Strengthening Supercompilation for Call-By-Value Languages

Peter A. Jonsson Johan Nordlander
Luleå University of Technology

Background

Context

- Timber: a pure and higher-order call-by-value language
- Program optimization is our goal
- We care about preserving semantics

Wadler's algorithm

T[app (app xs ys) zs] =

app xs ys = case xs of {[] -> ys; (x:xs) -> x:app xs ys}

Wadler's algorithm

```
T[app (app xs ys) zs] =
```

```
T[case app xs ys of
    [] -> zs
    (x:xs) -> x:app xs zs]
```

```
app xs ys = case xs of {[] -> ys; (x:xs) -> x:app xs ys}
```

T[(\x.y) e] = y

$T[(\backslash x.y)\ e] = y$

$e = \perp$

CBN: $(\backslash x.y)\ \perp$ -> y

CBV: $(\backslash x.y)\ \perp$ -> $(\backslash x.y)\ \perp$ -> ...

T[(\x.y) e] = y

e = ⊥

CBN: (\x.y) ⊥ -> y

CBV: (\x.y) ⊥ -> (\x.y) ⊥ -> ...

Modify CBN algorithm to transform arguments first?

Naïve algorithm

```
N[app (app xs ys) zs] =
```

```
app xs ys = case xs of {[] -> ys; (x:xs) -> x:app xs ys}
```

Naïve algorithm

N[app (app xs ys) zs] =

N[app (case xs of {[] -> ys; (x:xs) -> x:app xs ys}) zs] =

app xs ys = case xs of {[] -> ys; (x:xs) -> x:app xs ys}

Naïve algorithm

```
N[app (app xs ys) zs] =
```

```
N[app (case xs of {[] -> ys; (x:xs) -> x:app xs ys}) zs] =
```

```
case xs of
  [] -> N[app (ys) zs]
  (x:xs) -> N[app (x:app xs ys) zs]
```

```
app xs ys = case xs of {[] -> ys; (x:xs) -> x:app xs ys}
```

```
app (app xs ys) zs
```

```
app (x:app xs ys) zs
```

```
app (app xs ys) zs
```

$<$

```
app (x:app xs ys) zs
```

```
N[app (app xs ys) zs] = case xs of
                          [] -> h1 ys zs
                          (x:xs) -> h1 (h2 x xs ys) zs
```

```
h1 xs ys = case xs of {[] -> ys; (x:xs) -> x:h1 xs ys}
h2 x xs ys = x:case xs of {[] -> ys; (x:xs) -> h2 x xs ys}
```

```
N[app (app xs ys) zs] = case xs of
                          [] -> h1 ys zs
                          (x:xs) -> h1 (h2 x xs ys) zs
```

```
h1 xs ys = case xs of {[] -> ys; (x:xs) -> x:h1 xs ys}
h2 x xs ys = x:case xs of {[] -> ys; (x:xs) -> h2 x xs ys}
```

Our algorithm

```
D[app (app xs ys) zs] =
```

```
app xs ys = case xs of {[] -> ys; (x:xs) -> x:app xs ys}
```

Our algorithm

```
D[app (app xs ys) zs] =
```

```
D[let xs = app xs ys, ys = zs in case xs of
                                  [] -> ys
                                  (x:xs) -> x:app xs ys]
```

```
app xs ys = case xs of {[] -> ys; (x:xs) -> x:app xs ys}
```

Our algorithm

```
D[app (app xs ys) zs] =
```

```
D[let xs = app xs ys, ys = zs in case xs of
                                  [] -> ys
                                  (x:xs) -> x:app xs ys]
```

```
app xs ys = case xs of {[] -> ys; (x:xs) -> x:app xs ys}
```

Our algorithm

```
D[app (app xs ys) zs] =
```

```
D[let xs = app xs ys, ys = zs in case xs of
                                  [] -> ys
                                  (x:xs) -> x:app xs ys]
```

```
D[case app xs ys of
    [] -> zs
    (x:xs) -> x:app xs zs]
```

```
app xs ys = case xs of {[] -> ys; (x:xs) -> x:app xs ys}
```

```
D[app (app xs ys) zs] = h3 xs ys zs
```

```
h3 xs ys zs = case xs of
                  [] -> case ys of
                            [] -> zs
                            (y:ys) -> y:h4 ys zs
                  (x:xs) -> x:h3 xs ys zs
h4 xs ys = case xs of {[] -> ys; (x:xs) -> x:h4 xs ys}
```

```
D[app (app xs ys) zs] = h3 xs ys zs
```

```
h3 xs ys zs = case xs of
                    [] -> case ys of
                               [] -> zs
                               (y:ys) -> y:h4 ys zs
                    (x:xs) -> x:h3 xs ys zs
h4 xs ys = case xs of {[] -> ys; (x:xs) -> x:h4 xs ys}
```

(Almost) Follow Sørensen et al. (1996)

$E ::= [\,]\ |\ E\ es\ |\ \text{case } E \text{ of } alts$

$D[x] = x$

$D[k\ es] = k\ D[es]$

$D[E<(\backslash xs.e)\ es>] = D[E<\text{let } xs = es \text{ in } e>]$

$D[E<\text{let } x = e \text{ in } f>] = D[E<[e/x]f>] \text{ if } x \in strict(f)$

$\qquad \text{and } x \in linear(f)$

$\qquad = \text{let } x = D[e] \text{ in } D[E<f>]$

$D[E<\text{case } x \text{ of } \{k_i\ x_i \text{ -> } e_i\}>]$

$\qquad = \text{case } x \text{ of } \{k_i\ x_i \text{ -> } D[[k_i\ x_i/x]E<e_i>]\}$

$D[E<\text{case } k_j\ es \text{ of } \{k_i\ x_i \text{ -> } e_i\}>] = D[E<\text{let } x_j = es \text{ in } e_j>]$

(Almost) Follow Sørensen et al. (1996)

```
E ::= [] | E es | case E of alts
```

```
D[x]  = x
D[k es] = k D[es]
D[E<(\xs.e) es>] = D[E<let xs = es in e>]
D[E<let x = e in f>] = D[E<[e/x]f>] if x∈strict(f)
                                      and x∈linear(f)
                     = let x = D[e] in D[E<f>]
D[E<case x of {k_i x_i -> e_i}>]
          = case x of {k_i x_i -> D[[k_i x_i/x]E<e_i>]}
D[E<case k_j es of {k_i x_i  -> e_i }>] = D[E<let x_j = es in e_j>]
```

(Almost) Follow Sørensen et al. (1996)

E ::= [] | E es | case E of alts

D[x] = x

D[k es] = k D[es]

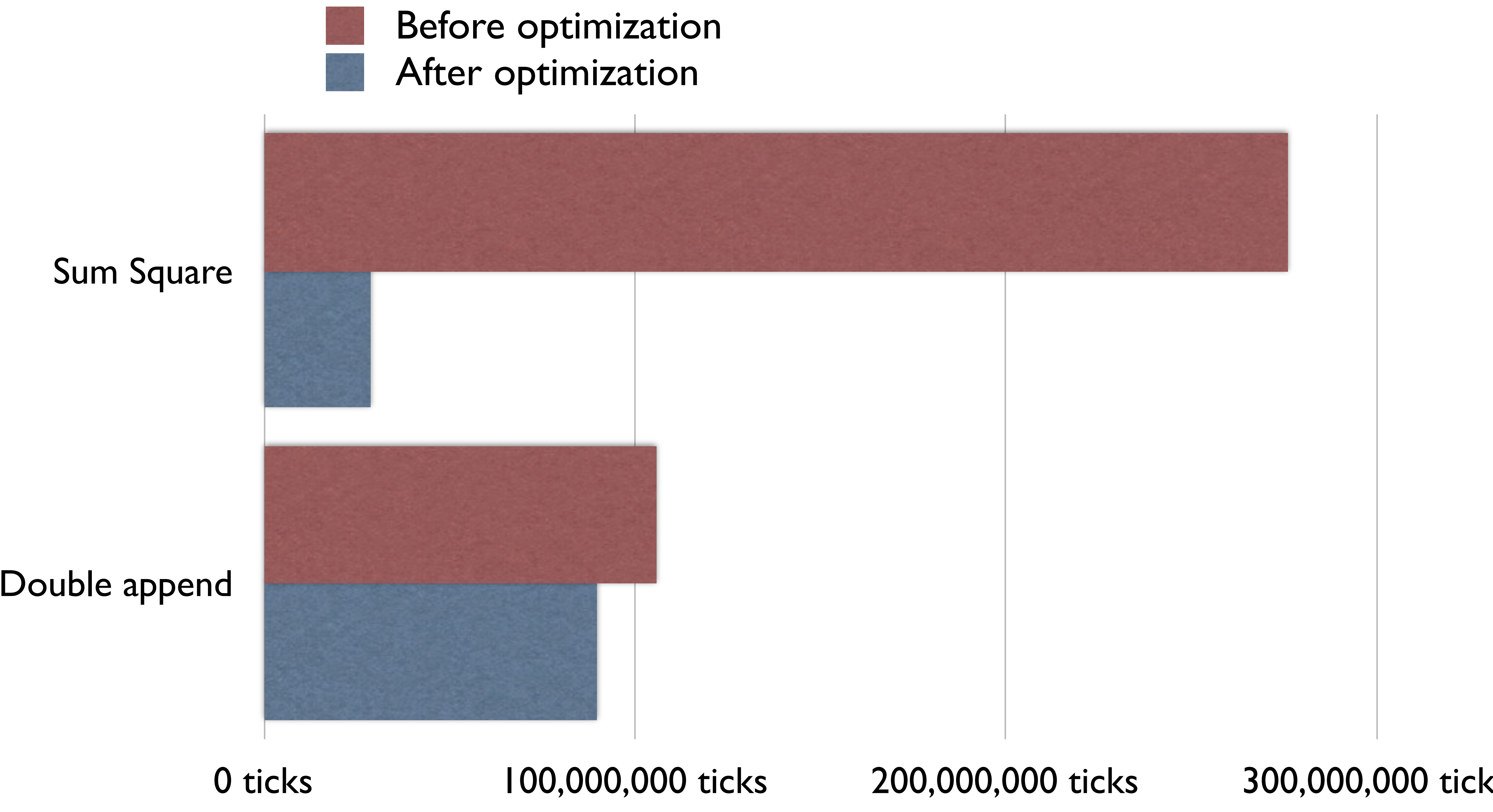
D[E<(\xs.e) es>] = D[E<let xs = es in e>]

D[E<let x = e in f>] = D[E<[e/x]f>] if x ∈ strict(f)
and x ∈ linear(f)
= let x = D[e] in D[E<f>]

D[E<case x of {k_i x_i -> e_i}>]
= case x of {k_i x_i -> D[[k_i x_i/x]E<e_i>]}

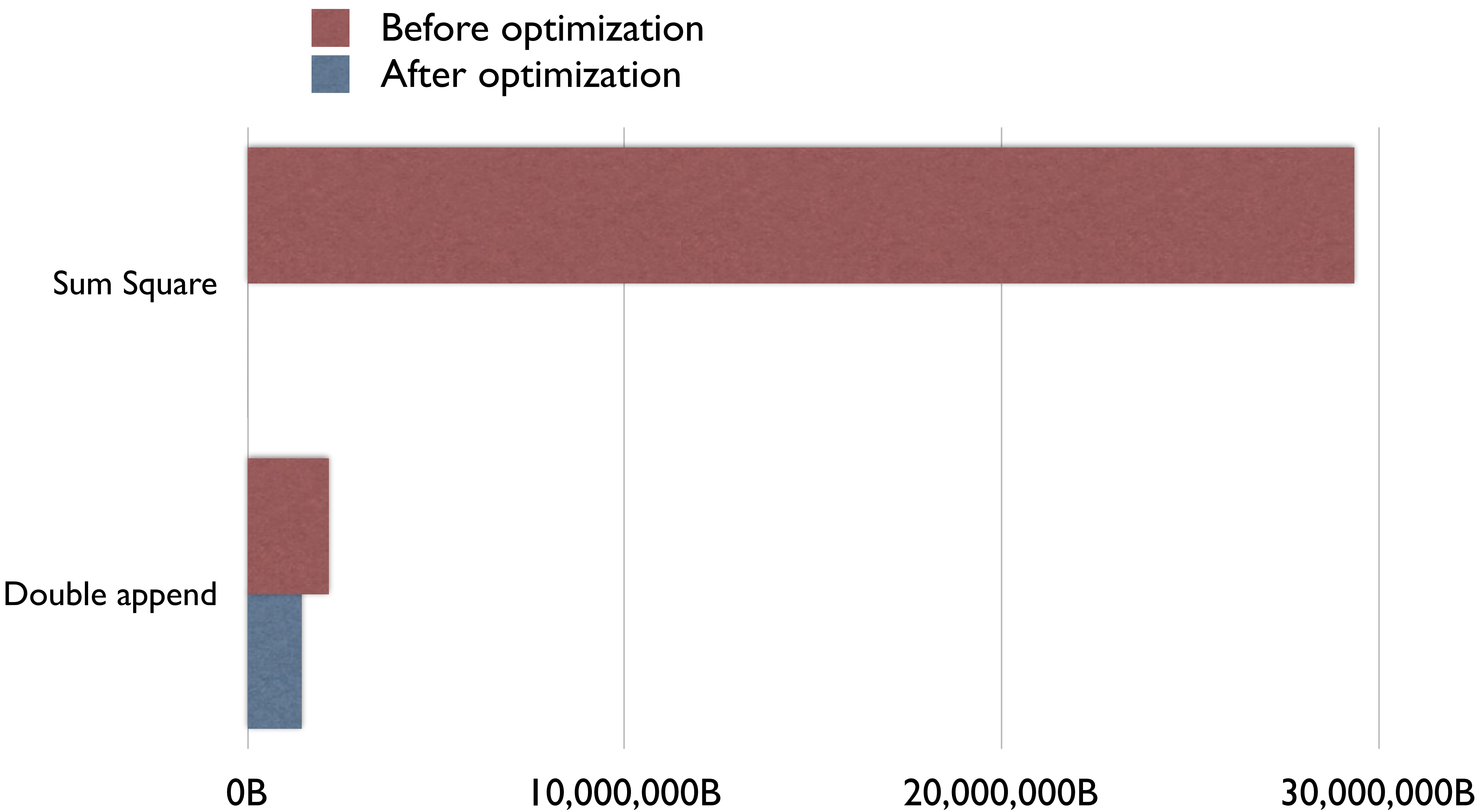
D[E<case k_j es of {k_i x_i -> e_i }>] = D[E<let x_j = es in e_j>]

Runtime



Before optimization
After optimization
Sum Square
Double append
0 ticks
100,000,000 ticks
200,000,000 ticks
300,000,000 tick

Memory allocated



But..

CBV-supercompilation is weaker than CBN-supercompilation

D[zip (map f xs') (map g ys')] =

CBV-supercompilation is weaker than CBN-supercompilation

```
D[zip (map f xs') (map g ys')] =
```

```
D[let ys = map g ys'
   in case map f xs' of
          [] -> []
          (x':xs') -> case ys of
                           [] -> []
                           (y':ys') -> (x', y'):zip xs' ys']
```

CBV-supercompilation is weaker than CBN-supercompilation

```
D[zip (map f xs') (map g ys')] =
```

```
D[let ys = map g ys'
   in case map f xs' of
          [] -> []
          (x':xs') -> case ys of
                            [] -> []
                            (y':ys') -> (x', y'):zip xs' ys']
```

Propagate Let-expressions

```
D[let ys = map g ys'
  in case (case xs' of
             [] -> []
             (x'':xs'') -> f x'':map f xs'') of
        [] -> []
        (x':xs') -> case ys of
                      [] -> []
                      (y':ys') -> (x', y'):zip xs' ys']
```

```
case xs of
   [] -> D[let ys = map g ys'
             in case [] of
                  [] -> []
                  (x':xs') -> case ys of
                                [] -> []
                                (y':ys') -> (x', y'):zip xs' ys']
   (x'':xs'') -> D[let ys = map g ys'
                     in case f x'':map f xs'' of
                          [] -> []
                          (x':xs') -> case ys of
                                        [] -> []
                                        (y':ys') -> (x', y'):zip xs' ys']
```

```
D[zip (map f xs') (map g ys')] = h5 f xs' g ys'
```

```
h5 f xs' g ys' =
      case xs' of
          [] -> let ys = map g ys' in []
          (z:zs) -> case ys' of
                        [] -> let x' = f z, xs' = map f zs in []
                        (z':zs') -> (f z, g z'):h5 f zs g zs'
```

The Case for Termination Analysis

```
h5 f xs' g ys' =
    case xs' of
      [] -> let ys = map g ys' in []
      (z:zs) -> case ys' of
                  [] -> let x' = f z, xs' = map f zs in []
                  (z':zs') -> (f z, g z'):h5 f zs g zs'
```

Extended Let Rule

D[E<let x = e in f>] B
= D[E<let B in f>] O if terminates(e)
and $x \notin fv(f)$
= D[E<let B in [e/x]f>] O if $x \in strict(f)$
and $x \in linear(f)$
= D[E<f>] (B U (x, e))

Future Work

- More measurements on real programs
- Investigate cheaper methods for ensuring termination
- What about accumulating parameters?

Related Work

- Supercompilers:
  - Scp4 (Nemytykh 2003)
  - Mitchell (2008, 2010)
  - Bolingbroke & Peyton Jones (2010)
  - Reich et al. (2010)

Conclusions

Supercompilation for call-by-value languages:

- can be strengthened to close in on call-by-name supercompilers