λ Haskell With Go Faster Stripes

Neil Mitchell

λ Catch: Project Overview

- Catch checks that a Haskell program won’t raise a pattern match error
  - head [] = error “no elements in list”
  - Infer preconditions, postconditions
- Lots of progress
  - Mainly in the details
  - Nothing both *new* and *exciting*

λ The Catch Pipeline

1. Haskell source code
2. Core Haskell language – using Yhc
3. Haskell Intermediate Little Language
4. Transform to First Order HILL
5. Analysis

Higher Order Code

- A *function* is passed around as a *value*

```
head (x:xs) = x

map f [] = []
map f (x:xs) = f x : map f xs

main x = map head x
```

λ Higher Order, Point Free

- Point free/pointless code
- Does not mention the data values

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

even x = not (odd x)
```

Step 1: Arity Raise

- If a function can take more arguments, give it more!
- `(.)` takes 3 arguments, `even` gives `(.)` 2, therefore `even` takes 1

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

Step 2: Specialise

- If a function is passed higher order, generate a version with that argument frozen in:

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

Fall back plan...

- Reynolds Style Defunctionalisation
- Generate a data value for each function

```
data Func = Not | Odd | …
ap Not x = not x
ap Odd x = odd x
…
```

λ First Order HILL

- We now have First Order HILL
- The analysis is now happy

- But have we got faster code?
  - Reynold's style defunc is slow, but rare
  - Longer code, not necessarily slower

Reordering the operations

1. Arity raising
2. Reynold’s Style Defunc
3. Specialisation

- Now both functions and data are specialised!

The Competition

- GHC – Glasgow Haskell Compiler
- Optimising compiler for Haskell
- A *lot* of work has been done with GHC
- Speed competes with C!
- Based on inlining

Inlining vs Specialisation

```
ex1 = cond True 0 1

cond x t f =
    case x of
        True  -> t
        False -> f
```

Inlining

```
ex1 = case True of
          True  -> 0
          False -> 1

ex1 = 0
```

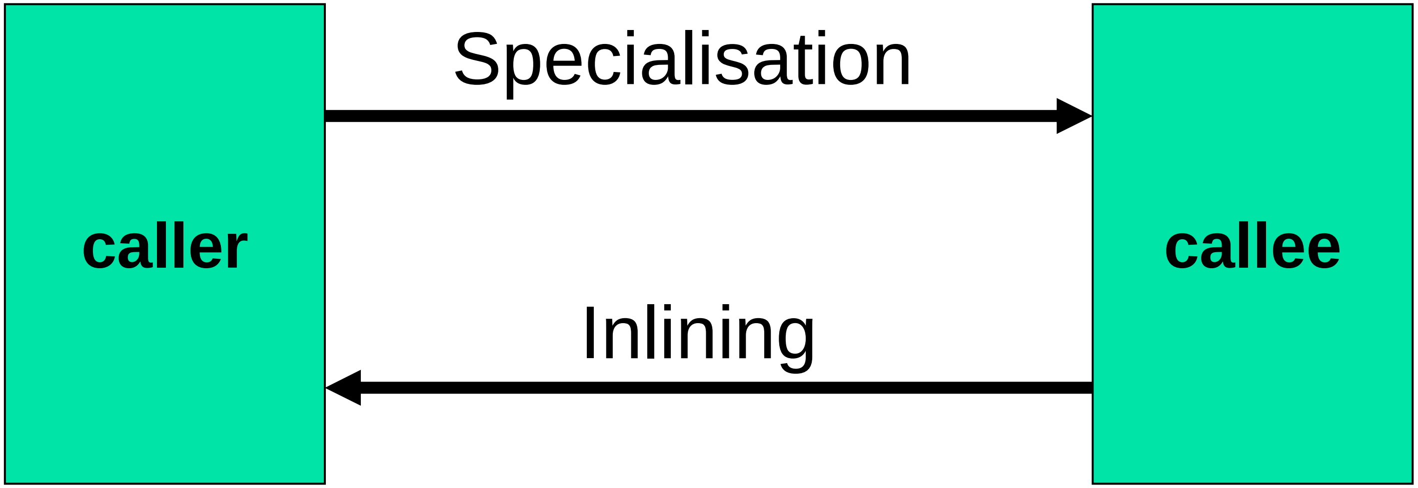
Specialisation

```
ex1 = cond<True> 0 1
cond<True> t f = t
```

`Cond<True>` is now just a “forwarder”, so is inlined

```
ex1 = 0
```

Inlining vs Specialisation



`… (f x) …`   `f x = …`

Termination condition

- Inlining
  - Do not inline recursive groups

- Specialisation
  - Based on types
  - `(1,'a':'b':[]):(3,[]):(4,[])`

Another few examples

```
map f [] = []
map f (x:xs) = f x : map f xs

ex2 f = map f []
ex3 x = map head x
```

- Inlining fails both of these*!

* Do not try this at home...

Specialisation

```
map<[]> f = []
ex2 f = map<[]> f
ex2 f = []
```

```
map<head> [] = []
map<head> (x:xs) = head x : map<head> xs
ex3 x = map<head> x
```

Specialisation Disadvantages

- Works best with *whole program*
  - Computers are now much faster
  - Does this really matter?
- Not as well studied
- Code blow up (in practice, small)

- Can use with inlining!

Pick a random benchmark...

- Calculate the n^th prime number
- In the standard `nofib` benchmark suite
- Lots of list traversals
- Quite a few higher order functions
- About 15 lines long

- Let's compare!

λ Executing HILL

- HILL is still very Haskell like
- Take the fastest Haskell compiler (GHC)
- Convert HILL to Haskell
- Compile Haskell using GHC

- Take note: benchmarking GHC against HILL + GHC (GHC wins regardless?)

Attempt 1: Draw

- Both are the same speed using –O2
- Using –O0, HILL beats GHC by 60%
- –O2 vs –O0 speeds HILL by 10%

- Suggests HILL is doing most of the work?

List fusion

- GHC has special foldr/build rules
- Specialise certain call sequences
- Built in, gives an advantage to GHC, but not to HILL

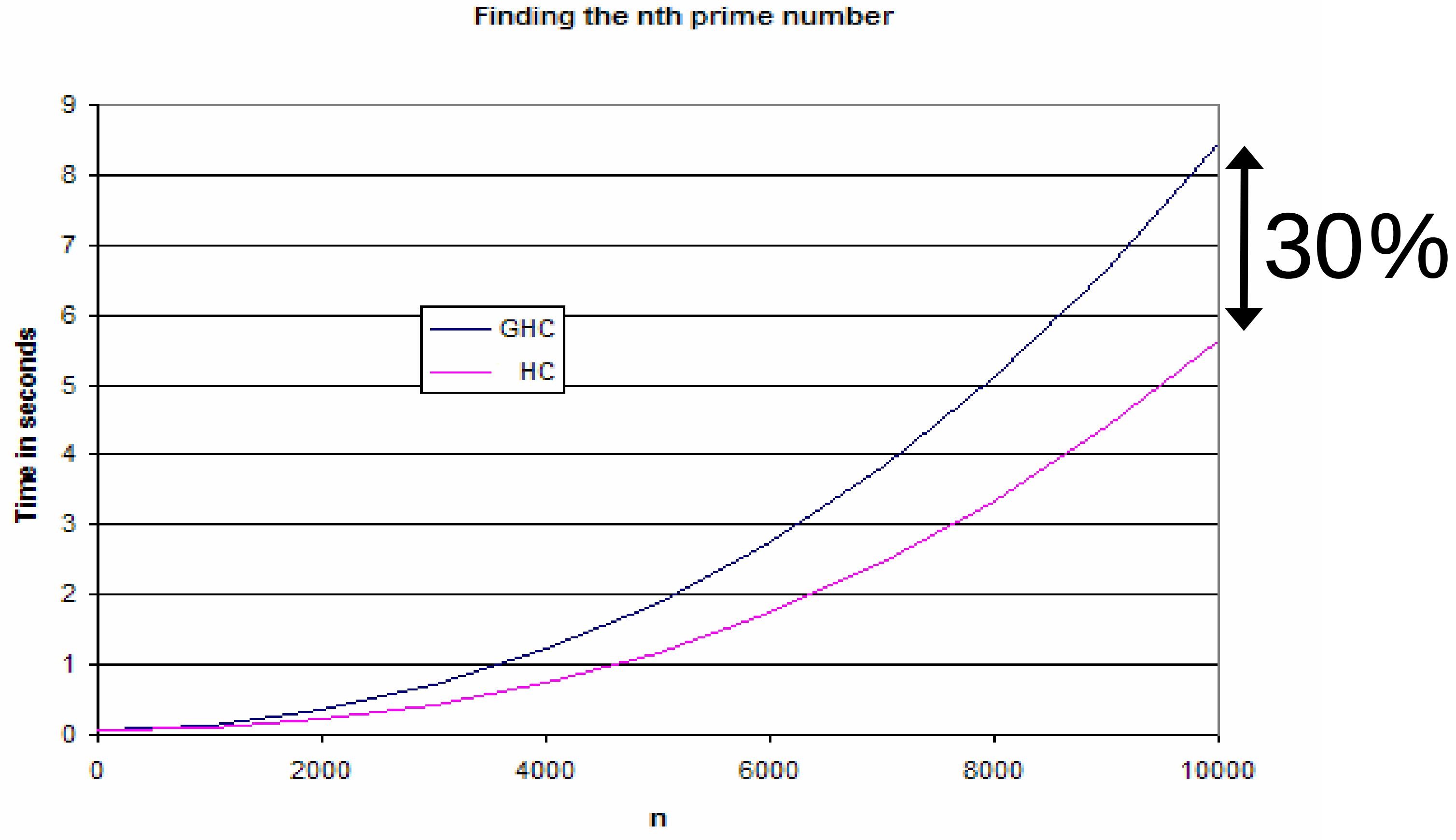
- Applies 4 places in the Primes benchmark

Add General Fusion to HILL

- Implemented, about an afternoon’s work (taking liberties)
- Works on all data types, even non-recursive ones
- Can deforest (!!), foldr/build can’t

- Applies 6 times

Results



Finding the nth prime number
Time in seconds
0
1
2
3
4
5
6
7
8
9
0
2000
4000
6000
8000
10000
n
GHC
HC
30%

Beware

- One benchmark
- Using GHC as the backend
- But consistent improvement

- One other benchmark (Exp3_8), 5% improvement, written close to optimal

λ Future Work

- Speed up transformations
- Be more selective about specialisation
- More benchmarks
  - Whole `nofib` suite
- Native C back end

C backend

- Catch: Haskell -> Yhc Core -> HILL -> First Order HILL -> Haskell
- GHC: Haskell -> GHC Core -> STG -> C

- Haskell can’t express some features of HILL (unnecessary case statements)
- STG copes with higher order functions

λ Conclusion

- All programs can be made first order
- Some Haskell programs can go faster
- Specialisation is an interesting technique

- More benchmarks will lead to more conclusions!