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

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

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

```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

\[ O[\text{case } x \text{ of } alts] = \text{case } O[x] \text{ of } alts \]
\[ O[\text{let } v = x \text{ in } y] = \text{let } v = O[x] \text{ in } O[y] \]
\[ O[x \ y] = O[x] \ y \]
\[ O[f] = \text{unfold } f, \text{ if } f \text{ is a not primitive} \]
\[ O^* = \text{apply } O \text{ until no further changes} \]

- Optimise the head of the expression
- Also apply standard simplification rules
The tie back

- Once an expression is optimised with $O^*$
  - 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

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

- $O^*$ 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

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

```haskell
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

- Haskell
- Yhc
- Core
- Supero
- Core
- Yhc.Core
- Haskell
- GHC
- Executable
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

- digits-e1
- digits-e2
- exp3
- primes
- queens
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