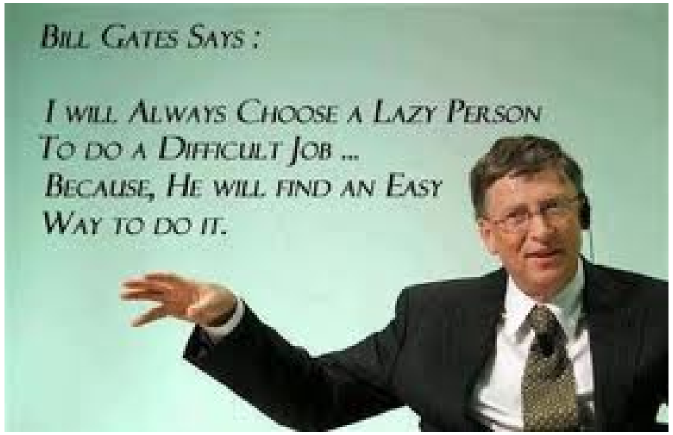
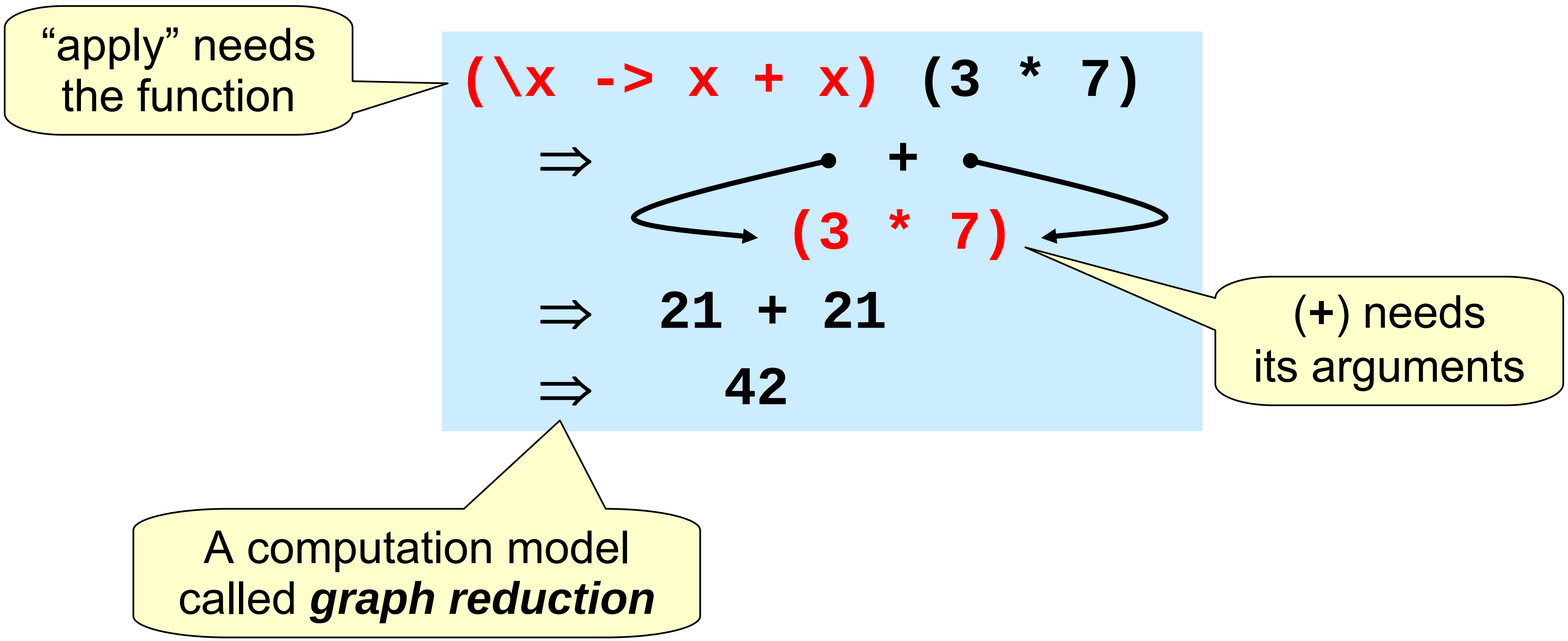
Haskell: From Basic to Advanced

Part 3 – A Deeper Look into Laziness



Laziness again

- Haskell is a *lazy* language
  - A particular function argument is only evaluated when it is *needed*, and
  - if it is needed then it is evaluated *just once*



When is a value “needed”?

```
strange :: Bool -> Integer
strange True  = 42
strange False = 42
```

```
Prelude> strange undefined
*** Exception: Prelude.undefined
```

An argument is evaluated when a pattern match occurs

But also primitive functions evaluate their arguments

use **`undefined`** or **`error`** to test if an argument is evaluated

Lazy programming style

- Clear separation between
  - Where the computation of a value is defined
  - Where the computation of a value happens

We naturally get modularity!

At most once?

```
fib n = head (drop n fibs)
```

```
foo :: Integer -> Integer
foo n = (fib n)^2 + fib n + 42
```

```
Prelude> foo (6 * 7)
717780702690899954
```

6 * 7 is evaluated once but **fib 42** is evaluated twice

```
bar :: Integer -> Integer
bar n = foo 42 + n
```

```
Prelude> bar 17 + bar 54
1435561405381799979
```

foo 42 is evaluated twice

Quiz: How to avoid such recomputation?

At most once!

```haskell
foo :: Integer -> Integer
foo x = t^2 + t + 42
        where t = fib x
```

```haskell
bar :: Integer -> Integer
bar x = foo42 + x

foo42 :: Integer
foo42 = foo 42
```

The compiler might also perform these optimizations with

ghc -O

ghc –ffull-laziness

Lazy iteration

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

```
Prelude> take 13 (iterate (*2) 1)
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096]
```

```
repeat :: a -> [a]
repeat x = x : repeat x

cycle :: [a] -> [a]
cycle xs = xs ++ cycle xs
```

Define these with **`iterate`**?

```
repeat :: a -> [a]
repeat x = iterate id x

cycle :: [a] -> [a]
cycle xs = concat (repeat xs)
```

Lazy replication and grouping

```
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

```
Prelude> replicate 13 42
[42,42,42,42,42,42,42,42,42,42,42,42,42]
```

```
group :: Int -> [a] -> [[a]]
group n =
          takeWhile (not . null)
        . map (take n)
        . iterate (drop n)
```

How to define this?

. connects stages like Unix pipe symbol **|**

```
Prelude> group 3 "abracadabra!"
["abr","aca","dab","ra!"]
```

Lazy IO

- Even IO is done lazily!

```
headFile f = do
  c <- readFile f
  let c' = unlines . take 5 . lines $ c
  putStrLn c'
```

Does not actually read in the whole file!

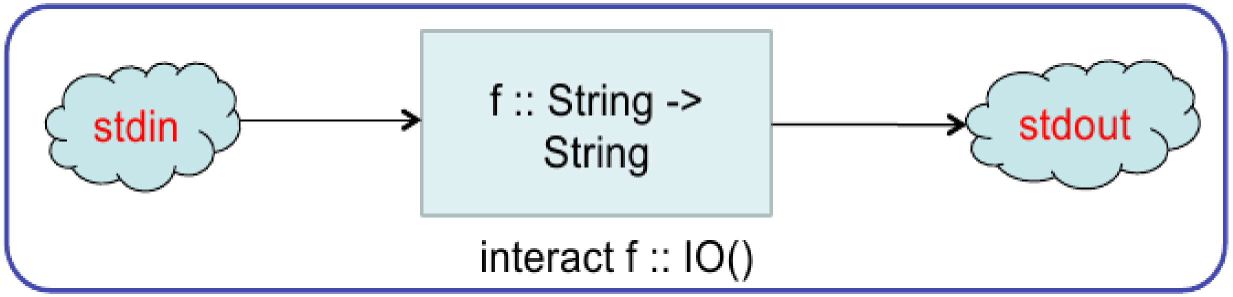
Need to print causes just 5 lines to be read

Aside: we can use names with **'** at their end (read: "prime")

Lazy IO

Common pattern: take a function from String to String, connect **stdin** to the input and **stdout** to the output

```
interact :: (String -> String) -> IO ()
```




```
import Network.HTTP.Base (urlEncode)

encodeLines = interact $
  unlines . map urlEncode . lines
```

```
Prelude> encodeLines
hello world
hello%20world
20+22=42
20%2B22%3D42
...
```

Other IO variants

- String is a list of Char:
  - each element is allocated individually in a cons cell
  - IO using String has quite poor performance

- Data.ByteString provides an alternative non-lazy array-like representation ByteString
- Data.ByteString.Lazy provides a hybrid version which works like a list of max 64KB chunks

Controlling laziness

- Haskell includes some features to reduce the amount of laziness, allowing us to decide *when* something gets evaluated
- These features can be used for performance tuning, particularly for controlling space usage
- Not recommended to mess with them unless you have to – hard to get right in general!

Tail recursion

- A function is tail recursive if its last action is a recursive call to itself and that call produces the function's result
- Tail recursion uses no stack space; a tail recursive call can be compiled to an unconditional jump
- Important concept in non-lazy functional programming

- Recall **`foldr`**

```
foldr op init [] = init
foldr op init (x:xs) = x `op` foldr op init xs
```

foldr op init [x1,x2,...,x42] ⇒ (x1 \`op\` (x2 \`op\` ... (x42 \`op\` init) ...

- The tail recursive "relative" of **`foldr`** is **`foldl`**

```
foldl op init [] = init
foldl op init (x:xs) = foldl op (init `op` x) xs
```

foldl op init [x1,x2,...,x42] ⇒ (...(init \`op\` x1) \`op\` x2) ... \`op\` x42

Tail recursion and laziness

- Recall **sum**

```
sum = foldr (+) 0
```

```
*Main> let big = 42424242 in sum [1..big]
*** Exception: stack overflow
*Main> let big = 42424242 in foldr (+) 0 [1..big]
*** Exception: stack overflow
```

- OK, we were expecting these, but how about **foldl**?

```
*Main> let big = 42424242 in foldl (+) 0 [1..big]
*** Exception: stack overflow
```

- What's happening!?
- Lazy evaluation is too lazy!

```
foldl (+) 0 [1..big]
⇒  foldl (+) (0+1) [2..big]
⇒  foldl (+) (0+1+2) [3..big]
⇒  ...
```

Not computed until needed; at the 42424242 recursive call!

Controlling laziness using **seq**

- Haskell includes a primitive function

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

- It evaluates its first argument and returns the second

"strict" is used to mean the opposite of "lazy"

The **`Prelude`** also defines a strict application operation

```
($!) :: (a -> b) -> a -> b
f $! x = x `seq` (f x)
```

Strictness

- A tail recursive lists sum function

```
sum :: [Integer] -> Integer
sum = s 0
  where s acc []     = acc
        s acc (x:xs) = s (acc+x) xs
```

- When compiling with **ghc -O** the compiler looks for arguments which will eventually be needed and will insert `seq` calls in appropriate places

```
sum’ :: [Integer] -> Integer
sum’ = s 0
  where s acc [] = acc
        s acc (x:xs) = acc `seq` s (acc+x) xs
```

force **acc** to be simplified on each recursive call

Strict tail recursion with `foldl'`

```haskell
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' op init [] = init
foldl' op init (x:xs) = let a = (init `op` x)
                        in  a `seq` foldl' op a xs
```

And now

```
*Main> let big = 42424242 in foldl' (+) 0 [1..big]
899908175849403
```

Or even better, we can use the built-in one

```
*Main> import Data.List (foldl')
*Main> let big = 42424242 in foldl' (+) 0 [1..big]
899908175849403
```

Are we there yet?

- One more example: average of a list of integers

```
average :: [Integer] -> Integer
average xs = sum' xs `div` fromIntegral (length xs)
```

> needed due to the types of **sum'** and **length**

```
*Sum> let big = 42424242 in length [1..big]
42424242
*Sum> let big = 42424242 in sum' [1..big]
899908175849403
*Sum> let big = 42424242 in average [1..big]
21212121
```

- Seems to work, doesn't it? Let's see:

```
*Sum> let bigger = 424242420 in length [1..bigger]
424242420
*Sum> let bigger = 424242420 in sum' [1..bigger]
89990815675849410
*Sum> let bigger = 424242420 in average [1..bigger]
... CRASHES THE MACHINE DUE TO THRASHING!
```

WTF?

Space leaks

- Making **`sum`** and **`length`** tail recursive and strict does not solve the problem ☹

- This problem is often called a space leak
  - **`sum`** forces us to build the whole **`[1..bigger]`** list
  - laziness ("at most once") requires us to keep the list in memory since it is going to be used by **`length`**
  - when we compute either the length or the sum, as we go along, the part of the list that we have traversed so far is reclaimed by the garbage collector

Fixing the space leak

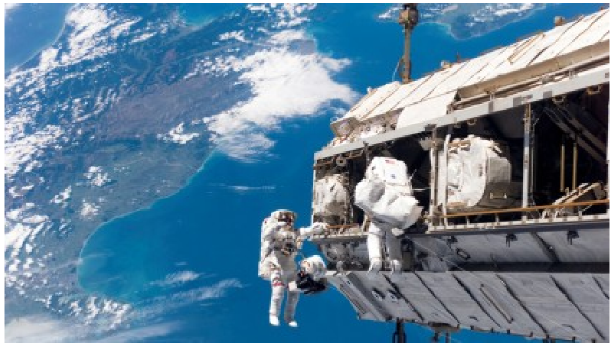
- This particular problem can be solved by making average tail recursive by computing the list sum and length at the same time

call to **`fromIntegral`** not needed anymore

```
average' :: [Integer] -> Integer
average' xs = av 0 0 xs where
  av sm len []     = sm `div` len
  av sm len (x:xs) = sm  `seq`
                     len `seq`
                     av (sm + x) (len + 1) xs
```

```
*Sum> let bigger = 424242420 in average [1..bigger]
212121210
```

fixing a space leak



Gotcha: **seq** is still quite lazy!

- **seq** forces evaluation of its first argument, but *only as far as the outermost constructor!*

evaluation to weak head-normal form

```
Prelude> undefined `seq` 42
*** Exception: Prelude.undefined
Prelude> (undefined,17) `seq` 42
42
```

the pair is already "evaluated", so a **seq** here would have no effect

```
sumlength = foldl' f (0,0)
  where f (s,l) a = (s+a,l+1)
```

```
sumlength = foldl' f (0,0)
  where f (s,l) a = let (s',l') = (s+a,l+1)
                    in  s' `seq` l' `seq` (s',l')
```

force the evaluation of components *before* the pair is constructed

Laziness and IO

```
count :: FilePath -> IO Int
count f = do contents <- readFile f
             let n = read contents
             writeFile f (show (n+1))
             return n
```

for the time being this will do

readFile is not computed until it is needed

```
Prelude> count "some_file"
*** Exception: some_file: openFile: resource busy (file is locked)
```

- We sometimes need to control lazy IO
  - Here the problem is easy to fix (see below)
  - Some other times, we need to work at the level of file handles

```
count :: (Num b,Show b,Read b) => FilePath -> IO b
count f = do contents <- readFile f
             let n = read contents
             n `seq` writeFile f (show (n+1))
             return n
```

Some lazy remarks

- Laziness
  - Evaluation happens on demand and “at most once”
  - \+ Can make programs more “modular”
  - \+ Very powerful tool when used right
  - − Different programming style / approach
- We do not have to employ it everywhere!
- Some performance implications are very tricky
  - Evaluation can be controlled by tail recursion and seq
  - Best avoid their use when not really necessary