Efficient Online Partial Evaluation

Eijiro Sumii
Naoki Kobayashi

University of Tokyo

({sumii,koba}@yl.is.s.u-tokyo.ac.jp)

Partial Evaluation

source program **p(s,d)**

+

static input **s´**

Partial Evaluation

source program **p(s,d)**

+

static input **s´**

|

Reduce/residualize static/dynamic portions of **p(s´,d)**

↓

Partial Evaluation

source program **p(s,d)**

+

static input **s´**

|

Reduce/residualize static/dynamic portions of **p(s´,d)**

↓

specialized program $\mathbf{p_{s'}(d)}$

Partial Evaluation

source program $\mathbf{p(s,d)}$

+

static input $\mathbf{s'}$

|

Reduce/residualize static/dynamic portions of $\mathbf{p(s',d)}$

↓

specialized program $\mathbf{p_{s'}(d)}$ s.t.

$\mathbf{p(s',d)} = \mathbf{p_{s'}(d)}$ for any dynamic input $\mathbf{d}$

(Hopefully, the r.h.s. is faster)

Online PE and Offline PE

When to decide "static or dynamic"?

Online PE and Offline PE

When to decide "static or dynamic"?

– During specialization, with static input

$\Rightarrow$ Online PE

Online PE and Offline PE

When to decide "static or dynamic"?

- During specialization, with static input

  $\Rightarrow$ Online PE

- Before specialization, without static input

  $\Rightarrow$ Offline PE

Merits and Demerits of Online PE and Offline PE

- Online PE is
  - more precise, but...

Merits and Demerits of Online PE and Offline PE

- Online PE is
  - more precise, but
  - less efficient
    - Takes 10-100 times as much PE time

Merits and Demerits of Online PE and Offline PE

- Online PE is
  - more precise, but
  - less efficient
    - Takes 10-100 times as much PE time
- Offline PE is
  - more efficient, but...

Merits and Demerits of Online PE and Offline PE

- Online PE is
  - more precise, but
  - less efficient
    - Takes 10-100 times as much PE time
- Offline PE is
  - more efficient, but
  - less precise
    - Requires more binding-time improvment

Our Goal and Approach

Combine

- the precision of online PE, and
- the efficiency of offline PE

by

- going back to a naive online partial evaluator, and
- optimizing it without losing its precision
  - State-based let-insertion
  - Cogen approach to online PE
  - Type-based use analysis

Overview

- Introduction
- Related work
- A naive online partial evaluator
- Our optimizations
  - State-based let-insertion
  - Cogen approach to online PE
  - Type-based use analysis
- Experiments
- Conclusion

Overview

- Introduction
- Related work
- A naive online partial evaluator
- Our optimizations
  - State-based let-insertion
  - Cogen approach to online PE
  - Type-based use analysis
- Experiments
- Conclusion

Related Work (I)

To reduce interpretive overheads

- Self-application in online PE [Ruf-93, Sperber-96, etc.]

  $\Rightarrow$ Cogen approach is simpler and faster

Related Work (I)

To reduce interpretive overheads

- Self-application in online PE [Ruf-93, Sperber-96, etc.]

  $\Rightarrow$ Cogen approach is simpler and faster

- Cogen approach to offline PE [Thiemann-96, etc.]

  $\Rightarrow$ We adopted it into online PE

Related Work (II)

To reduce unnecessary computations

- (Monovariant) BTA's
  - for offline PE [Henglein-91, Asai-99, etc.]
  - for online PE [Ruf-93, Sperber-96]

Related Work (II)

To reduce unnecessary computations

- (Monovariant) BTA's
  - for offline PE [Henglein-91, Asai-99, etc.]
  - for online PE [Ruf-93, Sperber-96]
- Abstract occurrence counting analysis [Bondorf-90]

$\Rightarrow$ Our analysis subsumes all of these in a single framework

Overview

- Introduction
- Related work
- A naive online partial evaluator
- Our optimizations
  - State-based let-insertion
  - Cogen approach to online PE
  - Type-based use analysis
- Experiments
- Conclusion

A Naive Online Partial Evaluator

E *(expression)* **::= <u>x</u> | <u>λx</u>.E | E<u>@</u>E**

A Naive Online Partial Evaluator

E *(expression)* **::=** **x** | l**x.E** | **E@E**

onpe : exp ® env ® value option ´ exp

(symbolic value)

A Naive Online Partial Evaluator

E *(expression)* **::= x | l x.E | E@E**

onpe : exp ® env ® value option ´ exp

(symbolic value)

onpe(x) r **=** r **(x)**

A Naive Online Partial Evaluator

E *(expression)* **::= x | l x.E | E@E**

onpe : exp ® env ® value option ´ exp

(symbolic value)

onpe(x) r **=** r **(x)**

onpe(l x.E) r **=** á***Some*(l v.onpe(E)** r **[x:=v]),**

¼ ñ

A Naïve Online Partial Evaluator

E *(expression)* **::= x | l x.E | E@E**

onpe : exp ® env ® value option ´ exp

(symbolic value)

onpe(x) r = r **(x)**

onpe(l x.E) r **=** á***Some*(l v.onpe(E)** r **[x:=v]),**

l xà.$\#_2$(onpe(E) r **[x:=á*None*, xàñ])**ñ

A Naïve Online Partial Evaluator

E *(expression)* **::= x | l x.E | E@E**

onpe : exp ® env ® value option ´ exp

(symbolic value)

onpe(x)r **=** r **(x)**

onpe(l x.E)r **=** á***Some*****(**l **v.onpe(E)**r **[x:=v]),**

l **x**à**.#**$_2$**(onpe(E)**r **[x:=**á***None*****, x**àñ**])**ñ

onpe(E$_1$**@E**$_2$**)**r **=** ***let*** **a = onpe(E**$_2$**)**r ***in*** ¼

A Naive Online Partial Evaluator

E *(expression)* **::= x | l x.E | E@E**

onpe : exp ® env ® value option ´ exp

(symbolic value)

onpe(x) r **=** r **(x)**

onpe(l x.E) r **=** á***Some*(**l **v.onpe(E)** r **[x:=v]),**

l xà**.#$_2$(onpe(E)** r **[x:=**á***None*, x**àñ**])**ñ

onpe(E$_1$@E$_2$) r **=** ***let*** **a = onpe(E$_2$)** r ***in***

case **onpe(E$_1$)** r ***of* …**

A Naive Online Partial Evaluator

E *(expression)* **::= x | l x.E | E@E**

onpe : exp ® env ® value option ´ exp

(symbolic value)

onpe(x) r **=** r **(x)**

onpe(l x.E) r **=** á***Some*****(**l **v.onpe(E)** r **[x:=v]),**

l xà.#$_2$(onpe(E) r **[x:=á*None*, xàñ])**ñ

onpe(E$_1$@E$_2$) r **=** ***let*** **a = onpe(E$_2$)** r ***in***

case **onpe(E$_1$)** r ***of*** á***Some*****(v), _**ñ Þ **v a | …**

A Naive Online Partial Evaluator

E *(expression)* **::= x | l x.E | E@E**

onpe : exp ® env ® value option ´ exp

(symbolic value)

onpe(x) r **=** r **(x)**

onpe(l x.E) r **=** á***Some*****(**l **v.onpe(E)** r **[x:=v]),**

l xà**.#$_2$(onpe(E)** r **[x:=**á***None*****, x**àñ**])**ñ

onpe(E$_1$@E$_2$) r **=** ***let*** **a = onpe(E$_2$)** r ***in***

case **onpe(E$_1$)** r ***of*** á***Some*****(v), _**ñ Þ **v a**

| á***None*****, e**ñ Þ á***None*****, e@#$_2$(a)**ñ

Examples

- l**x.(**l**y.y)x** ® l**x.x**

Examples

- l**x.(**l**y.y)x** ® l**x.x**

$\#_2$(onpe(l**x.(**l**y.y)@x)[ ])**

Examples

- **l x.(l y.y)x ® l x.x**

#$_2$(onpe(l x.(l y.y)@x)[ ])

® #$_2$á…, l xà.#$_2$(onpe((l y.y)@x)[x:=á*None*, xàñ])ñ

Examples

- l**x.(**l**y.y)x** ® l**x.x**

#$_2$(onpe(l**x.(**l**y.y)@x)[ ])**

® **#$_2$**á**…,** l**x**à**.#$_2$(onpe((**l**y.y)@x)[x:=**á***None***, x**àñ**])**ñ

® **#$_2$**á**…,** l**x**à**.#$_2$(*let* a = onpe(x)[x:=**á***None***, x**àñ**]** ***in*** **…)**ñ

Examples

- l **x.(**l **y.y)x** ® l **x.x**

$\#_2$(onpe(l **x.(**l **y.y)@x)[ ])**

® **$\#_2$**á**…,** l **x**à**.$\#_2$(onpe((**l **y.y)@x)[x:=**á***None*,** **x**àñ**])**ñ

® **$\#_2$**á**…,** l **x**à**.$\#_2$(*let* a = onpe(x)[x:=**á***None*,** **x**àñ**]** ***in*** **…)**ñ

® **$\#_2$**á**…,** l **x**à**.$\#_2$(*let* a =** á***None*,** **x**àñ ***in*** **…)**ñ

Examples

- lx.(ly.y)x ® lx.x

$\#_2$(onpe(lx.(ly.y)@x)[ ])

® $\#_2$á…, lxà.$\#_2$(onpe((ly.y)@x)[x:=á*None*, xàñ])ñ

® $\#_2$á…, lxà.$\#_2$(*let* a = onpe(x)[x:=á*None*, xàñ] *in* …)ñ

® $\#_2$á…, lxà.$\#_2$(*let* a = á*None*, xàñ *in* …)ñ

® $\#_2$á…, lxà.$\#_2$(*let* a = á*None*, xàñ *in*
case onpe(ly.y)[…]
of …)ñ

Examples

- l x.(l y.y)x ® l x.x

#$_2$(onpe(l x.(l y.y)@x)[ ])

® #$_2$á…, l xà.#$_2$(onpe((l y.y)@x)[x:=á*None*, xàñ])ñ

® #$_2$á…, l xà.#$_2$(*let* a = onpe(x)[x:=á*None*, xàñ] *in* …)ñ

® #$_2$á…, l xà.#$_2$(*let* a = á*None*, xàñ *in* …)ñ

® #$_2$á…, l xà.#$_2$(*let* a = á*None*, xàñ *in*
case onpe(l y.y)[…]
of …)ñ

® #$_2$á…, l xà.#$_2$(*let* a = á*None*, xàñ *in*
case á*Some*(l v.onpe(y)[…, y:=v]), …ñ
of …)ñ

Examples

- l**x.(**l**y.y)x** ® l**x.x**

#$_2$(onpe(lx.(ly.y)@x)[ ])

® …

® #$_2$á…, l**x**à.#$_2$(*let* **a =** á*None*, **x**àñ *in*

case á*Some*(l**v.onpe(y)[…, y:=v])**, …ñ

of á*Some*(**v**), _ñ Þ **v a**

| …)ñ

Examples

- **l x.(l y.y)x ® l x.x**

#$_2$(onpe(l x.(l y.y)@x)[ ])

® …

® **#$_2$**á**…, l x**à**.#$_2$(*let* a =** á***None*, x**àñ ***in***

case á***Some*(l v.onpe(y)[…, y:=v]), …**ñ

of á***Some*(v), _**ñ Þ **v a**

| …)ñ

® **#$_2$**á**…, l x**à**.#$_2$((l v.onpe(y)[…, y:=v])** á***None*, x**àñ**)**ñ

Examples

- **l x.(l y.y)x ® l x.x**

#$_2$(onpe(l x.(l y.y)@x)[ ])

® **…**

® **#$_2$á…, l xà.#$_2$(*let* a = á*None*, xàñ *in***
case á*Some*(l v.onpe(y)[…, y:=v]), …ñ
_of_ á*Some*(v), _ñ Þ v a
| …)ñ

® **#$_2$á…, l xà.#$_2$((l v.onpe(y)[…, y:=v]) á*None*, xàñ)ñ**

® **#$_2$á…, l xà.#$_2$(onpe(y)[…, y:=á*None*, xàñ])ñ**

Examples

- **l x.(l y.y)x ® l x.x**

#$_2$(onpe(l x.(l y.y)@x)[ ])

® **…**

® **#$_2$á…, l x^{à}.#$_2$(*let* a = á*None*, x^{à}ñ *in***

***case* á*Some*(l v.onpe(y)[…, y:=v]), …ñ**

***of* á*Some*(v), _ñ Þ v a**

| …)ñ

® **#$_2$á…, l x^{à}.#$_2$((l v.onpe(y)[…, y:=v]) á*None*, x^{à}ñ)ñ**

® **#$_2$á…, l x^{à}.#$_2$(onpe(y)[…, y:=á*None*, x^{à}ñ])ñ**

® **#$_2$á…, l x^{à}.#$_2$(á*None*, x^{à}ñ)ñ**

Examples

- **l x.(l y.y)x ® l x.x**

#$_2$(onpe(l x.(l y.y)@x)[ ])

® **…**

® **#$_2$á…, l xà.#$_2$(*let* a = á*None*, xàñ *in***

case á*Some*(l v.onpe(y)[…, y:=v]), …ñ

of á*Some*(v), _ñ Þ v a

| …)ñ

® **#$_2$á…, l xà.#$_2$((l v.onpe(y)[…, y:=v]) á*None*, xàñ)ñ**

® **#$_2$á…, l xà.#$_2$(onpe(y)[…, y:=á*None*, xàñ])ñ**

® **#$_2$á…, l xà.#$_2$(á*None*, xàñ)ñ**

® **#$_2$á…, l xà.xàñ**

Examples

- **lx.(ly.y)x ® lx.x**

$\#_2$(onpe(lx.(ly.y)@x)[ ])

® …

® **$\#_2$á…, lxà.$\#_2$(*let* a = á*None*, xàñ *in***
**   *case* á*Some*(lv.onpe(y)[…, y:=v]), …ñ**
**   *of* á*Some*(v), _ñ Þ v a**
** | …)ñ**

® **$\#_2$á…, lxà.$\#_2$((lv.onpe(y)[…, y:=v]) á*None*, xàñ)ñ**

® **$\#_2$á…, lxà.$\#_2$(onpe(y)[…, y:=á*None*, xàñ])ñ**

® **$\#_2$á…, lxà.$\#_2$(á*None*, xàñ)ñ**

® **$\#_2$á…, lxà.xàñ**

® **lxà.xà**

Examples

- l**x.(**l**y.y)x** ® l**x.x**
  - Reduces expressions inside functions

Examples

- l**x.(**l**y.y)x** ® l**x.x**
  - Reduces expressions inside functions
- ***let* f =** l**x.1** ***in*** á**f 2, f**ñ ® á**1,** l**x.1**ñ
  - Uses both the static value and the dynamic expression of **f**

    cf. "both" BTA [Asai-99]

Examples

- l **x.(**l **y.y)x** ® l **x.x**
  - Reduces expressions inside functions
- ***let* f =** l **x.1 *in*** á**f 2, f**ñ ® á**1,** l **x.1**ñ
  - Uses both the static value and the dynamic expression of **f**

    cf. "both" BTA [Asai-99]
- l **x.1+(*if* true *then* 2 *else* x)** ® l **x.3**
  - Doesn't require context duplication
    - cf. continuation-based PE [Lawall-Danvy-94]

Let-Insertion is Necessary

- to avoid code duplication

  let* f = l x.1+2 *in á**f, f**ñ

  ® ***let* f = l x.3 *in*** á**f, f**ñ

  rather than á l **x.3,** l **x.3**ñ

Let-Insertion is Necessary

- to avoid code duplication
- to preserve semantics under side-effects

  l**f.1+(*let* x = f 2 *in* 3)**

  ® l**f.*let* x = f 2 *in* 4**

  rather than l**f.4**

Continuation-Based Let-Insertion [Lawall-Danvy-94]

Insert let-bindings by manipulating delimited continuations

- Creates a let-insertion point by delimiting a context

$$\textbf{delimit-let(e)} \equiv \textbf{\textit{reset}(e)}$$

Continuation-Based Let-Insertion [Lawall-Danvy-94]

Insert let-bindings by manipulating delimited continuations

- Creates a let-insertion point by delimiting a context

delimit-let(e) ° *reset*(e)

- Inserts a let-binding by extracting the delimited context

add-let(e) ° *shift*(l k. let xà = e in k xà)

Continuation-Based Let-Insertion in Offline PE

λf.(1+(*let* x = f@2 *in* 3))

Continuation-Based Let-Insertion in Offline PE

$\underline{\lambda f}$.$\underline{(}$1+(*let* x = $\underline{f@2}$ *in* 3))$\underline{)}$

$\Rightarrow$ **$\underline{\lambda f}$.delimit-let$\underline{(}$1+(*let* x = add-let($\underline{f@2}$) *in* 3))$\underline{)}$**

Continuation-Based Let-Insertion in Offline PE

l f.(1+(*let* x = f@2 *in* 3))

Þ **l f.delimit-let(1+(*let* x = add-let(f@2) *in* 3))**

® **l f.*reset*(1+(*let* x =**

shift(l k. let xà = (f@2) in k xà) _in_ 3))

Continuation-Based Let-Insertion in Offline PE

l f.(1+(*let* x = f@2 *in* 3))

Þ **l f.delimit-let(1+(*let* x = add-let(f@2) *in* 3))**

® **l f.*reset*(1+(*let* x =**

shift(l k. let xà = (f@2) in k xà) _in_ 3))

® **l f.let xà = (f@2) in (k xà)**

where **k[ ] = 1+(*let* x = [ ] *in* 3)**

Continuation-Based Let-Insertion in Offline PE

l f.(1+(*let* x = f@2 *in* 3))

Þ l f.delimit-let(1+(*let* x = add-let(f@2) *in* 3))

® l f.*reset*(1+(*let* x =

shift(l k. let xà = (f@2) in k xà) *in* 3))

® l f.let xà = (f@2) in (k xà)

where k[ ] = 1+(*let* x = [ ] *in* 3)

® l f.let xà = (f@2) in (1+(*let* x = xà *in* 3))

Continuation-Based Let-Insertion in Offline PE

lf.(1+(*let* x = f@2 *in* 3))

Þ lf.delimit-let(1+(*let* x = add-let(f@2) *in* 3))

® lf.*reset*(1+(*let* x =

shift(lk. let xà = (f@2) in k xà) *in* 3))

® lf.let xà = (f@2) in (k xà)

where k[ ] = 1+(*let* x = [ ] *in* 3)

® **lf.let xà = (f@2) in (1+(*let* x = xà *in* 3))**

® **lf.let xà = (f@2) in 4**

Continuation-Based Let-Insertion in Online PE

onpe(x) r = r(x)

onpe(l x.E) r = á*Some*(l v.onpe(E) r [x:=v]),

(l xà.

(#$_2$(onpe(E) r [x:=á*None*, xàñ])))ñ

onpe(E$_1$@E$_2$) r = *let* a = onpe(E$_2$) r *in*

case onpe(E$_1$) r *of* á*Some*(v), _ñ Þ v a

| á*None*, eñ Þ á*None*, (e@#$_2$(a))ñ

Continuation-Based Let-Insertion in Online PE

onpe(x) r **=** r**(x)**

onpe(l **x.E)** r **=** á***Some*****(**l **v.onpe(E)** r **[x:=v]),**

add-let(l **x**à**.**

delimit-let(#$_2$(onpe(E) r **[x:=**á***None*****, x**àñ**])))**ñ

onpe(E$_1$@E$_2$) r **=** ***let*** **a = onpe(E$_2$)** r ***in***

case **onpe(E$_1$)** r ***of*** á***Some*****(v), _**ñ Þ **v a**

| á***None*****, e**ñ Þ á***None*****, add-let(e@#$_2$(a))**ñ

Problems of the Naive Online Partial Evaluator

Correct but inefficient, because of

- shift/reset for let-insertion

Problems of the Naive Online Partial Evaluator

Correct but inefficient, because of

- shift/reset for let-insertion
- interpretive overheads
  - environment manipulation
  - syntax dispatch
  - universal domain (in typed languages)

Problems of the Naive Online Partial Evaluator

Correct but inefficient, because of

- shift/reset for let-insertion
- interpretive overheads
  - environment manipulation
  - syntax dispatch
  - universal domain (in typed languages)
- unnecessary computations
  - unnecessary let-insertions
  - unnecessary ***Some/None*** tags
  - unused values/expressions

Overview

- Introduction
- Related work
- A naive online partial evaluator
- Our optimizations
  - State-based let-insertion
  - Cogen approach to online PE
  - Type-based use analysis
- Experiments
- Conclusion

Problems of the Naive Online Partial Evaluator

Correct but inefficient, because of

- shift/reset for let-insertion
- interpretive overheads
  - environment manipulation
  - syntax dispatch
  - universal domain (in typed languages)
- unnecessary computations
  - unnecessary let-insertions
  - unnecessary ***Some/None*** tags
  - unused values/expressions

State-Based Let-Insertion

context := l **e.e**

State-Based Let-Insertion

context := l e.e

add-let(e) º

(context := !context ∘ l x.<u>let</u> <u>z</u>à = e <u>in</u> x; <u>z</u>à)

State-Based Let-Insertion

context := l e.e

add-let(e) º

(context := !context ∘ l x.<u>let</u> <u>z</u>à = e <u>in</u> x; <u>z</u>à)

delimit-let(e) º

***let* tmp = !context *in* …**

State-Based Let-Insertion

context := l e.e

add-let(e) º

(context := !context ∘ l x.<u>let</u> <u>z</u>à = e <u>in</u> x; <u>z</u>à)

delimit-let(e) º

let* tmp = !context *in

(context := l e.e; …

State-Based Let-Insertion

context := l e.e

add-let(e) º

(context := !context ∘ l x.let zà = e in x; zà)

delimit-let(e) º

let* tmp = !context *in

(context := l e.e;

***let* body = e *in* …**

State-Based Let-Insertion

context := l e.e

add-let(e) º

(context := !context ∘ l x.<u>let</u> <u>z</u>à = e <u>in</u> x; <u>z</u>à)

delimit-let(e) º

let* tmp = !context *in

(context := l e.e;

let* body = e *in

***let* head = !context *in* …**

State-Based Let-Insertion

context := l e.e

add-let(e) º

  (context := !context ∘ l x.let zà = e in x; zà)

delimit-let(e) º

  let* tmp = !context *in

  (context := l e.e;

  let* body = e *in

  let* head = !context *in

  (context := tmp; …

State-Based Let-Insertion

context := l e.e

add-let(e) º

(context := !context ∘ l x.let zà **= e in x; z**à**)**

delimit-let(e) º

let **tmp = !context** *in*

(context := l e.e;

let **body = e** *in*

let **head = !context** *in*

(context := tmp;

head body))

Overview

- Introduction
- Related work
- A naive online partial evaluator
- Our optimizations
  - State-based let-insertion
  - Cogen approach to online PE
  - Type-based use analysis
- Experiments
- Conclusion

Problems of the Naive Online Partial Evaluator

Correct but inefficient, because of

- shift/reset for let-insertion
- interpretive overheads
  - environment manipulation
  - syntax dispatch
  - universal domain (in typed languages)
- unnecessary computations
  - unnecessary let-insertions
  - unnecessary ***Some/None*** tags
  - unused values/expressions

Interpretive Approach

source program **p(s,d)**

+

static input ***s´***

|

Partially-evaluate **p(s´,d)**

with an interpreter

↓

specialized program $\mathbf{p_{s'}(d)}$

Cogen Approach

source program **p(s,d)**

Cogen Approach

source program **p(s,d)**

↓

generating extension $\mathbf{cogen_p(s)}$

Cogen Approach

source program **p(s,d)**

↓

generating extension $\mathbf{cogen_p(s)}$

+

static input **s´**

Cogen Approach

source program **p(s,d)**

↓

generating extension $\mathbf{cogen_p(s)}$

+

static input **s´**

|

Execute $\mathbf{cogen_p(s')}$ *directly*

without an interpreter

↓

Cogen Approach

source program $\mathbf{p(s,d)}$

↓

generating extension $\mathbf{cogen_p(s)}$

+

static input $\mathbf{s'}$

|

Execute $\mathbf{cogen_p(s')}$ *directly*

without an interpreter

↓

specialized program $\mathbf{p_{s'}(d)}$

Offline-PE Combinators [Thiemann-96]

From an interpretive partial evaluator, derive a cogen

Offline-PE Combinators [Thiemann-96]

From an interpretive partial evaluator, derive a cogen using

- Higher-order abstract syntax [Pfenning-Elliott-88]
  - Substitute object-level bindings with meta-level bindings

Offline-PE Combinators [Thiemann-96]

From an interpretive partial evaluator, derive a cogen using

- Higher-order abstract syntax [Pfenning-Elliott-88]
  - Substitute object-level bindings with meta-level bindings
- Deforestation
  - Compose the syntax constructors with the partial evaluator

Online-PE Combinators

Define **abs** and **app** s.t.

onpe(lx.E)r » **abs(lx.E¢)** under r¢

onpe(E_1**@**E_2**)**r » **app(**E_1¢, E_2¢**)** under r¢

Online-PE Combinators

Define **abs** and **app** s.t.

onpe(lx.E)r » **abs(lx.E¢)** under r¢

onpe(E$_1$@E$_2$)r » **app(E$_1$¢, E$_2$¢)** under r¢

e.g.,

onpe(lx.(ly.y)@x)[ ]

» **abs(lx.app(abs(ly.y),x))**

= á**…, lx**à**.x**àñ

Online-PE Combinators

Define **abs** and **app** s.t.

onpe(lx.E)r » **abs(**l**x.E**¢**)** under r¢

onpe(E$_1$**@E**$_2$**)**r » **app(E**$_1$¢**, E**$_2$¢**)** under r¢

e.g.,

onpe(l**x.(**l**y.y)@x)[ ]**

» **abs(**l**x.app(abs(**l**y.y),x))**

= á**…,** l**x**à**.x**àñ

abs(f) = á***Some*(**l**v.f v), **l**x**à**.#**$_2$**(f** á***None*, x**àñ**]))**ñ

app(p, a) = ***case* p *of*** á***Some*(v), _**ñ Þ **v a**

| á***None*, e**ñ Þ á***None*, e@#**$_2$**(a)**ñ

Overview

- Introduction
- Related work
- A naive online partial evaluator
- Our optimizations
  - State-based let-insertion
  - Cogen approach to online PE
  - Type-based use analysis
- Experiments
- Conclusion

Problems of the Naive Online Partial Evaluator

Correct but inefficient, because of

- shift/reset for let-insertion
- interpretive overheads
  - environment manipulation
  - syntax dispatch
  - universal domain (in typed languages)
- unnecessary computations
  - unnecessary let-insertions
  - unnecessary ***Some/None*** tags
  - unused values/expressions

Unnecessary Computations in Online PE

- Unused values

  e.g., in l**x.1+2** ® l**x.3**,

  abs(l**x.**¼**)**

  = á***Some(*****l x.** ¼ **)**, l**x**à**.** ¼ ñ

  can be simplified to

  á**( )**, l**x**à**.** ¼ ñ

Unnecessary Computations in Online PE

- Unused values
- Unnecessary tags & unused expressions

e.g., in **(**l**x.x)a** ® **a**,

app(abs(l**x.x),a)**

= ***case*** **abs(**l**x.x)**

of á***Some*****(v), _**ñ Þ **v a** | á***None*****,** ¼ñ Þ ¼

= ***case*** á***Some*****(**l**x.x),** l**x**à**.** ¼ñ

of á***Some*****(v), _**ñ Þ **v a** | á***None*****,** ¼ñ Þ ¼

can be simplified to

case ál**x.x, ( )**ñ ***of*** á**v, _**ñ Þ **v a**

= ***let*** **v** = l**x.x** ***in*** **v a**

Unnecessary Computations in Online PE

- Unused values
- Unnecessary tags & unused expressions
- Unnecessary let-insertions

  An expression requires no let-insertion

  - if it has no side-effects, and
  - if it appears at most once
    in the specialized program

Type-Based Use Analysis

Count the uses of values/expressions

r *(raw type)* **::=** a | t ® t | t ´ t

t *(annotated type)* **::=** $r^{(s,d)}$

Type-Based Use Analysis

Count the uses of values/expressions

r *(raw type)* **::=** a | t ® t | t ´ t

t *(type)* **::=** r$^{(\mathbf{s},\mathbf{d})}$

s *(static use)* **::=** **0** *(never)* | w *(always)* | **T** *(sometimes)*

– Whether a static value is available or not

Type-Based Use Analysis

Count the uses of values/expressions

r *(raw type)* **::=** a | t ® t | t ´ t

t *(type)* **::=** $r^{(s,d)}$

s *(static use)* **::=** **0** *(never)* | w *(always)* |
T *(sometimes)*

– Whether a static value is available or not

d *(dynamic use)* **::=** **0** *(never)* |
1 *(at most once)* | w *(any number of times)*

– How many times a dynamic expression appears in the specialized program

Examples

- ***let*** **f =** l **x.x in f 3**

Examples

- ***let*** **f : *int*** ® $^{(w,0)}$ ***int*** **=** l**x.x in f 3**

  Þ **3**

Examples

- ***let*** **f :** ***int*** ® $^{(w,0)}$ ***int*** **= l x.x in f 3**

  Þ **3**

- ***let*** **f = l x.x** ***in*** **f**

Examples

- ***let*** **f :** ***int*** ®$^{(w,0)}$ ***int*** **= l x.x in f 3**

  Þ **3**

- ***let*** **f :** ***int*** ®$^{(0,1)}$ ***int*** **= l x.x** ***in*** **f**

  Þ l **x.x**

Examples

- ***let* f : *int* ®$^{(w,0)}$ *int* = l x.x in f 3**
  Þ 3
- ***let* f : *int* ®$^{(0,1)}$ *int* = l x.x *in* f**
  Þ l x.x
- ***let* f = l x.x *in* áf 3, fñ**

Examples

- ***let*** **f :** ***int*** ® **(**w**,0)** ***int*** **=** l**x.x in f 3**
  Þ **3**
- ***let*** **f :** ***int*** ® **(0,1)** ***int*** **=** l**x.x** ***in*** **f**
  Þ l**x.x**
- ***let*** **f :** ***int*** ® **(**w**,1)** ***int*** **=** l**x.x** ***in*** á**f 3, f**ñ
  Þ á**3,** l**x.x**ñ

Examples

- ***let*** **f :** ***int*** ® $^{(w,0)}$ ***int*** **=** l**x.x in f 3**

  Þ **3**
- ***let*** **f :** ***int*** ® $^{(0,1)}$ ***int*** **=** l**x.x** ***in*** **f**

  Þ l**x.x**
- ***let*** **f :** ***int*** ® $^{(w,1)}$ ***int*** **=** l**x.x** ***in*** á**f 3, f**ñ

  Þ á**3,** l**x.x**ñ
- ***let*** **f =** l**x.x** ***in*** á**f 3, f, f**ñ

Examples

- ***let*** **f :** ***int*** ® (w,0) ***int*** **= l x.x in f 3**

  Þ **3**

- ***let*** **f :** ***int*** ® (0,1) ***int*** **= l x.x** ***in*** **f**

  Þ l **x.x**

- ***let*** **f :** ***int*** ® (w,1) ***int*** **= l x.x** ***in*** á**f 3, f**ñ

  Þ á**3,** l **x.x**ñ

- ***let*** **f :** ***int*** ® (w,w) ***int*** **= l x.x** ***in*** á**f 3, f, f**ñ

  Þ ***let*** **f = l x.x** ***in*** á**3, f, f**ñ

More Examples

- ***let*** **f = l x.x** ***in***
  let **g = l y.f** ***in***
  g

More Examples

- ***let* f : $\alpha \rightarrow^{(0,1)} \alpha = \lambda$x.x *in***

  let* g : $\beta \rightarrow^{(0,1)} \alpha \rightarrow^{(0,1)} \alpha = \lambda$y.f *in

  g

  $\Rightarrow$ λ**y.**λ**x.x**

More Examples

- ***let*** **f :** a ®$^{(0,1)}$ a = l**x.x** ***in***
  let **g :** b ®$^{(0,1)}$ a ®$^{(0,1)}$ a = l**y.f** ***in***
  g
  Þ l**y.**l**x.x**
- ***let*** **f =** l**x.x** ***in***
  let **g =** l**y.f** ***in***
  á**g 1, g 2**ñ

More Examples

- ***let*** **f :** a ®$^{(0,1)}$ a = l**x.x** ***in***
  let **g :** b ®$^{(0,1)}$ a ®$^{(0,1)}$ a = l**y.f** ***in***
  g
  Þ l**y.**l**x.x**
- ***let*** **f :** a ®$^{(0,w)}$ a = l**x.x** ***in***
  let **g :** b ®$^{(w,0)}$ a ®$^{(0,1)}$ a = l**y.f** ***in***
  á**g 1, g 2**ñ
  Þ ***let*** **f** = l**x.x** ***in*** á**f, f**ñ

Typing Rule

Example: λ-abstraction

$G_{\mathbf{0}}$**, x :** $r_{\mathbf{1}}^{\mathbf{(s1,d1)}} \vdash$ **e :** $t_{\mathbf{2}}$

¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾

$G \vdash l$ **x.e :** $r_{\mathbf{1}}^{\mathbf{(s1,d1)}}$ ® $^{\mathbf{(s,d)}}$ $t_{\mathbf{2}}$

Typing Rule

Example: λ-abstraction

$$G \;?\; (s,d) \cdot G_0$$

$$G_0, x : r_1^{(s1,d1)} \vdash e : t_2$$

¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾

$$G \vdash l\, x.e : r_1^{(s1,d1)} \text{®}^{(s,d)}\; t_2$$

Typing Rule

Example: λ-abstraction

G ? **(s,d)** · G_0

d ¹ **0** Þ **s_1** ¹ w

G_0, **x :** $r_1^{(s1,d1)}$ ⊢ **e :** t_2

¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾ ¾

G ⊢ l**x.e :** $r_1^{(s1,d1)}$ ® $^{(s,d)}$ t_2

Type Inference

- Construct the type derivation
  - assigning variables for uses, and
  - generating constraints for the variables

Type Inference

- Construct the type derivation
  - assigning variables for uses, and
  - generating constraints for the variables
- Solve the constraints
  - beginning with the most conservative approximation ($\mathbf{s} = \mathbf{T}$ and $\mathbf{d} = \mathrm{w}$ for all $\mathbf{s}$ and $\mathbf{d}$), and
  - refining it with iterations (linear w.r.t. the number of use variables)

Overview

- Introduction
- Related work
- A naive online partial evaluator
- Our optimizations
  - State-based let-insertion
  - Cogen approach to online PE
  - Type-based use analysis
- Experiments
- Conclusion

Conditions

- Mobile Pentium II 400MHz
- 128MB Main Memory
- Debian/Linux 2.2.10
- SML/NJ 110.0.3

Applications

- imp

  An interpreter for a simple imperative language

- match1

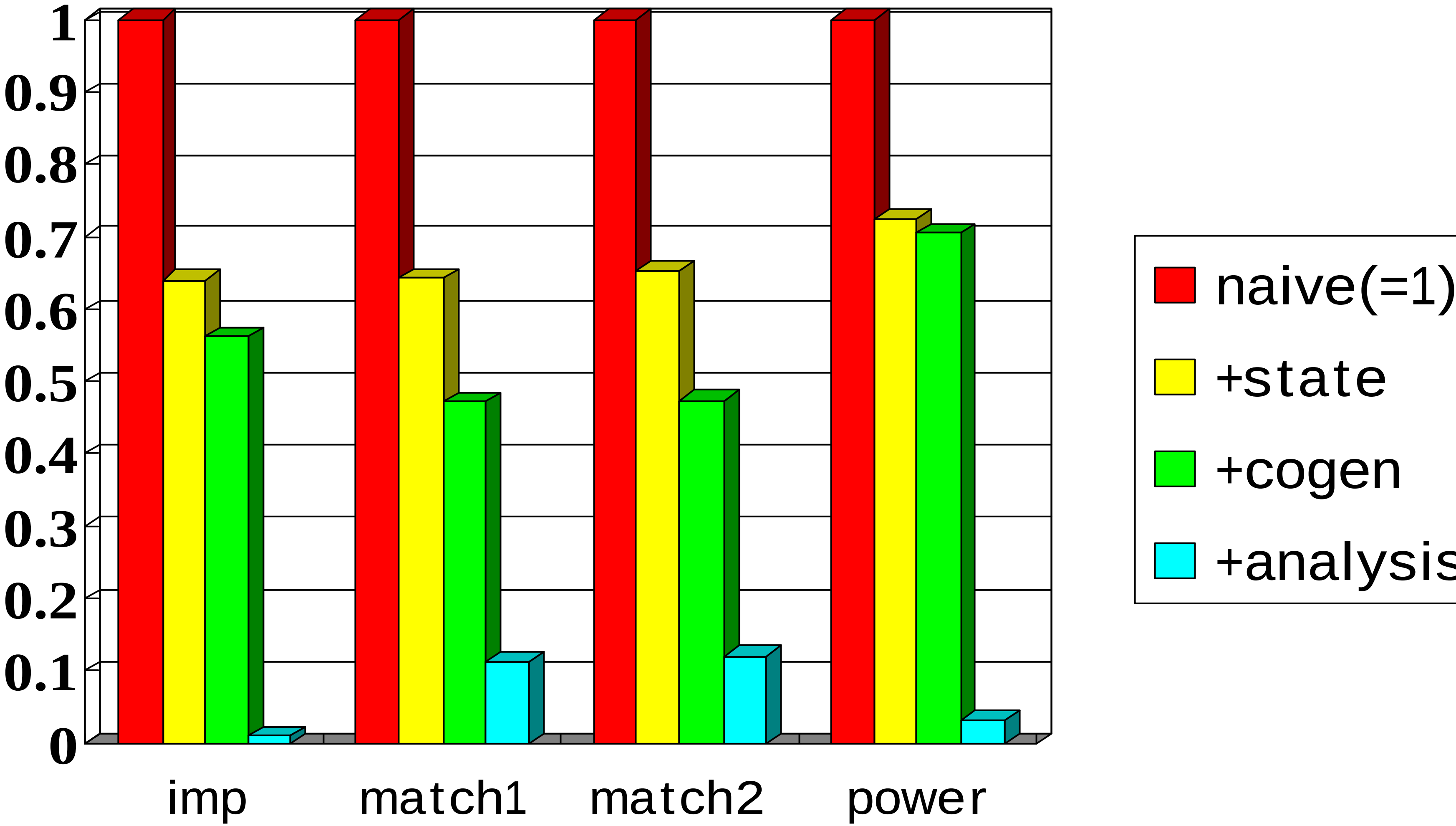
  A pattern matcher with the pattern static

- match2

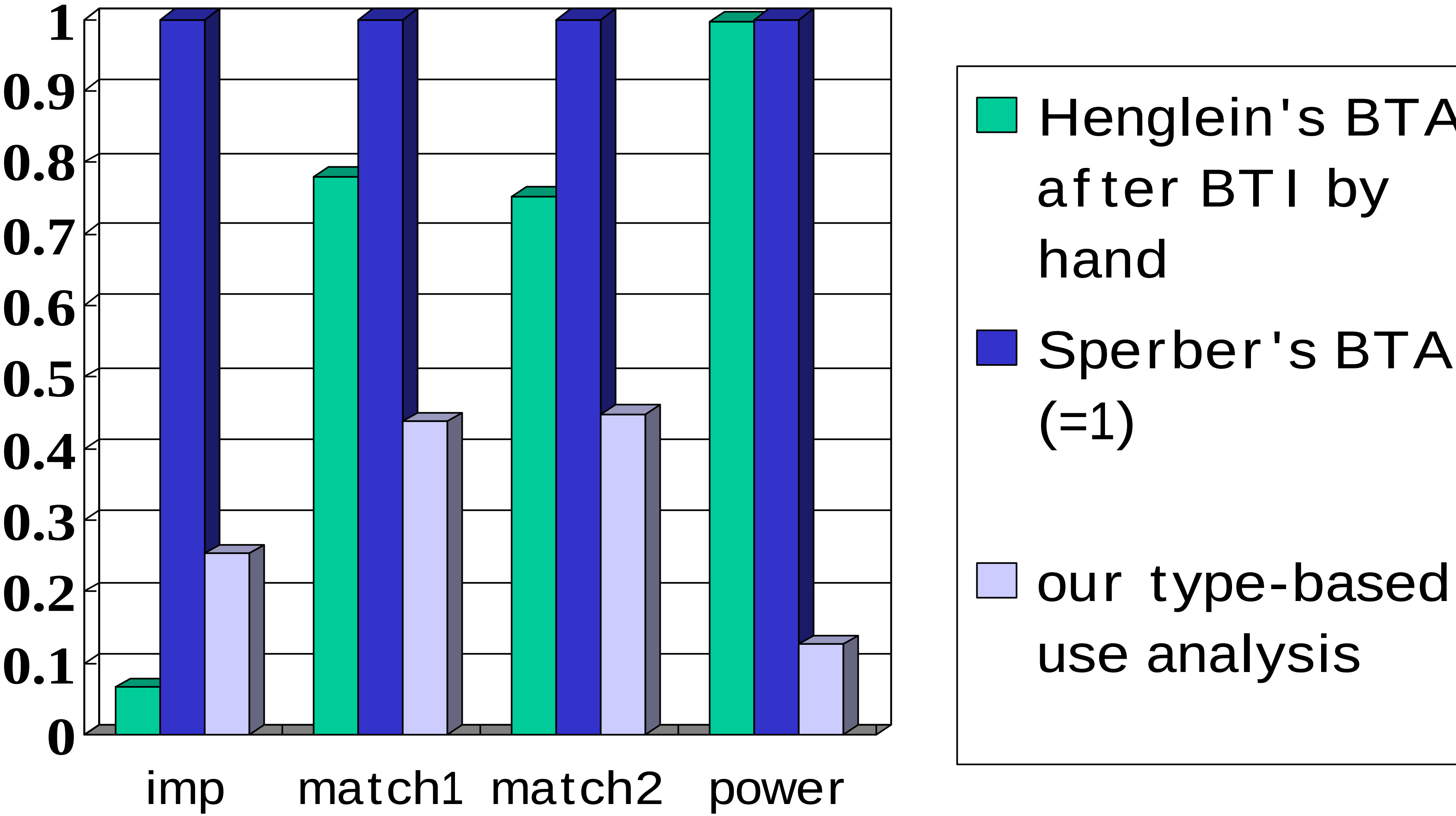
  The same pattern matcher with the string static

- power

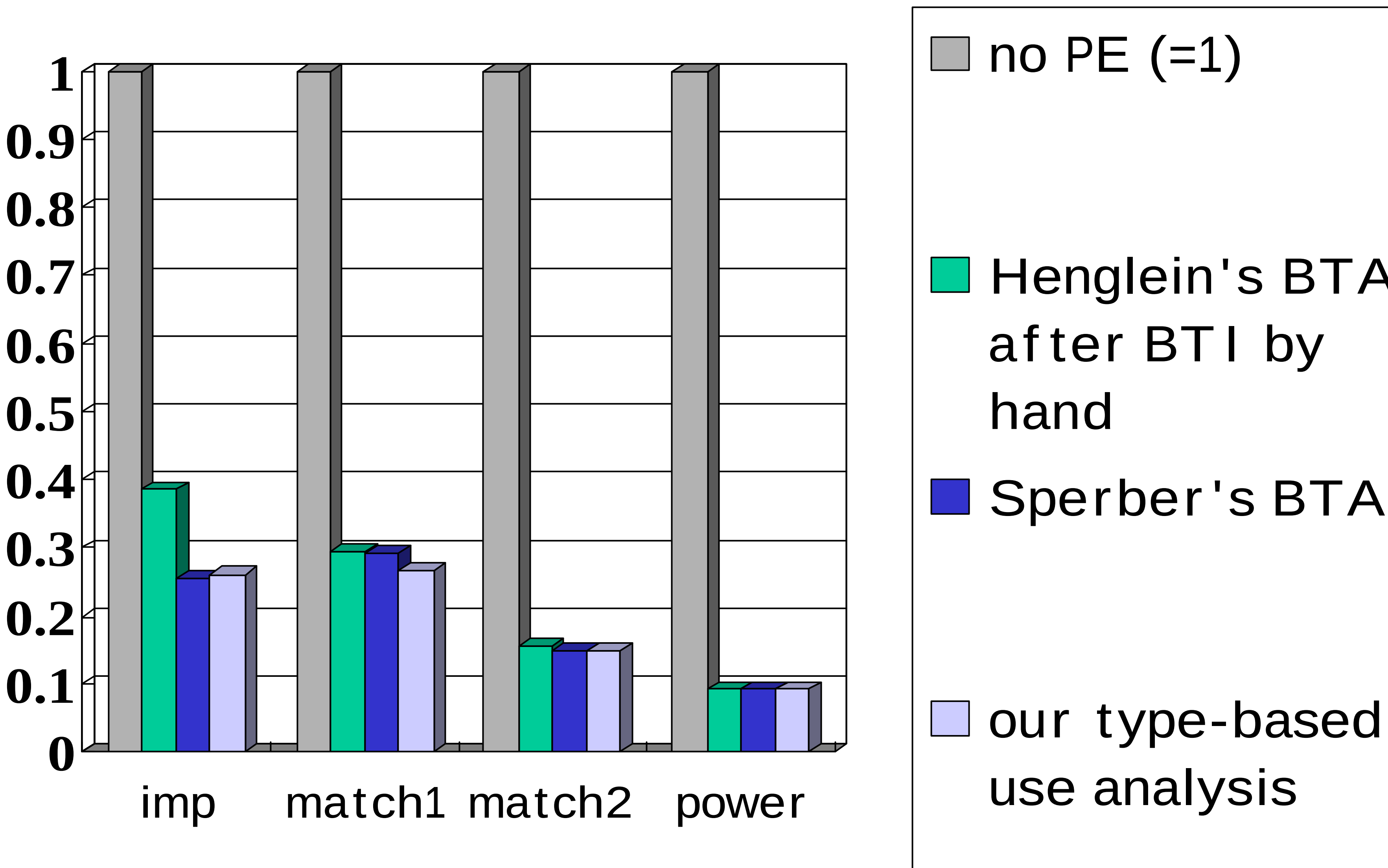
Effects of Optimizations: Time for Partial Evaluators



Comparison of BTA's: Time for Generating Extensions



Comparison of BTA's: Time for Specialized Programs



Conclusion

- We presented "hybrid" PE that combines
  - the precision of online PE, and
  - the efficiency of offline PE

Conclusion

- We presented "hybrid" PE that combines
  - the precision of online PE, and
  - the efficiency of offline PE
- Our future work includes
  - correctness proof
  - termination guarantee

Conclusion

- We presented "hybrid" PE that combines
  - the precision of online PE, and
  - the efficiency of offline PE
- Our future work includes
  - correctness proof
  - termination guarantee
- The paper to appear in PEPM'00 is available from **http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/**