

Laziness in GHC Haskell

The features and principles

Presented by chip

ZJU Lambda
From here to World

ZJU-Lambda Conference, May 2019



Contents

- 1 Appetizers
- 2 Thunk? What's it?
- 3 Why we need strictness?
- 4 Optimization
- 5 Tasty seq



```
possiblyBottom b =  
  case b of  
    True   → fst tup  
    False  → snd tup  
  where tup = (0, undefined)
```

If we apply **possiblyBottom** to **True**, we will get a 0.



Course 1: Outside in

A slightly arcane form:

```
possiblyBottom =  
  \f → f fst snd (0, undefined)
```

```
-- booleans in lambda form
```

```
true  :: a → a → a  
true = \a → (\b → a)
```

```
false :: a → a → a  
false = \a → (\b → b)
```



Course 1: Outside in

Nesting lambdas and reducing from the outside in:
(They are not in fact decomposed this way by the compiler)

$(\lambda f \rightarrow f \text{ fst snd } (0, \text{undefined})) (\lambda a \rightarrow (\lambda b \rightarrow a))$

$(\lambda a \rightarrow (\lambda b \rightarrow a)) \text{ fst snd } (0, \text{undefined})$

$(\lambda b \rightarrow \text{fst}) \text{ snd } (0, \text{undefined})$

$\text{fst } (0, \text{undefined})$

0



Course 2: Evaluate to WHNF

```
length'  :: [a] → Int
length' lst = go lst 0 where
    go [] acc      = acc
    go (x:xs) acc = go xs (acc+1)

main = let x = product [1..]
      in print $ length' [1, x]
```

It prints 2 !

What happened here?



Example 2: Evaluate to WHNF

The actual evaluation process:

```
length' [1, x]
= length' 1:(x:[])      -- 1:(x:[]) matches (x:xs)
= 1 + length' (x:[])    -- (x:[]), same with above
= 1 + 1 + length' []    -- [] matches []
= 1 + 1 + 0
= 2
```

Concept

In WHNF, we only evaluate the outermost constructor



Contents

- 1 Appetizers
- 2 Thunk? What's it?
- 3 Why we need strictness?
- 4 Optimization
- 5 Tasty seq



The Haskell Heap

The Haskell heap is a rather strange place.



Every item is wrapped up nicely in a box:
The Haskell heap is a heap of *presents* (thunks).



Present

When you actually want what's inside the present, you *open it up* (evaluate it).



Gift card

Sometimes you open a present, you get a *gift card* (data constructor). Gift cards have two traits.

- A name. (the **Just** gift card or **Right** gift card)
- And they tell you where the rest of your presents are.

There might be more than one (the tuple gift card), if you're a lucky duck!



Presents on the Haskell heap are rather mischievous.



Explode when you open it



Haunted by ghosts that open other presents when disturbed



What is a *thunk*?

<thunk: expression-to-be-evaluated>

- A box containing unevaluated expressions.
- Being evaluated when *needed*.
- Basically **anything** creates a thunk in (GHC) Haskell, by default

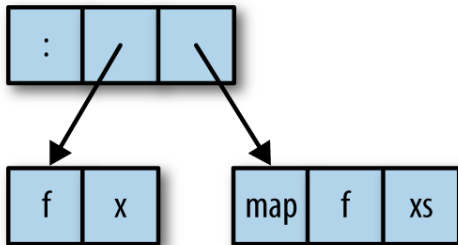


Figure: Thunks created by a map



Example: Evaluate a thunk

```
Prelude> let xs = map (+1) [1..10]
```

```
Prelude> seq xs ()
```

```
Prelude> :sprint xs
```

```
xs = _ : _
```

```
Prelude> length xs
```

```
Prelude> :sprint xs
```

```
xs = [_,_,_,_,_,_,_,_,_,_,_]
```

```
Prelude> head . tail $ xs
```

```
Prelude> :sprint xs
```

```
xs = [_,3,_,_,_,_,_,_,_,_,_]
```

Important

Once evaluated, the thunk is replaced by its *actual* value.

Thunk brings us...

- On-demand data types.
- Call-by-need strategy.
- Result sharing on CAF (Constant Applicative Forms).
- ...

```
fibs :: [Integer]  
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```



Contents

- 1 Appetizers
- 2 Thunk? What's it?
- 3 Why we need strictness?
- 4 Optimization
- 5 Tasty seq



Thunks are good, but...


```
foldl (+) 0 (1:2:3:[])  
  = foldl (+) (0 + 1)                (2:3:[])  
  = foldl (+) ((0 + 1) + 2)          (3:[])  
  = foldl (+) (((0 + 1) + 2) + 3) []  
  =                               (((0 + 1) + 2) + 3)
```

What about **foldl** (+) 0 [1..10000000000] ?

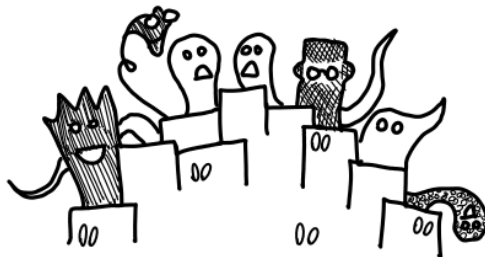


Memory leak

After executing **foldl** (+) 0 [1..1000000000]

Process Name	Status	% CPU	Nice	ID	Memory ▾
 ghc	Running	44	0	30047	4.0 GiB

A veritable ghost jamboree in our memory!



RTS - a non-trivial Runtime System

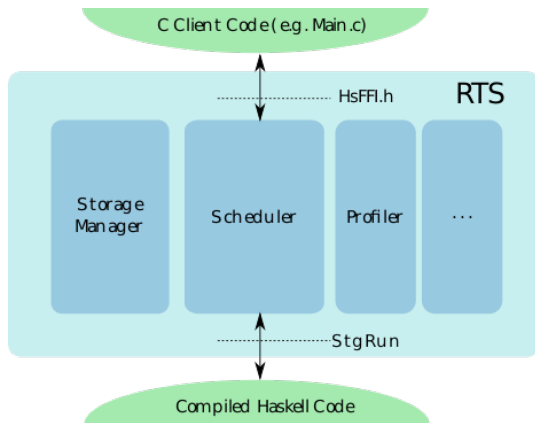


Figure: RTS Overview



Example: Profiling

```
import System.Environment
import Text.Printf

main = do
    [d] ← map read `fmap` getArgs
    printf "%f\n" (mean [1..d])

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



Compile it.

```
ghc -make -rtsopts -O2 a.hs  
./a 1e7 +RTS -sstderr
```

Output:

```
...  
1519 MB total memory in use
```

MUT	time	2.905s	(2.907s elapsed)
GC	time	8.936s	(8.956s elapsed)
Total	time	11.865s	(11.887s elapsed)

%GC	time	75.3%	(75.3% elapsed)
-----	------	-------	-----------------



Basic Profiling

Mark the cost centres

- SCC pragma
- Option **-auto-all**
- and **-caf-all**, if needed

Then, compile with option **-prof**

Run **./a 1e7 +RTS -p**, we get a file **a.prof**



Basic Profiling

```
1 Wed May 22 17:43 2019 Time and Allocation Profiling Report (Final)
2
3 a +RTS -p -RTS 1e7
4
5 total time = 2.57 secs (2570 ticks @ 1000 us, 1 processor)
6 total alloc = 1,680,116,384 bytes (excludes profiling overheads)
7
8 COST CENTRE MODULE %time %alloc
9
10 main Main 87.2 100.0
11 mean Main 12.8 0.0
12
13
14
15 COST CENTRE MODULE no. entries individual inherited
16 %time %alloc %time %alloc
17 MAIN MAIN 60 0 0.0 0.0 100.0 100.0
18 main Main 121 0 87.2 100.0 100.0 100.0
19 mean Main 124 1 12.8 0.0 12.8 0.0
20 CAF:main1 Main 118 0 0.0 0.0 0.0 0.0
21 main Main 120 1 0.0 0.0 0.0 0.0
22 CAF:main2 Main 117 0 0.0 0.0 0.0 0.0
23 main Main 122 0 0.0 0.0 0.0 0.0
24 CAF:main9 Main 114 0 0.0 0.0 0.0 0.0
25 main Main 123 0 0.0 0.0 0.0 0.0
26 CAF GHC.IO.Handle.FD 106 0 0.0 0.0 0.0 0.0
27 CAF Text.Read.Lex 102 0 0.0 0.0 0.0 0.0
28 CAF GHC.Conc.Signal 101 0 0.0 0.0 0.0 0.0
29 CAF GHC.Float 100 0 0.0 0.0 0.0 0.0
30 CAF GHC.IO.Encoding 99 0 0.0 0.0 0.0 0.0
31 CAF GHC.IO.Encoding.Iconv 79 0 0.0 0.0 0.0 0.0
```

Figure: Profiling message generated by RTS



Heap Profiling

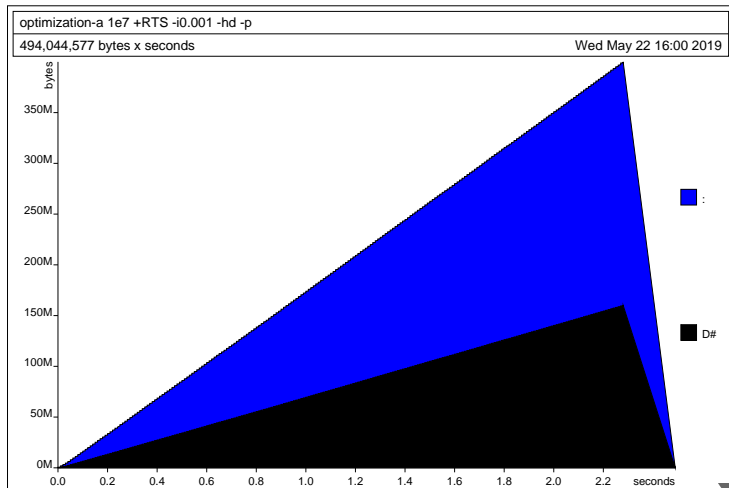


Figure: Break by constructor/closure



Contents

- 1 Appetizers
- 2 Thunk? What's it?
- 3 Why we need strictness?
- 4 Optimization**
- 5 Tasty seq



Example

Where might be the cost centres?

```
import System.Environment
import Text.Printf

main = do
    [d] ← map read `fmap` getArgs
    printf "%f\n" (mean [1..d])

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



Method 1: Using strict functions

```
mean :: [Double] → Double
mean xs = s / fromIntegral n
  where
    (n, s)      = foldl' k (0, 0) xs
    k (n, s) x = n `seq` s `seq` (n+1, s+x)
```

Why can't we just write `(n,s) `seq` (n+1, s+x)`?



Method 2: Bang patterns

```
{-# LANGUAGE BangPatterns #-}  
  
...  
mean :: [Double] → Double  
mean xs = s / fromIntegral n  
  where  
    (n, s)      = foldl' k (0, 0) xs  
    k (!n, !s) x = (n+1, s+x)
```



Method 3: Strict data types

```
data Pair a b = Pair !a !b
...
mean :: [Double] → Double
mean xs = s / fromIntegral n
  where
    Pair n s      = foldl' k (Pair 0 0) xs
    k (Pair n s) x = Pair (n+1) (s+x)
```

The **Pair** will always store its fields in WHNF.



Method 4: Unbox values

Introduction: Core

- An intermediate language used by GHC.
- Resembles a subset of Haskell
- Explicit type annotations (F_C)
- Type equality constraints and safe coercions

To check the Core, compile with **-ddump-simpl**



Method 4: Unbox values

Simplified Core: inlined **fold'**

```
lgo :: Integer → [Double]
      → Double# → (# Integer, Double #)
lgo = \ n xs s →
  case xs of
    []      → (# n, D# s #);
    (:) x ys →
      case plusInteger n 1 of
        n' → case x of
          D# y → lgo n' ys (+### s y)
```

More to optimize?



Method 4: Unbox values

Avoid heap checking: **data Pair = Pair !Int !Double**

Unbox the **Pair**: using flag **-funbox-strict-fields**

```
lgo :: Int# → Double#  
      → [Double] → (# Int#, Double# #)  
lgo = \ n s xs →  
      case xs of  
        []      → (# n, s #)  
        (:) x ys →  
          case x of  
            D# y → lgo (+# n 1) (+## s y) ys
```

Now, except the nodes of list, all the values are stored in registers.



Compile it.

```
ghc -make -funbox-strict-fields -rtsops -O2 a.hs  
./a 1e7 +RTS -sstderr
```

Output:

```
...  
1 MB total memory in use  
  
MUT      time      1.490s  (  1.496s elapsed)  
GC        time      0.016s  (  0.015s elapsed)  
Total     time      1.509s  (  1.511s elapsed)  
  
%GC       time      1.0%   (1.0% elapsed)
```



Summary

	Memory used	MUT	GC	Total
Original	1519 MB	2.905s	8.936s(75.3%)	11.865s
Optimized	1 MB	1.490s	0.016s(1.0%)	1.509s

What we can learn is:

- Compile with **-O2** flag.
- Go profiling(Time/Heap) when confused.
- Avoid calculations piling up (using strictness).
- Unbox atom types (**Int**, **Double**, ...)
- Use types that can be transformed into primitives (**Int** instead of **Integer**)



Further optimization

- Deforestation (remove intermediate structures)
- Rely on gcc -O2 (**-fvia-C -optc-O2**)

In our example, we can use a deforestation called **stream fusion**. It turns *recursive list generation* and *transformation functions* into *non-recursive unfolds*



Stream fusion

```
import System.Environment
import Text.Printf
import Data.Array.Vector

main = do
    [d] ← map read `fmap` getArgs
    printf "%f\n" (mean (enumFromToFracU 1 d))
```

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

```
mean :: UArr Double → Double
```

```
mean xs = s / fromIntegral n
```

```
where
```

```
    Pair n s      = foldlU k (Pair 0 0) xs
```

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



Contents

- 1 Appetizers
- 2 Thunk? What's it?
- 3 Why we need strictness?
- 4 Optimization
- 5 Tasty seq



Using *seq*

seq :: $a \rightarrow b \rightarrow b$

seq evaluates its first argument to **WHNF**, and return the second one.

$(\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b$ -- |infixr 0|

$\$!$ is similar with $\$$, but evaluates its argument to **WHNF**.



```
deepseq :: NFData a  $\Rightarrow$  a  $\rightarrow$  b  $\rightarrow$  b  
($!!) :: NFData a  $\Rightarrow$  (a  $\rightarrow$  b)  $\rightarrow$  a  $\rightarrow$  b -- |infixr 0|
```

```
force :: NFData a  $\Rightarrow$  a  $\rightarrow$  a
```

```
force x = x `deepseq` x
```

```
class NFData a where
```

```
  rnf :: a  $\rightarrow$  ()
```

```
  rnf a = a `seq` ()
```



par :: a → b → b -- |infixr 0|

Indicates that it may be beneficial to evaluate the first argument in parallel with the second. Returns the value of the second argument.

pseq :: a → b → b -- |infixr 0|

Guarantee the order of evaluation in parallelism.



par :: a → b → b -- |infixr 0|

Indicates that it may be beneficial to evaluate the first argument in parallel with the second. Returns the value of the second argument.

pseq :: a → b → b -- |infixr 0|

Guarantee the order of evaluation in parallelism.

Possible transformation on **seq**:

a `seq` b \iff b `seq` a `seq` b



More on parallel programming

Please refer to:

Control.Parallel.Strategies (deterministic parallelism)

Control.Concurrent (non-deterministic parallelism)

Seq no more: Better Strategies for Parallel Haskell



Thank you!

This work is licensed under **CC-BY-SA 4.0**
Creative Commons Attribution Share Alike 4.0 International

You can find the slides on my **Github**

(The hyperlinks really exist, but they are not colored. x)

