Laziness in GHC Haskell

The features and principles

Presented by chip

ZJU Lambda
From here to World

ZJU-Lambda Conference, May 2019

Contents

Course 1: Outside in

```
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`.

A slightly arcane form:

```haskell
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)

```
(\f → f fst snd (0, undefined)) (\a → (\b → a))

(\a → (\b → a)) fst snd (0, undefined)

(\b → fst) snd (0, undefined)

fst (0, undefined)

0
```

```haskell
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

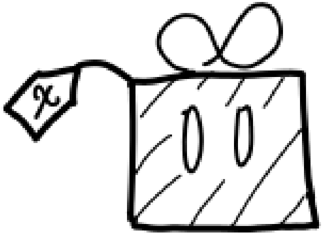
Contents

The Haskell Heap

The Haskell heap is a rather strange place.

Box

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



Present

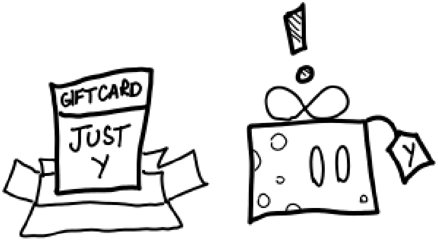
When you actually want what's inside the present, you *open it up* (evaluate it).

Gift card

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

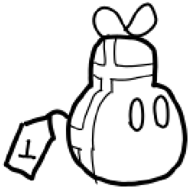
- A name. (the **Just** gift card or **Right** gift card)
- And they tell you where the rest of your presents are.

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



Tricksters

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

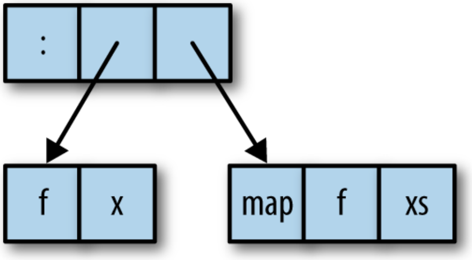


Figure: Thunks created by a map

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).
- ...

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

Contents

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)
```

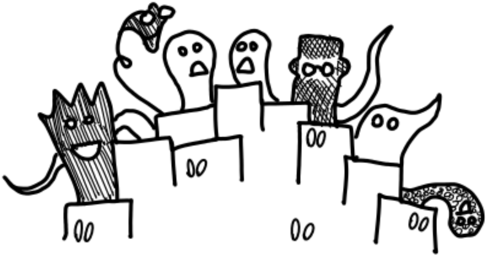
What about `foldl (+) 0 [1..1000000000]` ?

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

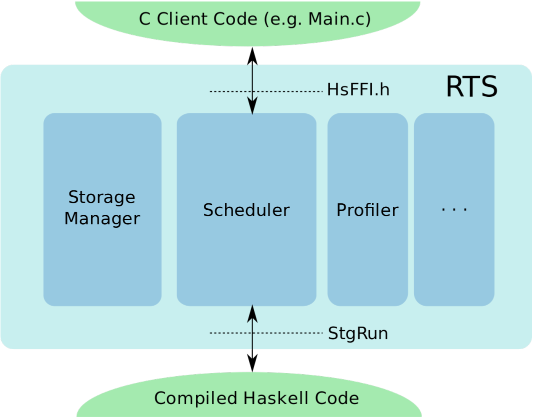


Figure: RTS Overview

Example: Profiling

```haskell
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)
```

Statistics

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

```
    Wed May 22 17:43 2019 Time and Allocation Profiling Report  (Final)

       a +RTS -p -RTS 1e7

    total time  =        2.57 secs   (2570 ticks @ 1000 us, 1 processor)
    total alloc = 1,680,116,384 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

main        Main     87.2  100.0
mean        Main     12.8    0.0


                                                     individual      inherited
COST CENTRE MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                     60           0    0.0    0.0   100.0  100.0
 main       Main                    121           0   87.2  100.0   100.0  100.0
  mean      Main                    124           1   12.8    0.0    12.8    0.0
 CAF:main1  Main                    118           0    0.0    0.0     0.0    0.0
  main      Main                    120           1    0.0    0.0     0.0    0.0
 CAF:main2  Main                    117           0    0.0    0.0     0.0    0.0
  main      Main                    122           0    0.0    0.0     0.0    0.0
 CAF:main9  Main                    114           0    0.0    0.0     0.0    0.0
  main      Main                    123           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Handle.FD        106           0    0.0    0.0     0.0    0.0
 CAF        Text.Read.Lex           102           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal         101           0    0.0    0.0     0.0    0.0
 CAF        GHC.Float               100           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding          99           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv    79           0    0.0    0.0     0.0    0.0
```

Figure: Profiling message generated by RTS

Heap Profiling

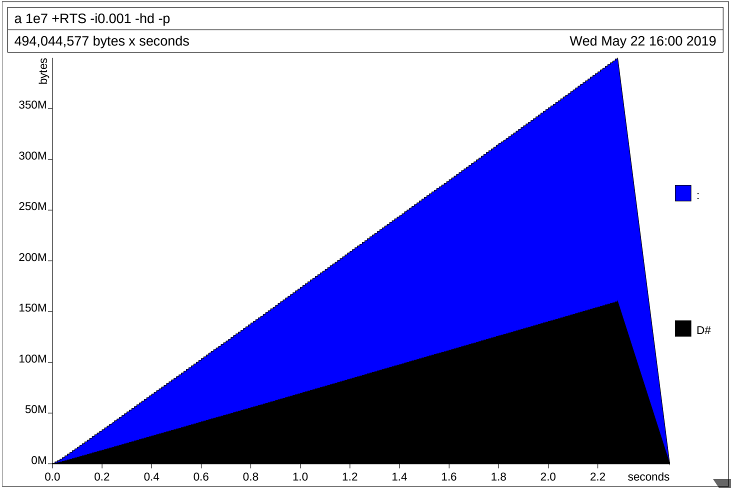


Figure: Break by constructor/closure

Contents

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

```haskell
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

```haskell
{-# 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

```haskell
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, thus NF for atoms.

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?

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.

Statistics

Compile it.

ghc –make -funbox-strict-fields -rtsopts -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

```haskell
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

Using *seq*

```haskell
seq :: a → b → b
```

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

```haskell
($!) :: (a → b) → a → b -- |infixr 0|
```

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

```haskell
deepseq :: NFData a ⇒ a → b → b
($!!) :: NFData a ⇒ (a → b) → a → b -- |infixr 0|

force :: NFData a ⇒ a → a
force x = x `deepseq` x

class NFData a where
    rnf :: a → ()
    rnf a = a `seq` ()
```

Control.Parallel

```
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.

Control.Parallel

```
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 ⟺ 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!

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

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