Laziness in GHC Haskell

The features and principles

Presented by chip

ZJU Lambda
From here to World

ZJU-Lambda Conference, May 2019

Contents

Example 1: No evaluation

```haskell
f :: Int -> Int -> Int
f x y = case x > 0 of
    True  -> x - 1
    False -> x + 1

main = print $ f 1 (product [1..])
```

Well, it prints 0

Example 2: Evaluate to WHNF

```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

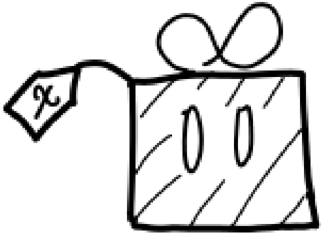
Contents

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

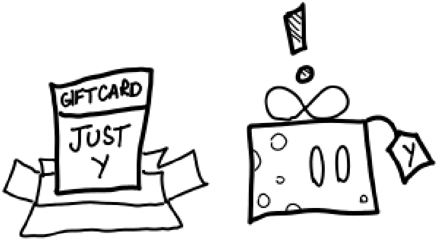
When you actually want what's inside the present, you *open it up* (evaluate it).

Gift card

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

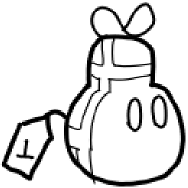
- A name. (the **Just** gift card or **Right** gift card)
- And they tell you where the rest of your presents are.

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

Example: Evaluate a thunk

How will this expression be evaluated?

```haskell
map negate [1,2,3]
```

Example: Evaluate a thunk

How will this expression be evaluated?

```
map negate [1,2,3]
```

```
<@: map negate <@: (1:2:3:[])>>
```

Example: Evaluate a thunk

How will this expression be evaluated?

```
map negate [1,2,3]

<@: map negate <@: (1:2:3:[])>>

<@: negate <@: 1> : <@: map negate <@: [2,3]>>
```

Example: Evaluate a thunk

How will this expression be evaluated?

```
map negate [1,2,3]
```

```
<@: map negate <@: (1:2:3:[])>>

<@: negate <@: 1> : <@: map negate <@: [2,3]>>

-<@: 1> : <@: map negate <@: [2,3]>>
```

Example: Evaluate a thunk

How will this expression be evaluated?

```
map negate [1,2,3]

<@: map negate <@: (1:2:3:[])>>

<@: negate <@: 1> : <@: map negate <@: [2,3]>>

-<@: 1> : <@: map negate <@: [2,3]>>

-1 : <@: map negate <@: [2,3]>>
```

Thunk brings us...

- On-demand data types.
- Call-by-need strategy.
- Memory reuse 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!