Partial Evaluation

6.894 (OODL) Spring 2001

Gregory T. Sullivan

MIT AI Lab

Outline

- Introduction to Partial Evaluation
- Offline PE
- Online PE
- Runtime PE
- PE for OO
- Dynamic PE
- Futamura projections

Introduction to Partial Evaluation

The "Bible" of PE: [JGS93].

Partial evaluation takes a *program* P with n inputs, and a list of *static values* $v_1 \ldots v_m$, $m \leq n$, and produces a *specialized* program P' such that (for all values $v_{m+1} \ldots v_n$):

$$P'(v_{m+1}, \ldots, v_n) = P(v_1, \ldots, v_m, v_{m+1}, \ldots, v_n)$$

Partial evaluation can be viewed as a function PE from programs to programs satisfying the following equation:

$$[\![PE(P, \langle v_1, \ldots, v_m \rangle)(v_{m+1}, \ldots, v_n)]\!] = [\![P(v_1, \ldots, v_n)]\!]$$

where $[\![e]\!]$ means "evaluate e".

This is an instance of Kleene's S_{MN} theorem (1952), which has a constructive proof.

Why Do Partial Evaluation?

Simple Example:

A two-input program to compute x^n:

```
pow(n, x) = if n = 0 then 1
            else if even(n) then pow(n/2, x)^2
            else x * pow(n-1, x)
```

If specialized for n = 5, we get:

```
pow5(x) = x * ((x^2)^2)
```

- Partial Evaluation is useful when some inputs change much less frequently than others (are "glacial").
- Note that specializing `pow` on the argument x is not so useful.

Successful Uses of Partial Evaluation

Operating Systems: For example, specializing file reads for the single-reader case.

Graphics - Ray Tracing: Specializing with respect to a background scene.

Compiling and Compiler Generation: More on this later.

Binding Time Analysis

Annotate all expressions as either static or dynamic.

The idea is that, given actual values for the static inputs, we can produce a specialized version of the program by slavishly obeying the annotations. We will *evaluate* the static expressions and *residualize* the dynamic expressions.

Rules:

1. Static input variables are static.
2. Base values (integers, booleans) are static.
3. For an assignment `x = e`, x is static only if e is static.
4. A conditional is static if its test expression is static.
5. A call is static only if all of its argument expressions are static.

Annotate pow

n static, x dynamic:

```
pow(n, x) = if n = 0 then 1
            else if even(n) then pow(n/2, x)^2
            else x * pow(n-1, x)
```

Specialize pow, n = 2

Every dynamic expression is residualized:

```
pow-2(x) = pow1(x)^2
pow-1(x) = x * pow0(x)
pow-0(x) = 1
```

And by unfolding calls to `pow-1` and `pow-0` we get:

```
pow-2(x) = (x * 1)^2 = x^2
```

Termination

What about (x static, y dynamic) in:

```
iterate(x, y) = if y ≠ 0 then iterate(x+1, y-1)
                else 0
```

Specialize iterate, x = 0

```
iterate-0(y) = if y ≠ 0 then iterate-1(y - 1) else 0
iterate-1(y) = if y ≠ 0 then iterate-2(y - 1) else 0
 ...
```

Specialization may infinitely loop. One solution is to note that in pow example, loop is on strictly decreasing values of static variable n.

Static infinite loops: The loops may not be encountered in actual usage, or it may be a bug in the program, but the compiler/partial evaluator should still avoid going into an infinite loop.

Another Example: (append)

Consider append, with first list (xs) static and second list (ys) dynamic:

```
(define (append xs ys)
  (if (null? xs)
      ys
      (cons (car xs)
            (append (cdr xs) ys ) ) ) )
```

Specialize append, xs = '(a b)

```
(define (append-ab ys) (cons 'a (append-b ys)))
(define (append-b ys) (cons 'b (append-() ys)))
(define (append-() ys) ys)
```

With unfolding (inlining), we get:

```
(define (append-ab ys) (cons 'a (cons 'b ys)))
```

Binding Time Analysis, (somewhat) formalized

$t \in \{S, D\}$	Binding Times
$\rho \in \mathit{VarId} \rightarrow t$	BT Environments
$\mathit{div} \in \mathit{FunId} \rightarrow \rho$	Divisions

Ordering on Binding Times:

$$S \leq D$$

extended to BT Environments and Divisions in the obvious way.

Goal:

Find the smallest Division.

Use *least upper bound* $\sqcup$: $S \sqcup D = D$.

Binding Time Analysis Function $\mathcal{B}_e[\![]\!]$

$\mathcal{B}_e[\![e]\!]$: BT Env$\rightarrow$Binding Time

$$
\begin{aligned}
\mathcal{B}_e[\![c]\!]\rho &= S \\
\mathcal{B}_e[\![x]\!]\rho &= \rho(x) \\
\mathcal{B}_e[\![(\mathsf{if}\ e_1\ e_2\ e_3)]\!]\rho &= \mathcal{B}_e[\![e_1]\!]\rho \sqcup \mathcal{B}_e[\![e_2]\!]\rho \sqcup \mathcal{B}_e[\![e_3]\!]\rho \\
\mathcal{B}_e[\![(f\ e_1 \ldots e_a)]\!]\rho &= \sqcup_{j=1}^{a} \mathcal{B}_e[\![e_j]\!]\rho
\end{aligned}
$$

Binding Time Propagation Function $\mathcal{B}_v[\![]\!]$

$\mathcal{B}_v[\![e]\!]$: BT Env$\rightarrow$ *Fun ID* $\rightarrow$BT Env

$\mathcal{B}_v[\![e]\!]\rho g$ is the least upper bound of the argument binding times in all calls to g in expression e. $x_1, \ldots, x_n$ are the parameters of g.

$$
\begin{aligned}
\mathcal{B}_v[\![c]\!]\rho g &= [x_i \mapsto S]_{i=1}^{n} \\
\mathcal{B}_v[\![x]\!]\rho g &= [x_i \mapsto S]_{i=1}^{n} \\
\mathcal{B}_v[\![(\mathsf{if}\ e_1\ e_2\ e_3)]\!]\rho g &= \mathcal{B}_v[\![e_1]\!]\rho g \sqcup \mathcal{B}_v[\![e_2]\!]\rho g \sqcup \mathcal{B}_v[\![e_3]\!]\rho g \\
\mathcal{B}_v[\![(f\ e_1 \ldots e_a)]\!]\rho &= t = \sqcup_{j=1}^{a} \mathcal{B}_v[\![e_j]\!]\rho g \quad \text{if } f \neq g \\
&= t \sqcup [x_i \mapsto \mathcal{B}_e[\![e_i]\!]\rho]_{i=1}^{n} \quad \text{if } f = g
\end{aligned}
$$

Monovariant Binding Time Division

For all functions $f_k(x_1 \ldots) = e_k$, $k = 1 \ldots m$,

$$(div\ \ f_k) = \sqcup_{i=1}^{m} \mathcal{B}_v[\![e_i]\!](div\ \ f_i) f_k \quad \text{for } k = 1 \ldots m$$

This definition is circular.

To find the smallest division, start with $(div\ \ f_k) = [x_i = S]_{i=1}^{n}$ for all functions f_k. Then compute $\sqcup_{i=1}^{m} \mathcal{B}_v[\![e_i]\!](div\ \ f_i) f_k$ for all k. If the division changes, update *div* and iterate.

Note that all variables mapping to time D is an upper bound.

Specialization

After annotating program according to BTA, core of specialization is:

```
(define (reduce e ρ)
  (case e of
    (number n) => n
    (static-if e1 e2 e3) =>
       (if (reduce e1 ρ)
           (reduce e2 ρ)
           (reduce e3 ρ))
    (dynamic-if e1 e2 e3) =>
       (list 'if (reduce e1 ρ)
                 (reduce e2 ρ)
                 (reduce e3 ρ))
    (static-call f e1 ...  en) =>
       (reduce ef [xi -> vi])  i ∈ 1 ... n
       where  vi = (reduce ei ρ)
    (dynamic-call f e1 ...  en) =>
       (list f::e'1..j e'j+1 ...  e'n
       where  e'_i = (reduce ei ρ)
       and the first j arguments of f are static)))
```

Polyvariant Specialization

Consider (and generate definitions for) different static-dynamic signatures for each function.

Online Partial Evaluation

Excellent reference: Erik Ruf's thesis, [Ruf93]

More like an interpreter, with "Symbolic Values".

A symbolic value is a type, a value, and code. The code component is guaranteed to return the given value in the current context.

Example of a symbolic value type: $(1\ \top_{\mathsf{int}}\ .\ \top)$ for a list with first element 1, second element an integer, and rest unknown.

Advantages of Online PE: Accuracy.

Advantages of Online PE: Speed (reuse BTA). Simplicity of specializer.

Runtime Partial Evaluation

Important reference: [CN96].

Advantages of Runtime PE: Can wait until runtime to discover values against which you will specialize.

Example (x static, y dynamic) in C:

```
int f(int x, int y) {
  int l;
  l = 2 * x;
  if (l == 2)
    l = l + y;
    else l = y * x;
    return l;
}
```

Runtime PE Example

```
int f_t(int y) {  // t1
  int l;          // t1
  switch unkonwn {
    case 1:
      l = [h1] + y;// t2
      break;
    case 2:
      l = y * [h2];// t3
      break;
    }
  return l;  // t4
}            // t4
```

```
void rt_spec_f(int x) {
  int l;
  dump_template(t1);
  l = 2 * x;
  if (l == 2) {
    dump_template(t2);
    instantiate_hole(t2, l);
  } else {
    dump_template(t3);
    instantiate_hole(t3, x);
  }
  dump_template(t4);
}
```

Partially Static Data

Parameterized partial evaluation [CK91] deals with data structures that may contain some static and some dynamic elements.

Saves the programmer from having to divid up data into strictly static and strictly dynamic components in order to help the partial evaluator.

Partial Evaluation for Object Oriented Languages

May specialize with respect to the type of an object, not just its value. This makes sense if you consider the type of an object a field, subject to parameterized partial evaluation.

In [SLCM99], they use declarations to drive the specializer.

When an object matches a set of criteria, the specified methods are specialized. Criteria are classified as stable or unstable.

[see the handout]

Eliminating Reflection Overhead in Java

From [BN00]:

```
public static void dumpFields(Object anObj)
        throws java.lang.IllegalAccessException {
    Field[] fields = anObj.getClass().getFields();
    for (int i = 0; i < fields.length; i++)
        System.out.println(fields[i].getName() +
            ":  " + fields[i].get(anObj));
}
```

Suppose `dumpFields` is called often with a parameter of type `Point`; we want a specialized version:

```
public static void dumpFieldsPoint(Point anObj) {
    System.out.println("x: "+anObj.x);
    System.out.println("y: "+anObj.y); }
```

Dynamic Partial Evaluation

[Sul01]

Question: can you accomplish partial evaluation at runtime without any compile time analysis?

Idea: Evaluation of an expression in a given environment returns a symbolic value: (value, type, expr). If type is fully static, original expression may be replaced by value.

Control: A method is called, say `foo(x:Object)`, with x an instance (3, 4) of `Point`. A decision is made to enable dynamic partial evaluation, with the `foo`'s argument specialized to `Point`, during this execution of `foo`. When `foo` is finished, we have not only the value of the call, but a new method `foo(x:Point)`, which is added to the `foo` generic function.

While `foo` is running, certain expressions, such as `x.bar()`, may be able to be optimized based on the knowledge that x is a `Point`.

Dynamic PE Example

$$(\mathsf{let}\ (a\ \top) = (\mathsf{if}\ (>\ b\ 0)\ 3\ 4)\ \mathsf{in}$$
$$(\mathsf{let}\ (c\ \tau_{\mathsf{int}}) = (\mathsf{if}\ (>\ d\ 0)\ 5\ 6)\ \mathsf{in}$$
$$(\mathsf{gf\text{-}call}\ \ g\ a\ b)))$$

in the following environment:

$$b \mapsto \langle 1, 1, eq(1)\rangle_{\mathsf{val}}$$
$$d \mapsto \langle 1, 1, \tau_{\mathsf{int}}\rangle_{\mathsf{val}}$$
$$g \mapsto \langle generic(\langle g_1, g_2\rangle, \langle \top\rangle, \top), -, eq(g)\rangle_{\mathsf{val}},$$
$$\text{where}\quad g_1 = closure(\langle x, y\rangle, \langle \top, \top\rangle, \top, [\![(+\ x\ y)]\!], \rho_{g_1})$$
$$g_2 = closure(\langle x, y\rangle, \langle \top, \tau_{\mathsf{int}}\rangle, \top, [\![x]\!], \rho_{g_2})$$

dynamically partially evaluates to:

$$\mathsf{let}\ (c\ \tau_{\mathsf{int}}) = (\mathsf{if}\ (>\ d\ 0)\ 5\ 6)\ \mathsf{in}\ \ 3$$

The identifier b is statically bound to the integer 1. d is also bound to 1, but has static type τ_{int}. The identifier g is statically bound[1] to a generic function of two methods – one with the most general specializers, and one specialized on integer values for its second argument. The first expression to evaluate is $(> \; b \; 0)$. Because b is completely static, the result value, *true* is completely static and the if expression can be folded. The result of the if expression is the fully static value 3. The identifier a is bound to the value 3, with residual expression 3, and gets static type $eq(3)$, which is the greatest lower bound (glb) of the declared type $\top$ and the result type $eq(3)$ of the expression. The comparison $(> \; d \; 0)$ also evaluates to *true*, but the value is not completely static because the static type of d is τ_{int}. The identifier c is bound to the value 5, residual expression $(\mathsf{if} \; (> \; d \; 0) \; 5 \; 6)$, and gets static type τ_{int}, which is the glb of the declared type τ_{int} and the type of the result of the expression, namely $\top$. The most applicable method for the call to g is g_2. Furthermore, static type τ_{int} of variable c is sufficient to statically select the most applicable method at the call, so the $\mathsf{gf\text{-}call}$ can be replaced by a simple call to

[1] $g \mapsto \langle -, -, eq(g) \rangle_{\mathsf{val}}$ means that g is completely static.

g_2. When the body of g_2 is executed, it returns the value of its first argument, a, which is fully static. Because the method can be statically selected, and because the result of the call is a fully static value, the whole gf-call expression can be folded to the literal expression 3. Thus the expression residualizes to:

$$\mathsf{let}\ (c\ \tau_{\mathsf{int}}) = (\mathsf{if}\ (>\ d\ 0)\ 5\ 6)\ \mathsf{in}\ \ 3$$

Dead variable elimination may eliminate the now useless let construct.

Dynamic Partial Evaluation Rules

$$\frac{\rho([\![x]\!]) = V \simeq \langle v, -, - \rangle_{\mathsf{val}}\ ; \quad check(v, \tau)}{[\![x]\!]\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle V \rangle, \rho, \tau)}$$

$$\frac{check(n, \tau)}{[\![n]\!]\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle \langle n, e, eq(n) \rangle_{\mathsf{val}} \rangle, \rho, \tau)}$$

$$\frac{\begin{array}{l}[\![e_0]\!]\ \ \rho\ \tau_{\mathsf{bool}} \Rightarrow V_0 \simeq \langle true, -, - \rangle_{\mathsf{val}} \\ [\![e_1]\!]\ \ \rho\ \tau \Rightarrow V\end{array}}{[\![(\mathsf{if}\, e_0\ \ e_1\ \ e_2)]\!]\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle V, V_0 \rangle, \rho, \tau)}$$

$$\frac{\begin{array}{l}[\![e_0]\!]\ \ \rho\ \tau_{\mathsf{bool}} \Rightarrow V_0 \simeq \langle false, -, - \rangle_{\mathsf{val}} \\ [\![e_2]\!]\ \ \rho\ \tau \Rightarrow V\end{array}}{[\![(\mathsf{if}\, e_0\ \ e_1\ \ e_2)]\!]\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle V, V_0 \rangle, \rho, \tau)}$$

$$\frac{\begin{array}{l}[\![e_0]\!]\ \ \rho\ \tau_{\mathsf{fun}} \Rightarrow \\ \quad V_f \simeq \langle closure(\langle [\![x_1]\!], \ldots, [\![x_n]\!] \rangle, \langle \tau_{\mathsf{arg}_1}, \ldots, \tau_{\mathsf{arg}_n} \rangle, \tau_{\mathsf{res}}, [\![e_f]\!], \rho_f), -, - \rangle_{\mathsf{val}} \\ [\![e_i]\!]\ \ \rho\ \tau_{\mathsf{arg}_i} \Rightarrow V_i \simeq \langle v_i, [\![e'_i]\!], \tau_i \rangle_{\mathsf{val}},\ \ i \in 1 \ldots n \\ [\![e_f]\!]\ \ \rho_f[x_i \mapsto \langle v_i, [\![e'_i]\!], glb(\tau_i, \tau_{\mathsf{arg}_i}) \rangle_{\mathsf{val}}],\ \ glb(\tau, \tau_{\mathsf{res}}) \Rightarrow V,\ \ i \in 1 \ldots n\end{array}}{[\![(\mathsf{call}\, e_0\ \ e_1\ \ \ldots\ \ e_n)]\!]\ \rho\ \tau \Rightarrow \mathcal{F}(e, \langle V, V_f, \langle V_1, \ldots, V_n \rangle \rangle, \rho, \tau)}$$

DVM Rules, continued

$$
\dfrac{
\begin{array}{l}
[\![e_0]\!]\ \rho\ \tau_{\mathsf{gf}} \Rightarrow V_g \simeq \langle generic(ms, \langle \tau_{\mathsf{g}_1}, \ldots, \tau_{\mathsf{g}_n} \rangle, \tau_{g_{\mathsf{res}}}), -, - \rangle_{\mathsf{val}} \\
[\![e_i]\!]\ \rho\ \tau_{\mathsf{g}_i} \Rightarrow V_i \simeq \langle v_i, [\![e'_i]\!], \tau_i \rangle_{\mathsf{val}},\ i \in 1 \ldots\ n \\
\textit{find-mam}(ms, \langle V_1, \ldots, V_n \rangle) \simeq \langle V_f, \textit{static?} \rangle \\
V_f \simeq \langle closure(\langle [\![x_1]\!], \ldots, [\![x_n]\!] \rangle, \langle \tau_{\mathsf{arg}_1}, \ldots, \tau_{\mathsf{arg}_n} \rangle, \tau_{\mathsf{res}},\ [\![e_f]\!],\ \rho_f), -, - \rangle_{\mathsf{val}} \\
(\textit{specialize?}, \langle \textit{spec-type}_1, \ldots \rangle) = \textit{choose-specialization}(V_g, V_f, \langle V_1, \ldots, V_n \rangle) \\
\textit{arg-type}_i = (\textit{specialize?} \Rightarrow \textit{spec-type}_i;\ glb(\tau_i, \tau_{\mathsf{arg}_i})),\ i \in 1 \ldots\ n \\
[\![e_f]\!]\ \rho_f[x_i \mapsto \langle v_i, [\![e'_i]\!], \textit{arg-type}_i \rangle_{\mathsf{val}}]\ glb(\tau, \tau_{\mathsf{res}}) \Rightarrow V,\ i \in 1 \ldots\ n
\end{array}
}{
\begin{array}{l}
[\![(\mathsf{gf\text{-}call}\, e_0\ e_1\ \ldots\ e_n)]\!]\, \rho\ \tau \Rightarrow \mathcal{F}(e, \textit{F-vals}, \rho, \tau),\ \text{where} \\
\quad \textit{F-vals} = \langle V, V_g, \langle V_1, \ldots, V_n \rangle, V_f, \mathsf{static?}, \textit{specialize?}, \langle \textit{spec-type}_1, \ldots \rangle \rangle
\end{array}
}
$$

$$
\dfrac{
\begin{array}{l}
[\![e_{\tau_i}]\!]\ \rho\ \tau_{\mathsf{type}} \Rightarrow V_{\tau_i} \simeq \langle v_{\tau_i}, -, - \rangle_{\mathsf{val}},\ i \in 1 \ldots\ n \\
[\![e_{\tau_{\mathsf{res}}}]\!]\ \rho\ \tau_{\mathsf{type}} \Rightarrow V_{\tau_{\mathsf{res}}} \simeq \langle v_{\tau_{\mathsf{res}}}, -, - \rangle_{\mathsf{val}} \\
v_f = closure(\langle [\![x_1]\!], \ldots, [\![x_n]\!] \rangle, \langle v_{\tau_1}, \ldots, v_{\tau_n} \rangle, v_{\tau_{\mathsf{res}}}, [\![e_0]\!], \rho) \\
V_f = \langle v_f, e, eq(v_f) \rangle_{\mathsf{val}}\ ; \quad check(v_f, \tau)
\end{array}
}{
\begin{array}{l}
[\![(\mathsf{lambda}(x_1\ e_{\tau_1}, \ldots, x_n\ e_{\tau_n}) e_{\tau_{\mathsf{res}}}\ .\ e_0)]\!]\, \rho\ \tau \Rightarrow \\
\quad \mathcal{F}(e, \langle V_f, \langle V_{\tau_1}, \ldots, V_{\tau_n} \rangle, V_{\tau_{\mathsf{res}}} \rangle, \rho, \tau)
\end{array}
}
$$

Finishing Rules for Dynamic Partial Evaluation

Variable Reference: $\mathcal{F}_{\mathrm{pe}}([\![x]\!], \langle V \rangle, \rho, \tau)$, where $V \simeq \langle v, e', \tau' \rangle_{\mathsf{val}}$

$= \mathit{if}\ \ \mathit{static?}(V)\ \&\ \ \mathit{expressible\text{-}as\text{-}literal?}(v)$ – if static and a literal

$\langle v, [\![v]\!], \tau' \rangle_{\mathsf{val}}$ – we can fold to a constant

V

Conditionals: $\mathcal{F}_{\mathrm{pe}}([\![(\mathsf{if}\, e_0\ e_1\ e_2)]\!], \langle V, V_0 \rangle, \rho, \tau),]$ where $V \simeq \langle v, e', \tau' \rangle_{\mathsf{val}}$ and $V_0 \simeq \langle v_0, e'_0, \tau_0 \rangle_{\mathsf{val}}$

$= \mathit{if}\ \ \mathit{static?}(V_0)$ – if test val is a constant,

V – we can eliminate the conditional

$\mathit{if}\ \ v_0$ – otherwise, rebuild the conditional

$\langle v, [\![(\mathsf{if}\ \ e'_0\ e'\ e_2)]\!], \top \rangle_{\mathsf{val}}$

$\langle v, [\![(\mathsf{if}\ \ e'_0\ e_1\ e')]\!], \top \rangle_{\mathsf{val}}$

Function call: $\mathcal{F}_{\text{pe}}([\![(\textsf{call}\, e_0\ e_1\ \ldots\ e_n)]\!], \langle V, V_f, \langle V_1, \ldots, V_n\rangle\rangle, \rho, \tau),]$ where
$V \simeq \langle v, e', \tau'\rangle_{\textsf{val}}$, $V_f \simeq \langle v_f, e'_f, \tau_f\rangle_{\textsf{val}}$, $V_i \simeq \langle v_i, e'_i, \tau_i\rangle_{\textsf{val}}, i \in 1 \ldots n$, and
$v_f \simeq \mathit{closure}(\langle [\![x_1]\!], \ldots, [\![x_n]\!]\rangle, \langle \tau_{\textsf{arg}_1}, \ldots, \tau_{\textsf{arg}_n}\rangle, \tau_{\textsf{res}},\ \rho_f)$

$= \mathit{if}\ \ \mathit{static?}(V_f)$ – if fun is a constant,

$\ldots \mathit{finish\text{-}known\text{-}function} \ldots$

– here if fun value is not constant

$\langle v, [\![(\textsf{call}\, e'_f\ e'_1 \ldots e'_n)]\!], \top\rangle_{\textsf{val}}$

where the code fragment *finish-known-function* is:

$(\mathit{if}\ \ \mathit{static?}(V)$ – if return val is a constant,

$\langle v, \mathit{val2exp}(v), \tau'\rangle_{\textsf{val}}$ – fold call

$\mathit{if}\ \ \mathit{inline?}(v_f)$ – if fun is a constant, may choose to inline

$\mathit{inline\text{-}call}(V, V_f, \langle V_1, \ldots, V_n\rangle, \rho, \tau)$

– here if fun is constant, but not inlining

$\langle v, [\![(\textsf{call}\ \ e'_f\ e'_1 \ldots e'_n)]\!], \tau_{\textsf{res}}\rangle_{\textsf{val}})$

Generic Function Call: $\mathcal{F}_{\text{pe}}(\llbracket(\textsf{gf-call}\, e_0\ e_1\ \ \ldots\ \ e_n)\rrbracket, \textit{F-vals}, \rho, \tau),]$
where

$\textit{F-vals} =$
$\langle V, V_g, \langle V_1, \ldots, V_n\rangle, V_f, \textsf{static?}, \textit{specialize?}, \langle \textit{spec-type}_1, \ldots\rangle\rangle,$
$V \simeq \langle v, e', \tau'\rangle_{\textsf{val}},\ V_g \simeq \langle v_g, -, -\rangle_{\textsf{val}},$
$v_g \simeq \textit{generic}(ms, \langle \tau_{\mathrm{g}_1}, \ldots, \tau_{\mathrm{g}_n}\rangle, \tau_{g_{\textsf{res}}}),$
$V_f \simeq \langle \textit{closure}(\langle \llbracket x_1\rrbracket, \ldots, \llbracket x_n\rrbracket\rangle, \langle \tau_{\textsf{arg}_1}, \ldots, \tau_{\textsf{arg}_n}\rangle, \tau_{\textsf{res}},\ e_f,\ \rho_f), -, -\rangle_{\textsf{val}},$
and $V_i \simeq \langle v_i, e'_i, \tau_i\rangle_{\textsf{val}},\ i \in 1 \ldots\ n$

$= \textit{if}\ \ \textit{specialize?}$

$\quad \textit{add-method}(v_g, \textit{closure}(\langle \llbracket x_1\rrbracket, \ldots, \llbracket x_n\rrbracket\rangle, \langle \textit{spec-type}_1, \ldots\rangle, e', \rho_f))$

$\quad \textit{if}\ \ \textit{static?}(V_g)$ – if generic is a constant,

$\quad\quad \textit{if}\ \ (\textit{static?})$ – if can statically dispatch

$\quad\quad\quad \ldots \textit{finish-known-function} \ldots$

$\quad\quad\quad \langle v, \llbracket(\textsf{gf-call}\ \ e'_g\ e'_1 \ldots e'_n)\rrbracket, \tau_{g_{\textsf{res}}}\rangle_{\textsf{val}}$ – can't statically dispatch

$\quad\quad$ – here if generic is not a constant

$\quad\quad \langle v, e, \top\rangle_{\textsf{val}}$

Dynamic Partial Evaluation, Overview

Two uses of Generic Functions:

1. As a language feature to be modeled.
2. As a way to manage newly created methods.

- Related to runtime *customization* of methods in Self & Cecil

Future Work

- When to specialize?
- Overhead / performance?
- Lower level (native)?
  - Get instruction traces, memory accesses using HP Dynamo
  - Post process/analyze – is same PC being hit with same subset of values in registers? Specialize w.r.t. those register-values, add to cache.
  - Consider how this might discover hotspots in interpreted code (as opposed to interpreter).

Futamura Projections

In 1971, Y. Futamura applied partial evualation to interpreters and compilers.

Defining Equation of Partial Evaluation:

$$[\![[\![PE]\!](prog, vals_1)]\!](vals_2) = [\![prog]\!](vals_1 \mid\mid vals_2)$$

First Futamura Projection

Given a partial evaluator *PE*, an interpreter *Int*, a source program *prog* (all in the same language), and some input values *vals*, we may "compile" the program to *obj* as follows:

$$obj = [\![PE]\!](Int, prog)$$

We can be assured, from the specification of *PE*, that

$$[\![obj]\!](vals) = [\![Int]\!](prog, vals)$$

Second Futamura Projection

We can generate a stand-alone compiler for the language that *Int* interprets as follows:

$$compiler = [\![PE]\!](PE, Int)$$

We can be verify its correctness as follows:

$$\begin{aligned} obj' &= [\![compiler]\!](prog) \\ &= [\![[\![PE]\!](PE, Int)]\!](prog) \\ &= [\![PE]\!](Int, prog) = obj \end{aligned}$$

Note that *PE* must be written in the same language it partially evaluates.

Third Futamura Projection

By exactly analogous reasoning, a *compiler generator* can be produced as follows:

$$compgen = [\![PE]\!](PE, PE)$$

which can be verified as follows:

$$\begin{aligned} compiler' &= [\![compgen]\!](int) \\ &= [\![[\![PE]\!](PE, PE)]\!](int) \\ &= [\![PE]\!](PE, int) = compiler \end{aligned}$$

Notes

Add lifts to examples (e.g. append example)?

References

[BN00] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in java. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM-00)*, pages 2–11, N.Y., January 22–23 2000. ACM Press.

[CK91] C. Consel and S. C. Khoo. Parameterized partial evaluation. In M. Billaud et al., editors, *Analyse Statique en Programmation Équationnelle, Fonctionnelle, et Logique, Bordeaux, France, Octobre 1991 (Bigre, vol. 74)*, pages 72–81. Rennes: IRISA, 1991.

[CN96] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg*

Beach, Florida, 21–24 January 1996, pages 145–156, New York, NY, USA, 1996. ACM Press.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[Ruf93] Erik Ruf. *Topics in online partial evaluation*. PhD thesis, Stanford Computer Systems Laboratory, March 1993.

[SLCM99] U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. *Lecture Notes in Computer Science*, 1628:367–??, 1999.

[Sul01] Gregory T. Sullivan. Dynamic partial evaluation. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects 2*, volume ? of *LNCS*, page ? Springer-Verlag, May 2001.