FFPiH - Performance!

Florian Hofmann

08.07.2016

Prolog

About

- Florian Hofmann
- Mail: fho@f12n.de
- Blog: fho.f12n.de

Obligatory XKCD referenz

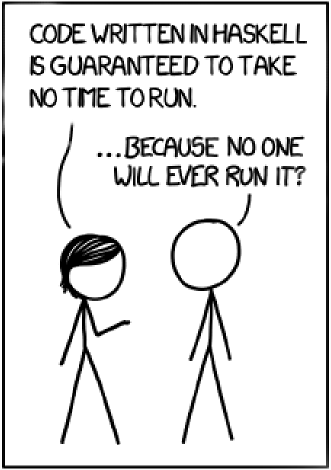

CODE WRITTEN IN HASKELL IS GUARANTEED TO TAKE NO TIME TO RUN.
...BECAUSE NO ONE WILL EVER RUN IT?

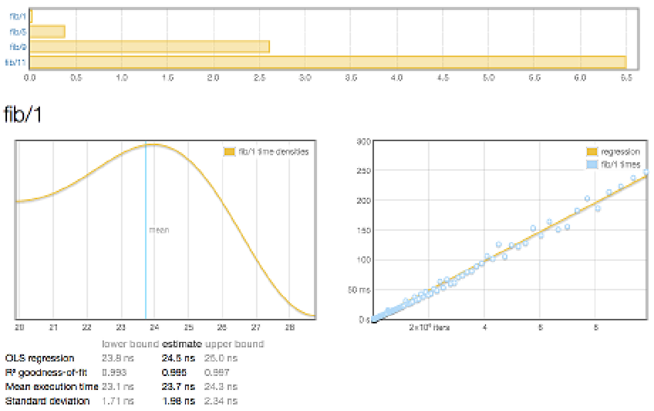
Figure 1: Adapted from XKCD #1312

Overview

- Benchmarking in Haskell
- Strictness/Laziness
- Boxing/Unboxing
- Inlining
- (just a little bit of) Core

Benchmarking / Criterion

Criterion Benchmark Report



fib/1

	lower bound	estimate	upper bound
OLS regression	23.9 ns	**24.5 ns**	25.0 ns
R^2 goodness-of-fit	0.993	**0.995**	0.997
Mean execution time	23.1 ns	**23.7 ns**	24.3 ns
Standard deviation	1.71 ns	**1.98 ns**	2.34 ns

Figure 2: Criterion HTML output

Setup

In *.cabal* file:

```
benchmark signal-bench
  type:             exitcode-stdio-1.0
  hs-source-dirs:   src, bench
  main-is:          MainBenchmarkSuite.hs
  build-depends:    base,
                    criterion,
                    random
  ghc-options:      -Wall
                    -O2
```

Run as:

```
stack bench --benchmark-arguments "-o filename.html"
```

Setup

In *MainBenchmarkSuite.hs*:

```haskell
import Criterion.Main

-- The function we're benchmarking.
fib :: Int -> Int
fib x = ...

-- Our benchmark harness.
main = defaultMain [
  bgroup "fib" [ bench "1"  $ whnf fib 1
               , bench "5"  $ whnf fib 5
               , bench "9"  $ whnf fib 9
               ]
  ]
```

Profiling

```
stack build --executable-profiling --library-profiling \
  --ghc-options="-fprof-auto -rtsopts"
stack exec -- program-exe +RTS -p -s
```

- Creates a file `program-exe.prof` in the current folder
- `-s` option give small report about runtime at program shutdown
- Profiling overhead is huge

Profiling Report

```
Wed Jun 29 16:33 2016 Time and Allocation Profiling Report

eventrate +RTS -s -p -RTS 2016-06-08-atis-trials/vp09-dotst

total time  =       44.01 secs   (44008 ticks @ 1000 us, 1
total alloc = 55,502,215,128 bytes  (excludes profiling ove

COST CENTRE            MODULE                          %time

decodeStreamWith       System.IO.Streams.Csv.Decode     50.3
handleToOutputStream.f System.IO.Streams.Handle         18.0
encodeRates            Main                             17.3
eventRate.go.s'        EventDriven.Rate                  4.3
eventRate.go           EventDriven.Rate                  2.4
contramap              System.IO.Streams.Combinators     1.1
```

Strictness

Strictness

- Haskell is lazy by default.
    - Allows some algorithms and datastructures to be written more efficient.
- But this results in problems, other languages don't face.
    - Spaceleaks
    - Bad intuition about runtime and space usage

A mean volunteer

```haskell
mean :: Fractional a => [a] -> a
mean xs = s / l
    where (s,l) = foldl go (0,0) xs
          go (a,b) x = (a+x,b+1)
```

-XBangPatterns

- Extension that allows us to use `!` (bangs) in pattern matches.
- Enabled by `-XBangPatterns` or `{-# LANGUAGE BangPatterns #-}`.

A (not so) mean volunteer

```
notSoMean:: Fractional a => [a] -> a
notSoMean xs = s / l
    where (s,l) = foldl' go (0,0) xs
          go (!a,!b) x = (a+x,b+1)
```

The difference

```
benchmarking mean
time                 39.84 ms   (39.49 ms .. 40.33 ms)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 40.02 ms   (39.83 ms .. 40.30 ms)
std dev              432.7 us   (272.0 us .. 679.1 us)

benchmarking notSoMean
time                 2.910 ms   (2.895 ms .. 2.925 ms)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 2.929 ms   (2.916 ms .. 2.954 ms)
std dev              56.65 us   (21.81 us .. 95.49 us)
```

- Success: ~13x faster

An alternative: strict datastructures

```haskell
data StrictTuple a b = ST !a !b

notSoMeanEither :: Fractional a => [a] -> a
notSoMeanEither xs = s / fromIntegral l
    where (ST s l) = foldl' go (ST 0 0) xs
          go (ST a b) x = (ST (a+x) (b+1))
```

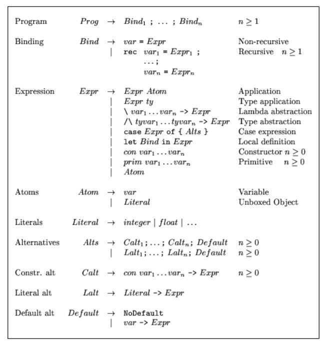
- Same speedup as with BangPatterns

Core

Core

- Core is a simplified version of Haskell
- Overview about external Core representation:
    - “An External Representation for the GHC Core Language” - Andrew Tolmach, Tim Chevalier and The GHC Team
    - https://downloads.haskell.org/~ghc/6.12.2/docs/core.pdf

(Hard-)Core



Program	*Prog*	→	$Bind_1$; … ; $Bind_n$	$n \geq 1$
Binding	*Bind*	→	*var* = *Expr*	Non-recursive
		|	`rec` var_1 = $Expr_1$;	Recursive $n \geq 1$
			… ;	
			var_n = $Expr_n$	
Expression	*Expr*	→	*Expr Atom*	Application
		|	*Expr ty*	Type application
		|	\\ $var_1 \ldots var_n$ -> *Expr*	Lambda abstraction
		|	/\\ $tyvar_1 \ldots tyvar_n$ -> *Expr*	Type abstraction
		|	`case` *Expr* `of` { *Alts* }	Case expression
		|	`let` *Bind* `in` *Expr*	Local definition
		|	*con* $var_1 \ldots var_n$	Constructor $n \geq 0$
		|	*prim* $var_1 \ldots var_n$	Primitive $n \geq 0$
		|	*Atom*	
Atoms	*Atom*	→	*var*	Variable
		|	*Literal*	Unboxed Object
Literals	*Literal*	→	*integer* | *float* | …	
Alternatives	*Alts*	→	$Calt_1$; … ; $Calt_n$; *Default*	$n \geq 0$
		|	$Lalt_1$; … ; $Lalt_n$; *Default*	$n \geq 0$
Constr. alt	*Calt*	→	*con* $var_1 \ldots var_n$ -> *Expr*	$n \geq 0$
Literal alt	*Lalt*	→	*Literal* -> *Expr*	
Default alt	*Default*	→	`NoDefault`	
		|	*var* -> *Expr*	

Figure 3: Syntax of the Core language

How to core

```
$ stack build --ghc-options "-ddump-to-file -ddump-simpl \
    -dsuppress-idinfo -dsuppress-coercions \
    -dsuppress-type-applications -dsuppress-uniques \
    -dsuppress-module-prefixes"
```

- `-ddump-simpl` enables (simplified) core output
- `-ddump-to-file` dumps the output to files
    - *stack:* `./.stack-work/dist/x86_64-linux/Cabal-1.22.5.0/ \`
      `build/prog/prog-tmp/src/Source.dump-simpl`
- `-ddump-suppress-*` removes lots of output to make it readable

Core

```
Rec {
$wgo :: [Double] -> Double# -> Int# -> (# Double#, Int# #)
$wgo = \ (w :: [Double]) (ww :: Double#) (ww1 :: Int#) ->
 case w of _ {
  [] -> (# ww, ww1 #);
  : y ys -> case y of _ { D# y1 -> $wgo ys (+## ww y1) (+#
 } end Rec }

mean05 :: [Double] -> Double
mean05 = \ (w :: [Double]) ->
 case $wgo w 0.0 0 of _ { (# ww1, ww2 #) ->
 case /## ww1 (int2Double# ww2) of ww3 {
  __DEFAULT -> D# ww3 }}
```

Core (the very brief version)

- Hashes are good, datatypes with hashes are *unboxed*
- each `case` is a *strict* evaluation
- each `let` is a lazy thunk
- constructors are applied in prefix notation

Unboxing

The riddling case

How much memory does this Haskell expression use?

```haskell
data IntPair = IP Int Int
```

(blatantly stolen from: Johan Tibell - ZuriHac2015 Performance)

Datatypes 101

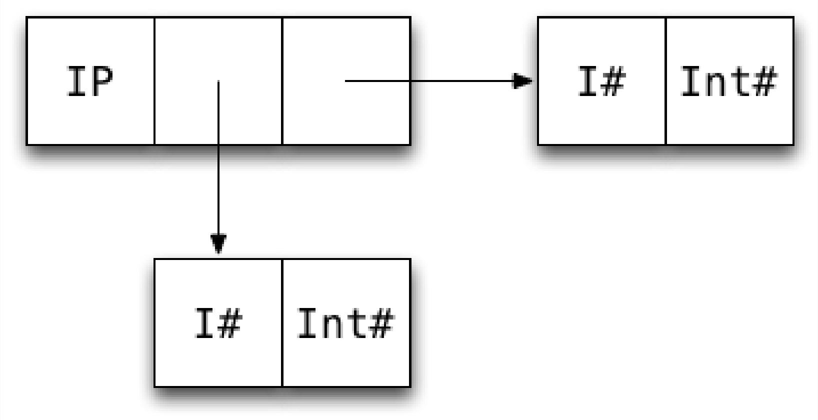


Figure 4: 7 machine words / 56 bytes on 64bit

Datatypes 102 - Unboxing

```
data IntPair =
  IP {-# UNPACK #-} !Int
     {-# UNPACK #-} !Int
```

Datatypes 102 - Unboxing

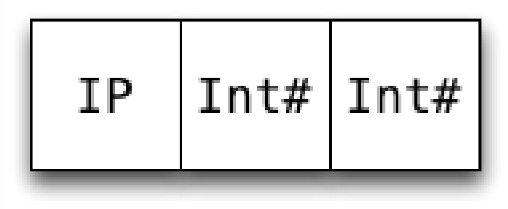


Figure 5: 3 machine words / 24 bytes on 64bit

Unboxing

- Unboxing/-packing uses Pragma *{-# UNPACK #-}*
- Generally improves performance
- no unboxing required
- reduces pointer count, improves cache locality

Unboxed Vectors

List Memory Layout

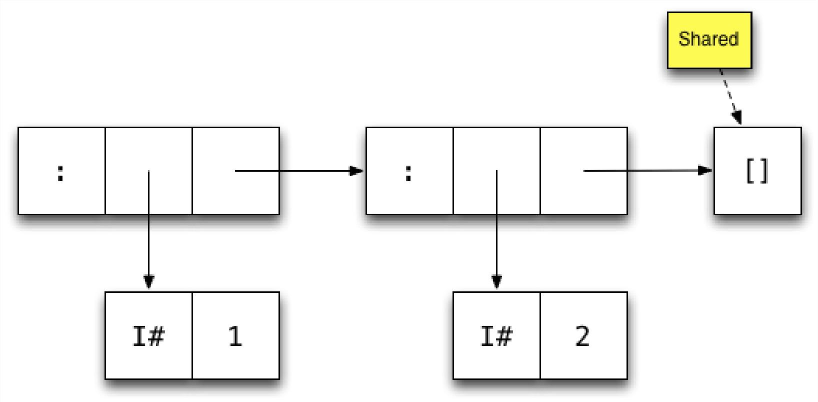


Figure 6: Lists in Memory

Solution: better datastructures

- `vector` package offers *C-style* zero-indexed arrays.
- `Data.Vector` stores references to elements in a plain array
- `Data.Vector.Unboxed` stores elements as a plain array.
  - Needs `Unboxed` instance, can be derived by GHC with a little help
- `Data.Vector.Storable` stores elements for exchange with foreign (C) programs
  - Needs `Storable` instance, *c-storable-deriving* package derives C compatible instances

A vectors mean

```haskell
mean06 :: (Fractional a, V.Unbox a) => V.Vector a -> a
mean06 v = V.sum v / fromIntegral (V.length v)

mean07 :: V.Vector Double -> Double
mean07 v = V.sum v / fromIntegral (V.length v)
```

Criterion report

```
benchmarking mean06
time                 2.949 ms   (2.922 ms .. 2.976 ms)
                     0.999 R²   (0.998 R² .. 1.000 R²)
mean                 2.914 ms   (2.902 ms .. 2.930 ms)
std dev              43.86 us   (32.44 us .. 72.83 us)

benchmarking mean07
time                 343.1 us   (340.9 us .. 345.4 us)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 345.4 us   (343.8 us .. 347.3 us)
std dev              5.789 us   (4.973 us .. 6.943 us)
```

Inlining

Inlining

- GHC inlines *small* functions by default, but only in modules
- `{-# INLINEABLE function #-}` allows GHC to inline over module borders
- `{-# INLINE function #-}` **forces** GHC to always inline this function

Some things to consider

- Use module export lists, this allows GHC to inline code that is not exported:

```
module Foo (bar,baz) where
```

- SPECIALIZE pragma makes GHC create specialized versions of a function:

```
foo a = a + 42
{-# SPECIALIZE foo :: Double -> Double #-}
```

Have my cake and eat it too

```
benchmarking mean06'
time                 339.4 us   (337.8 us .. 341.1 us)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 341.6 us   (340.1 us .. 343.8 us)
std dev              6.103 us   (4.610 us .. 8.465 us)
```

```
benchmarking mean07
time                 339.1 us   (336.9 us .. 341.4 us)
                     1.000 R²   (1.000 R² .. 1.000 R²)
mean                 339.9 us   (338.6 us .. 341.7 us)
std dev              4.897 us   (4.014 us .. 6.372 us)
```

Epilog

Random bits and pieces

- go functions, allows GHC to store data once (kind of a bug)

```haskell
bar a xs = go xs
  where go [] = 0
        go (x:xs) = a * x + go xs
```

- Use appropriate data-structures and algorithms
  - `Data.Vector` instead of `List`
  - `Data.Text` instead of `String`
  - Maybe use an alternative `Prelude`? (package: `basic-prelude`)

Random bits and pieces

- strict return from monadic functions (`$!`)

```haskell
foo = do
  x <- getData
  let x' = doComplexStuff x
  return $! x'  -- evaluates x' before returning
```

- don't use lazy IO, use `io-streams`, `pipes`, `conduit` instead.

Random bits and pieces

- GHC compile flags:
  - `-O2`: enable optimization
  - `-fexcess-precision`: faster floating point code (not *IEEE 754* compatible)
  - `-optc-O3`: enable optimizations in the C backend
  - `-optc-ffast-math`: allow the C backend to optimize floating point code more (see also the `fast-math` package)
  - `-fllvm`: use LLVM instead of GCC, may work better on numeric code

Random bits and pieces

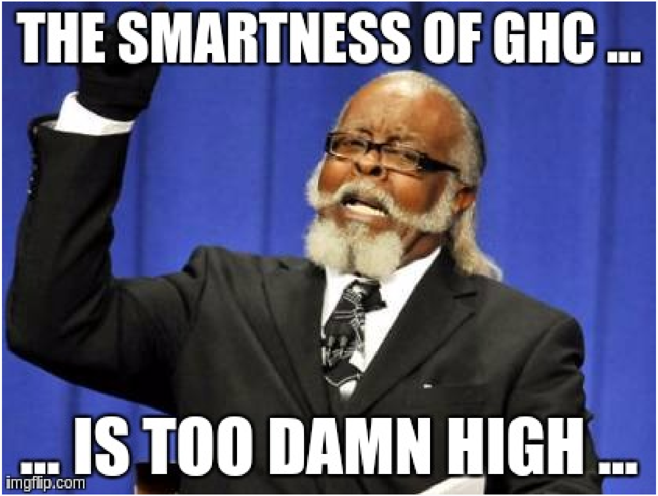


Figure 7: too smart

Conclusion

- Avoid boxing in hot loops:
  - Use *Unboxing*
  - Use *Strictness*
  - *Inlining* facilitates both
- Look out for non-strict accumulators
- Good guideline for datastructures: “lazy in the spine, strict in the leaves”

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%” - Donald Knuth

References

- ZuriHac2015 - Performance
- GHC User Guide