Plugging Space Leaks, Improving Performance

Neil Mitchell

https://github.com/ndmitchell/spaceleak

Should Haskell be strict? (No)

- Laziness is composable
  - all f = or . map f
- Laziness lets you express infiniteness
  - zip [1..] xs, primes !! 200
- Laziness matters for monads
  - putStrLn “Hello” >> error “done”
- Laziness is more natural
  - Most beginners *assume* Haskell is lazy

But....

The counterargument

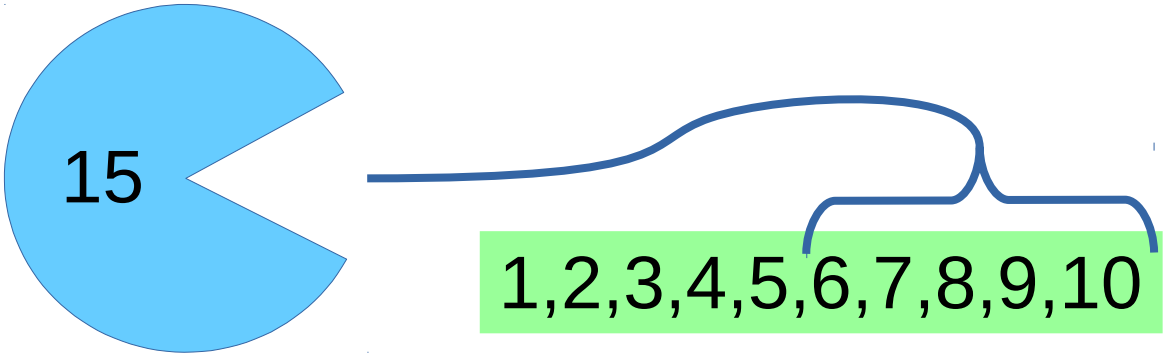
```
sum i [] = i
sum i (x:xs) = sum (i+x) xs

main = print $ sum 0 [1..10]
```

- What is the peak memory usage?

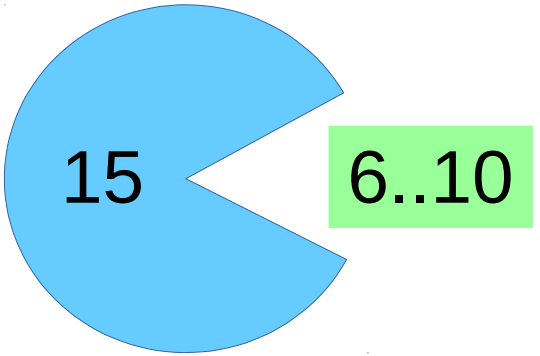
```
sum = foldl (+) 0
```

The execution

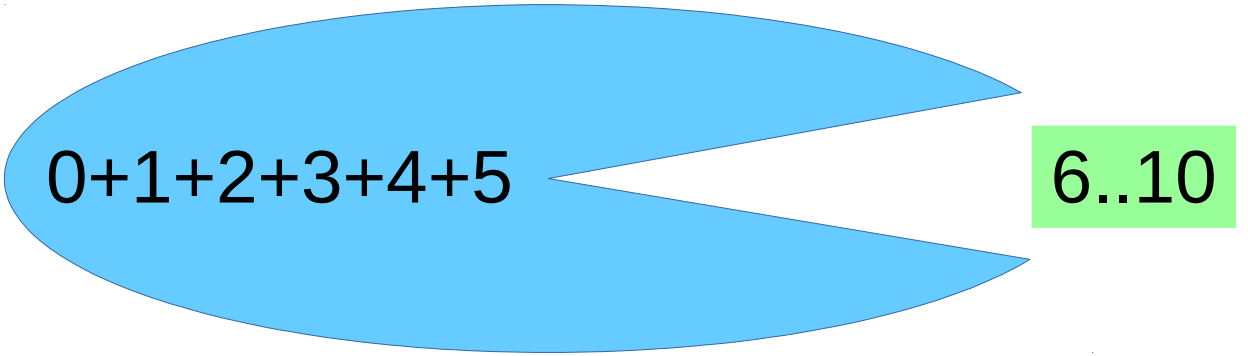
Strict



Ideal



Lazy



The solution

```
sum !i [] = i
sum i (x:xs) = sum (i+x) xs
```

- Annotate “the accumulator is strict”
- Each step reduces the accumulator
- Speedup: -O0 x13, -O1 or -O2 x17

```
sum = foldl’ (+) 0
```

Space leaks

- Relatively rare (1 per 2000 lines?)
- Not compositional property
- Not fatal, but significant performance hit
- Easy to fix (1m - 2h)
- Hard to spot

- This talk mostly fixes one of those issues
- And thus answers laziness vs strictness :)

One simple trick...

Limit the GHC stack

- GHC stack is used to evaluate deferred bits
- Limiting the stack turns space leaks into errors
- Track down errors, solve them

Idea with Tom Ellis + trains

The recipe

ghc --make Main.hs -rtsopts -prof -auto-all

- Compile with profiling

./Main +RTS -K${N}K

- Find lowest ${N} where program works

./Main +RTS -xc -K${N-1}K

- Get a stack trace, examine it

- Fix. Repeat until -K1K works

The output

```
C:\Neil\temp>Main +RTS -K100K -xc
*** Exception (reporting due to +RTS -xc): (THUNK_STATIC), stack trace:
  Main.sum1,
  called from Main.main,
  called from Main.CAF
*** Exception (reporting due to +RTS -xc): (THUNK_STATIC), stack trace:
  Main.sum1,
  called from Main.main,
  called from Main.CAF
Main: Stack space overflow: current size 33560 bytes.
Main: Use `+RTS -Ksize -RTS' to increase it.
```

Disclaimers

- Space leak investigation is sometimes not trivial
  - It’s a property of the way expressions are evaluated
  - Property does not compose!
  - Often it’s in the libraries you use
- Other things can use a lot of stack

Examples: Happy

- Parser generator for Haskell
- Medium (4800 lines), old, unfamiliar code base
- Run on one of the test examples (Calculator.ly)
- Found and fixed 3 space leaks
  - Now works at -K1K
  - 2 were trivial to fix
  - 1 took ~2 hours (5 min to fix, rest to check)

Example 1: Happy

```
indexInto :: Eq a => Int -> a -> [a] -> Maybe Int
indexInto _ _ [] = Nothing
indexInto i x (y:ys) =
    if x == y then Just i else indexInto (i+1) x ys
```

Example 2: Happy

```
foldr (\(a,b) (c,d) -> (a+b,b+d)) (0,0) conflictList
```

Example 2: Happy

```
foldr (\(a,b) (c,d) -> (a+b,b+d)) (0,0) conflictList


foldl' (\(a,b) (c,d) ->
    let !ac = a + c
        !bd = b + d
    in (ac,bd))
    (0,0) conflictList
```

The ugly truth: Stack limits

- GHC “mostly” obeys the stack limits
  - Stack limits can be exceeded while masked
  - Stack limits on the main thread are different

- Standard trick: join . onceFork

The ugly truth: Exception traces

- -xc prints out all exceptions
  - Your program may have *a lot* of exceptions
  - E.g. every ‘doesFileExist’ in some cases
  - Some exceptions may print more than once
- Usually the exception is near the end
- Worse if your program eats async exceptions
- Pipe them to a file, grep afterwards

The ugly truth: Stack contents

- The call stack elides adjacent duplicates
  - Which is exactly what we want to see!
- The stack probably doesn’t peek inside libraries
- Stack trace is more a list of hints, CAF’s get weird

```
{-# NOINLINE wrapper1 #-}
wrapper1 :: a -> a
wrapper1 x = x
```

Copy/Paste Toolbox

seq, deepseq, evaluate, force

```
foldl'' f = foldl' (\a b -> force $ f a b)
```

```
newThread a = unsafePerformIO $
    join $ onceFork return $! force a
```

False Positives

- reverse does *not* trigger a positive
- mapM/forM/sequence on IO does

```
main = do
    (t, _) <- duration $ mapM evaluate [1..100000]
    print t
```

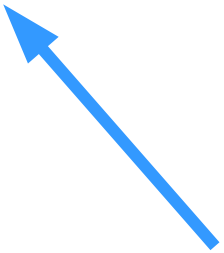
mapM stack trace

```
*** Exception (reporting due to +RTS -xc): (THUNK_STATIC), stack trace:
  Main.main,
  called from Main.CAF
  --> evaluated by: System.Time.Extra.duration,
  called from Main.main,
  called from Main.CAF
*** Exception (reporting due to +RTS -xc): (THUNK_STATIC), stack trace:
  Main.main,
  called from Main.CAF
Main: Stack space overflow: current size 33560 bytes.
```

Understanding the cause

- mapM f = sequence . map f

```
sequence :: [IO a] -> IO [a]
sequence [] = IO $ \r -> (# r, () #)
sequence (y:ys) = IO $ \r -> case unIO y r of
    (# r, y #) -> case unIO (sequence xs) r of
        (# r, ys #) -> (# r, y:ys #)
```



Recursion inside a case

Fixing mapM

- Use mapM_ if you don’t need the result
- Use mapIO
  - http://www.joachim-breitner.de/blog/684-Constructing_a_list_in_a_monad_revisited
- Use streaming (conduit/pipes)

- Definitely an annoyance

Example 3: QuickCheck

```
quickCheck $ \p ->
    label (if p > 0 then "+ve" else "-ve") True
+++ OK, passed 100 tests:
54% -ve
46% +ve
```

Example 3: QuickCheck

```
quickCheckWithResult stdArgs{maxSuccess=10000} $
    \(p :: Double) -> label "foo" True
(9999 tests)
Stack space overflow: current size 33624 bytes.
```

- At the end – a hint!
  - We’re detecting when the space leaks gets forced

Example 3: QuickCheck data

- Reproduce in QuickCheck to get better stack
- Found Map String Int, built with unionWith (+)

- Two “plausible” leaks:
  - unionWith (+) x1 $ unionWith x2 $ unionWith …
  - Map {foo = 1 + 1 + 1 + 1 ...}

Example 3: QuickCheck solution

```
-  import Data.Map
+ import Data.Map.Strict
```

- Fixed in QuickCheck 2.8.2
- Lay undiscovered for years, easy to fix
- O(n) extra memory required

Other examples

- base library: maximumBy
- Alex: lazy state monad
- Pretty: A strictness annotation
- Shake: three relatively small ones
- Hoogle: four or five (sum on Word16, strict Map with lazy pairs)
  - Uses -K1K in the test suite, so now they are fixed immediately

Weaknesses

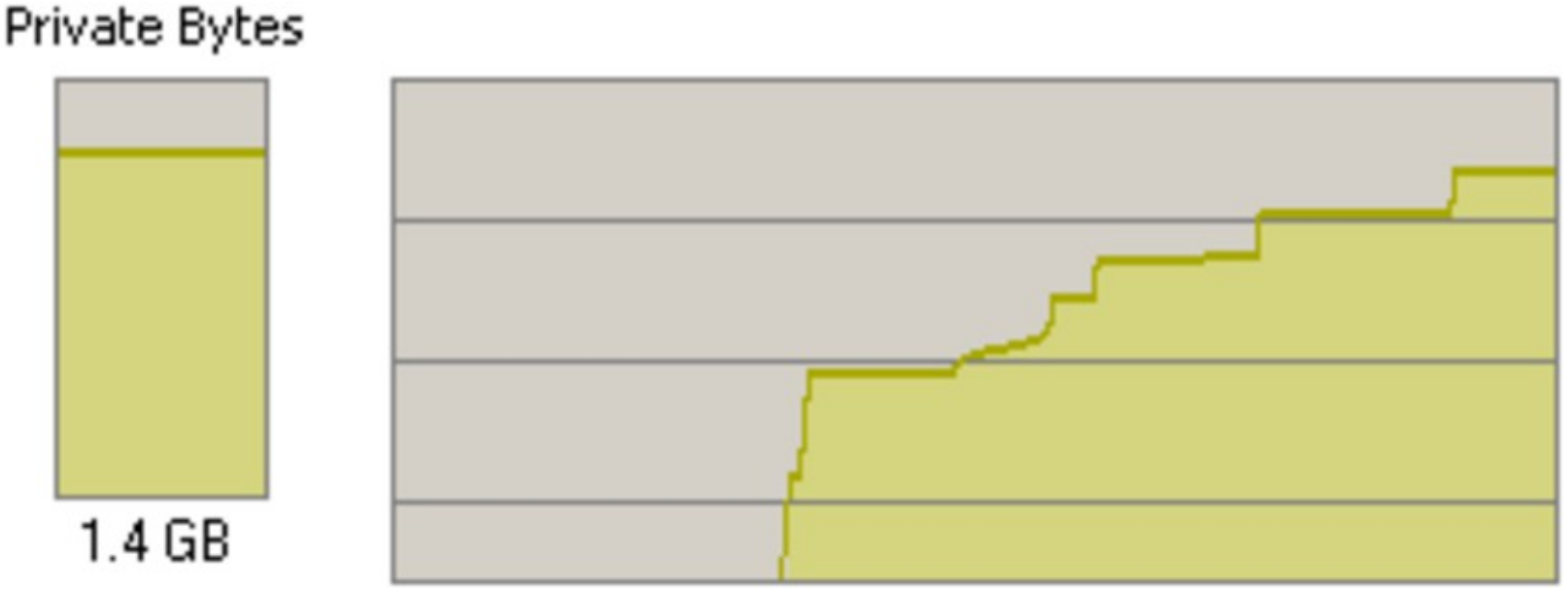
- There are memory issues that this doesn’t hit
  - Drag/lag/void/use problems
  - Genuine memory leaks
- Only finds the biggest space leak
  - Sometimes small space leaks are amplified
  - Your worst leak may not be the biggest
  - Serious leaks can be too small to detect

GHC etc. Requests

- -xc=StackOverflow, only show one type of exception
- Show repeat counts in the stack trace
- Call stacks inside libraries
  - At least the outer-most level
  - Can do with -auto-all when building (Cabal job?)
- “Exclude” mapM?
- Toolbox should be on Hackage

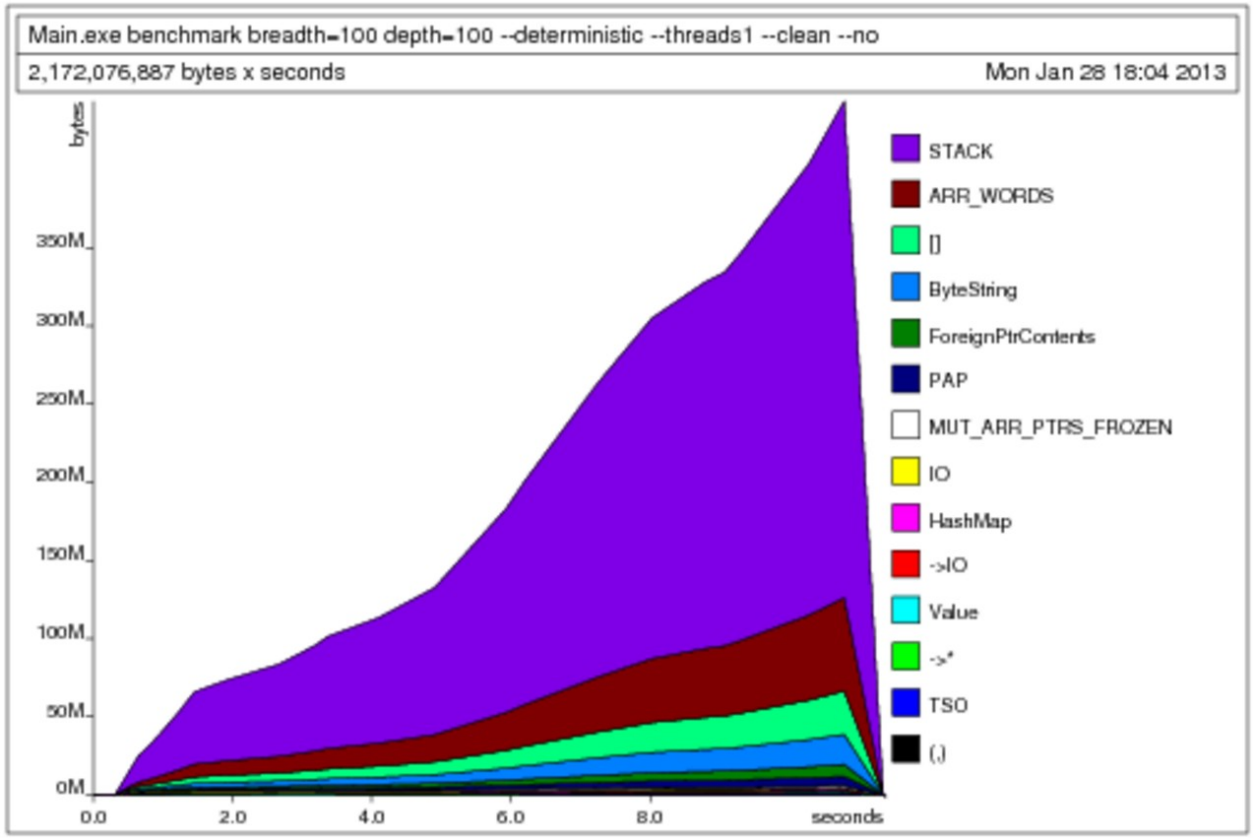
Example 4: Shake

- Shake v0.3 introduced a space leak
  - Went undetected for a year
  - Then blew up in production
  - Cost 1.5Gb memory (on a 32 bit system)



Example 4: Shake mem profile

- Compile: -rtsopts -prof -auto-all -caf-all
- Run: +RTS -xt -hy



Example 4: Shake diagnosis

- Two possibilities:
  - There are lots of threads in flight (there weren’t)
  - There are lots of stacks kept alive by ThreadId

Example 4: Shake understanding

- Shake thread pool had:

```
data Pool = Pool {threads :: Set ThreadId, …}
```

- Threads added when spawned, removed when finished
- Set of threads only used on exception cleanup
- Fix was trivial
- Significant space leak amplification

Example 4: Wrap up

- Space leak resulted in complete system failure
- Solved before my techniques were available
  - Took several painful weeks, not easy
  - Ended in a 1 character diff (plus comments)
- Set me on a journey…

… leading to today

Call to arms

- Fix your projects, fix other peoples projects
- A great way to get into a new project
  - Roughly all projects have such bugs
  - Fixing them is an awesome community service
- Add -K1K to your test suite
  - Much easier to fix with a breaking diff

Conclusion: Lazy > Strict

- Space leaks no longer worry me
- Relying on production Haskell no longer worries me (as much)
- Go forth and put Haskell in production!
- I am! Want to help?