Controlling Partial Evaluation

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Outline

Outline of talk:

- Partial Evaluators and the Futamura Projections
- Combinatorial Explosion and Binding Time Analysis (BTA)
- PEV's and Mogensen's Partial Evaluator
- Strategies
- Current Research Results and Research Goals

Why Partial Evaluation

In software engineering, we like two things to be true:

- Our programs run quickly.
- Our programmers write (reliable!) code quickly.

(Perceived) tradeoff: generality vs. efficiency

Partial evaluators can take a program that was written to solve a general problem and turn it into a program to solve a specific problem.

- e.g., circuit simulator can be specialized to simulate a specific circuit.

Definition of a Partial Evaluator

- Let M be a program such that takes inputs s and d, and produces x.
- An *interpreter* is a program I that takes a program M and inputs s and d and produces x by running M.
  $I(M,(s,d)) \Rightarrow x$
- A *partial evaluator* is a program P that takes a program M and a subset of the inputs s, and produces a new program M_s.
  $P(M,s) \Rightarrow M_s$, where $I(M_s,d) \Rightarrow x$

$$\texttt{power}(n, x) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ \texttt{power}(n/2, x * x) & \text{if } n \text{ is even} \\ x * \texttt{power}(n-1, x) & \text{if } n \text{ is odd} \end{cases}$$

- $I(\texttt{power}, (5, 3)) \Rightarrow 243$
- $P(\texttt{power}, 5) \Rightarrow \texttt{power}_5(x) = x * x * x * x * x$

Example: `printf`

- $P(\texttt{printf}, \texttt{"\%d"}) \Rightarrow \texttt{print_int}$
- Partial evaluation *removes a layer of interpretation*.

Other applications

- Circuit simulations
- Astronomy (planetary motion)
- Parser combinators
- Interpreters and compilers
- String and pattern matching

- The behavior of the partial evaluator under *self-application* is described by the *Futamura Projections*.

Futamura Projection I

$$P(I, \texttt{power}) \Rightarrow I_{\texttt{power}}, \text{ where } I_{\texttt{power}}(n, x) = x^n$$

- We say that `power` has been *compiled* into $I_{\texttt{power}}$.

Futamura Projection II

$$P(P, I) \Rightarrow P_I, \text{ where } P_I(\texttt{power}) \Rightarrow I_{\texttt{power}}$$

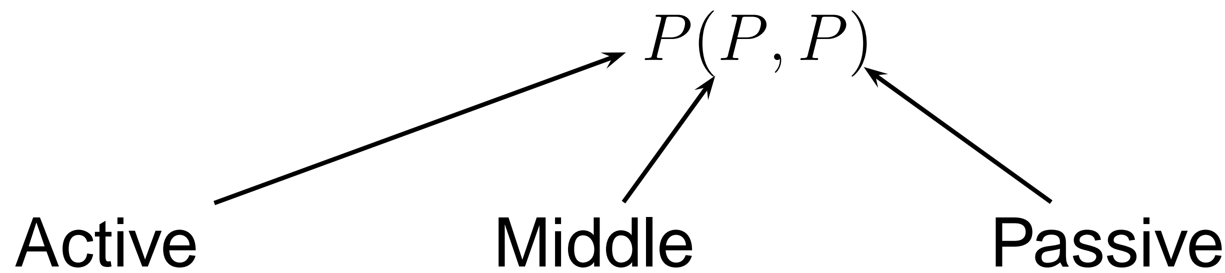
- P_I is a *compiler*
- $P_I(M)$ can run 4–10 times faster than $P(I, M)$.

Futamura Projection III

$$P(P, P) \Rightarrow P_P, \text{ where } P_P(I) \Rightarrow P_I$$

- P_P is also called P_{gen}, a *compiler generator*
- $P_P(I)$ can run 4–10 times faster than $P(P, I)$.

```
P(s,d) = ... if known(d) then compute(s,d)
                         else residualize(s,d) ...
```



- Active P knows both s and d, and so it reduces to result of `compute(s,d)`.
- Passive P knows nothing, so it stays the same.
- Middle P knows s, but does not know whether it will know d yet. Therefore the `if` statement is left, with `compute(s,d)` and `residualize(s,d)` (partially) expanded out.

```
P(s,d) = ... if known(d) then compute(s,d)
                          else residualize(s,d) ...
```

- We know that d in the middle evaluator will be known, even though we don't know what value it will have eventually. Therefore we can place an annotation that will allow `known(d)` to be reduced to `true`.
  Now, only `compute(s,d)` needs to be partially expanded.
- If a term is not annotated, it is assumed to be unknown, and `residualize(s,d)` is expanded.
- These annotations allowed the first successful self-application in Mix.

We have two kinds of partial evaluators.

Online

- Easy to write — very much like an interpreter.
- Accurate
- Unstable — difficult to self-apply

Offline

- A preprocessing stage is necessary
- Not very accurate — missed opportunities for specialization.
- Very stable

- Levels of accuracy and stability can be traded.
- Is there another way to accomplish self-application?

Different directions taken by PE research....

- Mix / Binding Time Annotations
- Expand number/features of languages
- Binding Time Analysis
  - Binding time improvements
  - Type-directed partial evaluation
- Non-self applied
  - BTA used for efficiency
  - Hand-written code generators

λ-Calculus

- The λ-calculus is one of the simplest programming languages.

  Variables x

  Functions $\lambda x.x$, $\lambda ab.b$

  Application $(\lambda x.x\ \lambda y.y)$
- This language is Turing complete!
- "The little white rat of programming languages"

Grammar:

$$\begin{aligned} \Lambda \rightarrow\ & x \\ & \mid \lambda x_1 x_2 \ldots x_n.\Lambda \\ & \mid (\Lambda_1\ \Lambda_2\ \cdots\ \Lambda_n) \end{aligned}$$

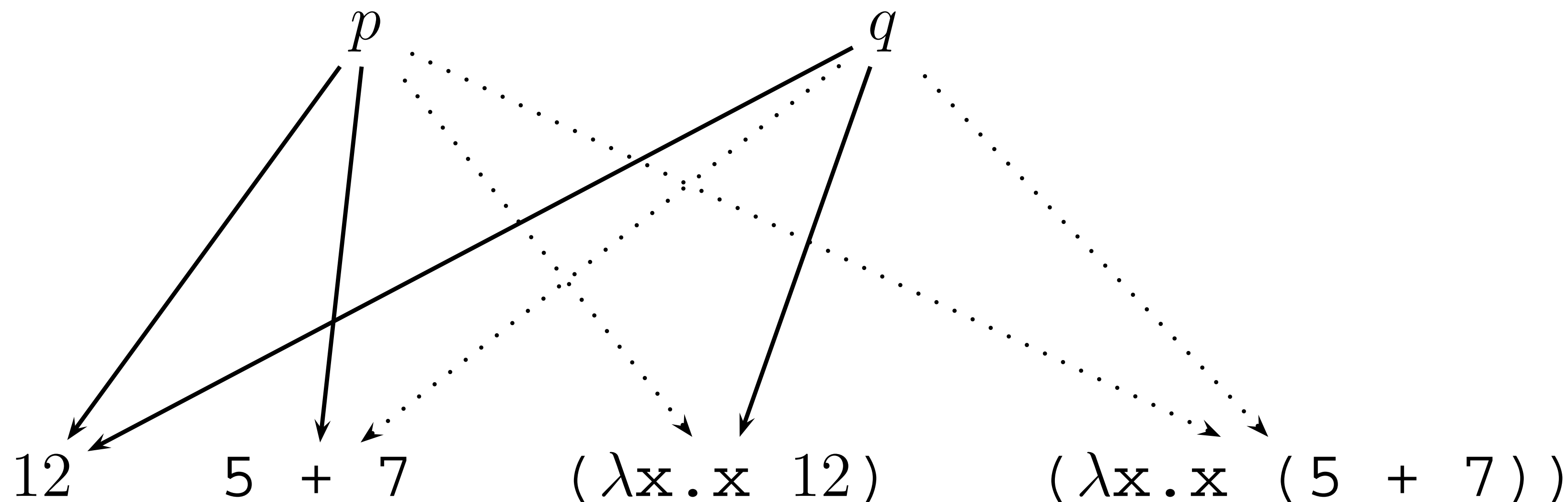
- Another method: *inline* the partial evaluator.
- Key observation 1: During partial evaluation, a code fragment has two identities — it is both code and data.
- We can represent that dual nature as a *Partial Evaluation Value* (*PEV*).
- Competitive advantage: simplicity.

$$PEV \equiv (PEV \rightarrow PEV) \times Exp$$

PEV Example

$$PEV \equiv (PEV \rightarrow PEV) \times Exp$$

Let $p = \lambda x.x$ and $q = 5 + 7$.



- We have four ways to combine two terms.

- Church Numerals:
  - e.g. Church Numeral 3 = $\lambda f\ x.(f\ (f\ (f\ x)))$
  - Numbers represented as functions
  - Argument is a function to be repeated
- Arbitrary Types:
  Suppose we have a type S, with constructors $S_1 \ldots S_n$. We can represent a term $t = S_i(t_1, t_2, \ldots, t_m)$ by

$$\bar{t} = \lambda x_1 \ldots x_n . x_i\ \overline{t_1} \cdots \overline{t_m}$$

The x_i represent functions that say what to do if the term t turns out to be an instance of the constructor S_i.

- Booleans and Pairs:
  - `T = ` $\lambda a\, b.\, a$ `F = ` $\lambda a\, b.\, b$
  - `Pair X Y = ` $\lambda pair.(pair\ X\ \ Y)$
- λ-calculus expressions:
  - e.g. $(\lambda x.x\ \lambda y.y)$ = $\lambda a\ b.(a\ (b\ \lambda x.x)\ (b\ \lambda y.y))$
  - Expressions represented as functions

A handles application:

```
1  A := λm n. (m T n);
```

D handles residualization:

$$1 \quad (\mathtt{D}\ \mathtt{x}\ \mathtt{y}) \longrightarrow (\mathtt{D}\ \overline{x})\ \mathtt{y} \longrightarrow (\mathtt{D}\ \overline{(x\ \ y)})$$

- Key Observation 2: The need to make decisions in the partial evaluator can be reduced by making residualization look like specialization. $(D\ x)\ T\ m \Rightarrow (D\ (x\ m_2))$

B handles abstractions:

```
1 B := λg . (Pair g λa b. (b λw . (g (D λa b. w) F a b
```

Once these combinators are defined, the partial evaluator itself is very simple.

```
1 R := λm. (m A B F);
2 M := λm n. (R λa b . (a (m a b) (n a b)));
```

1. Convert expression to HOAS:
   $(\lambda x.x\ \lambda y.y) = \lambda a\ b.(a\ (b\ \lambda x.x)\ (b\ \lambda y.y))$
2. Apply result to the combinators $(A\ (B\ \lambda x.x)\ (B\ \lambda y.y))$

- Result: the partial evaluator has been *inlined*.

Limitations

- No control — we can only apply.
- Code explosion due to "non-critical reduction of nonlinear redexes"
- Inability to process terms not having a normal form (e.g., Y)

- Change the type:
  $Strategy \equiv PEV \to PEV \to Result$
  $PEV \equiv Strategy \to Result$
  $Result \equiv (PEV \to Result) \times Exp$
- Change A:

```
1 A := λm n. (m T n);     before
2 A := λm n s. (s m n); after
```

- Change R, M[a]

```
1 R := λm. (m A B);
2 S := λm n s. (R λa b. (a (m a b) (n a b)) s F);
```

[a] We call it S now to distinguish it from Mogensen's.

Strategy: Expand All

$$E_{\text{all}} \equiv (Y\ \lambda\sigma\ p.(p\ \sigma\ T))$$

- Algorithm: Expand everything.
- Equivalent to M.

$(A\ m\ n\ E_{\text{all}})$
$\Rightarrow (E_{\text{all}}\ m\ n)$
$\Rightarrow (m\ E_{\text{all}}\ T\ n)$
$\Rightarrow (m_1\ n) : Result$
(The subscript notation will be applied to PEVs to indicate the parts of the $Result$ resulting from an application to a strategy.)

Strategy: Expand None

$$E_{\text{none}} \equiv (Y\ \lambda\sigma\ p.(D(p\ \sigma\ F)\ \sigma\ T)$$

$$E_{\text{none}} \equiv (Y\ \lambda\sigma\ p\ q.(D(\lambda a\ b.a\ (p\ \sigma\ F)\ (p\ \sigma\ F))\ \sigma\ T)$$

- Algorithm: Don't expand anything.
- Equivalent to identity (except it takes longer).

$(A\ m\ n\ E_{\text{none}})$
$\Rightarrow (E_{\text{none}}\ m\ n)$
$\Rightarrow (D\ (m\ E_{\text{none}}\ F)\ E_{\text{none}}\ T\ n)$
$\Rightarrow (D\ m_2\ E_{\text{none}}\ T\ n)$
$\Rightarrow^*$ "$(D\ (m_2\ n_2))$": $Result$

Strategy: ExpandN

$$mkExpandN \equiv \lambda n.(n\ \lambda\sigma\ p.(p\sigma\ T)\ E_{\text{none}})$$

E.g., $(mkExpandN\ 3) \Rightarrow$
$(3\ \ \lambda\sigma\ p.(p\sigma\ T)\ E_{\text{none}})\ \ \Rightarrow$
$(\lambda\sigma\ p.(p\ \sigma\ T)\ \ \lambda\sigma\ p.(p\ \sigma\ T)\ \ \lambda\sigma\ p.(p\ \sigma\ T)\ \ E_{\text{none}})\ \ \Rightarrow$
$(\lambda\sigma\ p.(p\ \sigma\ T)\ \ \lambda\sigma\ p.(p\ \sigma\ T)\ \ \lambda p.(p\ E_{\text{none}}\ T)\ \ \Rightarrow)$
$(\lambda\sigma\ p.(p\ \sigma\ T)\ \ \lambda p.(p\ E_{n,1}\ T))$
$(\lambda p.(p\ E_{n,2}\ T))$

- If you consider Y as a Church representation of ∞, then *Expand All* is a special case of *ExpandN*.

Strategy: ExpandSecond

Consider $(\lambda ab.(f\ a\ b)\ A\ B)$, with A is dynamic but B is static.

- We would like to reduce it to $(\lambda a.(f\ a\ B)\ A)$
- The first A “blocks” the reduction.
- The η-rule: $M \equiv \lambda x.(M\ x)$
- Normal way to process $(M\ N)$: residualize and build.

$$\lambda y.(App(Abs(x, ((M_1\ (D\ Var(x)))_1\ y)_2), X))$$

Code for Expand Second

$$
\begin{array}{l}
\Sigma_{2nd,n} \;=\; \lambda \pi \pi'. \\
\quad \textbf{let } r \;=\; (\pi\ \Sigma_{2nd,n}) \textbf{ in} \\
\quad \textbf{let } r' \;=\; (\pi'\ \Sigma_{2nd,n}) \textbf{ in} \\
\qquad \textbf{if } (size\ r'_2) \;<\; n \\
\qquad\qquad \textbf{then } (r_1\ \pi') \\
\qquad\qquad \textbf{else } \lambda s.(s \\
\qquad\qquad\qquad \lambda y. App(Abs(\lambda x.((r_1\ (D\ Var(x)))_1\ y)), r'_2) \\
\qquad\qquad\qquad App(r\ F, r'\ F))
\end{array}
$$

- Expand Marked: By changing the representation of the HOAS, we can add annotations to the code that can tell the strategies whether or not to expand.
- Expand Linear: Mark abstractions that are nonlinear and residualize them.
    - This didn't work as well as we'd hoped. The problem with M is not "reduction of nonlinear redexes"—it's the *non-critical* reduction of nonlinear redexes....

We were able to show...

- Strategies have the ability to make tradeoffs — more time spent in the partial evaluator usually means less time spent running the residual program.
- First and Second Futamura Projections work as expected.
- Third Projection explodes. Why?
  - Reducer not efficient enough?
    - Tried FOAS/HOAS hybrid
    - Many interpreter optimizations
  - Looping?
    - S does have a normal form, though.
    - Transformation not that far from M.

Exp				Beta	CPU
M	M	M		232,621	38
M	M	S		350,358	61.48
M	S	M		26,630,792	3321 (55 min)
M	S	S		crashed	crashed
S	M	M	E_{all}	361,680	245
S	M	S	E_{all}	546,211	428
S	S	M	E_{all}	35,217,577	9762 (2h 42m)
S	S	S	E_{all}	crashed	crashed

Major Goals

- Fix S^3
  - We know what is causing non-termination.
  - Assertion: strategies can combine both on-line and off-line techniques.
- Other Questions to answer:
  - Can we have S^3 without resorting to BTA?
  - Can strategies make binding time improvements? (e.g, η-expansions, “the trick”)
  - How much language support will we need?

Other Goals

- Modularity
  - Strategies can be combined easily, built for specific programs.
  - Can strategies allow the reduction of terms with no normal form (e.g., Y recursion)?
- Scaling — can this work with a larger language?
  - Lambda calculus with integers, let.
  - Subset of Scheme?
  - Without HOAS?