Supercompilation by evaluation

Simon Peyton Jones (Microsoft Research)
Max Bolingbroke (University of Cambridge)

June 2010

Supercompilation

- Supercompilation is well studied…
- …but not deployed in any widely-used compiler
- Why not?
  - Not invented here?
  - Works only at small scale?
  - Too complicated for a practical programming language?

Our goals

- Understand the state of the art
- Try out supercompilation in the context of GHC
  - Small, tractable intermediate language:
    - Purely functional
    - Only eight constructors in the term language
    - Strongly typed
  - Haskell has a substantial code base: thousands of packages with millions of lines of code

Supercompilation is

- A mixture of interacting ideas:
  - Evaluation of open terms
  - Memoisation / dynamic programming (use previous results), including of as-yet-incomplete calls
  - Generalisation
  - Control of termination

I have spent months trying to get a "gut" understanding of what is going on, and I'm still finding it hard!

This talk

- A new, modular supercompiler
  - Based **directly** on an evaluator; nicely separates the pieces
  - Works for call-by-need with 'let'

```
    let ones = 1:ones in map (\x.x+1) ones
===>
    let xs = 2:xs in xs
```

  - Higher order
  - No special "top level"
  - De-emphasises the termination issues

How a supercompiler works

- Do compile-time evaluation until you get "stuck"
  - being careful about non-termination
  - cope with open terms (ie with free variables)
- Split the stuck term into a residual "shell" and sub-terms that are recursively supercompiled.
- Memoise the entire process to "tie the knot"

```
let inc = \x. x+1
    map = \f xs. case xs of [] -> []
                            (y:ys) -> f y : map f ys
in map inc zs
```

- Evaluate until stuck (zs is free)

```
let inc = …; map = … in
in case zs of [] -> []
              (y:ys) -> inc y : map inc ys
```

- Split:

```
case zs of [] -> []
           (y:ys) -> HOLE1 : HOLE2
```

HOLE1

```
let inc = … in inc y
```

HOLE2

```
let inc = …; map = … in
in map inc ys
```

- Recursively supercompile

```
let inc = \x.x+1 in inc y
```

```
y+1
```

- Memoise:

```
let inc = …; map = … in
in map inc ys
```

```
h0 ys
```

Result (as expected)

Output bindings

```
h0 = \xs. case xs of []       -> []
                     (y:ys) -> y+1 : h0 ys
```

Optimised term

```
h0 xs
```

A simple language

Values

v	$::=$	$\lambda x.\, e$	Lambda abstraction
	$\mid$	ℓ	Literal
	$\mid$	$\mathbf{C}\ \overline{x}$	Saturated constructed data

Terms

e	$::=$	x	Variable reference
	$\mid$	v	Values
	$\mid$	$e\ x$	Application
	$\mid$	$e \otimes e$	Binary primops
	$\mid$	$\mathbf{let}\ \overline{x\ =\ e}\ \mathbf{in}\ e$	Recursive let-binding
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}\ \overline{\alpha\ \rightarrow\ e}$	Case decomposition

Case Alternative

α	$::=$	ℓ	Literal alternative
	$\mid$	$\mathbf{C}\ \overline{x}$	Constructor alternative

No lambda-lifting

A-normal form: argument is a variable

Nested, possibly-recursive let bindings

Simple, flat case patterns

Things to notice

- No distinguished top level. A "program" is just a term

  let <defns> in ...

- "let" is mutually recursive
- "case" expressions rather than f and g-functions

The entire supercompiler

$$
\begin{array}{l}
sc, sc' :: History \to State \to ScpM\ Term \\
sc\ hist = memo\ (sc'\ hist) \\
sc'\ hist\ state = \mathbf{case}\ terminate\ hist\ state\ \mathbf{of} \\
\quad Continue\ hist' \to split\ (sc\ hist')\ (reduce\ state) \\
\quad Stop \qquad\qquad\quad\ \to split\ (sc\ hist)\ \ state
\end{array}
$$

- History, terminate: deals with termination
- State: a term with an explicit focus of evaluation
- ScpM, memo: memoisation
- reduce: an evaluator; also deals with termination

EVALUATION

Evaluation and State

Perform compile-time transitions on State, until it gets stuck

```
reduce :: State -> State

type State = (Heap, Term, Stack)
```

- "State" is the state of the abstract machine (sometimes called a "configuration")
- "reduce" takes successive steps, directly implementing the small-step transition semantics

State = ‹H, e, K›

Heaps $H ::= \overline{x \mapsto e}$ **Stacks** $K ::= \overline{\kappa}$

Stack Frames

κ	$::=$	**update** x	Update frame
	$\mid$	$\bullet\ x$	Apply to function value
	$\mid$	**case** $\bullet$ **of** $\overline{\alpha \to e}$	Scrutinise value
	$\mid$	$\bullet \otimes e$	Apply first value to primop
	$\mid$	$v \otimes \bullet$	Apply second value to primop

```
let map = …
in case map f x of …
```

Term

Heap

Term (the focus)

Stack

State

```
< [map = …],
   map,
  [• f, • x, case • of …] >
```

Small step transitions

$$\boxed{\langle H \mid e \mid K\rangle \rightsquigarrow \langle H \mid e \mid K\rangle}$$

$$\begin{array}{rcl}
\langle H, x \mapsto e \mid x \mid K\rangle & \rightsquigarrow & \langle H \mid e \mid \textbf{update}\ x, K\rangle \\
\langle H \mid v \mid \textbf{update}\ x, K\rangle & \rightsquigarrow & \langle H, x \mapsto v \mid v \mid K\rangle \\
\langle H \mid e\ x \mid K\rangle & \rightsquigarrow & \langle H \mid e \mid \bullet\ x, K\rangle \\
\langle H \mid \lambda x.\, e \mid \bullet\ x, K\rangle & \rightsquigarrow & \langle H \mid e \mid K\rangle \\
\langle H \mid e_1 \otimes e_2 \mid K\rangle & \rightsquigarrow & \langle H \mid e_1 \mid \bullet \otimes e_2, K\rangle \\
\langle H \mid v_1 \mid \bullet \otimes e_2, K\rangle & \rightsquigarrow & \langle H \mid e_2 \mid v_1 \otimes \bullet, K\rangle \\
\langle H \mid v_2 \mid v_1 \otimes \bullet, K\rangle & \rightsquigarrow & \langle H \mid \otimes\,(v_1, v_2) \mid K\rangle \\
\langle H \mid \textbf{case}\ e_{\text{scrut}}\ \textbf{of}\ \overline{\alpha \ \rightarrow \ e} \mid K\rangle & \rightsquigarrow & \langle H \mid e_{\text{scrut}} \mid \textbf{case} \bullet \textbf{of}\,\overline{\alpha \ \rightarrow \ e}, K\rangle \\
\langle H \mid \mathrm{C}\,\overline{x} \mid \textbf{case} \bullet \textbf{of}\ \{\ldots, \mathrm{C}\,\overline{x} \ \rightarrow \ e, \ldots\}, K\rangle & \rightsquigarrow & \langle H \mid e \mid K\rangle \\
\langle H \mid \ell \mid \textbf{case} \bullet \textbf{of}\ \{\ldots, \ell \ \rightarrow \ e, \ldots\}, K\rangle & \rightsquigarrow & \langle H \mid e \mid K\rangle \\
\langle H \mid \textbf{let}\ \overline{x \ = \ e}\ \textbf{in}\ e_{\text{body}} \mid K\rangle & \rightsquigarrow & \langle H, \overline{x \mapsto e} \mid e_{\text{body}} \mid K\rangle
\end{array}$$

Small step transitions

$$\langle H \mid \mathsf{case}\ e_{\mathrm{scrut}}\ \mathsf{of}\ \overline{\alpha \to e} \mid K \rangle \rightsquigarrow \langle H \mid e_{\mathrm{scrut}} \mid \mathsf{case} \bullet \mathsf{of}\,\overline{\alpha \to e}, K \rangle$$

$$\langle H \mid \mathrm{C}\,\overline{x} \mid \mathsf{case} \bullet \mathsf{of}\,\{\ldots, \mathrm{C}\,\overline{x} \to e, \ldots\}, K \rangle \rightsquigarrow \langle H \mid e \mid K \rangle$$

$$\langle H \mid \ell \mid \mathsf{case} \bullet \mathsf{of}\,\{\ldots, \ell \to e, \ldots\}, K \rangle \rightsquigarrow \langle H \mid e \mid K \rangle$$

- "Going into a case": push the continuation onto the stack, and evaluate the scrutinee
- "Reducing a case": when a case scrutinises a constructor application (or literal), reduce it

Small step transitions

$$\langle H \mid e\,x \mid K \rangle \quad \rightsquigarrow \quad \langle H \mid e \mid \bullet\ x, K \rangle$$
$$\langle H \mid \lambda x.\, e \mid \bullet\ x, K \rangle \quad \rightsquigarrow \quad \langle H \mid e \mid K \rangle$$

- Application: evaluate the function
- Lambda: do beta reduction

The heap

$$
\begin{array}{rcl}
\langle H, x \mapsto e \mid x \mid K \rangle & \rightsquigarrow & \langle H \mid e \mid \mathbf{update}\ x, K \rangle \\
\langle H \mid v \mid \mathbf{update}\ x, K \rangle & \rightsquigarrow & \langle H, x \mapsto v \mid v \mid K \rangle \\
\langle H \mid \mathbf{let}\ \overline{x\ =\ e}\ \mathbf{in}\ e_{\mathsf{body}} \mid K \rangle & \rightsquigarrow & \langle H, \overline{x \mapsto e} \mid e_{\mathsf{body}} \mid K \rangle
\end{array}
$$

- Let: add to heap
- Variable: evaluate the thunk
- Value: update the thunk

reduce

```
reduce :: State -> State
type State = (Heap, Term, Stack)
```

- Taking small steps: direct transliteration of operational semantics
- Works on open terms
- **Gets "stuck" when trying to evaluate**
  <H, x, K>
  where H does not bind x (because term is open)
- Needs a te`let f = \x. f x in f y`id divergence

Implementing reduce

Stuck

Check termination less often

$$
\begin{array}{l}
\mathit{reduce} :: \mathit{State} \to \mathit{State} \\
\mathit{reduce} = \mathit{go}\ \mathit{emptyHistory} \\
\quad \mathbf{where} \\
\quad\quad \mathit{go}\ \mathit{hist}\ \mathit{state} = \mathbf{case}\ \mathit{step}\ \mathit{state}\ \mathbf{of} \\
\quad\quad\quad \mathit{Nothing} \to \mathit{state} \\
\quad\quad\quad \mathit{Just}\ \mathit{state}' \\
\quad\quad\quad\quad \mid \mathit{intermediate}\ \mathit{state}' \to \mathit{go}\ \mathit{hist}\ \mathit{state}' \\
\quad\quad\quad\quad \mid \mathit{otherwise} \to \mathbf{case}\ \mathit{terminate}\ \mathit{hist}\ \mathit{state}'\ \mathbf{of} \\
\quad\quad\quad\quad\quad \mathit{Stop} \quad\quad\quad\quad \to \mathit{state}' \\
\quad\quad\quad\quad\quad \mathit{Continue}\ \mathit{hist}' \to \mathit{go}\ \mathit{hist}'\ \mathit{state}' \\
\quad\quad \mathit{intermediate}\ (\_, \mathit{Var}\ \_, \_) = \mathit{False} \\
\quad\quad \mathit{intermediate}\ \_ \quad\quad\quad\quad\quad = \mathit{True} \\
\\
\mathit{step} :: \mathit{State} \to \mathit{Maybe}\ \mathit{State} \\
\quad \text{-- Implements Figure 3}
\end{array}
$$

```
terminate :: History -> State -> TermRes
data TermRes = Stop | Continue History
```

TERMINATION

Termination

```
terminate :: History -> State -> TermRes
data TermRes = Stop | Continue History
```

- Use some kind of well-quasi ordering, so you cannot get an infinite sequence

  terminate h0 s0 = Continue h1
  terminate h1 s1 = Continue h2
  terminate h2 s2 = Continue h3
  …etc…

- Want to test as infrequently as possible:
  - Nearby states look similar => whistle may blow unnecessarily

Two separate termination checks

- "Horizontal": check that evaluation does not diverge
- "Vertical": check that recursive calls to "sc" don't see bigger and bigger expressions
- The two checks carry quite separate "histories"; but both use the same "terminate" function
- This is like [Mitchell10] but otherwise

MEMOISATION

The supercompiler

$$
\begin{array}{l}
sc, sc' :: History \to State \to ScpM\ Term \\
sc\ hist = memo\ (sc'\ hist) \\
sc'\ hist\ state = \mathbf{case}\ terminate\ hist\ state\ \mathbf{of} \\
\quad Continue\ hist' \to split\ (sc\ hist')\ (reduce\ state) \\
\quad Stop \qquad\qquad\quad\ \to split\ (sc\ hist)\ \ state
\end{array}
$$

- ScpM, memo: memoisation

```
memo :: (State -> ScpM Term)
     -> (State -> ScpM Term)
```

Memoisation

```
memo :: (State -> ScpM Term)
     -> (State -> ScpM Term)
```

- Goal: re-use previously-supercompiled states
- ScpM monad is a state monad with state:
  - Supply of **fresh names**, [h0, h1, h2, …]
  - Set of **Promises**
    - States that have previously been supercompiled
    - Grows monotonically
  - Set of **optimised bindings**

What's in ScpM?

- Supply of fresh names, [h0, h1, h2, …]
- Set of Promises (states that have previously been supercompiled)

```
data Promise
  = P { meaning :: State
      , name    :: Var
      , fvs     :: [Var] }
```

```
P { meaning = <[map=…], map f (map g) xs, []>
  , name = h4
  , fvs = [f,g,xs] }
```

```
h4 f g xs = case xs of { [] -> []
                         (y:ys) -> f (g y) : h4 f g ys }
```

```
memo :: (State -> ScpM Term)
     -> (State -> ScpM Term)
```

What (`memo f s`) does:

1. Check if s is in the current Promises (modulo alpha-renaming of free variables)
   - If so, return (`h4 f g xs`)
2. Allocate fresh name, `h7`
3. Add a promise for `s`, with meaning s, name `h7`, and free vars of `s` (say, `f,g,x`)
4. Apply `f` to `s`, to get a Term `t`

Things to notice

- We memoise:
  - **States** we have seen before, **not**
  - **Function calls** we have seen before

```
f x y = ..x.... (..y..y..) ....x....
```

- Specialising function f would make duplicate copies for the red code for two calls

  f True y

  f False y

States

Heaps $H ::= \overline{x \mapsto e}$ **Stacks** $K ::= \overline{\kappa}$

Stack Frames

κ	$::=$	**update** x	Update frame
	$\mid$	$\bullet\ x$	Apply to function value
	$\mid$	**case** $\bullet$ **of** $\overline{\alpha \to e}$	Scrutinise value
	$\mid$	$\bullet \otimes e$	Apply first value to primop
	$\mid$	$v \otimes \bullet$	Apply second value to primop

- States and Terms are inter-convertible
- A State <H,e,K> is really just a Term with an explicit "focus". (The H,K are like a "zipper".)
- Distinct terms may evaluate to the same State; more equivalence is good.
- Splitting States is much, much easier than

SPLITTING

Splitting

$$
\begin{array}{l}
sc, sc' :: History \to State \to ScpM\ \ Term \\
sc\ hist = memo\ (sc'\ hist) \\
sc'\ hist\ state = \mathbf{case}\ terminate\ hist\ state\ \mathbf{of} \\
\quad Continue\ hist' \to split\ (sc\ hist')\ (reduce\ state) \\
\quad Stop \qquad\qquad\quad\ \to split\ (sc\ hist)\ \ state
\end{array}
$$

```
split :: (State -> ScpM Term)
      -> (State -> ScpM Term)
```

- The argument state is stuck
- Supercompile its sub-terms and return the

```
split :: (State -> ScpM Term)
      -> (State -> ScpM Term)
```

What (`split sc s`) does:

- Find sub-terms (or rather sub-states) of s
- Use sc to supercompile them
- Re-assemble the result term

```
split :: (State -> ScpM Term)
      -> (State -> ScpM Term)
```

What (`split sc s`) does:

e.g. split sc <[g=...], f, [• (g True)]>

- Stuck because f is free
- Supercompile (g True); or, more precisely
  <[g=...], g True, []> → e
- Reassemble result term: f e

Splitter embodies heuristics

- Example:
  - Duplicate let-bound values freely
  - Do not duplicate (instead supercompile) let-bound non-values

E.g. **<[x = f y], v, [• (h x), • (k x)]>**

- So we recursively supercompile

  <[], h x, []>

- **NOT** **<[x = f y], h x, []>**

See the paper for lots more on splitting

CHANGING THE EVALUATOR

Lazy evaluation

- The update frames of the evaluator deal with lazy (call-by-need) evaluation
- Adding this to our supercompiler was relatively easy. Care needed with the splitter (see paper).
- And it's useful: some of our benchmark programs rely on local letrec

Changing evaluator

- We thought that we could get call-by-value by simply changing "reduce" to cbv.
- Result: correct, but you get very bad specialisation

```
(xs ++ ys) ++ zs
➔   (case xs of
     [] -> []
     (p:ps) -> x : ps++ys)   ++   zs
```

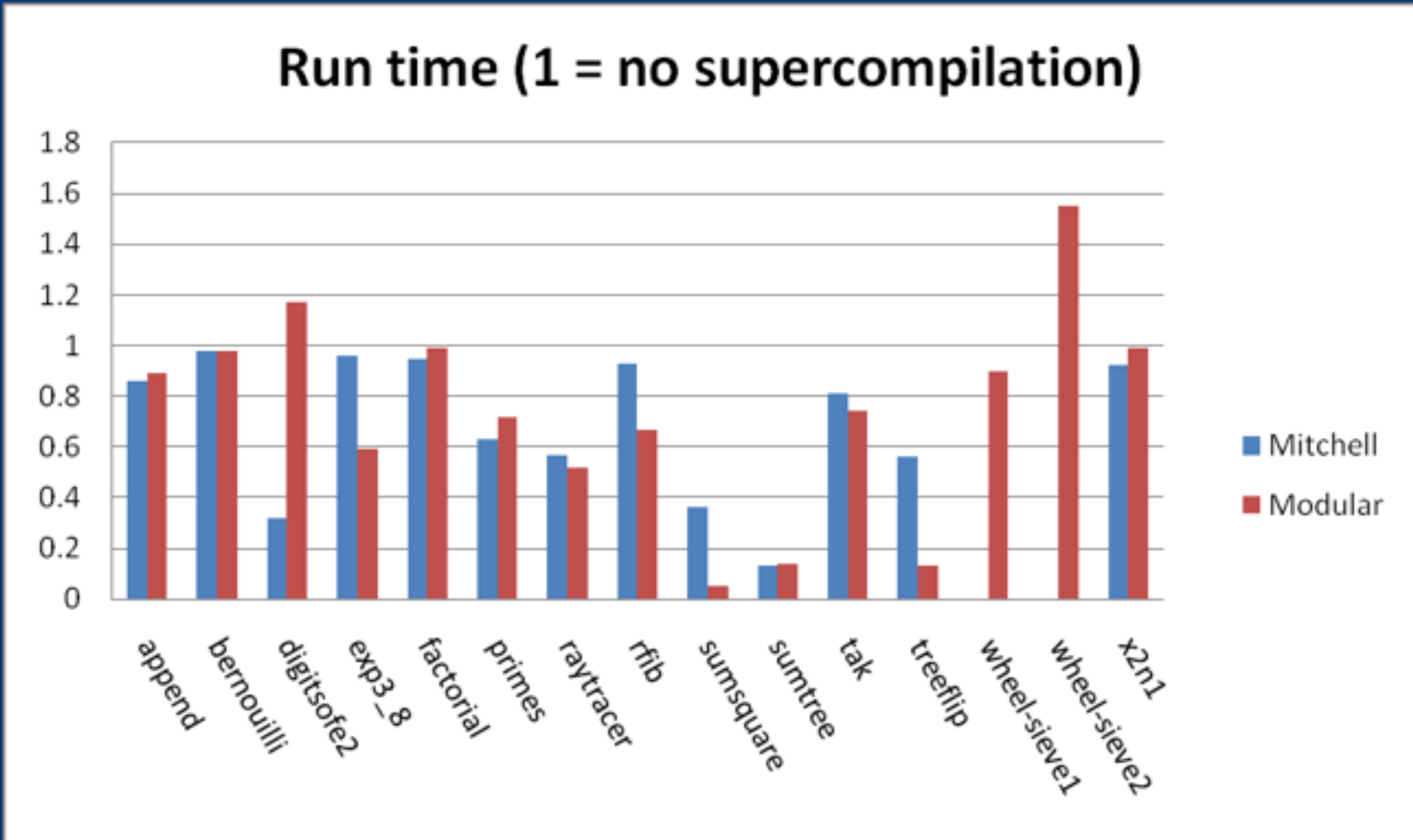
Under cbv, the focus is now (ps++ys) but we

Changing the evaluator

- Apparent conclusion: even for cbv we must do outermost first (ie call by name/need) reduction
- So how can we supercompile an impure language, where the evaluation order is fixed?

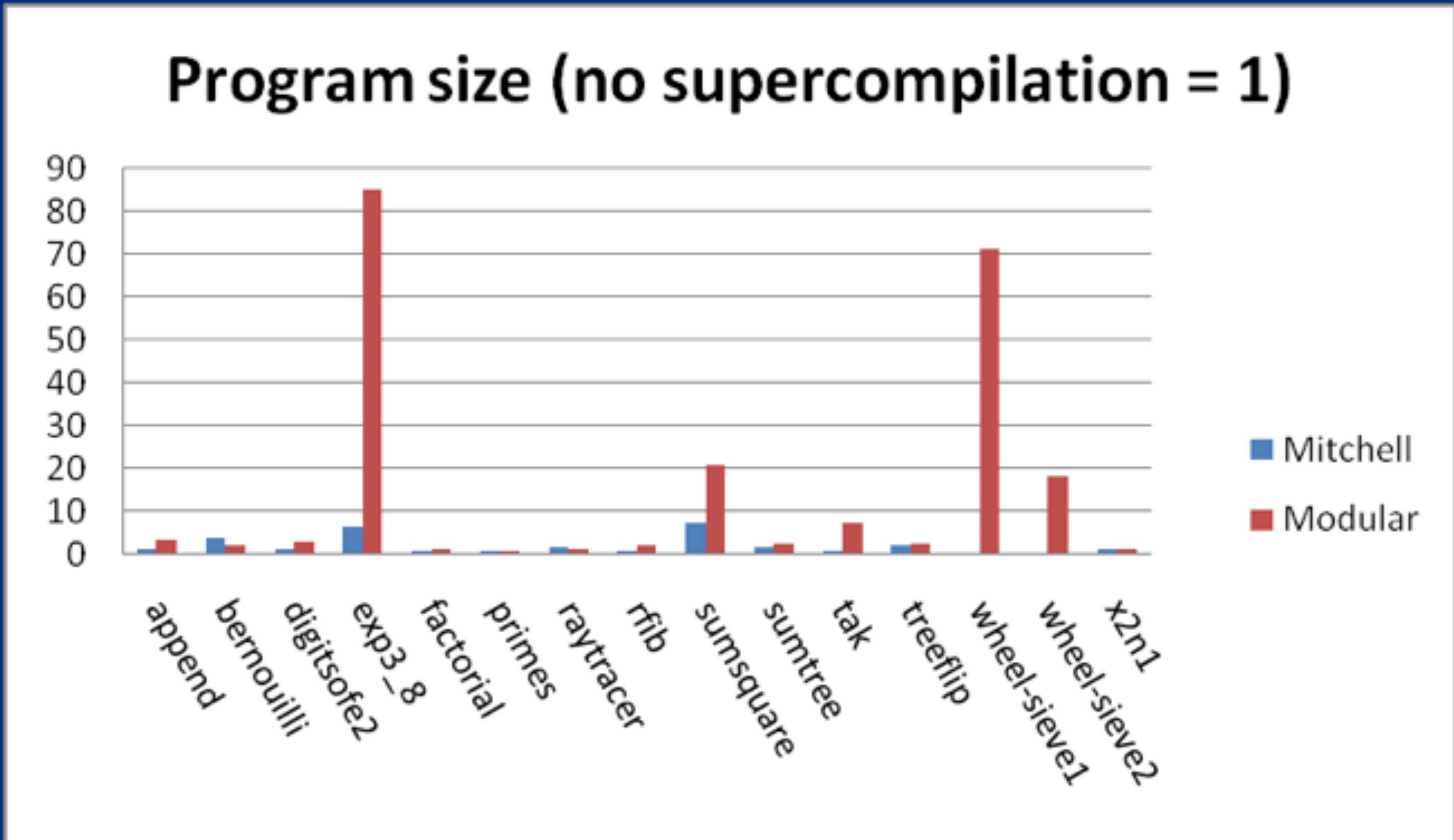
RESULTS

Results

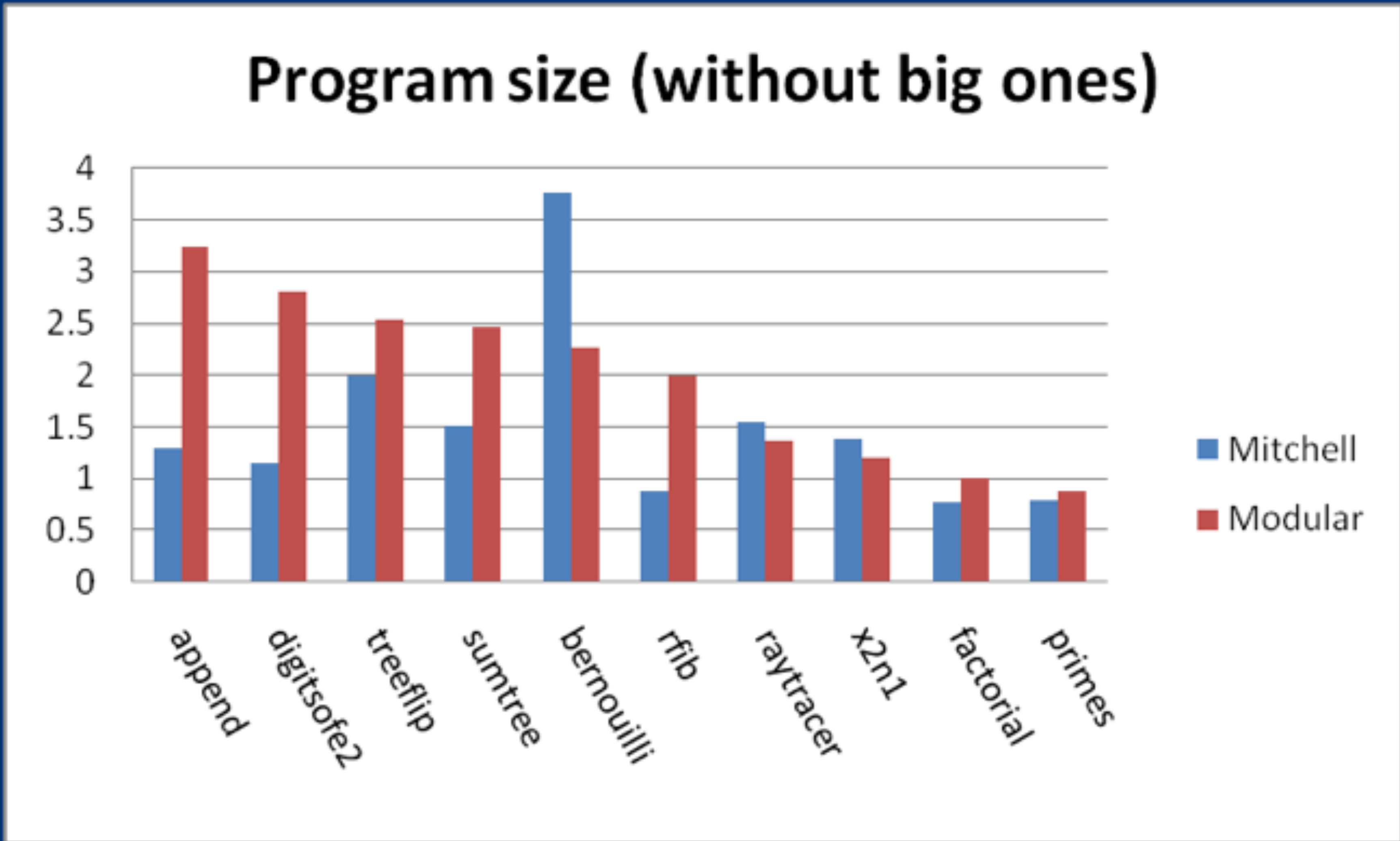


Run time (1 = no supercompilation)
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
append
bernouilli
digitsofe2
exp3_8
factorial
primes
raytracer
rfib
sumsquare
sumtree
tak
treeflip
wheel-sieve1
wheel-sieve2
x2n1
Mitchell
Modular

Program size



Program size again



Comments

- Results are patchy, and rather fragile
- Sometimes programs run slower
- Some programs get grotesquely larger
- Quite hard to "debug" because one gets lots in a morass of terms

THE big issue: size

- A reliable supercompiler simply must not generate grotesque code bloat. 3x maybe; 80x no.
- Some small benchmark programs simply choke every supercompiler we have been able to try (gen-regexps, digits-of-e)
- To try to understand this, we have identified one pattern that generates an supercompiled program that is exponential in the size of the

Exponential code blow-up

```
f1 x = f2 y ++ f2 (y + 1)
  where y = (x + 1) * 2

f2 x = f3 y ++ f3 (y + 1)
  where y = (x + 1) * 2

f3 x = [x + 1]
```

- Supercompile f1
- Leads to two distinct calls to f2
- Each leads to two distinct calls to f3
- And so on

- This program takes exponential time to run, but that's not necessary (I think)

What to do?

- The essence of supercompilation is specialising a function for its calling contexts
- That necessarily means code duplication!
- No easy answers

Idea 1: thrifty supercompilation

- Supercompilation often over-specialises

```
replicate 3 True
➔ h0
where
   h0 = [True, True, True]
```

  - No benefit from knowing True
  - Instead, make a more-re-usable function

```
  h0 True
where
  h0 x = [x,x,x]
```

Size limits

- Fix an acceptable code-bloat factor
- Think of the tree, where each node is a call to split
- We can always just stop and return the current term
- Somehow do so when the code size gets too big. Something like breadth-first traversal?

Conclusion

- Supercompilation has the potential to dramatically improve the performance of Haskell programs
- But we need to work quite a lot harder to develop heuristics that can reliably and predictably optimise programs, without code blow-up