Introduction to optimizing Haskell code

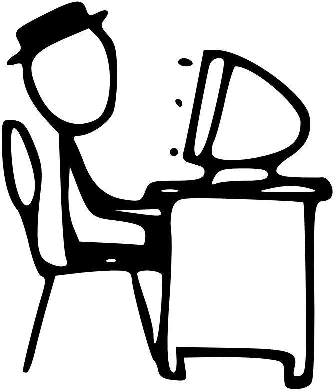
8th GhentFPG meeting

Jasper Van der Jeugt

May 30, 2011

Hello!

My name is Jasper
Student at UGent
I write Haskell
GhentFPG
`@jaspervdj`
`jaspervdj.be`



Overview

Credit where credit is due

High-Performance Haskell
(And general advice)
Johan Tibell

Overview

Introduction

Strictness analysis

Benchmarking pitfalls

GHC Core

Overview

Introduction

Strictness analysis

Benchmarking pitfalls

GHC Core

Strictness analysis

Haskell has lazy evaluation as default

Strictness analysis

Lazy evaluation leads to more composable code

Disadvantage: too much laziness

Strictness analysis

A function can be strict in it's arguments

```
null :: [a] -> Bool
null [] = True
null _  = False
```

Strictness analysis

```haskell
hello :: String -> String
hello name =
  "Hello, " ++ name ++ "!"
```

Strictness analysis

```
quadr :: Floating a
      => a -> a -> a -> a -> a
quadr a b c x =
  a * x ^ 2 + b * x + c
```

Strictness analysis

```
if' :: Bool -> a -> a -> a
if' True  x _ = x
if' False _ y = y
```

Strictness analysis

```
maybe ::
   b -> (a -> b) -> Maybe a -> b
maybe d _ Nothing  = d
maybe _ f (Just x) = f x
```

Strictness analysis

Functions can easily be made strict

```
seq :: a -> b -> b
```

Strictness analysis

```haskell
quadr :: Floating a
      => a -> a -> a -> a -> a
quadr a b c x =
  a `seq` b `seq` c `seq`
    a * x ^ 2 + b * x + c
```

Strictness analysis

Useful syntactic sugar

```
{-# LANGUAGE BangPatterns #-}
quadr :: Floating a
      => a -> a -> a -> a -> a
quadr !a !b !c x =
  a * x ^ 2 + b * x + c
```

Strictness analysis

Some notes about seq

Strictness analysis

Right usage

```
f x = x `seq` g x
```

Strictness analysis

Wrong usage

```
x ‘seq‘ x
```

Strictness analysis

Most important: `seq` is no magic!
Translates to a `case` statement

Overview

Introduction

Strictness analysis

Benchmarking pitfalls

GHC Core

Benchmarking pitfalls

Haskell is a lazy language
This makes benchmarking hard

Benchmarking pitfalls

Two types of benchmarks:
Functions and programs

(we focus on the former)

Benchmarking pitfalls

Benchmarking some function

```
f :: Int -> Int
```

Benchmarking pitfalls

In e.g. Python

```
total = 0
for i in range(100):
    start = time.time()
    f()
    end = time.time()
    total += (end - start) / 100
```

Benchmarking pitfalls

In Haskell?

```
replicateM 100 $ do
    start <- getTime
    let y = f x
    end <- y ‘seq‘ getTime
```

This is pretty hard to get right

Benchmarking pitfalls

Conclusion?

Never write your own benchmarking code

Benchmarking pitfalls

Criterion

By Bryan O'Sullivan

Benchmarking pitfalls

Criterion

```
bench "f" $ nf   f x
bench "g" $ whnf g x
```

Benchmarking pitfalls

Eq for string types

```
whnf (== T.init t
     `T.snoc` '\xfffd') t

whnf (== BL.init bl
     `BL.snoc` '\xfffd') bl
```

Benchmarking pitfalls

But `ByteString.Lazy`
is a little faster

`Text`: 2.489305 us
`ByteString.Lazy`: 39.29312 **ns**

Benchmarking pitfalls

Digging into the code...

```
eq (Chunk a as) (Chunk b bs) =
   case compare (S.length a)
                (S.length b) of
     ...
     EQ -> a == b && eq as bs
     ...
```

Benchmarking pitfalls

Digging further...

```
eq a@(PS p s l) b@(PS p' s' l')
    -- short cut on length
    | l /= l'                = False
    -- short cut for same string
    | p == p' && s == s' = True
    | ...
```

Benchmarking pitfalls

Conclusion?

Libraries can be smarter than you think they are, make sure you know what you are benchmarking!

Benchmarking pitfalls

Benchmarking IO

```
bench "HtmlCombinator" $ do
  putStr "Content-Type:␣..."
  ...
  putStr "<table>"
  putStr $ toLazyText $
    makeTable 20000
  putStr "</table>"
```

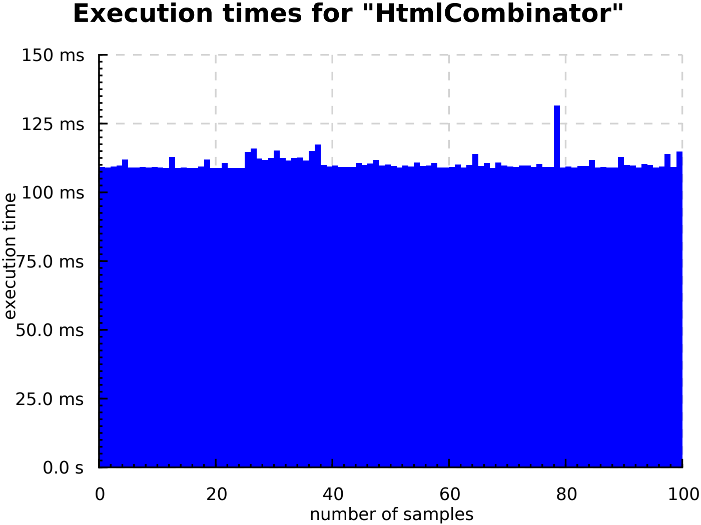
Benchmarking pitfalls

This looks suspicious

```
benchmarking HtmlCombinator
collecting 100 samples (...)
    estimated 30.80161 s
mean: 107.6378 ms (...)
```

$$100 * 100ms \neq 30s$$

Benchmarking pitfalls



Benchmarking pitfalls

Where is the issue?

```
bench "HtmlCombinator" $ do
  putStr "Content-Type:␣..."
  ...
  putStr "<table>"
  putStr $ toLazyText $
    makeTable 20000
  putStr "</table>"
```

Benchmarking pitfalls

```
putStr . toLazyText .
    makeTable =<< rows
...
where
  rows :: IO Int
  rows = return 20000
  {-# NOINLINE rows #-}
```

Benchmarking pitfalls

Conclusion?

GHC is pretty smart as well

Overview

Introduction

Strictness analysis

Benchmarking pitfalls

GHC Core

GHC Core

What is GHC Core?
Why should we care?

GHC Core

What is GHC Core?

Internal representation used by GHC

A kernel language

Optimizations are applied here

GHC Core

Why should we care?

Understanding benchmark results
Know what is going on
Impress your friends!

GHC Core

A few basic rules

GHC Core

Function pattern matching, guards, if's are translated to `case`

GHC Core

`where` is translated to `let`

GHC Core

Type annotations everywhere

GHC Core

Reading core

Clean up qualified names
Use proper variable names
Remove unnecessary type annotations

GHC Core

Demo

Questions?