# Control.Monad.Writer

A Journal of Haskell Programming

## Write Haskell as fast as C: exploiting strictness, laziness and recursion

In a recent **mailing list thread** Andrew Coppin complained of poor performance with "nice, declarative" code for computing the mean of a very large list of double precision floating point values:

```
import System.Environment
import Text.Printf

mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)

main = do
    [d] <- map read `fmap` getArgs
    printf "%f\n" (mean [1 .. d])
```

Which, when executed, churns away for a while, but eventually runs out of memory:

```
$ ghc -O2 --make A.hs
$ time ./A 1e9
A: out of memory (requested 1048576 bytes)
./A 1e9  12.74s user 1.20s system 99% cpu 13.997 total
```

Well, it got 13 seconds into the job doing something. At least it was making progress.

While Andrew was citing this as an example of GHC Haskell being unpredictable (it's not an example of that, as we'll see), there is something here — this little program reveals a lot about the interaction between strictness, laziness and approaches to compiler optimisation in Haskell.

This post is about writing reliably fast code, and then how to write more naive code that is also reliably fast. In particular, how recursion consistently produces excellent code, competitive with heavily optimised C, and how to get similar results from higher order functions. Throughout this post I'll be using GHC 6.8.2, with -O2. Sometimes I'll switch to the C backend (-fvia-C -optc-O2). Also, I don't care about smarter ways to implement this — we're simply interested in understanding the compiler transformations involved in the actual loops involved.

## Understanding strictness and laziness

First, let's work out why this ran out of memory. The obvious hard constraint is that we request a list of 1e9 doubles: that is very, very large. It's 8 * 10^9 bytes, if allocated directly, or about 7.5G. Inside a list there is yet further overhead, for the list nodes, and their pointers. So no matter the language, we simply cannot allocate this all in one go without burning gigabytes of memory. The only approach that will work is to lazily generate the list somehow. To confirm this, we can use a strict list data type, just to see how far we get.

First, let's define a strict list data type. That is one whose head and tail are always fully evaluated. In Haskell, strictness is declared by putting bangs on the fields of the data structure, though it can also be inferred based on how values are used:

```
data List a = Empty
            | Cons !a !(List a)
```

This is a new data type, so we'll need some new functions on it:

```
length' :: List a -> Int
length' = go 0
    where
        go n Empty       = n
        go n (Cons _ xs) = go (n+1) xs

sum' :: Num a => List a -> a
sum' xs = go 0 xs
    where
        go n Empty       = n
        go n (Cons x xs) = go (n+x) xs

enumFromTo' :: (Num a, Ord a) => a -> a -> List a
enumFromTo' x y | x > y     = Empty
                | otherwise = go x
    where
        go x = Cons x (if x == y then Empty else go (x+1))
```

I've written these in a direct worker/wrapper transformed, recursive style. It's a simple, functional idiom that's guaranteed to get the job done.

So now we can compute the length and sum of these things, and generate one given a range. Our 'mean' function stays much the same, just the type changes from a lazy to strict list:

```
mean :: List Double -> Double
mean xs = sum' xs / fromIntegral (length' xs)
```

Testing this, we see it works fine for small examples:

```
$ time ./B 1000
500.5
./A 1000  0.00s user 0.00s system 0% cpu 0.004 total
```

But with bigger examples, it quickly exhausts memory:

```
$ time ./B 1e9
Stack space overflow: current size 8388608 bytes.
Use `+RTS -Ksize' to increase it.
./A 1e9  0.02s user 0.02s system 79% cpu 0.050 tota
```

To see that we can't even get as far as summing the list, we'll replace the list body with undefined, and just ask for the list argument to be evaluated before the body with a bang patterns (a strictness hint to the compiler, much like a type annotation suggests the desired type):

```
mean :: List Double -> Double
mean !xs = undefined

main = do
    [d] <- map read `fmap` getArgs
    printf "%f\n" (mean (enumFromTo' 1 d))
```

And still, there's no hope to allocate that thing:

```
$ time ./A 1e9
Stack space overflow: current size 8388608 bytes.
Use `+RTS -Ksize' to increase it.
./A 1e9  0.01s user 0.04s system 93% cpu 0.054 total
```

Strictness is not the solution here.

## Traversing structures and garbage collection

So why is the naive version actually allocating the whole list anyway? It was a lazy list, shouldn't it somehow avoid allocating the elements? To understand why it didn't work, we can look at what:

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)
```

actually means. To reason about the code, one way is to just look at the final, optimised Haskell GHC produces, prior to translation to imperative code. This reduced Haskell is known as "core", and is the end result of GHC's optimisations-by-transformation process, which iteratively rewrites the original source into more and more optimised versions.

Referring to the core is useful if you're looking for C-like performance, as you can state precisely, to the register level, what your program will do — there are no optimisations to perform that would confuse the interpretation. It is an entirely unambiguous form of Haskell, so we'll use it to clarify any uncertainties. (For everyday programming, a comfortable knowledge of laziness and strictness is entirely adequate, so don't panic if you don't read core!).

To view the core, we can use ghc -O2 -ddump-simpl, or the **ghc-core** tool. Looking at it for the naive program we see that 'mean' is translated to:

```
$wlen :: [Double] -> Int# -> Int#
...

$wsum'1 :: [Double] -> Double# -> Double#
...

main = ...
   shows (
    let xs :: [Double]
        xs = enumFromTo1 lvl3 d_a6h
    in
         case $wsum'1 xs 0.0 of {
             _ -> case $wlen xs 0 of {
                 z -> case ww_s19a /## (int2Double# z) of {
                      i -> D# i
   )
```

It looks like a lot of funky pseudo-Haskell, but its made up of very simple primitives with a precise meaning. First, length and sum are specialised — their polymorphic components replaced with concrete types, and in this case, those types are simple, atomic ones, so Double and Int are replaced with unlifted (or unboxed) values. That's good to know — unlifted values can be kept in registers.

We can also confirm that the input list is allocated lazily.

```
let xs :: [Double]
    xs = enumFromTo1 lvl3 d_a6h
in ..
```

The 'let' here is the lazy allocation primitive. If you see it on non-unboxed type, you can be sure that thing will be lazily allocated on the heap. Remeber: core is like Haskell, but there's only 'let' and 'case' (one for laziness, one for evaluation).

So that's good. It means the list itself isn't costing us anything to write. This lazy allocation also explains why we're able to make some progress before running out of memory resources — the list is only allocated as we traverse it.

However, all is not perfect, the next lines reveal:

```
         case $wsum'1 xs 0.0 of {
             _ -> case $wlen xs 0 of {
                 z -> case ww_s19a /## (int2Double# z) of {
                      i -> D# i
```

Now, 'case' is the evaluation primitive. It forces evaluation to the outermost constructor of the scrutinee — the expression we're casing on — right where you see it.

In this way we get an ordering of operations set up by the compiler. In our naive program, first the sum of the list is performed, then the length of the list is computed, until finally we divide the sum by the length.

That sounds fine, until we think about what happened to our lazily allocated list. The initial 'let' does no work, when allocating. However, the first 'sum' will traverse the entire list, computing the sum of its elements. To do this it must evaluate the entire huge list.

Now, on its own, that is still OK — the garbage collector will race along behind 'sum', deallocating list nodes that aren't needed anymore. However, there is a fatal flaw in this case. 'length' still refers to the head of the list. So the garbage collector cannot release the list: it is forced to keep the whole thing alive.

'sum', then, will allocate the huge list, and it won't be collected till after 'length' has started consuming it — which is too late. And this is exactly as we observed. The original poster's unpredictably is an entirely predictable result if you try to traverse a 7G lazy list twice.

Note that this would be even worse if we'd been in a strict language: the initial 'let' would have forced evaluation, so you'd get nowhere at all.

Now we have enough information to solve the problem: make sure we make only one traversal of the huge list. so we don't need to hang onto it.

## Lazy lists are OK

The simple thing to do then, and the standard functional approach for the last 40 years of functional programming is to write a loop to do both sum and length at once:

```
mean :: [Double] -> Double
mean = go 0 0
    where
        go :: Double -> Int -> [Double] -> Double
        go s l []     = s / fromIntegral l
        go s l (x:xs) = go (s+x) (l+1) xs
```

We just store the length and sum in the function parameters, dividing for the mean once we reach the end of the list. In this way we make only one pass. Simple, straight forward. Should be the standard approach in the Haskell hackers kit.

By writing it in an obvious recursive style that walks the list as it is created, we can be sure to run in constant space. Our code compiles down to:

```
$wgo :: Double# -> Int# -> [Double] -> Double#
$wgo x y z =
    case z of
      []             -> /## x (int2Double# y);
      x_a9X : xs_a9Y ->
         case x_a9X of
            D# y_aED -> $wgo (+## x y_aED) (+# y 1) xs_a9Y
```

We see that it inspects the list, determining if it is an empty list, or one with elements. If empty, return the division result. If non-empty, inspect the head of the list, taking the raw double value out of its box, then loop, On a small list:

```
$ time ./B 1000
500.5
./B 1000  0.00s user 0.00s system 0% cpu 0.004 total
```

On the 7 gigabyte list:

```
$ time ./B 1e9
500000000.067109
./B 1e9  68.92s user 0.19s system 99% cpu 1:09.57 total
```

Hooray, we've computed the result! The lazy list got us home.

But can we do better? Looking at the GC and memory statistics (run the program with +RTS -sstderr ) we can see how much work was going on:

```
            24,576 bytes maximum residency (1 sample(s))

      %GC time       1.4%  (4.0% elapsed)
  Alloc rate    2,474,068,932 bytes per MUT second
  Productivity           98.6% of total user
```

What does this tell us? Firstly, garbage collection made up a tiny fraction of the total time (1.4%), meaning the program was doing productive thing 98.6% of the time — that's very good.

Also, we can see it ran in constant space: 24k bytes maximum allocated in the heap. And the program was writing and releasing these list nodes at an alarming rate. 2G a second (stunning, I know). Good thing allocation is cheap.

However, this does suggest that we can do better — much better — by avoiding any list node allocation at all and keeping off the bus. We just need to prevent list nodes from being allocated at all — by keeping only the current list value in a register — if we can stay out of memory, we should see dramatic improvements.

## Recursion kicks arse

So let's rewrite the loop to no longer take a list, but instead, the start and end values of the loop as arguments:

```
mean :: Double -> Double -> Double
mean n m = go 0 0 n
    where
        go :: Double -> Int -> Double -> Double
        go s l x | x > m      = s / fromIntegral l
                 | otherwise  = go (s+x) (l+1) (x+1)

main = do
    [d] <- map read `fmap` getArgs
    printf "%f\n" (mean 1 d)
```

We've moved the list lower and upper bounds into arguments to the function. Now there are no lists whatsover. This is a fusion transformation, instead of two separate loops, one for generation of the list, and one consuming it, we've fused them into a single loop with all operations interleaved. This should now avoid the penalty of the list memory traffic. We've also annotated the types of the functions, so there can be no ambiguity about what machine types are used.

Looking at the core:

```
$wgo_s17J :: Double# -> Int# -> Double# -> Double#
$wgo_s17J x y z =
  case >## z m of
    False ->
      $wgo_s17J
        (+## x z)
        (+#  y 1)
        (+## z 1.0);
    True -> /## x (int2Double# y)
```

it is remarkably simple. All parameters are unboxed, the loop is tail recursive, and we can expect zero garbage collection, all list parameters in registers, and we'd expect excellent performance. The >## and +## are GHC primitives for double comparison and addition. +# is normal int addition.

Let's compile it with GHC's native code generator first (-O2 -fexcess-precision):

```
$ time ./C 1e9
500000000.067109
./C 1e9  3.75s user 0.00s system 99% cpu 3.764 total
```

Great. Very good performance! The memory statistics tell a similar rosy story:

```
     20,480 bytes maximum residency (1 sample(s))
      %GC time       0.0%  (0.0% elapsed)
  Alloc rate    10,975 bytes per MUT second
  Productivity 100.0% of total user
```

So it ran in constant space, did no garbage collection, and was allocating only a few bytes a second. Recursion is the breakfast of champions.

And if we use the C backend to GHC: (-O2 -fexcess-precision -fvia-C -optc-O2), things get seriously fast:

```
$ time ./C 1e9
500000000.067109
./C 1e9  1.77s user 0.00s system 99% cpu 1.776 total
```

Wow, much faster. GCC was able to really optimise the loop GHC produced. Good job.

Let's see if we can get this kind of performance in C itself. We can translate the recursive loop directly into a for loop, where function arguments becomes the variables in the loop:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

        double d = atof(argv[1]);

        double n;
        long long a; // 64 bit machine
        double b;

        // go_s17J :: Double# -> Int# -> Double# -> Double#
        for (n = 1,
             a = 0,
             b = 0; n <= d; b+=n,
                            n++,
                            a++)
            ;

        printf("%f\n", b / a);

        return 0;
}
```

That was quite straight forward, and the C is nicely concise. It is interesting to see how function parameters are updated in place explicitly in the C code, while its implicitly reusing stack slots (or registers) in the functional style. But they're really describing the same code. Now, compiling the C without optimisations:

```
$ time ./a.out 1e9
500000000.067109
./a.out 1e9  4.72s user 0.00s system 99% cpu 4.723 total
```

Ok. That's cool. We got the same results, and optimised Haskell beat unoptimsed C. Now with -O, gcc is getting closer to catch GHC:

```
$ time ./a.out 1e9
500000000.067109
./a.out 1e9  2.10s user 0.00s system 99% cpu 2.103 total
```

And with -O2 it makes it in front by a nose:

```
$ time ./a.out 1e9
500000000.067109
./a.out 1e9  1.76s user 0.00s system 99% cpu 1.764 total
```

gcc -O2 wins the day (barely?), just sneaking past GHC -O2, which comes in ahead of gcc -O and -Onot.

We can look at the generated assembly (ghc -keep-tmp-files) to find out what's really going on.. GCC generates the rather nice:

```
.L6:
    addsd   %xmm1, %xmm2
    incq    %rax
    addsd   %xmm3, %xmm1
    ucomisd %xmm1, %xmm0
    jae .L6

.L8:
    cvtsi2sdq   %rax, %xmm0
    movl    $.LC2, %edi
    movl    $1, %eax
    divsd   %xmm0, %xmm2
```

Note the very tight inner loop, and final division. Meanwhile, GHC produces, with its native code backend:

```
s1bn_info:
  ucomisd 5(%rbx),%xmm6
  ja .Lc1dz
  movsd %xmm6,%xmm0
  addsd .Ln1dB(%rip),%xmm0
  leaq 1(%rsi),%rax
  addsd %xmm6,%xmm5
  movq %rax,%rsi
  movsd %xmm0,%xmm6
  jmp s1bn_info

.Lc1dz:
  cvtsi2sdq %rsi,%xmm0
  divsd %xmm0,%xmm5
```

Quite a bit more junk in the inner loop, which explains the slowdown with -fasm. That native code gen needs more work for competitve floating point work. But with the GHC C backend:

```
s1bn_info:
  ucomisd     5(%rbx), %xmm6
  ja  .L10
  addsd       %xmm6, %xmm5
  addq        $1, %rsi
  addsd       .LC1(%rip), %xmm6
  jmp s1bn_info

.L10:
  cvtsi2sdq   %rsi, %xmm7
  divsd       %xmm7, %xmm5
```

Almost identical to GCC, which explains why the performance was so good!

## Some lessons

Lesson 1: To write predictably fast Haskell — the kind that competes with C day in and out — use tail recursion, and ensure all types are inferred as simple machine types, like Int, Word, Float or Double that simple machine representations. The performance is there if you want it.

Lesson 2: Laziness has an overhead — while it allows you to write new kinds of programs (where lists may be used as control structures), the memory traffic that results can be a penalty if it appears in tight inner loops. Don't rely laziness to give you performance in your inner loops.

Lesson 3: For heavy optimisation, the C backend to GHC is still the way to go. Later this year a new bleeding edge native code generator will be added to GHC, but until then, the C backend is still an awesome weapon.

In a later post we'll examine how to get the same performance by using higher order functions.

Uncategorized

fusion, performance

May 6, 2008

14 Comments

# 14 comments

Do you want to comment?

Comments RSS and TrackBack URI

1.
***Itkovian***
Cool post. However, the real case one should look at is when the list comes from a file, and cannot be 'generated' by peeking at the final element. In that case, the recursion with the allocation would be the way forward, no?

Permalink
*January 21, 2009* ***10:26 am***

2.
***Stacy***
I believe the intent of the original problem was to calculate the mean of an arbitrary list of doubles not of a list [1..n] for which there's a trivial non recursive solution 1/2(n+1).

Permalink
*January 23, 2009* ***4:51 pm***

3.
***George Giorgidze***
For some reason I can't explain, I am not able to reproduce Don's results on my system. Unfortunately, in my experiments GCC wins with the huge advantage:

> gcc -Wall -O3 -o c_mean src/mean.c
> ghc -Wall -O3 -o h_mean src/Mean.hs

> time ./c_mean 1e9
500000000.500000
real 0m2.611s

> time ./h_mean 1e9
5.00000000067109e8
real 0m17.968s

> ghc –version
The Glorious Glasgow Haskell Compilation System, version 6.10.1

> uname -a
Linux 2.6.28-ARCH #1 SMP PREEMPT i686 Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz GenuineIntel GNU/Linux

I played with various GHC options (including -fvia-C) but without any success.

Any comments and suggestions will be very much appreciated. Can anyone else confirm the similar behaviour?

Understanding this results will help me to improve some of my own numerical Haskell code.

Cheers, George

Permalink
*March 9, 2009* ***12:08 am***

4.
*dons00*
@George

The absolutely only way to see what is going on is to look at the assembly produced. This is highly dependent on which GCC version you use. In this case,

gcc (GCC) 4.3.3

The other issue I see is that you're using different flags:

gcc -O2

and

ghc -O2 -fvia-C -optc-O3

Then check the resulting assembly (e.g. with the ghc-core tool):

```
s1u7_info:
ucomisd 5(%rbx), %xmm6
ja .L19
addsd %xmm6, %xmm5
addq $1, %rsi
addsd .LC0(%rip), %xmm6
jmp s1u7_info

.L19:
cvtsi2sdq %rsi, %xmm0
divsd %xmm0, %xmm5
```

Permalink
*March 9, 2009* ***12:14 am***

5.
***George Giorgidze***
Hi Don,

Thanks for your timely reply.

I am using exactly the same version of GCC (4.3.3 the one which comes with Arch Linux) and tried exactly the same flags. Unfortunately, results for GHC got even worse.

```
> gcc -O2 -o c_mean src/mean.c
> ghc -O2 -fvia-C -optc-O3 -o h_mean src/Mean.hs

> time ./c_mean 1e9
500000000.500000
real 0m2.572s

> time ./h_mean 1e9
5.00000000067109e8
real 0m41.289s
```

You are right I should dig into the assembly codes and I am digging now.

Meanwhile, can you confirm that you get the similar (to your original post) results with newer versions of GHC. Results in your post are obtained with GHC 6.8.

Once again thanks for your excellent post and your help.

Cheers, George

Permalink

*March 9, 2009* ***1:07 am***

6.
*dons00*
@George

Yes, the assembly above was obtained today (2009-03-08) with GHC 6.10, and GCC 4.3.3 on Linux x86_64. Runtimes are still much the same:

GCC -O2 1.966s
GHC -O2 -fvia-C -optc-O3 2.080

Are you sure you're using the non-list based version?

Permalink
*March 9, 2009* ***1:14 am***

7.
***George Giorgidze***
I am sure. Here is the code:

```
module Main where

import System.Environment

mean :: Double -> Double -> Double
mean n m = go 0 0 n
where
go :: Double -> Int -> Double -> Double
go s l x | x > m = s / fromIntegral l
| otherwise = go (s+x) (l+1) (x+1)

main :: IO ()
main = do
[d] <- map read `fmap` getArgs
print (mean 1 d)
```

Permalink
*March 9, 2009* ***1:29 am***

8.
*dons00*
Usual benchmarking rules apply: if you're out by a factor of 20x, then you're doing something wrong.

Check carefully it is being compiled properly, that the flags you think it is using are correct, and if all else fails, look at the generated code (the ghc-core tool is helpful here).

Permalink
*March 9, 2009* ***1:36 am***

9.

***George Giorgidze***

Using ghc-core I had a look at inner loop instructions (I am not expert in this stuff, I hope i got it right). I was looking for a division instruction and for the preceding loop.

C version:

```
.L9:
fxch %st(2)
fxch %st(1)
.L4:
fadd %st(1), %st
fxch %st(1)
addl $1, %edx
fadds .LC0
fxch %st(2)
fucom %st(2)
fnstsw %ax
sahf
jae .L9
fstp %st(0)
fstp %st(1)
pushl %edx
fildl (%esp)
addl $4, %esp
.L3:
fdivrp %st, %st(1)
movl $.LC2, (%esp)
fstpl 4(%esp)
call printf
```

Haskell:

```
.type s1rK_info, @function
# 97 "/tmp/ghc10380_0/ghc10380_0.hc" 1
# 0 "" 2
fldl 12(%ebp)
fstpl 32(%esp)
fldl 32(%ebp)
fstpl 24(%esp)
fldl 32(%esp)
fldl 24(%esp)
fxch %st(1)
fucom %st(1)
fnstsw %ax
fstp %st(1)
sahf
ja .L12
```

```
fld %st(0)
movl 8(%ebp), %eax
fadds .LC0
addl $1, %eax
movl %eax, 8(%ebp)
fstpl 48(%esp)
fldl (%ebp)
fstpl 16(%esp)
faddl 16(%esp)
fstpl 56(%esp)
fldl 48(%esp)
fstpl 12(%ebp)
fldl 56(%esp)
fstpl (%ebp)
jmp s1rK_info
.p2align 4,,7
.p2align 3
.L12:
fstp %st(0)
fildl 8(%ebp)
fstpl 40(%esp)
fldl (%ebp)
addl $40, %ebp
fstpl 8(%esp)
fldl 8(%esp)
fdivl 40(%esp)
fstpl 64(%esp)
fldl 64(%esp)
fstpl 56(%ebx)
movl (%ebp), %eax
jmp *%eax
.size s1rK_info, .-s1rK_info
```

It seems to me that Haskell loop has lots of un-necessary fstp/fld (storing and loading of floating points) in the loop. However, in the core worker function for mean uses only unboxed types.

Cheers, George

P.S. I liked this ghc-core tool.

Permalink
*March 9, 2009* ***2:43 am***

10.
***George Giorgidze***
If used with -fexcess-precision some of the fstp/fld (storing and loading of floating points) instructions are removed from the loop, but not all of them:

```
.type s1qL_info, @function
```

```
# 169 "/tmp/ghc5427_0/ghc5427_0.hc" 1
# 0 "" 2
fldl 16(%ebp)
fldl 1(%esi)
fxch %st(1)
fucom %st(1)
fnstsw %ax
fstp %st(1)
sahf
ja .L21
fld1
fldl 8(%ebp)
fadd %st(1), %st
fldl (%ebp)
fadd %st(3), %st
fxch %st(3)
faddp %st, %st(2)
fxch %st(1)
fstpl 16(%ebp)
fstpl 8(%ebp)
fstpl (%ebp)
jmp s1qL_info
.p2align 4,,7
.p2align 3
.L21:
fstp %st(0)
fldl (%ebp)
fdivl 8(%ebp)
addl $24, %ebp
fstpl 56(%ebx)
movl (%ebp), %eax
jmp *%eax
```

This makes the Haskell program a lot faster:

```
> ghc -O2 -fexcess-precision -fvia-C -optc-O2 -o h_mean src/Mean.hs
> time ./h_mean 1e9
> real 0m16.634s
```

By inlineing the mean function into the main function things get even more faster:

```
{-# INLINE mean #-}
mean :: Double -> Double -> Double
…

> ghc -O2 -fexcess-precision -fvia-C -optc-O2 -o h_mean src/Mean.hs
> time ./h_mean 1e9
> real 0m9.625s
```

By replacing the length counter of type Int with type Double things get even more faster:

```
go :: Double -> Double -> Double -> Double
go s l x | x > m = s / l
| otherwise = go (s + x) (l + 1) (x + 1)
```

```
> ghc -O2 -fexcess-precision -fvia-C -optc-O2 -o h_mean src/Mean.hs
> time ./h_mean 1e9
> real 0m4.842s
```

To be honest I do not really understand (yet) why the last two transformations speeded the Haskell version up. In the case of C version the last transformation had no effect on its performance.

To summarise, hear are the final results (for now) on my system:

```
> ghc -O2 -fexcess-precision -fvia-C -optc-O2 -o h_mean src/Mean.hs
> time ./h_mean 1e9
> real 0m4.842s
```

```
> gcc -O2 -o c_mean src/mean.c
> time ./c_mean 1e9
> real 0m2.770s
```

So, on my system C remains about twice as fast. For now, I am not able to make the Haskell version any faster. Any pointers in this direction will be very much appreciated. In particular, how can I get rid off the remaining fstp/fld instructions in the Haskell version.

I should also note that Don uses x86_64 architecture and I am using i686 (though my CPU, see above, has 64 bit support). I am not sure but this might be the main reason of different results. I do not have access to x86_64 machine to confirm this doubt.

Maybe it is time to upgrade my Arch Linux to x86_64 version.

Anyway, thanks for your post and helpful comments on my struggles. I learned a lot from this example.

Cheers, George

Permalink
*March 10, 2009* ***4:28 pm***

11.
***Don Stewart***
@George

Yes, looks like on 32 bits GCC is being smarter with the floating point generated from C, compared to the Haskell stuff. While on x86_64 it does the same (better) thing.

Permalink
*March 10, 2009* ***4:42 pm***

# Trackbacks

1. Using Mutable Arrays for Faster Sorting « Haskellville
2. Can GHC really never inline map, scanl, foldr, etc.? | Ngoding
3. How to: Reading GHC Core | SevenNet