Supero: Making Haskell Faster

Neil Mitchell,
Colin Runciman

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

The Goal

- Make Haskell *‘faster’*
  - Reduce the runtime
  - But keep high-level declarative style

- Without user annotations
  - Different from foldr/build, steam/unstream

Word Counting

- In Haskell

```
main = print . length . words =<< getContents
```

- Very high level
- A nice ‘specification’ of the problem

And in C

```
int main() {
    int i = 0, c, last_space = 1;
    while ((c = getchar()) != EOF) {
        int this_space = isspace(c);
        if (last_space && !this_space) i++;
        last_space = this_space;
    }
    printf("%i\n", i);
    return 0;
}
```

About 3 times faster than Haskell (gcc vs ghc)

Why is Haskell slower?

- Intermediate lists! (and other things)
  - GHC allocates and garbage collects memory
  - C requires a fixed ~13Kb
- length . words =<< getContents
  - getContents produces a list
  - words consumes a list, produces a list of lists
  - length consumes the outer list

Removing the lists

- GHC already has foldr/build fusion
  - e.g. map f (map g x) == map (f . g) x
- But getContents is trapped under IO
  - Much harder to fuse automatically
  - Don’t want to rewrite everything as foldr
  - Easy to go wrong (take function in GHC 6.6)

Supero: Optimiser

- No annotations or special functions
- Uses ideas of supercompilation
- Whole program
- Evaluate the program at *compile* time
  - Start at main, and execute
- Residuate when you reach a primitive
  - The primitive is in the optimised program

Optimising an Expression

$\mathcal{O}$ [**case** *x* **of** *alts*] = **case** $\mathcal{O}$ [*x*] **of** *alts*

$\mathcal{O}$ [**let** *v* = *x* **in** *y*] = **let** v = $\mathcal{O}$ [*x*] in $\mathcal{O}$ [*y*]

$\mathcal{O}$ [*x y*] = $\mathcal{O}$ [*x*] *y*

$\mathcal{O}$ [*f*] = unfold *f*, if *f* is a not primitive

$\mathcal{O}$* = apply $\mathcal{O}$ until no further changes

- Optimise the head of the expression
- Also apply standard simplification rules

The tie back

- Once an expression is optimised with $\mathcal{O}^*$
  - The outmost expression is frozen
  - The inner expressions are assigned names
- Each name and expression is then optimised further
- Identical expressions receive identical names
  - Finitely many expressions/names

An Example

```
sum x = case x of
            []    → 0
            x:xs  → x + sum xs

range i n = case i > n of
              True  → []
              False → i : range (i+1) n

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

Evaluation proceeds

Generalise

```
main n
sum (range 0 n)
main n = main2 0 n
    where main2 i n = sum (range i n)
case range i n of {[] → 0; x:xs → x + sum xs}
case (case i > n of {True → []; False → …}) of …
case i > n of {True → 0
                 ;False → i + sum (range (i+1) n)}
```

tie back: main2 (i+1) n

The Residual Program

```
main n = main2 i n

main2 i n = if i > n then 0 else i + main2 (i+1) n
```

- Lists have gone entirely
- Everything is now strict
- Using sum as foldl or foldl’ would have given accumulator version

Termination

- $\mathcal{O}$* does not necessarily terminate
- Some expressions may keep getting bigger
- Size bound on an expression
  - If an expression exceeds a threshold
  - Then freeze the outermost expression shell

```
case map head xs of
        []     → True
        (y:ys) → and ys
```

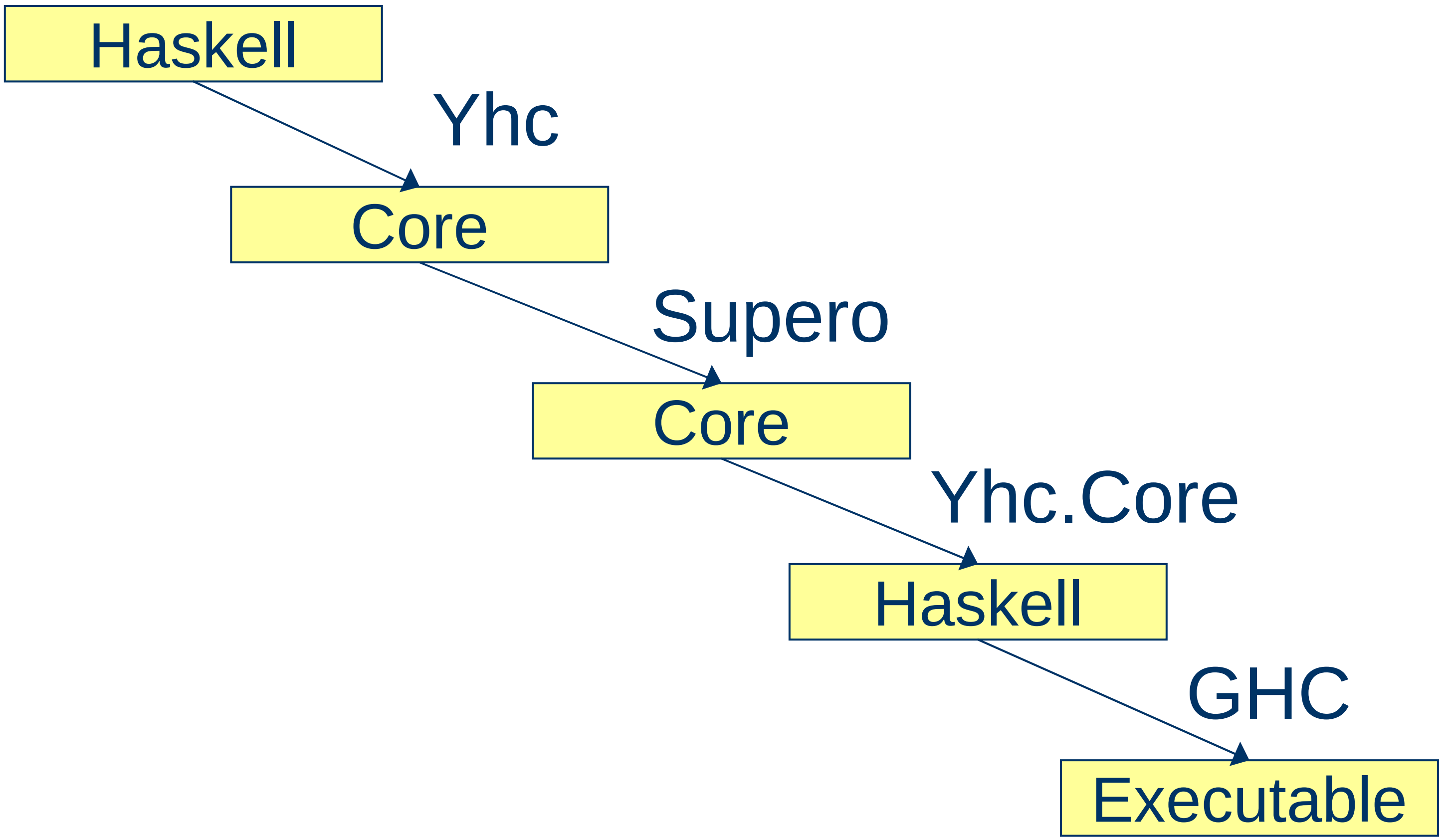
```
case [map head xs] of
        []     → [True]
        (y:ys) → [and ys]
```

Termination Problems

- Some programs like different bounds
- Ad hoc numeric parameters

- A better method may be based on homeomorphic embedding
  - *Positive Supercompilation for a higher order call-by-value language*, by Peter A. Jonsson

'Supero' Compilation

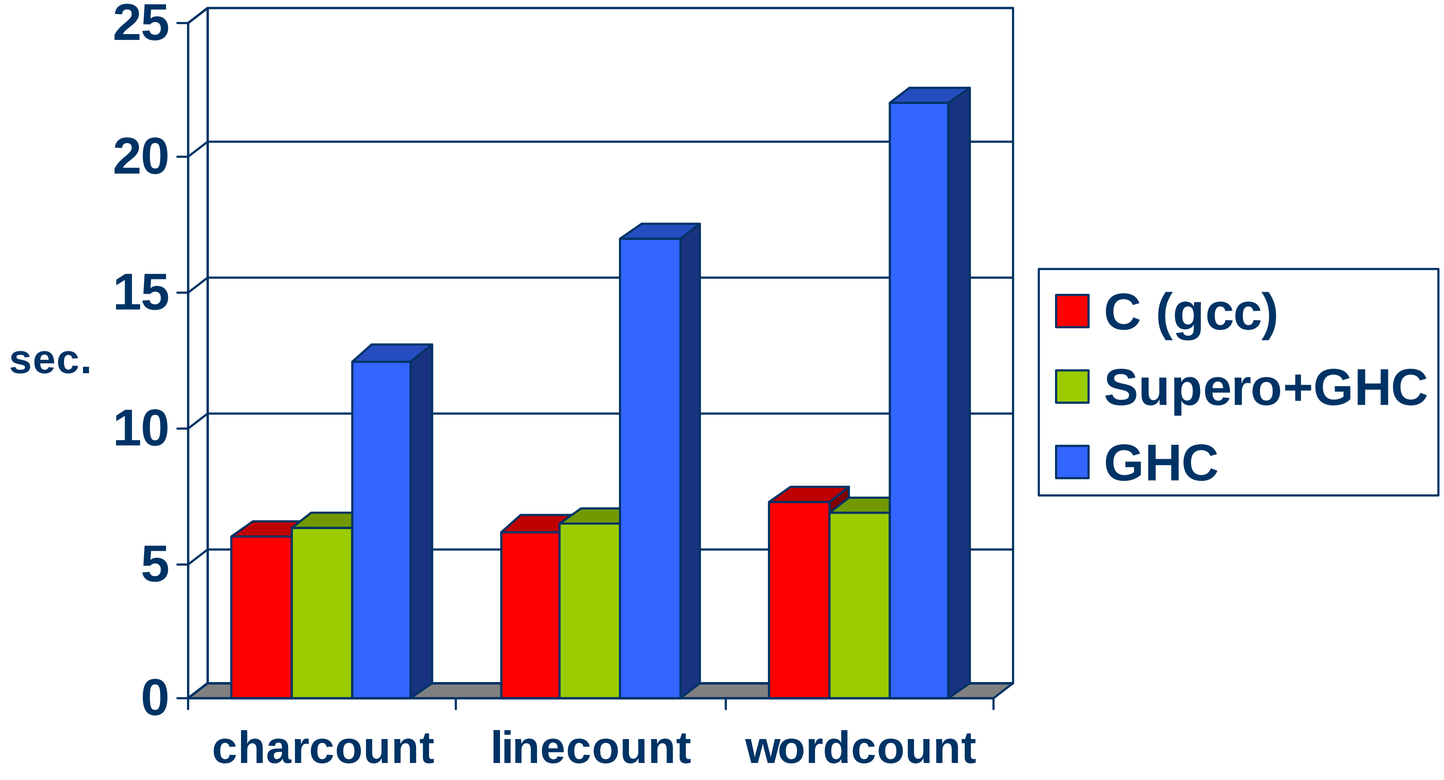


GHC’s Contributions

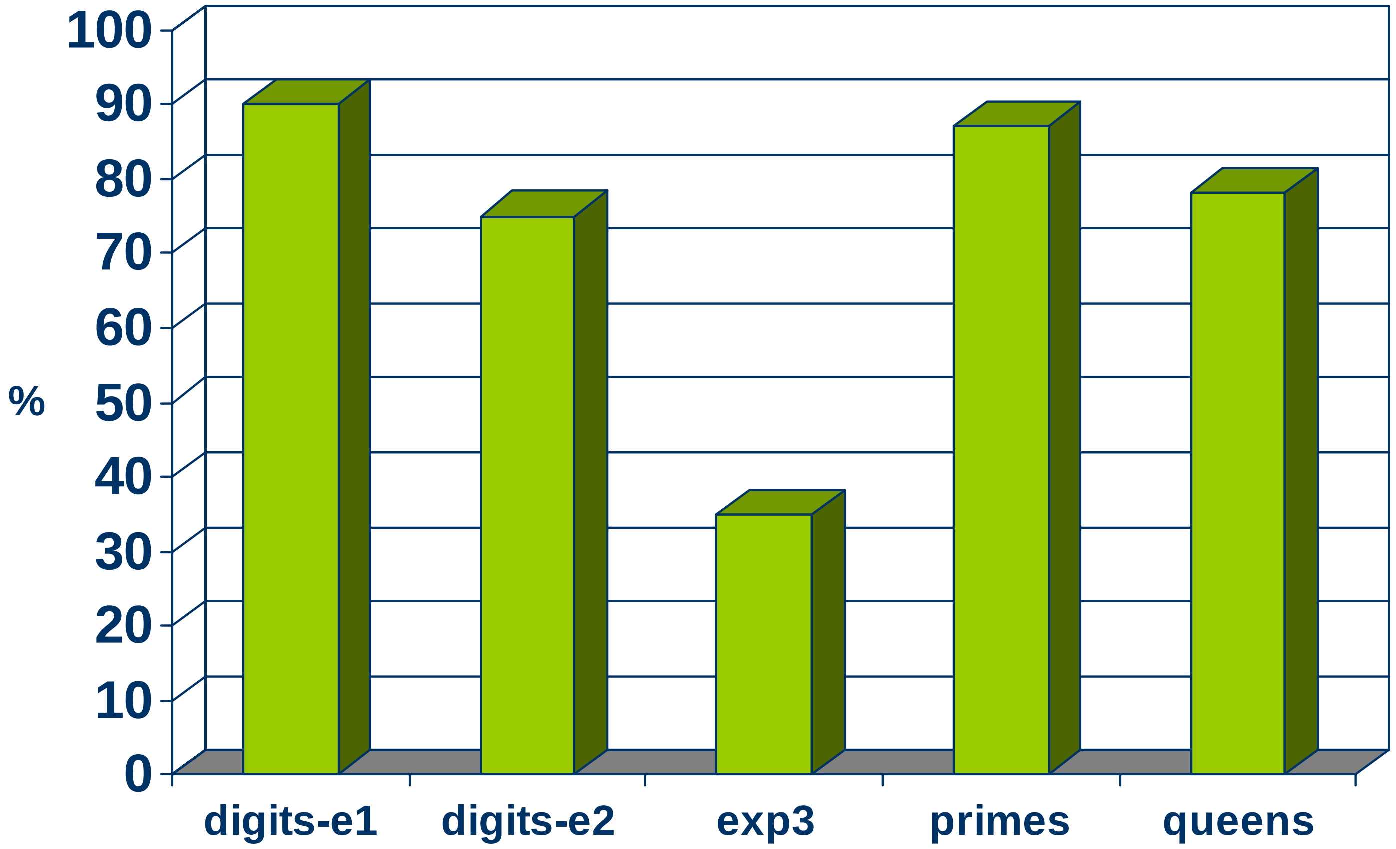
- GHC is a mature optimising compiler

- Primitives (Integer etc)
- Strictness analysis and unboxing
- STG code generation
- Machine code generation

Comparative Runtime (40Mb file)



Runtime as % of GHC time



Conclusions

- Still more work to be done
  - Complete nofib suite is the target
  - Termination is the ‘open issue’

- Haskell can perform as fast as C
- Haskell programs can go faster