Supercompilation by Evaluation

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

To supercompile, evaluate!

```
let fact = λn. if n == 0 then 1 else n * fact (n - 1)
in fact 1
```

$$\Downarrow$$

```
1
```

- **Supercompilation complete!**
- That was almost too easy...
- It is not always so straightforward - it gets more interesting, as we will see

Problem one: free variables

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

$$\Downarrow$$

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

- Supercompilers need to deal with terms with **missing information**
- We **can't know** at compile time what list we are `map`ping over. What to do?

Solution one: split around the free variable

↡

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

- To supercompile when **stuck**, "split" to supercompile some subcomponents:

"**Residual**" shell

Recursively supercompiled expressions: result is plugged into the holes ◉

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

```
[]
```

```
let inc = ...; map = ...
in inc y : map inc ys
```

Problem two: values containing fields

```
let inc = ...; map = ...
in map inc [1, 2, 3]
```

↓

```
let inc = ...; map = ...
in inc 1 : map inc [2, 3]
```

- Another kind of problem can arise when **call-by-{name,need}** reduction reaches a value
- Due to **laziness**, evaluation by itself can't touch e.g. the `inc 1` subterm... but we'd really like for the supercompiler to reduce it to `2`

Solution two: split around the structure of the value

```
let inc = ...; map = ...
in inc 1 : map inc [2, 3]
```

- Play exactly the same game as before: “split” the term to find some subcomponents amenable to driving

Residual shell

```
◉ : ◉
```

Recursively supercompiled expressions: result is plugged into the holes ◉

```
let inc = ...
in inc 1
```

```
let inc = ...; map = ...
in map inc [2, 3]
```

Problem three: infinite programs

```
map inc ys   ↠  case ys of
                  []         -> []
                  (y0:ys0) -> inc y0 : map inc ys0
             ⇒  case ys of
                  []         -> []
                  (y0:ys0) -> y0+1 : ●

map inc ys0  ↠  case ys0 of
                  []         -> []
                  (y1:ys1) -> inc y1 : map inc ys1
             ⇒  case ys0 of
                  []         -> []
                  (y1:ys1) -> y1+1 : ●

map inc ys1  ↠  case ys1 of
                  []         -> []
                  (y2:ys2) -> inc y2 : map inc ys2
             ⇒  case ys1 of
                  []         -> []
                  (y2:ys2) -> y2+1 : ●
```

- ... and so on
- It looks like the supercompiler will produce an **infinite output program**, and thus **never terminate**!

Infinite programs?

```
map inc ys
```

```
case ys of
  []        -> []
  (y0:ys0) -> y0+1 : case ys0 of
                       []        -> []
                       (y1:ys1) -> y1+1 : case ys1 of
                                            []        -> []
                                            (y2:ys2) -> case ys2 of ...
```

- Maybe not what we had in mind...

Solution three: memoisation

- If we **memoise** the supercompiler by `let`-binding supercompiled terms, we can get **finite output programs**:

```
map inc ys
```

```
case ys of
  []         -> []
  (y0:ys0) -> inc y0 : map inc ys0
```

```
case ys of
  []         -> []
  (y0:ys0) -> y0+1 :
```

```
h0 ys == map inc ys
```

```
map inc ys0
```

```
h0 ys0
```

Final expression:

```
let h0 ys = case ys of []         -> []
                       (y0:ys0) -> y0+1 : h0 ys0
in h0 ys
```

Supercompilation $\approx$ Memoisation + Evaluation

... but no other supercompiler does evaluation right

Existing work

- Existing supercompilers are based on evaluators only in an **ad-hoc** way
  - Typically based on a **driving function** which intermingles the evaluation and splitting steps
- Supero 2010 is slightly better: it separates out a **simplifier** `:: Term -> Maybe Term`
  - Still not quite an evaluator - doesn't follow standard evaluation order
  - Does not terminate on some terms...
- The ad-hoc evaluators used so far are **incapable** of reducing some call-by-need terms (they don't like local **recursive** `let` bindings)

Recursive `let`

```
let ones = 1 : ones
in head ones
```

```
let repeat = λx -> let xs = x : xs
                   in xs
in repeat 1
```

```
map (\x -> x + 1) (repeat 1)
```

```
let twos = 2 : twos in twos
```

- My feeling is that you can't claim to have a supercompiler for a call by need language if you leave some features unimplemented like this

The solution

- Clean semantics for evaluating languages that include recursive `let` are well known [Sestoft 1993, Launchbury 1993, Wadler 1998]
- Stop trying to reinvent the wheel - just take one of those **unmodified** and build our implementation on top

```
step :: State -> Maybe State
```

The state of an abstract machine for evaluating the language

Sestoft's machine

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

An environment mapping variables to `Terms`

The focus of evaluation: we want to reduce this to a value

What is around the focus - what to do with the value once we have it

```
< ε
| let t = True
  in case (\x -> x) t of
       True  -> 1
       False -> 2
| ε >
```

```
< t = True
| case (\x -> x) t of
    True  -> 1
    False -> 2
| ε >
```

```
< t = True
| (\x -> x) t
| case • of
    True  -> 1
    False -> 2 >
```

```
< t = True
| \x -> x
| • t,
  case • of
    True  -> 1
    False -> 2 >
```

```
< t = True
| t
| case • of
    True  -> 1
    False -> 2 >
```

```
< t = True
| True
| case • of
    True  -> 1
    False -> 2 >
```

Sestoft's machine vs Mitchell

```
let x1 = e1
    ...
    xn = en
in xi
```

```
([x1 |-> e1, ..., xn |-> en], xi,[])
```

Sestoft's machine vs Jonsson

$$\mathcal{R} ::= [\,] \mid \mathcal{R}\,\overline{e} \mid \textbf{case}\,\mathcal{R}\ \textbf{of}\ \{p_i \to e_i\} \mid \mathcal{R} \oplus e \mid e \oplus \mathcal{R} \mid \mathcal{R}.l$$

$$\begin{array}{rcll}
\kappa & ::= & \textbf{update}\,x & \text{Update frame} \\
& \mid & \bullet\, x & \text{Apply to function value} \\
& \mid & \textbf{case} \bullet \textbf{of}\,\overline{\alpha \ \to\ e} & \text{Scrutinise value} \\
& \mid & \bullet \otimes\ e & \text{Apply first value to primop} \\
& \mid & v \otimes\ \bullet & \text{Apply second value to primop}
\end{array}$$

Apotheosis

```
scp :: Term -> ScpM Term
```

```
scp :: State -> ScpM Term
```

See paper for the gory details


Supercompilation by Evaluation

Benefits of using a real evaluator

- **Modularity**: the issues of evaluation are separated from those specific to supercompilation (memoisation, splitting...)
- **Generality**: it is easy to swap in *alternative* evaluation strategies (call-by-value, garbage collecting, `let`-speculating...)
- **Optimisation**: we can supercompile programs that no existing call-by-need supercompiler can deal with:

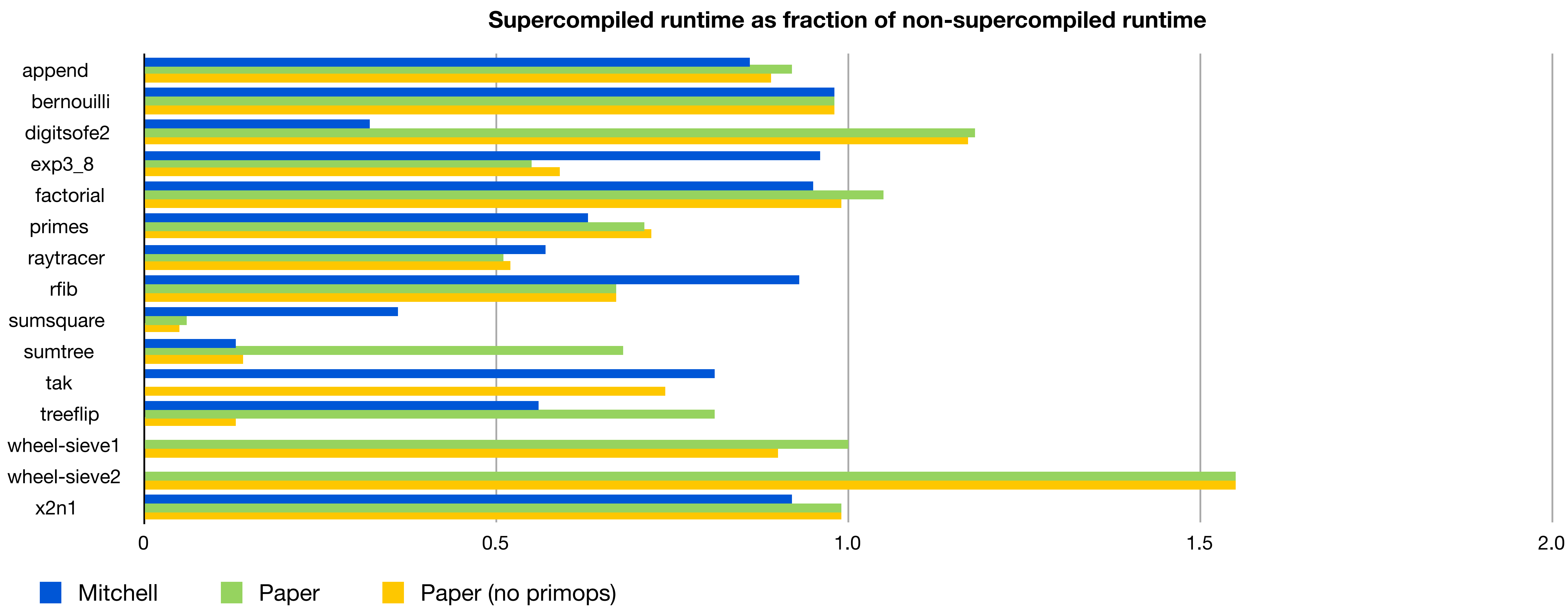
```
map (\x -> x + 1) (repeat 1)
```

↓

```
let twos = 2 : twos in twos
```

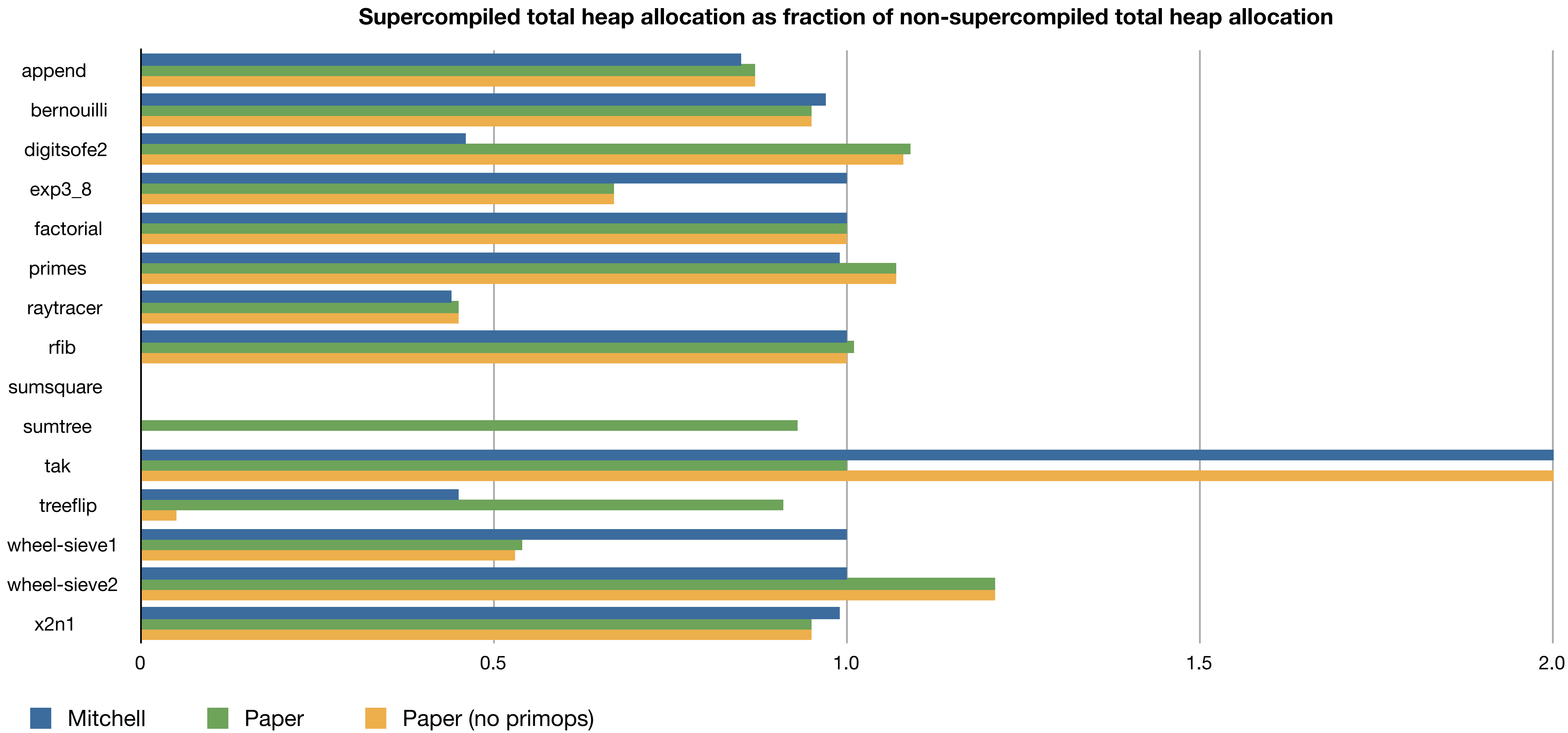
Results

- Our implementation, CHSC, is a preprocessor which produces supercompiled Haskell code for later standard compilation by GHC



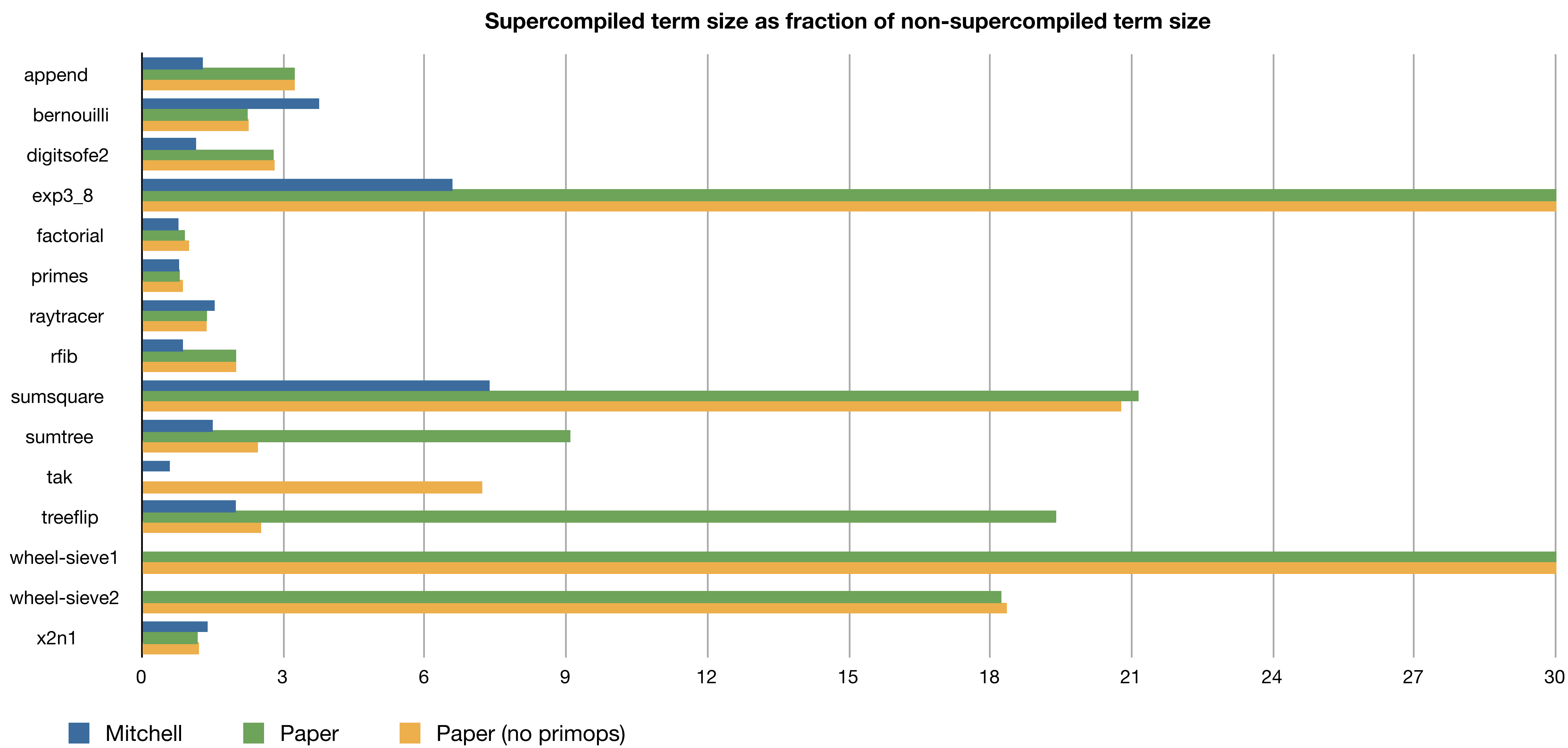
- On benchmarks we can both supercompile, average runtime reduction is ~30% for both Mitchell and our supercompiler (without primops)

Results



- Heap allocation totally eliminated for e.g. sumsquare
- 600x worse (Mitchell) or 19,000x worse (us) for e.g. tak

Results



- On programs we both supercompile, the average size increase is ~2.5x (Mitchell) / ~15x (us)

Related work

	Language	**Recursive `let`?**
SCP4 [Nemytykh]	Refal (CBV)	✖
Timber supercompiler [Jonsson and Nordlander]	Lambda calculus (CBV)	✖
Supero 2010 [Mitchell]	Lambda calculus (Lazy)	✖
HOSC [Klyuchnikov and Romanenko]	Lambda calculus (CBN)	✓
CHSC [Bolingbroke and Peyton Jones]	Lambda calculus (Lazy)	✓

Future work

- **Better optimisation** for output programs
  - In particular, use **generalisation** to prevent specialisation on accumulators
- Make supercompilation **faster**
  - Several imaginary suite benchmarks **cannot be supercompiled in reasonable amounts of time** by any existing system
  - Even some of the examples that did supercompile successfully are annoyingly slow: wheel-sieve1 takes 22s to supercompile

Conclusion

- Takeaway: supercompilers are better when built on top of **real evaluators**
  - **Simpler** to implement and read
  - More **powerful**

Thanks!

http://www.github.com/batterseapower/chsc