Part 3 – A Deeper Look into Laziness
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

\[
\begin{align*}
(\lambda x \to x + x) (3 \times 7) \\
\Rightarrow 21 + 21 \\
\Rightarrow 42
\end{align*}
\]
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?

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

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

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

Prelude> foo (6 * 7)
71778070269089954

Prelude> bar 17 + bar 54
143556140538179979
```

Quiz: How to avoid such recomputation?
foo :: Integer -> Integer
foo x = t^2 + t + 42
    where t = fib x

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

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)

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

• Even IO is done lazily!

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

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

- The tail recursive “relative” of `foldr` is `foldl`

  foldl op init [] = init
  foldl op init (x:xs) = foldl op (init `op` x) xs
Tail recursion and laziness

- Recall `sum`  
  \[
  \text{sum} = \text{foldr} \ (+) \ 0
  \]

  *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!

  \[
  \text{foldl} \ (+) \ 0 \ [1..\text{big}]
  \Rightarrow \text{foldl} \ (+) \ (0+1) \ [2..\text{big}]
  \Rightarrow \text{foldl} \ (+) \ (0+1+2) \ [3..\text{big}]
  \Rightarrow \ldots
  \]

  Not computed until needed; at the 42424242 recursive call!
Controlling laziness using \texttt{seq}

- Haskell includes a primitive function

\[
\texttt{seq} :: a \rightarrow b \rightarrow b
\]

- It evaluates its first argument and returns the second

The \texttt{Prelude} also defines a strict application operation

\[
\texttt{($)!} :: (a \rightarrow b) \rightarrow a \rightarrow b
\]
\[
f \texttt{($)!} \; x = x \; \texttt{`seq`} \; (f \; x)
\]

“\textit{strict}” is used to mean the opposite of “lazy”
Strictness

• A tail recursive lists sum function

```haskell
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

```haskell
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 \texttt{foldl’}

\texttt{foldl’} :: (a \to b \to a) \to a \to [b] \to a
\texttt{foldl’} \texttt{op init} [] = \texttt{init}
\texttt{foldl’ op init} (x:xs) = \texttt{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

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

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

```haskell
*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
```

- CRASHES THE MACHINE DUE TO THRASHING!

WTF?

needed due to the types of `sum'` and `length`
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.

```haskell
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
```

* Sum: call to fromIntegral not needed anymore fixing a space leak
Gotcha: \texttt{seq} is still quite lazy!

- \texttt{seq} forces evaluation of its first argument, but only as far as the outermost constructor!

Prelude> \texttt{undefined \ `seq` \ 42}
*** Exception: Prelude.undefined
Prelude> (\texttt{undefined,17} \ `seq` \ 42)
42

\texttt{sumlength = foldl' \ f \ (0,0)}
\texttt{where \ f \ (s,l) \ a = (s+a,l+1)}

\texttt{sumlength = foldl' \ f \ (0,0)}
\texttt{where \ f \ (s,l) \ a = let \ (s',l') = (s+a,l+1)}
\texttt{\textit{in} \ s' \ `seq` \ l' \ `seq` \ (s',l')}

force the evaluation of components \textit{before} the pair is constructed.

evaluation to weak head-normal form.

the pair is already “evaluated”, so a \texttt{seq} here would have no effect.

Laziness and IO

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

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

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

```haskell
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