Fastest Lambda First

Neil Mitchell

www.cs.york.ac.uk/~ndm/

The Problem

- Count the number of lines in a file
  - “” = 0
  - “test” = 1
  - “test\n” = 1
  - “test\ntest” = 2
- Read from the console
  - Using *getchar* only
  - No buffering

The Haskell

main = print . length . lines =<< getContents

- getContents :: IO String
- lines :: String $\rightarrow$ [String]
- length :: [a] $\rightarrow$ Int
- print :: Show a $\Rightarrow$ a $\rightarrow$ String

The C

```
int main() {
    int count = 0, last_newline = 1, c;
    while ((c = getchar()) != EOF) {
        if (last_newline) count++;
        last_newline = (c == '\n');
    }
    printf("%i\n", count);
    return 0;
}                        /* Is this correct? */
```

The Results

Disclaimer Slide

- Uses GHC as a backend
  - GHC does some really cool optimisation
  - Inlining, strictness, unboxing
- Only one benchmark presented
  - Promising results on others, but not enough yet

Other Benchmarks

- Three results
  - wc -c 13% faster GHC, 3% slower C
  - wc -l 47% faster GHC, 2% slower C
  - wc -w 70% faster GHC, 20% slower C
- All very similar programs…

Overview

- Different approach
- First order code
- First order code without data
- Termination
- What could be improved
- Conclusion

Whole program analysis

- Look at all the code at once
- Done by *a few* compilers (MLton, JHC)
- Usually compilation is *really* slow

- Linking is whole-program
- Mine is quite quick

Bullets versus a nuclear bomb

- Most (all?) optimising compilers use “bullets”
  - Small, targeted transformations
  - Hit programs with a hail of bullets
- I use one single optimisation
  - No issues of “enabling transformations”
  - No optimisation “dials”
  - No “swings and roundabouts”

Alpha Renaming

- Some optimisers rely on special names
  - foldr/build
  - stream/unstream
- Achieves good practical results
  - Limits what can be optimised well
  - Requires functions to be defined unnaturally
  - They tend to go wrong (take in GHC 6.6)

First Order Haskell

- Remove all lambda abstractions (lambda lift)
- Leaving only partial application/currying

```
odd = (.) not even
(.) f g x = f (g x)
```

- Generate templates (specialised fragments)

Oversaturation

f x y z, where arity(f) < 3

main = odd 12

<odd _> x = (.) not even x

main = <odd _> 12

Undersaturation

f x (g y) z, where arity(g) > 1

<odd _> x = (.) not even x

<(.) not even _> x = not (even x)

<odd _> x = <(.) not even _> x

Special Rules

let z = f x y, where arity(f) > 2 (let-under)

- inline z, after sharing x and y

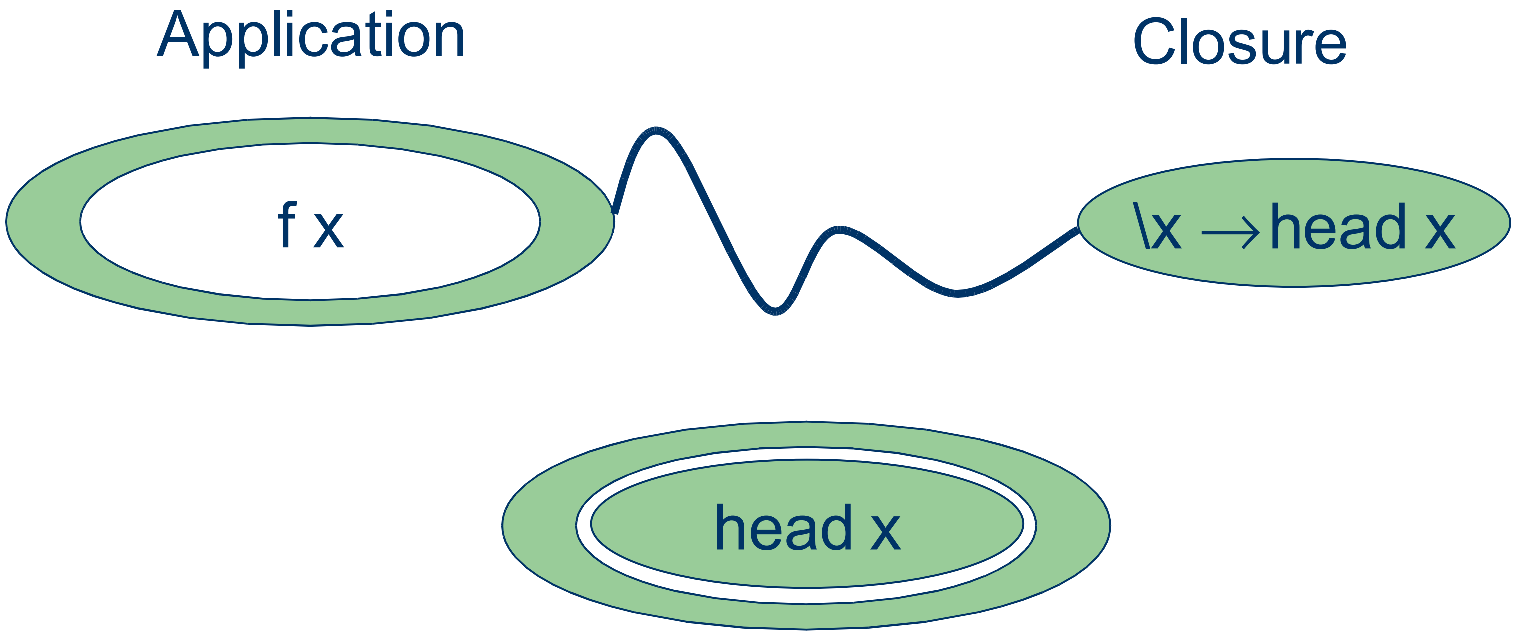
d = Ctor (f x) y, where arity(f) > 1 (ctor-under)

- inline d
- The “dictionary” rule

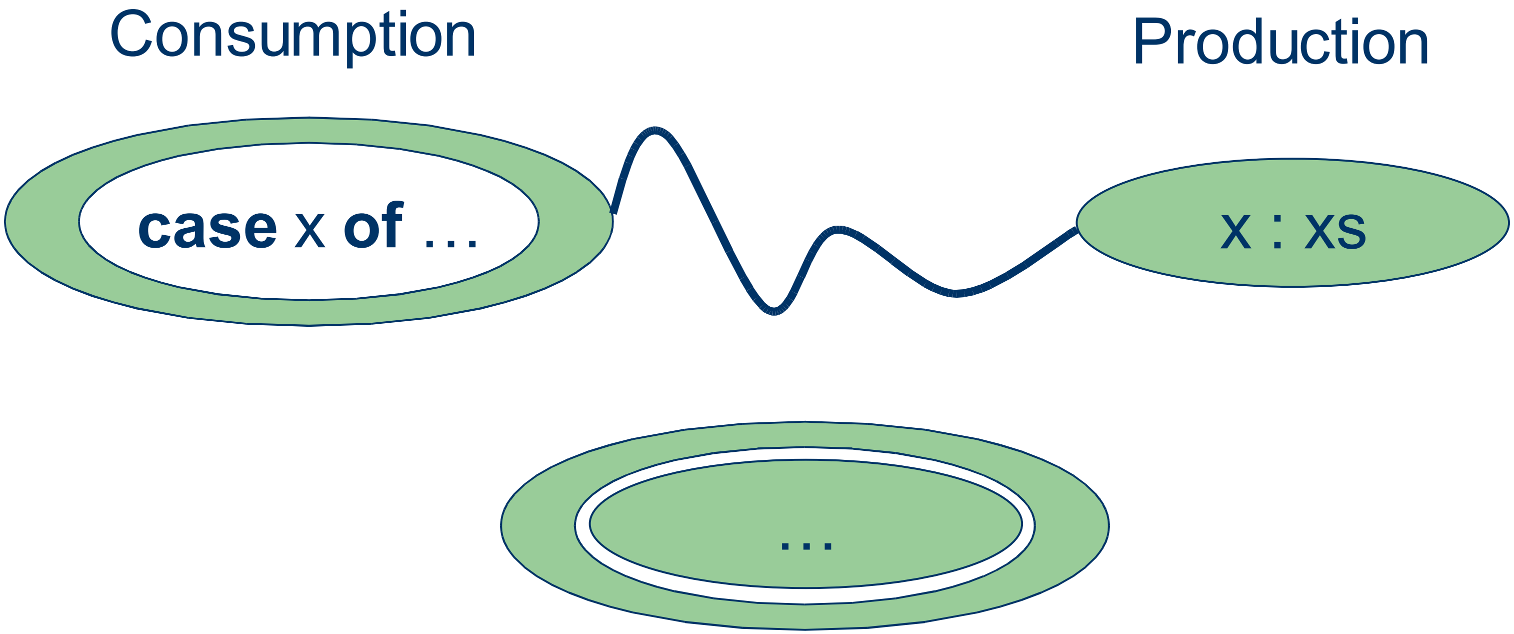
Standard Rules

- **let** x = (**let** y = z **in** q) **in** … (let/let)
- **case** (**let** x = y **in** z) **of** … (case/let)
- **case** (**case** x **of** …) **of** … (case/case)
- (**case** x **of** …) y z (app/case)
- **case** C x **of** … (case/ctor)

Removing functions



Removing data



Church Encoding

```
data List a =                    nil       = \n c → n
    Nil                          cons x xs = \n c → c x xs
  | Cons a (List a)

len x = case x of                len x = x
  Nil → 0                          0
  Cons y ys → 1 + len ys           (\y ys → 1 + len ys)
```

Optimisation Algorithm

1. Remove higher-order functions
2. Church encode
3. Remove higher-order functions

Proof: It doesn’t work

- A program has no data, and no functions
- Implies its not Turing complete!

- Linear Bounded Turing Machine
- Therefore, removing HO cannot be perfect

Failing Example

```
showPosInt x = f x “”
f 0  acc = acc
f i   acc = f (i / 10) (c:acc)
   where c = ord ‘0’ + (i % 10)
```

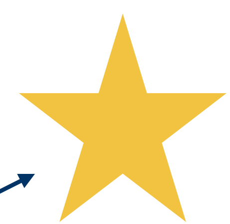
- Requires a buffer $O(\log_{10} n)$
- Cannot be removed automatically

Failing pleasantly

- Keep running
- At some point, stop
  - 1000 new functions created
  - 100 based on a particular function
  - Some particular name recurring
- Leaves higher-order functions around

Failing Church Encoding

- Church encoding requires rank-2 types
  - Cannot be inferred automatically
  - Makes some things more complex
- Why not merely “pretend” Church Encode
  - Failure is now left-over data
  - Much more pleasant



Pretend we are Church encoding

Summing the Integers

```
main n = sum (range 0 n)

sum xs = case xs of
                  [] → 0
                  (y:ys) → y + sum ys

range i n = if i > n then [] else i : range (i+1) n
```

Undersaturation of Data

- A constructor is higher-order

main n = sum (range 0 n)

<sum (range#2)> i n = **case** range i n **of** …

main n = <sum (range#2)> 0 n

Oversaturation of Data

- A case is an application

case range i n **of** {[] → 0; (y:ys) → y + sum ys}

<**case** range#2 {[] → 0; (y:ys) → y+sum ys}> i n =

if i > n **then** 0 **else** i + sum (range (i+1) n)

Final Result

```
main n       = sum' 0 n
sum' i n     = range' i n
range' i n   = if i > n then 0 else i + sum' (i+1) n
```

- All constructors have disappeared
- First-order with Church encoding

Special Cases

let x = C y z

- inline x, after sharing y and z

let x = f y z, where f produces data

- inlining may break sharing
- only if *one* use of x

What isn’t Optimised?

- This optimisation does a lot
- But doesn’t always produce optimal code

- What can we do better?
  - Ignore “better algorithms”

Call overhead

f1 x y = f2 x y

f2 x y = f3 y x

f3 y x = g x + y

- My optimisation gives loads of these!

Again, GHC is good at this

Strictness/Boxing

- Lazy evaluation requires "thunks"
- Strictness avoids these thunks

- Int is box stored in the heap
- Int# is more like a C int

Can cause space leaks

Sharing/lets

g (f x) (f x) $\Rightarrow$ **let** y = f x **in** g y y

- Common sub expression

map (g 100) ys

g x y = f x + y

- Strength reduction

Constant movement

```
countLines xs = count '\n' xs
count n (x:xs) | n == x       = 1 + count xs
               | otherwise  = count n xs
```

- This one remains in linecount example
- Should make the Haskell faster

Can Haskell beat C?

- A question of abstraction
  - In C, abstraction is painful
  - For linecount, not worth it

- Haskell can remove abstraction better than C
  - Won’t win on micro-benchmarks (may draw)
  - *May* win on real programs

http://shootout.alioth.debian.org/

Faster than C

print . sum . map readInt . lines =<< getContents

readInt :: Int → String

- Haskell can optimise sum/readInt
- C can't optimise between them

- NB. Not actually tried, yet…

More Benchmarks

- Needs refactoring
  - Some transformations in Yhc.Core
  - Some in the optimiser
  - Don’t glue together nicely
- GHC sometimes “over-optimises”
  - Turns getchar into a constant!
  - Need to integrate with GHC’s IO Monad

Conclusion

- Haskell can be made faster
  - Nearly the speed of C (sometimes)
  - But always more beautiful
- You can't draw conclusions from small benchmarks