Supercompilation for Haskell

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. \( \text{map } f \ (\text{map } g \ x) = \text{map } (f \ . \ g) \ x \)

- 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)
Supercompilation

- An old idea (Turchin 1982)
- Whole program
- Evaluate the program at compile time
  - Start at main, and execute
- If you can’t evaluate (primitives) leave a residual expression
  - The primitive is in the optimised program
Optimising an expression

expression → simplify → inline

When should we terminate?

What should we inline?

How should we generalise?

generalise → residual → named*

When should we terminate?

What should we inline?

How should we generalise?
An example (specialisation)

map (\b → b+1) as -- named as map’

- inline map
  case as of {[]} → []; x:xs -> (\b → b+1) x : map (\b → b+1) xs

- simplify
  case as of {[]} -> []; x:xs → x+1 : map (\b → b+1) xs

- no generalisation and residuate
  case as of {[]} -> []; x:xs → x+1 : ? xs

  ? xs = map (\b → b+1) xs

- use existing name
  ? xs = map’ xs

map’ xs = case as of {[]} → []; x:xs → x+1 : map’ xs
An example (deforestation)

map f (map g as) -- named as map'

- inline outer map
  case map g as of {[] → []; x:xs → f x : map f xs}

- inline remaining map
  case (case … of …) of {[] → []; x:xs → f x : map f xs}

- simplify
  case as of {[] → []; x:xs → f (g x) : map f (map g xs)}

- generalise, residuate and use existing name
  map’ f g as = case as of {[] → []; x:xs → f (g x) : map’ f g xs}
An example (with generalisation)

```
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

\[
\text{sum (range 0 n)} \\
\text{case range 0 n of \{[] \to 0; x:xs \to x + \text{sum} \ xs\}} \\
\text{case (case 0 > n of \{True \to []; False \to \ldots\}) of \ldots} \\
\text{case 0 > n of \{True \to 0; False \to i + \text{sum (range (0+1) n)}\}} \\
\text{sum (range (0+1) n)} \\
\bullet \ \text{Now we terminate and generalise!} \\
\text{sum (range i n)} \\
\text{case range i n of \{[] \to 0; x:xs \to x + \text{sum} \ xs\}} \\
\ldots
\]
The Residual Program

```
main n = if 0 > n then 0 else 0 + main2 (0+1) 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
When do we terminate?

- When the expression we are currently at is an extension of a previous one

\[
\text{sum (range (0+1) n)} > \text{sum (range 0 n)}
\]

\[
a > b \iff a \rightarrow^{\text{emb}} b, \text{ where } \text{emb} = \{f(x_1,\ldots,x_n) \rightarrow x_i\}
\]

- This relation is a homeomorphic embedding
  - Guarantees termination as a whole
How do we generalise?

- When we terminated which bit had emb applied?
  \[ \text{sum (range (0+1) n)} \]

- Generalise those bits
  \[ \text{let } i = 0+1 \]
  \[ \text{in sum (range } i \ n) \]
What should we inline?

- Obvious answer: whatever would be evaluated next. But…

  ```
  let x = (==) $ 1
  in x 1 : map x ys
  ```

- We want to evaluate $, as map will terminate
- Inline by evaluation order, unless will terminate, in which case try others
GHC’s Contributions

- GHC is great 😊
  - Primitives (Integer etc)
  - Strictness analysis and unboxing
  - STG code generation
  - Machine code generation

- How do we do on word counting now?
Problem 1: isSpace

- On GHC, isSpace is too slow (bug 1473)
  - C's isspace: 0.375
  - C's iswspace: 0.400
  - Char.isSpace: 0.672

- For this test, I use the FFI

SOLVED!
Problem 2: words (spot 2 bugs!)

```haskell
words :: String → [String]
words s = case dropWhile isSpace s of
    [] → []
    s2 → w : words s3
        where (w, s3) = break isSpace s2
```

- Better version in Yhc

SOLVED!
Other Problems

- Wrong strictness information (bug 1592)
  - IO functions do not always play nice
- Badly positioned heap checks (bug 1498)
  - Tight recursive loop, where all time is spent
  - Allocates only on base case (once)
  - Checks for heap space every time
- Unnecessary stack checks
- Probably ~15% slowdown
Performance

- Now Supero+GHC is 10% faster than C!
  - Somewhat unexpected…
  - Can anyone guess why?

```c
while ((c = getchar()) != EOF)
    int this_space = isspace(c);
    if (last_space && !this_space) i++;
    last_space = this_space;
```
The Inner Loop

- Haskell encodes space/not in the program counter!
- Hard to express in C
Comparative Runtime (40Mb file)

- C (gcc)
- Supero+GHC
- GHC

Bar chart showing runtime comparison for different counts:
- charcount
- linecount
- wordcount

Runtime measured in seconds (sec.)
Runtime as % of GHC time
Conclusions

- Still more work to be done
  - More benchmarks, whole nofib suite
  - Compilation time is currently too long
- Haskell can perform as fast as C
- Haskell programs can go faster