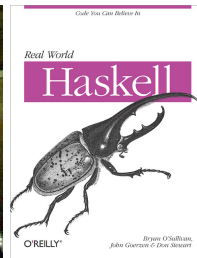
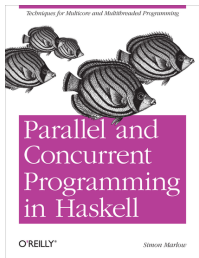


Lazy Evaluation and Seq

Stephen A. Edwards

Columbia University

Fall 2023



Laziness in Haskell

Haskell follows a *call-by-need*[†] evaluation strategy in which expressions are evaluated **only when their values are needed** and **at most once**.

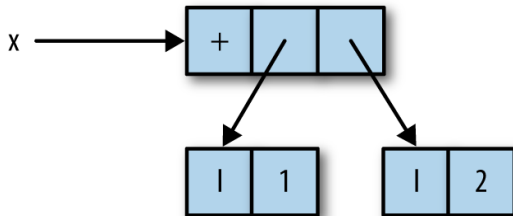
```
ghci> let x = 1 + 2 :: Int
ghci> :t x
x :: Int
ghci> :sprint x
x = _
ghci> x + 1
4
ghci> :sprint x
x = 3
```

_ denotes an unevaluated **"thunk"**

[†]C, Java, etc. are *call-by-value*: arguments are evaluated before a function call;
Algol-68 is *call-by-name*: arguments are (re)evaluated at each reference



Thunk Crood



[Marlow, Figure 2-1]

Thunks all the way down: seq forces evaluation

seq :: a -> b -> b

seq x y = evaluate x and y; return y

```
ghci> let x = 1 + 2 :: Int
```

```
ghci> let y = x + 1
```

```
ghci> :sprint x
```

```
x = _
```

```
ghci> :sprint y
```

```
y = _
```

```
ghci> seq y ()
```

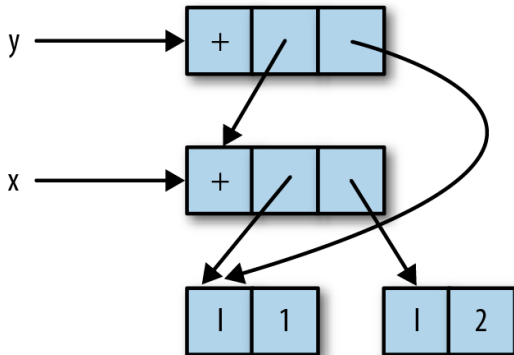
```
()
```

```
ghci> :sprint x
```

```
x = 3
```

```
ghci> :sprint y
```

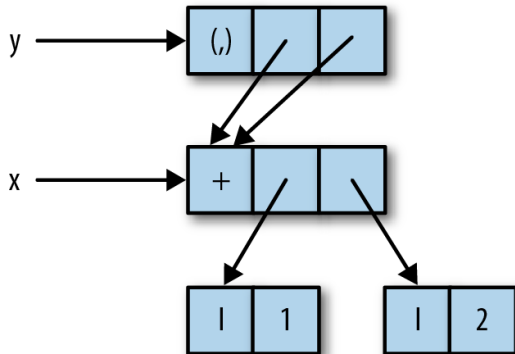
```
y = 4
```



[Marlow, Figure 2-2]

Weak Head Normal Form: Lazy Data Structures

```
ghci> let x = 1 + 2 :: Int
ghci> let y = (x, x)
ghci> let swap(a, b) = (b, a)
ghci> let z = swap (x,x+1)
ghci> :sprint z
z = _
ghci> seq z ()
()
ghci> :sprint z
z = (_,_)
ghci> seq x ()
()
ghci> :sprint z
z = (_,3)
```



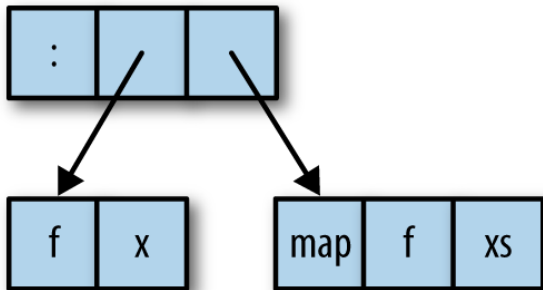
[Marlow, Figure 2-3]

Weak head normal form: top is data constructor or lambda, not application

Functions Build Thunks

```
ghci> let xs =  
      map (+1) [1..10] :: [Int]  
ghci> :sprint xs  
xs = _  
ghci> seq xs ()  
( )  
ghci> :sprint xs  
xs = _ : _  
ghci> seq (tail xs) ()  
( )  
ghci> :sprint xs  
xs = _ : _ : _  
ghci> length xs  
10  
ghci> :sprint xs  
xs = [_,-,-,-,-,-,-,-,-,-,-]
```

```
map :: (a -> b) -> [a] -> [b]  
map f []      = []  
map f (x:xs) = let x' = f x  
                xs' = map f xs  
                in x' : xs'
```



foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr _ z []      = z  
foldr f z (x:xs) = x `f` foldr f z xs
```

```
ghci> s = foldr (:) [] "ABC"  
ghci> :sprint s  
s = _  
ghci> seq s ()  
ghci> :sp s  
s = 'A' : _  
ghci> seq (tail s) ()  
ghci> :sp s  
s = 'A' : 'B' : _  
ghci> seq (tail (tail s)) ()  
ghci> :sp s  
s = 'A' : 'B' : 'C' : _
```

```
foldr (:) [] "ABC"
```

```
'A' : foldr (:) [] "BC"
```

```
'A' : 'B' : foldr (:) [] "C"
```

```
'A' : 'B' : 'C' : foldr (:) [] ""
```

```
'A' : 'B' : 'C' : []
```

Perfect: lazy list built from the left; thunk the same size each time

foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr _ z []      = z  
foldr f z (x:xs) = x `f` foldr f z xs
```

```
ghci> t = foldr (+) 0 [1,2,3] :: Int  
ghci> :sp t  
t = _  
ghci> seq t ()  
()  
ghci> :sp tr  
t = 6
```

```
foldr (+) 0 [1,2,3]  
1 + foldr (+) 0 [2,3]  
1 + (2 + foldr (+) 0 [3])  
1 + (2 + (3 + foldr (+) 0 []))  
1 + (2 + (3 + 0))  
1 + (2 + 3)  
1 + 5  
6
```

Yikes: unevaluated thunk grows: $1 + _$, $1 + (2 + _)$, $1 + (2 + (3 + _))$

foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b  
foldl _ z []      = z  
foldl f z (x:xs) = foldl f (z `f` x) xs
```

```
ghci> s = foldl (flip (:)) [] "ABC"  
ghci> :sprint s  
s = _  
ghci> seq s ()  
ghci> :sp s  
s = 'C' : _  
ghci> seq (tail s) ()  
ghci> :sp s  
s = 'C' : 'B' : _  
ghci> seq (tail (tail s)) ()  
ghci> :sp s  
s = "CBA"
```

```
foldl (flip (:)) [] "ABC"  
foldl (flip (:)) ('A' : []) "BC"  
foldl (flip (:)) ('B' : ('A' : [])) "C"  
foldl (flip (:)) ('C' : ('B' : ('A' : []))) ""  
('C' : ('B' : ('A' : [])))
```

Lazy result from huge unevaluated thunk

foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b  
foldl _ z []      = z  
foldl f z (x:xs) = foldl f (z `f` x) xs
```

```
ghci> t = foldl (+) 0 [1,2,3] :: Int  
ghci> :sp t  
t = _  
ghci> seq t ()  
()  
ghci> :sp t  
t = 6
```

```
foldl (+) 0 [1,2,3]  
foldl (+) (0 + 1) [2,3]  
foldl (+) ((0 + 1) + 2) [3]  
foldl (+) (((0 + 1) + 2) + 3) []  
(((0 + 1) + 2) + 3)  
((1 + 2) + 3)  
(3 + 3)  
6
```

Unevaluated thunk growing again
because function application is lazy
and it's trying to reduce to WHNF

foldl'

```
foldl' :: (b -> a -> b) -> b -> [a] -> b  
foldl' _ z []      = z  
foldl' f z (x:xs) = let z' = z `f` x in z' `seq` foldl' f z' xs
```

```
foldl' (+) 0 [1,2,3]  
let z' = 0 + 1 in z' `seq` foldl' (+) z' [2,3]  
foldl' (+) 1 [2,3]  
let z' = 1 + 2 in z' `seq` foldl' (+) z' [3]  
foldl' (+) 3 [3]  
let z' = 3 + 3 in z' `seq` foldl' (+) z' []  
foldl' (+) 6 []  
6
```

OK: no growing unevaluated thunk, but result is not lazy.
This is what `foldl'` is intended for

foldl'

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
```

```
foldl' _ z [] = z
```

```
foldl' f z (x:xs) = let z' = z `f` x in z' `seq` foldl' f z' xs
```

```
ghci> s = foldl' (flip (:)) [] "ABC"
```

```
ghci> :sprint s
```

```
s = _
```

```
ghci> seq s ()
```

```
ghci> :sp s
```

```
s = "CBA"
```

Interesting: reducing s to WHNF gave us the whole list, not just the first element. Why?

foldl'

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
```

```
foldl' _ z []      = z
```

```
foldl' f z (x:xs) = let z' = z `f` x in z' `seq` foldl' f z' xs
```

```
foldl' (flip (:)) [] "ABC"
```

```
let z' = flip (:) [] 'A' in z' `seq` foldl' (flip (:)) z' "BC"
```

```
foldl' (flip (:)) "A" "BC"
```

```
let z' = flip (:) "A" 'B' in z' `seq` foldl' (flip (:)) z' "C"
```

```
foldl' (flip (:)) "BA" "C"
```

```
let z' = flip (:) "BA" 'C' in z' `seq` foldl' (flip (:)) z' []
```

```
foldl' (flip (:)) "CBA" []
```

```
"CBA"
```

OK: no growing unevaluated thunk, but result is not lazy

Optimizing Laziness and Space Leaks

Artificial example program to optimize:

Calculate the average value of a list of Doubles [1,2,3,...] by summing their values and dividing by the length

```
import System.Environment (getArgs)
import Text.Printf (printf)

main :: IO ()
main = do
    [d] <- map read <$> getArgs
    printf "%f\n" $ mean [1..d]

mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)
```

```
$ stack ghc -- --make -O2 -Wall prof1.hs
[1 of 2] Compiling Main                ( prof1.hs, prof1.o )
[2 of 2] Linking prof1
$ /usr/bin/time -f "elapsed %e s" ./prof1 1e5
50000.5
elapsed 0.03 s
$ /usr/bin/time -f "elapsed %e s" ./prof1 1e6
500000.5
elapsed 0.12 s
$ /usr/bin/time -f "elapsed %e s" ./prof1 1e7
5000000.5
elapsed 1.19 s
$ /usr/bin/time -f "elapsed %e s" ./prof1 1e8
50000000.5
elapsed 10.24 s
$ /usr/bin/time -f "elapsed %e s" ./prof1 1e9
Command terminated by signal 9
elapsed 200.01 s
```

Scales linearly, but
this exhausted
memory on a
64 GB machine

What is actually
going on?

How do we make
this work in
bounded memory?

```
$ ./prof1 1e8 +RTS -s
```

```
50000000.5
```

```
12,800,142,880 bytes allocated in the heap
```

```
10,020,680,440 bytes copied during GC
```

```
2,517,947,400 bytes maximum residency (13 sample(s))
```

```
431,184,888 bytes maximum slop
```

```
5718 MiB total memory in use (0 MB lost due to fragmentation)
```

				Tot time (elapsed)	Avg pause	Max pause
Gen 0	3039 colls,	0 par	2.278s	2.286s	0.0008s	0.0038s
Gen 1	13 colls,	0 par	5.422s	5.431s	0.4178s	2.1034s

```
INIT    time    0.001s ( 0.001s elapsed)
```

```
MUT     time    2.068s ( 2.059s elapsed)
```

```
GC      time    7.699s ( 7.717s elapsed)
```

```
EXIT    time    0.000s ( 0.003s elapsed)
```

```
Total   time    9.768s ( 9.780s elapsed)
```

```
%GC      time      0.0% (0.0% elapsed)
```

```
Alloc rate    6,190,156,893 bytes per MUT second
```

```
Productivity  21.2% of total user, 21.1% of total elapsed
```

MUT = "mutator" = useful work

GC = "garbage collection"

**Spending about 79% of its time
doing garbage collection**

%GC time is wrong


```
$ stack ghc -- --make -O2 prof1.hs \  
-rtsopts \           # Enable more +RTS options  
-prof \             # Enable profiling  
-fprof-auto          # Profile every top-level function  
[1 of 2] Compiling Main          ( prof1.hs, prof1.o )  
[2 of 2] Linking prof1 [Objects changed]
```

```
$ /usr/bin/time -f "elapsed %e s" ./prof1 1e6 +RTS -p  
500000.5  
elapsed 0.16 s      # vs. 0.12 s without profiling
```

Generates prof1.prof file because of -p option

prof1 +RTS -p -RTS 1e6

total time = 0.06 secs (62 ticks @ 1000 us, 1 processor)

total alloc = 128,107,088 bytes (excludes profiling overheads)

COST CENTRE MODULE SRC %time %alloc

main Main prof1.hs:(5,1)-(7,29) 77.4 99.9

mean Main prof1.hs:10:1-43 22.6 0.0

COST CENTRE MODULE			SRC	no. entries	individual		inherited	
					%time	%alloc	%time	%alloc
MAIN	MAIN		<built-in>	129	0	0.0	0.0	100.0
CAF	Main		<entire-module>	257	0	0.0	0.0	0.0
main	Main		prof1.hs:(5,1)-(7,29)	258	1	0.0	0.0	0.0
CAF	GHC.Conc.Signal		<entire-module>	250	0	0.0	0.0	0.0
CAF	GHC.Float		<entire-module>	244	0	0.0	0.0	0.0
CAF	GHC.IO.Encoding		<entire-module>	237	0	0.0	0.0	0.0
CAF	GHC.IO.Encoding.Iconv		<entire-module>	235	0	0.0	0.0	0.0
CAF	GHC.IO.Handle.FD		<entire-module>	227	0	0.0	0.0	0.0
CAF	GHC.Read		<entire-module>	212	0	0.0	0.0	0.0
CAF	Text.Printf		<entire-module>	193	0	0.0	0.0	0.0
main	Main		prof1.hs:(5,1)-(7,29)	259	0	77.4	99.9	100.0
mean	Main		prof1.hs:10:1-43	260	1	22.6	0.0	22.6

```

main :: IO ()
main = do
    [d] <- map read <$> getArgs
    printf "%f\n" $ mean [1..d]

mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)

```

COST	CENTRE	MODULE	SRC	%time	%alloc
main	Main	prof1.hs:(5,1)-(7,29)		79.4	99.9
mean	Main	prof1.hs:10:1-43		20.6	0.0

Most allocation is for the list of Doubles (in main)

Only 20% of the time is spent traversing the list

-fprof-auto gives insufficient precision; need to define smaller *cost centers*

SCC : "Set Cost Center" annotation in a `{-# SCC id #-}` comment.
id may be a Haskell identifier or a quoted string

```
import System.Environment (getArgs)
import Text.Printf (printf)
main :: IO ()
main = do [d] <- map read <$> getArgs
         printf "%f\n" $ mean ({-# SCC list #-} [1..d])

mean :: [Double] -> Double
mean xs =
    ({-# SCC sum      #-} sum      xs) /
    fromIntegral ({-# SCC length #-} length xs)
```

```
$ stack ghc -- --make -O2 prof2.hs -prof -fprof-auto
[1 of 2] Compiling Main                ( prof2.hs, prof2.o )
[2 of 2] Linking prof2
$ /usr/bin/time -f "elapsed %e s" ./prof2 1e6 +RTS -p
500000.5
elapsed 0.19 s
```

```
main = do [d] <- map read <$> getArgs
          printf "%f\n" $ mean ({-# SCC list #-} [1..d])
mean xs =      ({-# SCC sum      #-} sum      xs) /
               fromIntegral ({-# SCC length #-} length xs)
```

Thu Nov 2 17:24 2023 Time and Allocation Profiling Report (Final)

prof2 +RTS -p -RTS 1e6

total time = 0.06 secs (59 ticks @ 1000 us, 1 processor)

total alloc = 128,107,104 bytes (excludes profiling overheads)

COST CENTRE	MODULE	SRC	%time	%alloc
list	Main	prof2.hs:6:50-55	72.9	99.9
sum	Main	prof2.hs:9:44-52	13.6	0.0
length	Main	prof2.hs:10:44-52	13.6	0.0

COST CENTRE	MODULE	SRC	no.	entries	individual %time %alloc	inherited %time %alloc
MAIN	MAIN	<built-in>	129	0	0.0 0.0	100.0 100.0
CAF	Main	<entire-module>	257	0	0.0 0.0	0.0 0.0
main	Main	prof2.hs:(5,1)-(6,56)	258	1	0.0 0.0	0.0 0.0
CAF	GHC.Conc.Signal	<entire-module>	250	0	0.0 0.0	0.0 0.0
CAF	GHC.Float	<entire-module>	244	0	0.0 0.0	0.0 0.0
CAF	GHC.IO.Encoding	<entire-module>	237	0	0.0 0.0	0.0 0.0
CAF	GHC.IO.Encoding.Iconv	<entire-module>	235	0	0.0 0.0	0.0 0.0
CAF	GHC.IO.Handle.FD	<entire-module>	227	0	0.0 0.0	0.0 0.0
CAF	GHC.Read	<entire-module>	212	0	0.0 0.0	0.0 0.0
CAF	Text.Printf	<entire-module>	193	0	0.0 0.0	0.0 0.0
main	Main	prof2.hs:(5,1)-(6,56)	259	0	0.0 0.0	100.0 99.9
list	Main	prof2.hs:6:50-55	260	1	72.9 99.9	72.9 99.9
mean	Main	prof2.hs:(9,1)-(10,53)	261	1	0.0 0.0	27.1 0.0
length	Main	prof2.hs:10:44-52	263	1	13.6 0.0	13.6 0.0
sum	Main	prof2.hs:9:44-52	262	1	13.6 0.0	13.6 0.0

```

main = do [d] <- map read <$> getArgs
          printf "%f\n" $ mean ({-# SCC list #-} [1..d])
mean xs =      ({-# SCC sum      #-} sum      xs) /
              fromIntegral ({-# SCC length #-} length xs)

```

```

Thu Nov  2 17:24 2023 Time and Allocation Profiling Report  (Final)
  prof2 +RTS -p -RTS 1e6
total time   =          0.06 secs   (59 ticks @ 1000 us, 1 processor)
total alloc  = 128,107,104 bytes   (excludes profiling overheads)

```

COST CENTRE	MODULE	SRC	%time	%alloc
list	Main	prof2.hs:6:50-55	72.9	99.9
sum	Main	prof2.hs:9:44-52	13.6	0.0
length	Main	prof2.hs:10:44-52	13.6	0.0

The list is virtually all of the memory allocated

Sum and length are both taking a fair amount of time (to traverse the list)

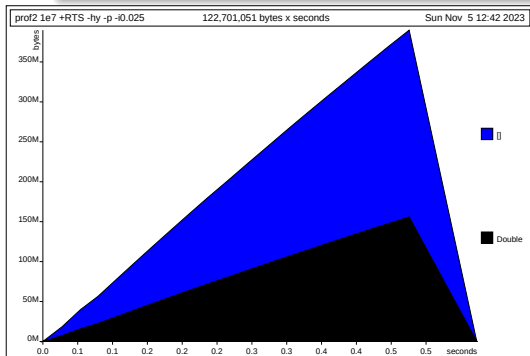
```
$ stack ghc -- --make -O2 prof2.hs -prof -fprof-auto -rtsopts  
[1 of 2] Compiling Main ( prof2.hs, prof2.o )
```

```
[2 of 2] Linking prof2
```

```
$ /usr/bin/time -f "elapsed %e s" ./prof2 1e7 +RTS -hy -p -i0.025  
5000000.5
```

```
elapsed 3.22 s
```

```
$ hp2ps prof2.hp # Creates prof2.ps
```



-hy Profile heap by type into prof2.hp

-i0.025 Sample heap size every 0.025 s

1e7 element list:

160e6 bytes of Doubles (16 bytes ea.);

240e6 bytes of Cons cells (24 bytes ea.)

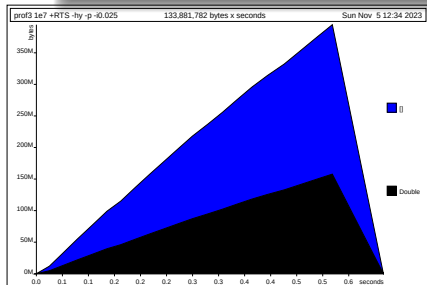
List is being created slowly (lazily) then deallocated as it's being traversed again

Laziness doesn't reduce memory because the list needs to be traversed *twice*.

Idea: make two equivalent lists

```
main = do
  [d] <- map read <$> getArgs
  printf "%f\n" $ mean [1..d] [1..d]

mean :: [Double] -> [Double] -> Double
mean xs1 xs2 = sum xs1 / fromIntegral (length xs2)
```



```
$ ghc --make -O2 prof3.hs \
  -prof -fprof-auto -rtsopts
$ ./prof3 1e7 +RTS -hy -p -i0.025
5000000.5
elapsed 6.66 s
```

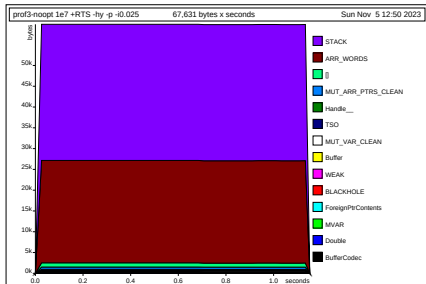
No luck: compiler recognized that `[1..d]` and `[1..d]` are identical and merged

Laziness doesn't reduce memory because the list needs to be traversed *twice*.

Idea: make two equivalent lists

```
main = do
  [d] <- map read <$> getArgs
  printf "%f\n" $ mean [1..d] [1..d]

mean :: [Double] -> [Double] -> Double
mean xs1 xs2 = sum xs1 / fromIntegral (length xs2)
```



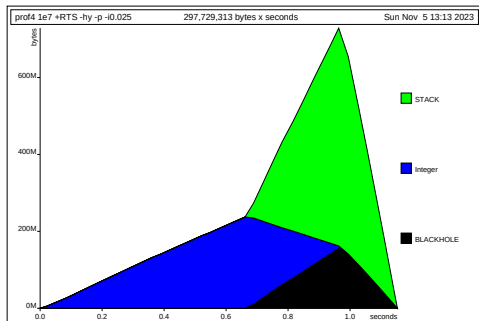
```
$ ghc --make prof3.hs -o prof3-noopt \
  -prof -fprof-auto -rtsospts
$ ./prof3-noopt 1e7 +RTS -hy -p -i0.025
5000000.5
elapsed 1.17 s
```

6× speedup from *disabling* optimization

This is what we want but...yuck

Sum the list and assess its length simultaneously:

```
mean :: [Double] -> Double
mean xs = total / fromIntegral count
  where (count, total) = foldl step (0, 0) xs
        step (n, s) x = (n + 1, s + x)
```



```
$ ghc --make -O2 prof4.hs \
  -prof -fprof-auto -rtsopts
$ /usr/bin/time -f "elapsed %e s" \
  ./prof4 1e7 +RTS -hy -p -i0.025
5000000.5
elapsed 12.51 s
```

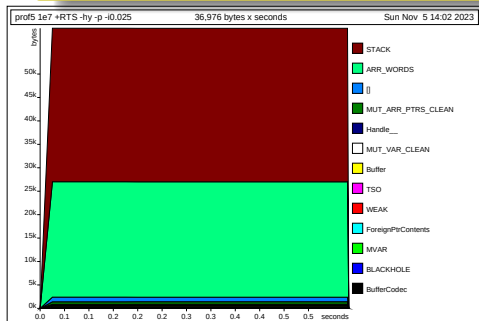
We traded 400M of heap for 700M of stack

The whole computation remains lazy: each call of *step* remains a thunk on the stack

That there are a lot of *Integer*-type objects is a hint: these are complex data structures being evaluated lazily.

Force the use of *Ints*:

```
mean :: [Double] -> Double
mean xs = total / fromIntegral count
  where (count, total) = foldl step (0::Int, 0) xs
        step (n, s) x = (n + 1, s + x)
```



```
$ ghc --make -O2 prof5.hs \
    -prof -fprof-auto -rtsopts
$ /usr/bin/time -f "elapsed %e s" \
    ./prof5 1e7 +RTS -hy -p -i0.025
5000000.5
elapsed 0.68 s
```

Much, much better

Initial version:

1,280,142,368 bytes allocated
1,070,855,432 bytes copied by GC

670 MiB total memory in use

INIT	time	0.001s
MUT	time	0.208s
GC	time	0.831s
EXIT	time	0.000s
Total	time	1.040s

Productivity 20.0%

Using *foldl* with *Ints*

1,280,142,368 bytes allocated
35,520 bytes copied by GC

6 MiB total memory in use

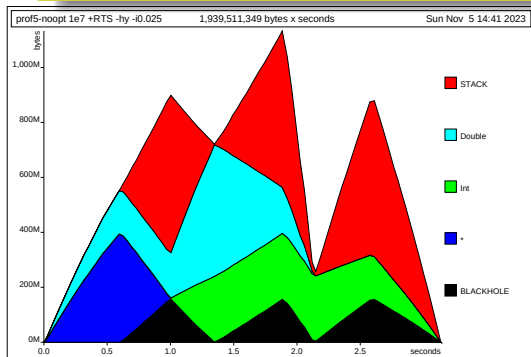
INIT	time	0.001s
MUT	time	0.184s
GC	time	0.002s
EXIT	time	0.000s
Total	time	0.187s

Productivity 98.3%

5× speedup: GC 400× faster

However, without optimization, we get

```
$ ghc --make prof5.hs -o prof5-noopt -prof -rtsopts
$ ./prof5-noopt 1e7 +RTS -hy -i0.025 -s
MUT      time      3.099s
GC       time      37.322s
```



Using an enormous amount of stack space, building a big Double list

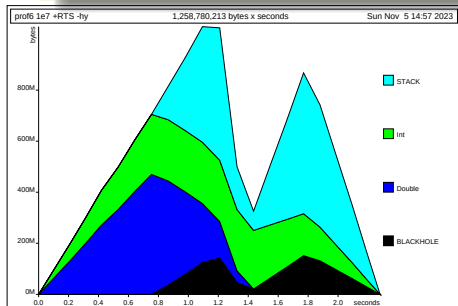
It's being too lazy

For this simple example `-O2` was able to do strictness analysis to eliminate needless laziness; we won't always be so lucky

First trick: use *Data.List.foldl'*, which accumulates state strictly

```
import Data.List (foldl')

mean :: [Double] -> Double
mean xs = total / fromIntegral count
  where (count, total) = foldl' step (0::Int, 0) xs
        step (n, s) x = (n + 1, s + x)
```



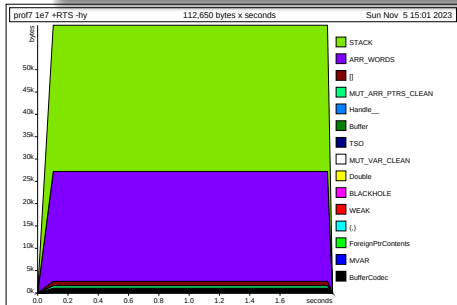
```
$ ghc --make prof6.hs -prof -rtsops
$ ./prof6 1e7 +RTS -hy
```

Better than before, but it's still too lazy

Problem is that *foldl'* is only strict to WHNF: the accumulated pair state is left unevaluated

Second trick: use `seq` to force strict evaluation of the components of the pair

```
mean :: [Double] -> Double
mean xs = total / fromIntegral count
  where (count, total) = foldl' step (0, 0) xs
        step (n, s) x = n `seq` s `seq` (n + 1, s + x)
```



```
$ ghc --make prof7.hs -prof -rtsops
$ ./prof7 1e7 +RTS -hy
```

Back to a very fast, constant-memory implementation

Another approach: force the accumulated state datatype to be strict

Adding ! to fields forces them to be strict

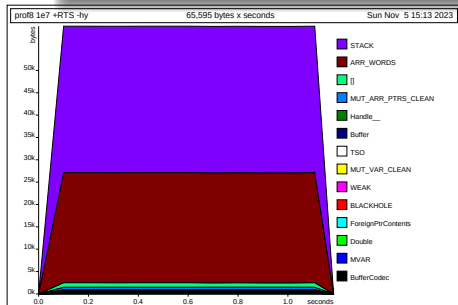
```
data Pair = Pair !Int !Double
```

```
mean :: [Double] -> Double
```

```
mean xs = total / fromIntegral count
```

```
  where Pair count total = foldl' step (Pair 0 0) xs
```

```
    step (Pair n s) x = Pair (n + 1) (s + x)
```



```
$ ghc --make prof8.hs -prof -rtspts
```

```
$ ./prof8 1e7 +RTS -hy
```

Similar effect as using *seq*; slightly less intrusive

The BangPatterns language extension is another way to force strictness

Adding ! to patterns forces arguments to be strict

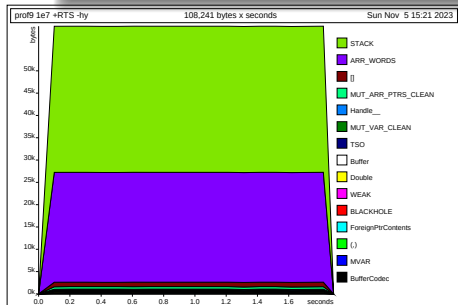
```
{-# LANGUAGE BangPatterns #-}
```

```
mean :: [Double] -> Double
```

```
mean xs = total / fromIntegral count
```

```
  where (count, total) = foldl' step (0::Int, 0) xs
```

```
    step (!n, !s) x = (n + 1, s + x)
```



```
$ ghc --make prof9.hs -prof -rtsospts
```

```
$ ./prof9 1e7 +RTS -hy
```

Even less intrusive than using a different data type