CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 16: Lazy Evaluation in Haskell

- Review: Lazy Evaluation and Simultaneous Let
- Lazy Evaluation and Pattern Matching
- Infinite Lists
- Infinite Trees

Programming with Infinite Lists

The power of infinite lists leads to some very interesting algorithms in Haskell, particularly when generating useful infinite series.

Prime Numbers:

```
Main> factors x = filter (\y -> x `mod` y == 0) [1..x]
Main> primes = [ x | x <- [1..], factors x == [1,x] ]
Main> take 20 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

Factorials:

```
Main> fact = map (\n -> product [1..n]) [1..]
Main> take 10 fact
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

Fibonacci Numbers:

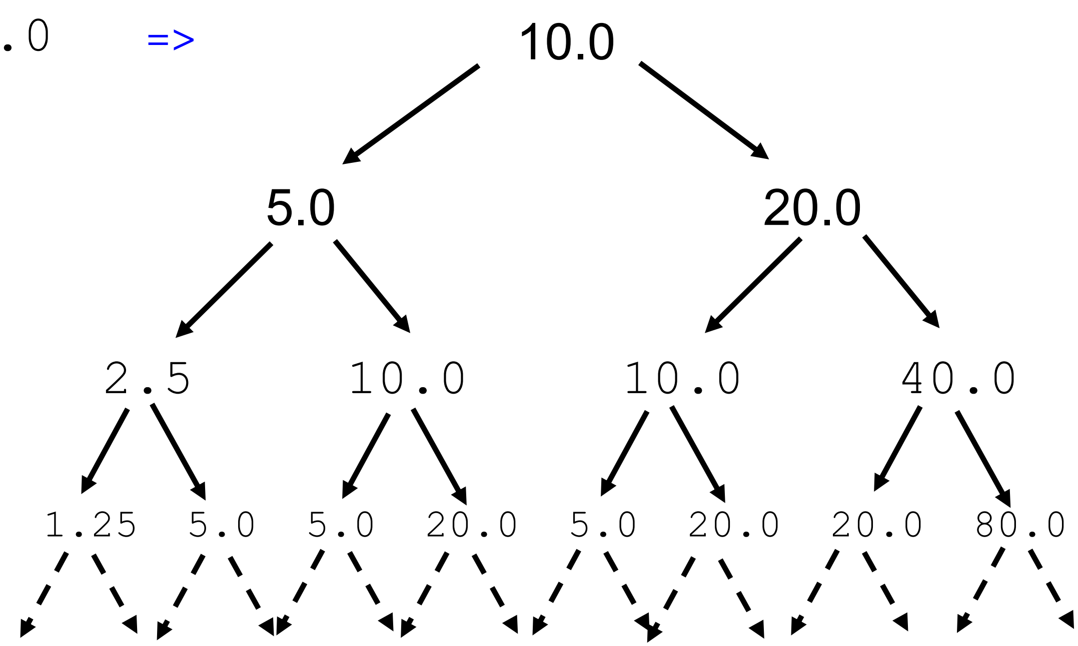
```
Main> fib = 1:1:[ x+y | (x,y) <- zip fib (tail fib) ]
Main> take 18 fib
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584]
```

Infinite Trees

You can create infinite data structures using constructors – just leave off the base case!

```
data Tree = Node Tree Double Tree deriving Show

tree :: Double -> Tree
tree x = Node (tree (x / 2)) x (tree (x * 2))
```

tree 10.0 =>



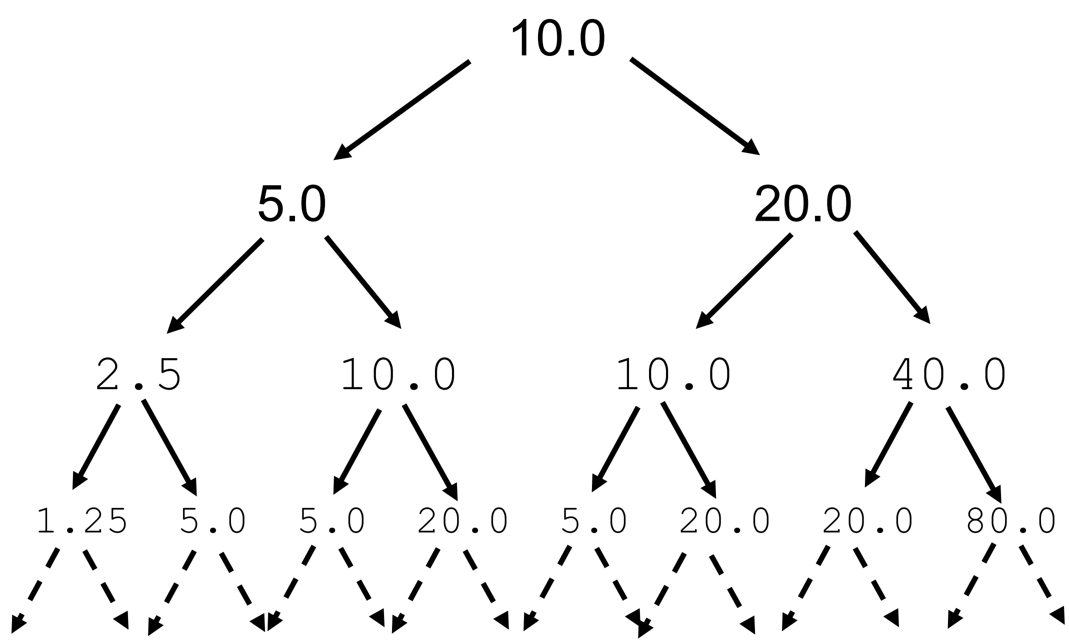
Infinite Trees

```haskell
data Tree = Node Tree Double Tree deriving Show

tree :: Double -> Tree
tree x = Node (tree (x / 2)) x (tree (x * 2))
```

```
Main> tree 10   -- NOOOOO, WAYNE, DON'T DO IT   AHHHHH!
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
(Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node (Node
```

Infinite Trees

```haskell
data Tree = Node Tree Double Tree deriving Show

tree :: Double -> Tree
tree x = Node (tree (x / 2)) x (tree (x * 2))

level :: Integer -> Tree -> [Double]
level 0 (Node _ x _) = [x]
level n (Node left x right) = (level (n-1) left) ++ (level (n-1) right)
```




```
Main> level 0 $ tree 10
[10.0]

Main> level 1 $ tree 10
[5.0,20.0]

Main> level 2 $ tree 10
[2.5,10.0,10.0,40.0]

Main> level 3 $ tree 10
[1.25,5.0,5.0,20.0,5.0,20.0,20.0,80.0]
```

Problems with Lazy Evaluation

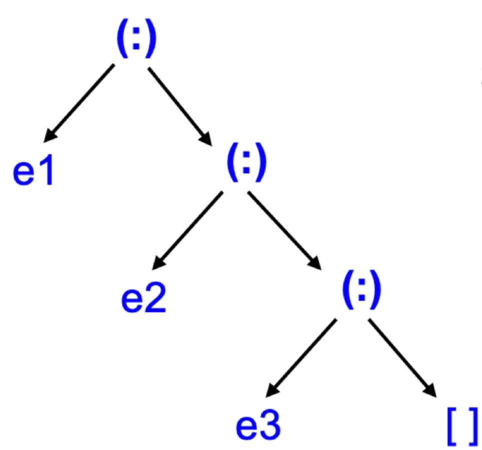
But lazy evaluation can be **very inefficient**, due to the space and time required to create thunks (stored, unevaluated, expressions)!

The classic example is the `foldl` function, which is analogous to `foldr`, but for left-associative functions. Recall how `foldr` works:

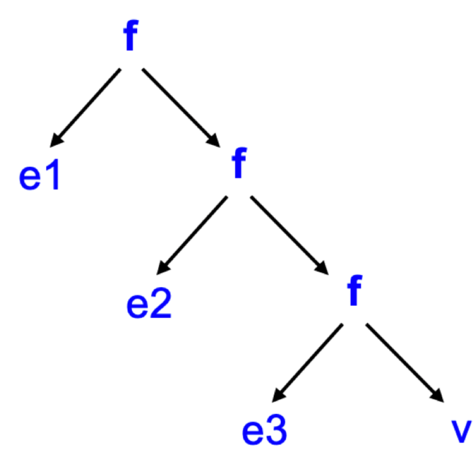
`foldr` right takes a list (constructed with the cons operator :) and effectively replaces a (prefix) cons with a function of two arguments, and the empty list with an "initial value" to get the recursion started:

```
[ e1, e2, e3 ]

e1 : ( e2 : e3 : [] )
```

```
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 [e1,e2,e3]`



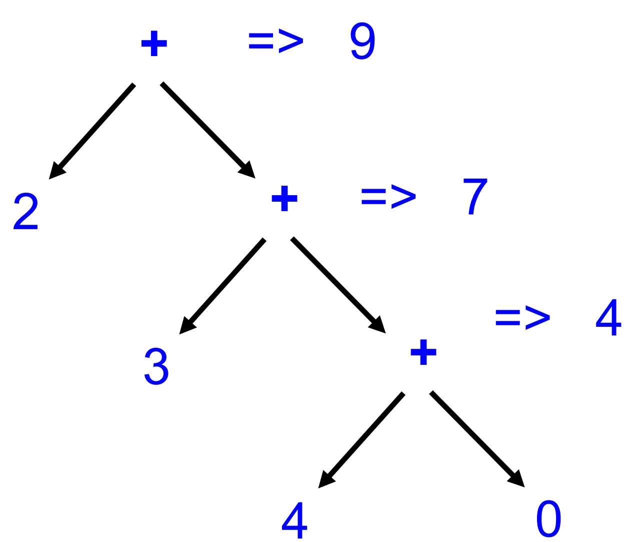
Problems with Lazy Evaluation

Essentially, `foldr` inserts an infix version of `f` between every member of the list, and ends with `v`:

$\texttt{foldr f v [}e_1, e_2, \ldots, e_n\texttt{]} = e_1 \;\texttt{`f`}\; e_2 \;\texttt{`f`}\; \ldots \;\texttt{`f`}\; e_n \;\texttt{`f`}\; v$

But the important point for now is that this makes the infix `` `f` `` right associative:

`foldr (+) 0 [2,3,4]   =>   9`



`( 2 + ( 3 + ( 4 + 0 ) ) )`

Problems with Lazy Evaluation

Foldr is a common function in Haskell, but it does not do well with Lazy Evaluation:

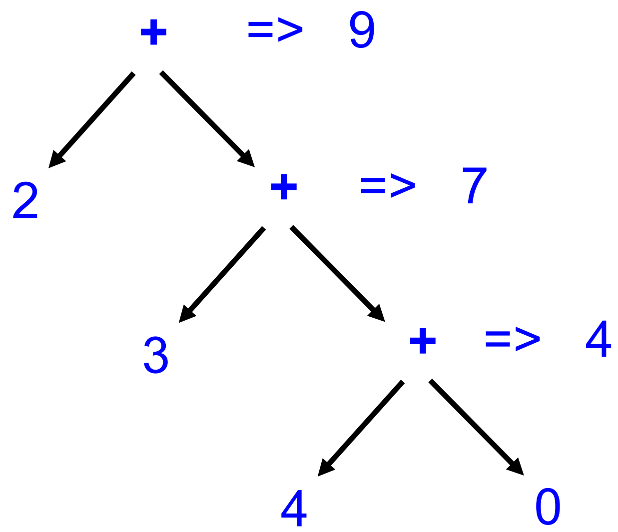
```
Prelude> :set +s          -- print out performance data

Prelude> foldr (+) 0 [1..1000000]      -- 10^6
500000500000
(0.78 secs, 161,594,000 bytes)

Prelude> foldr (+) 0 [1..10000000]     -- 10^7
50000005000000
(7.46 secs, 1,615,380,104 bytes)

Prelude> foldr (+) 0 [1..100000000]    -- 10^8
*** Exception: stack overflow
```

Q: Why is Haskell using so much space to just add a bunch of integers??

Problems with Lazy Evaluation

A: Lazy evaluation is creating thunks for each subexpression! Each one has to be stored!

```
foldr (+) 0 [2,3,4]    =>  9
```




```
( 2 + ( 3 + ( 4 + 0 ) ) )
```

You can't evaluate this until you get to the end! You have to go down the whole list, storing all the subexpressions, and then go back and add them all up!

Problems with Lazy Evaluation

A: Lazy evaluation is creating thunks for each subexpression! Each one has to be stored!

```
foldr (+) 0 veryBigList -->

foldr (+) 0 [1..1000000] -->
1 + (foldr (+) 0 [2..1000000]) -->
1 + (2 + (foldr (+) 0 [3..1000000])) -->
1 + (2 + (3 + (foldr (+) 0 [4..1000000]))) -->
1 + (2 + (3 + (4 + (foldr (+) 0 [5..1000000])))) -->
-- ...
-- ...  My stack overflows when there's a chain of around 500000 (+)'s !!!
-- ...  But the following would happen if you got a large enough stack:
-- ...
1 + (2 + (3 + (4 + (... + (999999 + (foldr (+) 0 [1000000]))...)))) -->
1 + (2 + (3 + (4 + (... + (999999 + (1000000 + ((foldr (+) 0 []))))...)))) -->

1 + (2 + (3 + (4 + (... + (999999 + (1000000 + 0))...)))) -->
1 + (2 + (3 + (4 + (... + (999999 + 1000000)...)))) -->
1 + (2 + (3 + (4 + (... + 1999999 ...)))) -->

1 + (2 + (3 + (4 + 500000499990))) -->
1 + (2 + (3 + 500000499994)) -->
1 + (2 + 500000499997) -->
1 + 500000499999 -->
500000500000
```

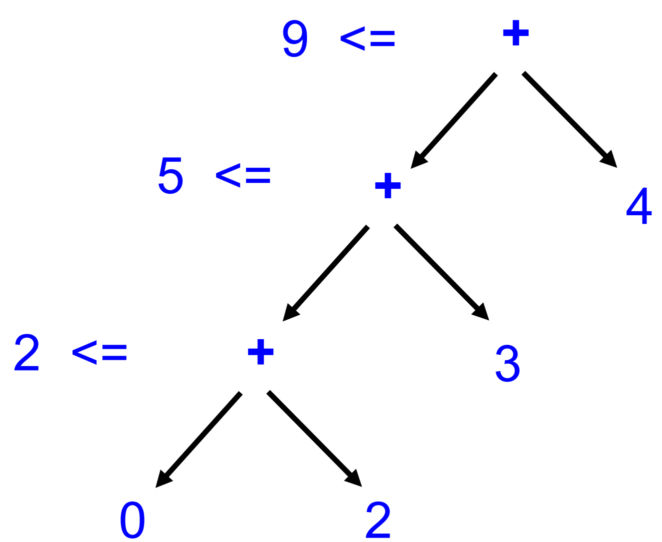
Problems with Lazy Evaluation

What's the solution? Well, for foldr, we can try to rearrange the computation so that we have something to evaluate at each step. The function which does this is `foldl` (fold left):

$$\texttt{foldl f v [}e_1,e_2,\ \ldots,\ e_n\texttt{]} \;=\; \texttt{v `f`}\ e_1\ \texttt{`f`}\ e_2\ \texttt{`f`}\ \ldots\ \texttt{`f`}\ e_n$$

but with **left associativity**, and where the initial value `v` is now on the left:

`foldl (+) 0 [2,3,4]   =>   9`



`( ( ( 0  + 2 ) + 3 ) + 4 )`

Now we can evaluate this from the front of the list:

0

0 + 2 = 2

2 + 3 = 5

5 + 4 = 9

Problems with Lazy Evaluation

Ok, let's try it!

```
Main> :set +s

Main> foldr (+) 0 [1..1000000]              - 10^6
500000500000
(0.75 secs, 346,961,056 bytes)

Main> foldl (+) 0 [1..1000000]              - 10^6
500000500000
(0.53 secs, 241,696,928 bytes)

Main> foldr (+) 0 [1..10000000]              - 10^7
 Exception: stack overflow

Main> foldl (+) 0 [1..10000000]              - 10^7
50000005000000
(19.34 secs, 2,416,474,776 bytes)
```

?? Foldl seems to be better, but not by much! ??

Problems with Lazy Evaluation

The problem is that foldl still uses lazy evaluation and still stores thunks without evaluating the subexpressions!

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

```
foldl (+) 0 veryBigList -->

foldl (+) 0 [1..1000000] -->

let z1 =  0 + 1
in foldl (+) z1 [2..1000000] -->

let z1 =  0 + 1
    z2 = z1 + 2
in foldl (+) z2 [3..1000000] -->

let z1 =  0 + 1
    z2 = z1 + 2
    z3 = z2 + 3
in foldl (+) z3 [4..1000000] -->

let z1 =  0 + 1
    z2 = z1 + 2
    z3 = z2 + 3
    z4 = z3 + 4
in foldl (+) z4 [5..1000000] -->
```

Problems with Lazy Evaluation

The solution is to force evaluation of one of the arguments:

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

```
foldl' (+) 0 veryBigList -->

foldl' (+) 0 [1..1000000] -->
foldl' (+) 1 [2..1000000] -->
foldl' (+) 3 [3..1000000] -->
foldl' (+) 6 [4..1000000] -->
foldl' (+) 10 [5..1000000] -->
-- ...
-- ... You see that the stack doesn't overflow
-- ...
foldl' (+) 499999500000 [1000000] -->
foldl' (+) 500000500000 [] -->
500000500000
```

The strict application operator

`$!`

forces a function to evaluate its argument immediately.

`foldl'` is defined in `Data.Foldable` this way.

Problems with Lazy Evaluation

By importing Data.Foldable, we can use this more efficient version of foldl. The same strategy can be used throughout Haskell to improve performance, but it is tricky!

```
Prelude> foldr (+) 0 [1..10000000]
50000005000000
(2.67 secs, 1,615,379,304 bytes)

Prelude> foldl (+) 0 [1..10000000]
50000005000000
(2.06 secs, 1,612,377,936 bytes)

Prelude> import Data.Foldable
(0.00 secs, 0 bytes)

Prelude Data.Foldable> foldl' (+) 0 [1..10000000]
50000005000000
(0.24 secs, 880,077,648 bytes)
```

Scope of Non-Local References (Free Variables)