ `f ...(f (f (f a 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) 1 [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`
- `($) :: (a -> b) -> a -> b`
  `($) f x = f x`
- `($!) :: (a -> b) -> a -> b`
  `($!) f x = y `seq` f y`
  `where y = x`
  behaviour

– This is not the official definition
– Only conveys the intended