

# Haskell Performance Measurements

what to do if laziness has bitten you  
(or you've eaten too much memory)

Bohdan Liesnikov

[@phittacus](#)

2018.04.22

# Overview

Spoilers!

# Overview

Spoilers!

This talk isn't an eye-opening one.

# Overview

Spoilers!

This talk isn't an eye-opening one.

- there are profiling tools in haskell

# Overview

Spoilers!

This talk isn't an eye-opening one.

- there are profiling tools in haskell
- they are actually usable

# Overview

Spoilers!

This talk isn't an eye-opening one.

- there are profiling tools in haskell
- they are actually usable
- there is a couple of funny quirks and techniques along the way

# Overview (like, a serious one)

- why do you need it?  
(isn't Haskell ponies and butterflies anyway?)
- time profiling
- memory profiling

# Ponies and butterflies

Space leak



# Ponies and butterflies

Space leak

- Not similar to memory leaks

# Ponies and butterflies

## Space leak

- Not similar to memory leaks
- “Space leak” simply means that we can do better

# Ponies and butterflies

## Space leak

- Not similar to memory leaks
- “Space leak” simply means that we can do better
- Classic example

```
let xs = [1..10000000::Integer]  
in sum xs * product xs
```

# Optimizations

An interesting theme for all<sup>1</sup>of us<sup>2</sup>

---

<sup>1</sup>Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. 1991. E. Meijer, M. Fokkinga, R. Paterson [\[pdf\]](#)

<sup>2</sup>Blog posts by Don Stewart

<https://donsbot.wordpress.com/tag/fusion/>

# Optimizations

An interesting theme for all<sup>1</sup> of us<sup>2</sup>

## ■ Fusion

**map** f . **map** g → **map** (f . g)

---

<sup>1</sup>Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. 1991. E. Meijer, M. Fokkinga, R. Paterson [\[pdf\]](#)

<sup>2</sup>Blog posts by Don Stewart

<https://donsbot.wordpress.com/tag/fusion/>

# Optimizations

An interesting theme for all<sup>1</sup> of us<sup>2</sup>

- Fusion

**map** f . **map** g → **map** (f . g)

- Strictness (in some arguments)

---

<sup>1</sup>Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. 1991. E. Meijer, M. Fokkinga, R. Paterson [\[pdf\]](#)

<sup>2</sup>Blog posts by Don Stewart

<https://donsbot.wordpress.com/tag/fusion/>

# Optimizations

- Metamorphism  
Catamorphism followed by anamorphism  
(fold then unfold)

---

these were initially mixed up, thanks for correcting me!

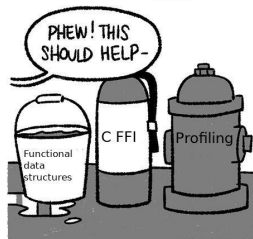
# Optimizations

- Metamorphism  
Catamorphism followed by anamorphism  
(fold then unfold)
- Hylomorphism  
Anamorphism followed by catamorphism  
(unfold then fold)

---

these were initially mixed up, thanks for correcting me!





# Example

- takes three file names (initial, series of diffs and final)
- parses them
- checks if they are fine
- prints a verdict out

# Example

- takes three file names (initial, series of diffs and final)
- parses them
- checks if they are fine
- prints a verdict out

initial:	diff:	final:
me you	# 2018.04.22 13:00 < me > them	you them

# The simplest approach

## ■ In ghci

```
*Main> :set +s
*Main> process Files { initial = "i.list"
                        , final   = "f.list"
                        , diff    = "d.list"}
this is a correct diff
(10.11 secs, 11,753,183,352 bytes)
```

Not the same timing you would get in a real setting

# The simplest approach

## ■ Debug statements

```
import Debug.Trace (trace)
...
flip trace () $ "before" ++ show getCurrentTime
...
flip trace () $ "after"  ++ show getCurrentTime
```

Meh

# Compiling

You can't just profile it right away — we need to compile it properly beforehand

ghc-options:

```
...  
--enable-profiling # and/or --enable-library-profiling  
-fprof-auto  
-rtsopts
```

Yes, this actually means recompiling libraries with profiling enabled

# Simple approach

```
./dist/build/demo/demo +RTS  
-sstderr
```

```
...  
INIT    time    0.002s  ( 0.001s elapsed)  
MUT     time    0.343s  ( 0.339s elapsed) <- doing useful things  
GC      time    0.349s  ( 0.346s elapsed) <- gc is gc  
RP      time    0.000s  ( 0.000s elapsed)  
PROF    time    0.000s  ( 0.000s elapsed)  
EXIT    time    0.003s  ( 0.004s elapsed)  
Total   time    0.697s  ( 0.690s elapsed)
```

```
...  
Alloc rate    3,426,690,480 bytes per MUT second
```

```
Productivity  68.7% of total user, 68.8% of total elapsed
```

# Stack traces and flame graphs

```
./dist/build/demo/demo +RTS -p
```

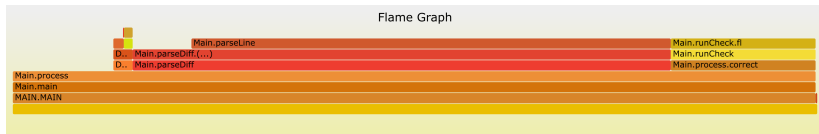
COST CENTRE	MODULE	SRC	no.	1 entries	2 %time	3 %alloc	4 individual %time	5 inherited %alloc
...								
parseDiff	Main	...	419	1	0.0	0.0	39.6	66.9
parseDiff(...)	Main	...	421	1	5.9	7.3	39.6	66.9
parseLine	Main	...	422	161080	26.7	59.6	33.7	59.6
...								

- 1 entries — number of times this particular point in the call tree was entered
- 2 %time — percentage of the total run time of the program spent at this point
- 3 %alloc — percentage of the total memory allocations of the program made by this call
- 4 %time — percentage of the total run time of the program spent below this point in the call tree.
- 5 %alloc — percentage of the total memory allocations of the program made by this call and all of its sub-calls



# Stack traces and flame graphs

- [FlameGraph](#) by Brendan Gregg
- [ghc-prof-flamegraph](#) by FP Complete



So good graphs, immediately runs faster

# Cost centers

- Cost centres are just program annotations  
`{-# SCC "name" #-} <expression>`
- `-fprof-auto` automatically insert a cost centre annotation around every binding not marked `INLINE` in your program
- You are entirely free to add cost centre annotations yourself.

# Space profiling

Not that easy with lazy evaluation and all the transformations <sup>3</sup>

All we can do — get some measurements from the beforementioned tools

---

<sup>3</sup>You can do that, just it isn't pleasant in any way

<sup>4</sup>Neil Mitchell. Detecting Space Leaks. 2015

<https://neilmitchell.blogspot.com/2015/09/detecting-space-leaks.html>

# Space profiling

Not that easy with lazy evaluation and all the transformations<sup>3</sup>

All we can do — get some measurements from the beforementioned tools

Or we can try to detect space leaks using a cute trick<sup>4</sup>

---

<sup>3</sup>You can do that, just it isn't pleasant in any way

<sup>4</sup>Neil Mitchell. Detecting Space Leaks. 2015

<https://neilmitchell.blogspot.com/2015/09/detecting-space-leaks.html>

# Space profiling

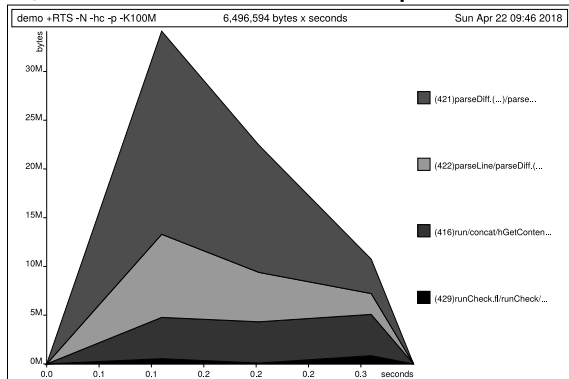
- Run the program with a specific stack size,  
`./demo +RTS -K100K` to run with a  
100Kb stack.
- Increase/decrease the stack size until you  
have determined the minimum stack for  
which the program succeeds
- Reduce the stack by a small amount and  
rerun

# Space profiling

- The `-xc` run will print out the stack trace on every exception, look for the one which says stack overflow
- Attempt to fix the space leak, confirm by rerunning
- Repeat until the test works with a small stack, typically `-K1K`.

# Space profiling can have nice graphs too!

add `-caf-all` to `ghc-flags` and then run as  
`./demo +RTS -hc -p -K100M`



Questions?

---

join us at [Papers We Love Kyiv](#) (@pwl\_kyiv)