Rethinking Supercompilation

Neil Mitchell
ICFP 2010

community.haskell.org/~ndm/supero
Supercompilation

- Whole program optimisation technique
  - From Turchin 1982

Run the program at compile time

Source Program → Residual Program
map :: (a → b) → [a] → [b]
map f x = case x of
  [] → []
  x:xs → f x : map f xs

root f g y = map f (map g y)
map f (map g y)
map f (map g y)

case map g y of
  [] → []
  x:xs → f x : map f xs
map f (map g y)
case map g y of
  [] → []
  x:xs → f x : map f xs

case (case y of [] → []; x:xs → g x : map g xs) of
  [] → []
  x:xs → f x : map f xs
map f (map g y)
case map g y of
  [] → []
  x:xs → f x : map f xs
case (case y of [] → []; x:xs → g x : map g xs) of
  [] → []
  x:xs → f x : map f xs

- Stuck, but y must be either [] or (:)

case y of
  [] → next slide
  z:zs → next slide + 1
let y = [] in

case (case y of [] → []; x:xs → g x : map g xs) of
  [] → []
  x:xs → f x : map f xs
let y = [] in

case (case y of [] → []; x:xs → g x : map g xs) of
  [] → []
x:xs → f x : map f xs

case [] of
  [] → []
x:xs → f x : map f xs

[]
let y = z:zs in

  case (case y of [] → []; x:xs → g x : map g xs) of
    [] → []
    x:xs → f x : map f xs

  case g z : map g zs of
    [] → []
    x:xs → f x : map f xs

  f (g z) : map f (map g zs)
let y = z:zs in

\[
\begin{aligned}
\text{case }& (\text{case } y \text{ of } []) \rightarrow []; x:xs \rightarrow g \ x : \text{map } g \ x s) \text{ of } \\
& [] \quad \rightarrow [] \\
& x:xs \rightarrow f \ x : \text{map } f \ x s \\
\text{case }& g \ z : \text{map } g \ zs \text{ of } \\
& [] \quad \rightarrow [] \\
& x:xs \rightarrow f \ x : \text{map } f \ x s \\
f (g \ z) : \text{map } f (\text{map } g \ zs) \\
\end{aligned}
\]

● Stuck, result must be _ : _
let y = z:zs in

case (case y of [] → []; x:xs → g x : map g xs) of
  [] → []
  x:xs → f x : map f xs

case g z : map g zs of
  [] → []
  x:xs → f x : map f xs

f (g z) : map f (map g zs)

● Stuck, result must be _ : _

...
Deforestation

root f g y = case y of
    [] → []
    z:zs → f (g z) : root f g zs

- Simple evaluation, no case/case transformation
- Works even if the user defines their own map
  - Semantic, not syntactic
Overview of Supercompilation

1 evaluation

- seen before?
  - Use previous result
- stuck?
  - Split residual and evaluate pieces
- otherwise
  - terminate?
    - Split residual and evaluate pieces

The paper

This talk
What is new?

- New Core language
  - Totally different treatment of let
  - let is often poorly handled by supercompilers

- New termination criteria
  - No more slow homeomorphic embedding

- These changes lead to many other changes
Core Language

- The root of an expression is a list of let bindings
- Most places allow variables, not expressions

\[
\text{root } f \ g \ y = \text{let } v_1 = \text{map } g \ y \\
\quad v_2 = \text{map } f \ v_1 \\
\text{in } v_2
\]
Evaluate 1: Case of constructor

\[
\begin{align*}
\text{let } v_1 &= [] \\
v_2 &= \text{case } v_1 \text{ of} \\
&\quad [] \rightarrow [] \\
&\quad x:xs \rightarrow xs \\
\text{in } v_2 \\
\end{align*}
\]
Evaluate 2: \( \beta \) reduce

```
let v_1 = map f z
in  v_1

let v_1 = case z of
        []    \rightarrow []
        x:xs \rightarrow let w_1 = f x; w_2 = map f xs
                   in  w_3 = w_1 : w_2; w_3

in  v_1
```
Evaluate 3: Root let

\[
\begin{align*}
  \text{let } v_1 &= \text{let } v_2 = [] \\
  &\quad \text{in } v_2 \\
  &\quad v_3 = \text{case } v_1 \text{ of } ... \\
  &\quad \text{in } v_3
\end{align*}
\]

\[
\begin{align*}
  \text{let } v_1 &= v_2 \\
  v_2 &= [] \\
  v_3 &= \text{case } v_1 \text{ of } ... \\
  \text{in } v_3
\end{align*}
\]
Evaluate 4: α rename

\[
\text{let } v_1 = v_2 \\
v_2 = [] \\
v_3 = \text{case } v_1 \text{ of } ... \\
\text{in } v_3
\]

+ more
Termination

- We never construct new subexpressions!
  - No case/case, no let substitution
  - We just move around and alpha rename source program subexpressions

- Finite number of source subexpressions
- A root let binding corresponds to a bag/multiset over a finite alphabet
Termination Strategy

History = list of previously seen expressions

- Empty history
- Perform a step (inline + simplify)
- Can this expression be added to the history?
  - Yes
  - No
- Add to history
Termination Function

- History is a list of previously seen values
- Values are a multiset over a finite alphabet

- Can only add $x$ to the history $ys$ if:
  - $\forall y \in ys \cdot x \subseteq y$
  - $x \subseteq y = \text{set}(x) \neq \text{set}(y) \lor \#x < \#y$
Performance Results

Disclaimer: For comparison purposes we compiled all the benchmarks with GHC 6.12.1, using the -O2 optimisation setting. For the supercompiled results we first ran our supercompiler, then compiled the result using GHC. To run the benchmarks we used a 32bit Windows machine with a 2.5GHz processor and 4Gb of RAM. Benchmarks may go up as well as down. Contents may settle during shipping. Benchmarks are very hard to get right.
Performance Summary

- Compared to GHC alone
  - Can sometimes be much faster
- Compared to previous supercompilers
  - No worse, perhaps even a bit better
- Compile time is much faster
  - In particular, termination testing < 5%, with most simple method possible
Why Supercompilation?

- Subsumes most other optimisations
  - Deforestation
  - Specialisation
  - Constructor specialisation
  - Inlining

- Requires no user annotations/special names

- Reasonably simple

- Great at removing abstraction overhead
Why Not Supercompilation?

- Some programs can get much bigger/take very long at compile time
  - See Bolingbroke and Peyton Jones 2010 (HS)
- Not yet ready for real use
- Some optimisations still aren’t integrated
  - Strictness
  - Unboxing
  - Changing data type representations
Conclusions

- Supercompilation is a simple and powerful program optimisation technique
- We can now handle let expressions properly
- Termination checks are now fast enough
- Even with all the excellent GHC work, supercompilation still gives big wins
Current Optimising Compilers

“Good compilers have a lot of bullets in their gun”
Simon Peyton Jones
Supercompilation

One powerful transformation