Hybrid Partial Evaluation

Amin Shali
Computer Science Department
University of Texas at Austin
amshali@cs.utexas.edu

William R. Cook
Computer Science Department
University of Texas at Austin
wcook@cs.utexas.edu

Abstract

Hybrid partial evaluation (HPE) is a pragmatic approach to partial evaluation that borrows ideas from both online and offline partial evaluation. HPE performs offline-style specialization using an online approach without static binding time analysis. The goal of HPE is to provide a practical and predictable level of optimization for programmers, with an implementation strategy that fits well within existing compilers or interpreters. HPE requires the programmer to specify where partial evaluation should be applied. It provides no termination guarantee and reports errors in situations that violate simple binding time rules, or have incorrect use of side effects in compile-time code. We formalize HPE for a small imperative object-oriented language and describe *Civet*, a straightforward implementation of HPE as a relatively simple extension of a Java compiler. Code optimized by Civet performs as well as and in some cases better than the output of a state-of-the-art offline partial evaluator.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Partial evaluation

General Terms Languages, Performance

Keywords Partial Evaluation, Object-Oriented Languages, Hybrid

1. Introduction

Object-oriented systems are increasingly based on configurable frameworks and reflection. These features are expensive at runtime, and the costs can limit the ambitions of framework developers in creating more powerful and general frameworks. These costs, however, are often unnecessary because a particular program typically configures and uses the frameworks in a specific way. Configuration files, data-driven programming and more sophisticated forms of model-driven development often involve dynamic interpretation of large amounts of relatively static data [22]. Avoiding the penalty of generality requires optimizations that cut across module boundaries to simplify the general framework operations with respect to the program-specific configuration data.

Partial evaluation is well suited to optimizing such programs. A partial evaluator can *specialize* a generic framework in the context of the usage pattern in a particular program. It can also optimize across interfaces, allowing programmers to write modular, general-purpose programs, with the assurance that they will be optimized automatically.

In this paper we present *hybrid partial evaluation* (HPE), a pragmatic approach to partial evaluation that is designed to be effective in existing object-oriented languages. Hybrid partial evaluation provides predictable and reliable optimizations, because the programmer explicitly identifies parts of the program that should be evaluated at *compile time* versus normal *runtime* evaluation [16]. The following example illustrates how HPE can be used to optimize a naive regular expression library.

```
1 Regex regex = CT(RegexParser.parse("(a|b)*(abb|a+b)"));
2 regex.execute(buffer);
```

The `CT` expression tells the compiler to instantiate the `Regex` object at compile time. The `execute` method is a simple, naive regular expression interpreter. When the `execute` method is invoked on a runtime buffer, HPE inlines and specializes the interpreter on the specific pattern, resulting in a set of static methods to efficiently interpret the finite state machine representing the regular expression. This example is discussed in more detail in Section 5.3.

We describe hybrid partial evaluation in the context of a small imperative object-oriented language. Like online partial evaluation, HPE does not perform binding time analysis. The system supports polyvariant specialization of methods and classes, and specialization of reflective operations. On the other hand, the kinds of specializations performed are similar to those performed by an offline partial evaluator. The goals of HPE are predictability, ease of implementation, and sufficient specialization to optimize common programs.

To achieve predictability, HPE requires programmer annotations to indicate which objects should be instantiated at compile time, and HPE prohibits migration of compile-time objects to runtime. HPE has a simple check to ensure that executing imperative code at compile time is consistent with the original semantics of the program. Hybrid partial evaluation rejects programs with incorrect binding times, rather than silently generating inefficient residual code. These restrictions allow developers to understand and rely on the optimizations performed by the partial evaluator.

To simplify implementation, a Hybrid partial evaluator is derived from an interpreter (or operational semantics) and avoids static binding time analysis. In addition, HPE provides no termination guarantee. If the partial evaluating compiler takes too long, the programmer must terminate it just as any other program with an infinite loop and rewrite the program to avoid the problem.

We have implemented hybrid partial evaluation within the JastAdd Java compiler [9] and used it to optimize a range of Java programs. Compared to JSpec [25], an existing offline partial evaluator for Java, hybrid partial evaluation generates code that is as efficient as JSpec's residual code. Initial results show an average 6 times speedup of specialized programs.

$\textbf{data}\ v = \texttt{null} \mid v_s \mid v_n \mid v_b \mid [v] \mid C{:}\rho$

$\textbf{type}\ \mathit{Prog} = \overline{\mathrm{CD}}$

$\textbf{data}\ \mathrm{CD} = \texttt{class}\ C\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}\overline{\texttt{var}\ x\texttt;}\ \texttt{init\{}e\texttt{\}}\ \overline{\mathrm{MD}}\texttt{\}}$

$\textbf{data}\ \mathit{mod} = \texttt{static} \mid \texttt{method}$

$\textbf{data}\ \mathrm{MD} = \mathit{mod}\ m\texttt{(}\overline{x}\texttt{)}\ \texttt{\{}e\texttt{\}}$

$\textbf{data}\ \mathit{op} = \texttt{+} \mid \texttt{-} \mid * \mid \texttt{/} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{\%}$

$\textbf{data}\ e = v$	*constant value*
$\mid x$	*variable*
$\mid \texttt{this}$	*self-reference*
$\mid C$	*class name*
$\mid \texttt{var}\ x = e\texttt;\ e$	*variable declaration*
$\mid x\ \texttt{:=}\ e$	*assignment*
$\mid e\texttt;\ e$	*sequence*
$\mid e\ \mathit{op}\ e$	*binary operator*
$\mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	*conditional*
$\mid \texttt{while}\ e\ \texttt{do}\ e$	*iteration*
$\mid e\texttt{.}m\texttt{(}\overline{e}\texttt{)}$	*method invocation*
$\mid \texttt{invoke(}e, e, \overline{e}\texttt{)}$	*reflective method invocation*
$\mid \texttt{new}\ C\texttt{(}\overline{e}\texttt{)}$	*constructor call*
$\mid CT(e, e)$	*execute at compile time*
$\mid RT(e)$	*execute at runtime*
$\mid IsCT(e)$	*tests for a compile-time value*

Figure 1. Syntax of MOOL

2. A Miniature Object-Oriented Language

A Miniature Object-Oriented Language (MOOL) is used to explain hybrid partial evaluation. MOOL is a dynamically typed imperative language based on Java [17]. It includes classes, static methods, mutable fields, local variables, and reflective method invocation. It does not include inheritance, interfaces, instanceof, static fields, or non-local control flow constructs such as return, goto or exceptions. Similar to Smalltalk [12], all fields are private and all methods are public. We believe that MOOL is sufficient to demonstrate the use of partial evaluation in real-world object-oriented languages. A more complete implementation in a real Java compiler is described in Section 4.

2.1 Syntax

Figure 1 gives the syntax for MOOL. A MOOL program is a list of class definitions. As in Scala, a class definition has a single constructor, whose arguments are listed after the class name. These constructor arguments also become fields of the object. The class contains a list of additional fields, methods, and an initialization expression. The fields of a class are initialized to an undefined value.

A method definition specifies the formal parameters and an expression which is the body of the method. The *static* modifier identifies the method as a class-level method, independent of any instance. This usage should not be confused with the traditional concept of "static" values in partial evaluation, which are called "compile-time values" in this paper. The $CT(e, e)$ and $RT(e)$ expressions mark expressions as compile time or runtime respectively. $IsCT(e)$ is a boolean expression which is used to test whether or not an expression is compile time.

Literal values are of types integer v_n, boolean v_b, string v_s or list $[v]$. Null is also a literal value. Value types also include object values, $C{:}\rho$, as described in the next section. Expressions include

```
1  class Circle(x0,y0,r0) {
2    var x;
3    var y;
4    var r;
5    init {
6      x := x0;
7      y := y0;
8      r := r0;
9    }
10   method resize(n) { r := n*r; }
11 }
12 class Main() {
13   static main() {
14     var s1 = CT(new Circle(3, 5, 10), True);
15     var s2 = new Circle(0, 1, CT(4, True));
16     s1.resize(2);
17     s2.resize(3);
18   }
19 }
```

Figure 2. An example program in MOOL syntax

operations on values and statements that affect control flow and the state: variable definitions, assignments, control constructs such as *if* and *while loop*, method calls, object creation and reflection. Figure 2 shows an example program written in MOOL syntax. It shows a `Circle` and a `Main` class which creates two `Circle` objects.

2.2 Notation

All the semantics definitions in this paper are written in Haskell [15], so they are executable. Literate Haskell [18] is used to render the definitions in more conventional style.

One non-standard aspect of the semantic definitions is the pervasive use of monads and Haskell's **do** notation to implicitly pass *state* through each definition in the interpreter. This implicit state is used for several purposes, but the most familiar one is to pass a *store* representing the mutable locations that are created as an object-oriented program is interpreted. While a complete discussion of monads is beyond the scope of this paper, we provide a quick explanation of the notation used in this paper which should be sufficient to understand the semantic definitions.

At a high level, the semantic functions have the following form:

$$
\begin{array}{l}
\mathit{command}\ x\ y = \textbf{do} \\
\quad z \leftarrow \mathit{command}\ x\ (y\ /\ 2) \\
\quad \mathit{put}\ z \\
\quad \textbf{if}\ x > y\ \textbf{then do} \\
\quad\quad a \leftarrow \mathit{command}\ (x - 1)\ y \\
\quad\quad \textbf{return}\ a \\
\quad \textbf{else} \\
\quad\quad \mathit{command}\ y\ z
\end{array}
$$

Each line is either a binding $x \leftarrow \mathit{expression}$ or an expression by itself. In either case, the expressions represent *commands* which may read or modify the implicit program state and produce a *value*, which is optionally bound to x. A command is just a function that is defined in the context of a hidden state. The final line in a **do** block must either be a command, whose value is used for the value of the block, or a **return** statement which returns a specific value.

The type of a state-based computation is specified as a monadic type $\mathit{State}\ S\ T$ where S is the type of the hidden state and T is the type of value produced. The hidden state can be, for example, a single value, a finite map of values, or a tuple of such types.

Since most semantic functions do not directly involve the state, it is useful to hide this state using a monad. When the hidden state is needed, it can be read or written using two commands, get and

```
data p = v | ⊤ | ⊥ | ˜v   -- concrete and abstract values
ρ :: x → l   -- environment maps variables to locations
σ :: l → p   -- store maps locations to values
E[[·]]· · ::e → ρ → v → State (Prog, σ, NameMap) v

E[[v]]ρo = return v

E[[e1 op e2]]ρo = do
  v1 ← E[[e1]]ρo
  v2 ← E[[e2]]ρo
  return op(v1, v2)

E[[x]]ρo = do
  (_, σ, _) ← get
  return σ(ρ(x))

E[[this]]ρo = return o

E[[var x = e1; e2]]ρo = do
  v ← E[[e1]]ρo
  [x ↦ l] ← allocate [x ↦ v]
  E[[e2]]([x ↦ l] + ρ)o

E[[x := e]]ρo = do
  v ← E[[e]]ρo
  update ρ(x) v
  return v

E[[e1; e2]]ρo = do
  E[[e1]]ρo
  E[[e2]]ρo

E[[if e1 then e2 else e3]]ρo = do
  b ← E[[e1]]ρo
  case b of
    True → E[[e2]]ρo
    False → E[[e3]]ρo

E[[while e1 do e2]]ρo = do
  E[[if e1 then (e2; while e1 do e2) else null]]ρo

E[[e.m(a)]]ρo = do
  C:ρ' ← E[[e]]ρo
  v ← mapM(E[[·]]ρo) a
  _ _(x) {eb} ← findMethod C m (length a)
  [x ↦ l] ← allocate [x ↦ v]
  E[[eb]]([x ↦ l] + ρ')(C:ρ')

E[[invoke(e, em, a)]]ρo = do
  m ← E[[em]]ρo
  E[[e.m(a)]]ρo

E[[new C(a)]]ρo = do
  class C(x) {f init{ec} _} ← findClass C
  v ← mapM(E[[·]]ρo) a
  [x ↦ l] ← allocate [x ↦ v]
  ρ' ← allocate [f ↦ ⊥]
  E[[ec]]([x ↦ l] + ρ')(C:ρ')
  return C:ρ'

update l v = do
  (P, σ, ν) ← get
  put (P, (l, v) : σ, ν)

allocate = mapM(allocate1)

allocate1 (x, v) = do
  (P, σ, ν) ← get
  let l = length σ
  put (P, (l, v) : σ, ν)
  return (x, l)
```

Figure 3. Full evaluation of MOOL expressions

put. For example, the following function ensures that the hidden state is at least n, and return the previous value of the hidden state.

```
ensure n = do
  x ← get
  if x < n then do
    put n
    return x
  else
    return x
```

The function $ensure$ has type $State\ Integer\ Integer$, meaning that it has a hidden integer state variable, and also returns an integer.

There are many papers and tutorials on monads which explain the details on the semantics and implementation of monads [28]. For the purposes of this paper, it is only necessary to understand that the store is passed through each line of a **do** block.

2.3 Semantics

The semantics of MOOL is shown in Figure 3. In the code, l refers to a location, x refers to a name, v refers to a value, and e and a are expressions. The Haskell source code for HPE can be found at the following URL:
`http://www.cs.utexas.edu/~amshali/Civet/`

An environment, ρ, maps variable or field names to locations. A store, σ, maps locations to potentially abstract values, p. Abstract values, $\tilde{v}$, are described in the next section. They are included here so that the full evaluator can have the same type signature as the partial evaluator. A $\perp$ value for a variable means that the variable has not been assigned yet. An object value, $C{:}\rho$, is a pair where C is the name of the class that the object is instantiated from. ρ is the environment for this object, which contains the locations of its fields.

The function $\mathcal{E}[\![\cdot]\!]\cdot\cdot$ is referred to as the "full evaluator" to distinguish it from the "partial evaluator" defined in Section 3. This function, $\mathcal{E}[\![e]\!]\rho o$, executes the program represented by an expression e in the context of an environment ρ and current object o. The full evaluator returns a value and potentially modifies the implicit state [28]. The implicit state has three components: the program, a store and a $NameMap$. The full evaluator only manipulates the store. The other components are included for consistency with the partial evaluator, which extends the program during evaluation.

The first two cases specify the behavior of value literals v and binary operators. The full evaluator applies the binary operation op to its operands and returns the result, taking into account the type of values that it receives with respect to the operation. The definition of op is omitted.

The next three cases concern variables, declarations, and assignment. All variables are bound to locations in the environment, and the locations are then looked up in the store. As mentioned in the previous section, get is a command which retrieves the program, the store and the NameMap in a tuple. All variables are assumed to be present in the environment, and their location defined in the store, otherwise an error is thrown.

A variable declaration, `var` $x = e_1$`;` e_2, evaluates e_1 to get a value, then stores the value into a new location, and then evaluates e_2 in an extended environment. The $allocate$ function takes a list of *name-value* pairs $[\overline{x} \mapsto \overline{v}]$ and returns a list of *name-location* pairs $[\overline{x} \mapsto \overline{l}]$. It updates the store so that each location contains the corresponding value. Assignment x `:=` e evaluates e and then updates the variable's location to the new value. The $update$ function gets the store, and then adds a new (l, v) pair to the store to associate location l with value v.

Evaluation of `if` and `while` expressions is standard.

The evaluation of a method call, $e.m(\overline{a})$, starts with evaluating the target expression, e, and all the arguments, $\overline{a}$. The evaluator then finds the method, m, based on the class of the target object. It then evaluates the body of the method in an environment which has the bindings actual parameters and the target object's fields, ρ'. The object context o is set to the target object $C{:}\rho'$.

The `invoke` expression supports reflective method invocation, where the method name is computed as a value rather than being explicit in the syntax of the call. The expression e is the target of the reflective call. e_m is an expression which evaluates to the name of the method and $\overline{a}$ is the list of actual parameters. To evaluate a reflective method invocation, the semantics first evaluates the method name expression, then performs a normal method call using the computed name.

The full evaluator evaluates the object creation expression, `new` $C(\overline{a})$, by finding the class C. It then evaluates all the actual arguments of the class constructor and binds them to their names in the environment. Then, it binds all the fields of the class to the undefined value, $\perp$, and evaluates the body of the constructor and returns an object $C{:}\rho'$. An object's fields are initialized when the full evaluator evaluates the body of the constructor(`init`).

3. Hybrid Partial Evaluator for MOOL

In this section we define a hybrid partial evaluator for MOOL. With partial evaluation, program execution is split into two stages. The first stage, where partial evaluation is performed, is *compile time*. The output of the compile-time stage is a modified program, called *residual code*, which is executed in the *runtime* stage. Values that exist during the first phase are called *compile-time* values, while all other values are called *runtime* values.

The key question for partial evaluation is how to identify what parts of a program should be evaluated at compile time. Hybrid partial evaluation is based on a few fundamental principles:

- A programmer identifies parts of the program to execute at compile time, creating compile-time values. Any subsequent operations that involve compile-time objects are executed at compile time, possibly creating more compile-time objects. Every object exists either at compile time or runtime and cannot move between phases. On the other hand, primitive values (integers, strings, dates) are automatically moved between phases as needed.
- All variables are assigned values at compile time, but the value may be a concrete (compile-time) value or an abstract value representing partial information about the future actual value of the variable at runtime. A variable's status, as either compile-time or runtime, never changes. Compile-time variables are eliminated from the program.
- Methods and constructors are specialized on every combination of specific compile-time arguments that arise during partial evaluation. When a constructor is specialized, its class is split into compile-time and runtime facets, in effect creating two partial objects that exist in different phases.

In the partial evaluator, concrete compile-time values are simply the normal values v, which can be primitive values or objects that exist only at compile time. There are two kinds of abstract values: unknown values $\top$ and abstract values, $\tilde{v}$. A completely unknown value is represented by $\top$. An abstract value $\tilde{v}$ can be either a primitive constant that has been marked to exist at runtime, or a *partial object* $\tilde{C}{:}\rho$. A partial object can specify just the class of a runtime object, or it can specify the class and some of its fields.

The key point is that compile-time values force specialization when used as arguments to methods or constructors, while abstract objects allow local propagation of compile-time information but do not trigger specialization. For instance, in the program shown in Figure 2, `s1` is a compile-time circle object whereas `s2` is an abstract runtime circle object, because only the value of its radius is marked to be known at compile time.

The type of hybrid partial evaluator, $\mathcal{P}[\![\cdot]\!]\cdot\cdot$, is shown in Figure 4. A hybrid partial evaluator, like an online one, works very much like a full evaluator. However, during partial evaluation, the store may contain abstract (or approximate) values ones represented by $\tilde{v}$. Operations on these abstract values are residualized to create code that executes at runtime, when the actual values are known.

The result of hybrid partial evaluation is an expression accompanied with a value, p, which may be a compile-time value or an abstraction of a runtime value (or $\top$). The expression represents the residual code. The value is the information about the partially evaluated expression. This information can be as concrete as a constant or as abstract as a $\top$ value. Online partial evaluators have traditionally been defined to return an residual expression *or* a compile-time value. Allowing both an expression and a value allows residual code to be generated while also returning partial information about the value computed by the residual code. Partial evaluation of basic expressions is given in Figure 4.

Partial evaluation of primitive value constants always produces abstract values. This may seem strange, because constants are fully known at compile time. However, if all constants were considered compile-time values, they would cause specialization whenever they were used, which would violate the principle that the programmer should indicate where specialization is to occur.

Binary operators, $e_1\ op\ e_2$, return a compile-time value if either e_1 or e_2 partially evaluate to a compile-time value, otherwise return an abstract value. This rule follows the principle that operations involving compile-time values produce compile-time values.

3.1 Variable Declaration and Assignment

Figure 4 also defines the hybrid partial evaluation of variables, variable declarations and variable assignments. A variable is compile-time if it is assigned a compile-time value and it is runtime if it is a $\top$ or an approximate value, $\tilde{v}$. HPE binds all the variables in the environment whether or not they are compile-time.

For variables, partial evaluator returns their value as the residual expression if they are compile-time. This is because compile-time variables are eliminated from the residual code. Otherwise, it re-

```
data PV = ⟨e, p⟩
𝒫⟦·⟧· · ::e → ρ → v → State (Prog, σ, NameMap) PV

𝒫⟦v⟧ρo = return ⟨⟦v⟧, ˜v⟩

𝒫⟦e1 op e2⟧ρo = do
  ⟨e1', p1⟩ ← 𝒫⟦e1⟧ρo
  ⟨e2', p2⟩ ← 𝒫⟦e2⟧ρo
  case (p1, p2) of
    (v1, v2) → let v = op(v1, v2)  in return ⟨⟦v⟧, v⟩
    (v1, ˜v2) → let v = op(v1, v2) in return ⟨⟦v⟧, v⟩
    (˜v1, v2) → let v = op(v1, v2) in return ⟨⟦v⟧, v⟩
    (˜v1, ˜v2) → let v = op(v1, v2) in return ⟨⟦v⟧, ˜v⟩
    else → return ⟨⟦e1' op e2'⟧, ⊤⟩

𝒫⟦this⟧ρo = return ⟨⟦this⟧, o⟩

𝒫⟦CT(e, e')⟧ρo = do
  v' ← ℰ⟦e'⟧ρo    -- Error if e' is not compile-time
  if v' ≡ True then do
    v ← ℰ⟦e⟧ρo    -- Error if e is not compile-time
    return ⟨⟦v⟧, v⟩
  else 𝒫⟦e⟧ρo

𝒫⟦IsCT(e)⟧ρo = do
  ⟨e', p⟩ ← 𝒫⟦e⟧ρo
  case p of
    v → return ⟨⟦True⟧, True⟩
    else → return ⟨⟦False⟧, False⟩

𝒫⟦RT(e)⟧ρo = do
  ⟨e', p⟩ ← 𝒫⟦e⟧ρo
  return ⟨⟦e'⟧, ⊤⟩

𝒫⟦x⟧ρo = do
  (_, σ, _) ← get
  case σ(ρ(x)) of
    v → return ⟨⟦v⟧, v⟩
    p → return ⟨⟦x⟧, p⟩

𝒫⟦var x = e1; e2⟧ρo = do
  ⟨e1', p1⟩ ← 𝒫⟦e1⟧ρo
  [x ↦ l] ← allocate [x ↦ p1]
  ⟨e2', p2⟩ ← 𝒫⟦e2⟧([x ↦ l] + ρ)o
  case p1 of
    v → return ⟨e2', p2⟩
    else → return ⟨⟦var x = e1'; e2'⟧, p2⟩

𝒫⟦x := e⟧ρo = do
  (_, σ, _) ← get
  case σ(ρ(x)) of
    v → do          -- compile-time variables not residualized
      v' ← ℰ⟦e⟧ρo   -- Error if e is not compile-time
      update ρ(x) v'
      return ⟨⟦v'⟧, v'⟩
    ˜v → do         -- abstract variable must stay abstract
      ⟨e', p⟩ ← 𝒫⟦e⟧ρo
      update ρ(x) ˜p
      return ⟨⟦x := e'⟧, p⟩
    ⊤ → do          -- unknown runtime value
      ⟨e', p⟩ ← 𝒫⟦e⟧ρo
      return ⟨⟦x := e'⟧, ⊤⟩
    ⊥ → do          -- variable is not yet defined
      ⟨e', p⟩ ← 𝒫⟦e⟧ρo
      update ρ(x) p
      case p of     -- first assignment determines status of variable
        v → return ⟨⟦v⟧, v⟩
        _ → return ⟨⟦x := e'⟧, ⊤⟩
```

Figure 4. Partial evaluation of basic values, variables, operators, variable declarations, and assignments

turns a residual code which contains the name of the variable along with the abstract value stored for that variable.

A variable declaration `var` $x = e$`;` $\cdot$ may introduce a compile-time or runtime variable. If the partial evaluated value of e is a compile-time value, then the variable is defined only at compile time, and has no existence at runtime. Otherwise, the variable is a normal runtime variable defined in the generated residual code.

Partial evaluation of a variable assignment, x `:=` e, depends on whether the x is a compile-time or runtime variable. For a compile-time variable x, the expression e must evaluate to a value and the value of x is updated in the store. For runtime variables, residual code is returned for the assignment. A value $\bot$ in the store for a variable means that the variable is a field and has not been assigned yet. Thus, it can accept any value and partial evaluator updates its value in the store accordingly. When a variable has the value $\top$, it means that we have no compile-time information about the variable. Such variables cannot be updated with any other values except $\top$.

3.2 Special Expressions

The special expression $CT(e, e')$ indicates which values should be created at compile time. If e' is $True$, then e is a evaluated at partial evaluation time using the full evaluator, to create a compile-time value. The result may be a primitive data type, or an object. The special expression, $IsCT(e)$, evaluates to $True$ when e is a compile-time value. $RT(e)$ expression marks an expression as runtime. The partial evaluator does not do any evaluation on the expression, e, and simply returns the same expression as the residual code along with a $\top$ value.

3.3 Control Flow

Figure 5 defines hybrid partial evaluation of control flow statements. Sequences are straightforward.

For an `if`-expression, if the condition is a compile-time value, then the partial evaluator selects the appropriate branch for further evaluation, just like the full evaluator. When the condition is a runtime value, it is desirable to partially evaluate both branches of the conditional. The problem is that branches may make incompatible changes to the store, so that it is not clear which modified store should be used for the evaluation of the remainder of the program.

This problem is illustrated in Figure 6 [20]. In this example, a is a runtime variable. Thus, the partial evaluation of the `if`-condition, $a < x$, results in the expression $a < 3$, which is not a value. During runtime, only one branch must take place, in which case, the value of x after the evaluation of `if`-expression would be 9 and the value of y can be either 4 or 6 based on the branch taken.

$\mathcal{P}[\![e_1\texttt{;}\ e_2]\!]\rho o = \mathbf{do}$
  $\langle e_1', p_1\rangle \leftarrow \mathcal{P}[\![e_1]\!]\rho o$
  $\langle e_2', p_2\rangle \leftarrow \mathcal{P}[\![e_2]\!]\rho o$
  return $\langle [\![e_1'\texttt{;}\ e_2']\!], p_2\rangle$

$\mathcal{P}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\rho o = \mathbf{do}$
  $\langle e_1', p_1\rangle \leftarrow \mathcal{P}[\![e_1]\!]\rho o$
  case p_1 **of**
    $True \rightarrow \mathcal{P}[\![e_2]\!]\rho o$
    $False \rightarrow \mathcal{P}[\![e_3]\!]\rho o$
    else $\rightarrow$ $checkStore\ \rho\ o\ (\texttt{if}\ e_1'\ \texttt{then}\ \cdot\ \texttt{else}\ \cdot)\ e_2\ e_3$

$\mathcal{P}[\![\texttt{while}\ e_1\ \texttt{do}\ e_2]\!]\rho o = \mathbf{do}$
  $\langle e_1', p_1\rangle \leftarrow \mathcal{P}[\![e_1]\!]\rho o$
  case p_1 **of**
    $True \rightarrow \mathcal{P}[\![e_2\texttt{;}\ \texttt{while}\ e_1\ \texttt{do}\ e_2]\!]\rho o$
    $False \rightarrow \mathcal{P}[\![\texttt{null}]\!]\rho o$
    else $\rightarrow$ **do**
      $sanitize\ \rho$
      $checkStore\ \rho\ o\ (\texttt{while}\ \cdot\ \texttt{do}\ \cdot)\ e_1\ e_2$

$checkStore\ \rho\ o\ f\ e_2\ e_3 = \mathbf{do}$
  $(P, store, \nu) \leftarrow get$ -- capture the initial store
  $\langle e_2', p_2\rangle \leftarrow \mathcal{P}[\![e_2]\!]\rho o$ -- evaluate the then branch
  $(\_, \sigma_1, \_) \leftarrow get$ -- snapshot the resulting store
  $put\ (P, store, \nu)$ -- reset store to initial conditions
  $\langle e_3', p_3\rangle \leftarrow \mathcal{P}[\![e_3]\!]\rho o$ -- run the else branch
  $(\_, \sigma_2, \_) \leftarrow get$ -- snapshot the else store
  $cmp \leftarrow \sigma_1 =_\rho \sigma_2$ -- check that changes are consistent
  $sanitize\ \rho$ -- erase abstract values
  if cmp **then** -- success
    return $\langle [\![f\ e_2'\ e_3']\!], \bot\rangle$
  else -- report inconsistency
    $inconsistentChangeError\ \rho\ \sigma_1\ \sigma_2$

Figure 5. Partial evaluation of control flow constructs

```
1  method iftest(a) {
2    var x = CT(3, True);
3    var y = CT(4, True);
4    if (a < x) {
5      y := 2 * x;
6      x := 3 + y;
7    }
8    else
9      x := 5 + y;
10 }
```

Figure 6. The problematic example of an `if`-expression for the partial evaluation

A polyvariant computation scheme [10] deals with this problem by partially evaluating both branches and inserting necessary assignments called *explicator* at the end of new residual branches. Meyer [20] proposed a solution that joins the environments resulted from the two branches. In a semantics based on continuation, the rest of the program is specialized separately for each branch [23, 27]. However, this has the potential to duplicate large amounts of code.

HPE has a pragmatic approach to this problem. The $checkStore$ function (See Figure 5) evaluates both branches and then looks for inconsistencies in the state. It also *sanitizes* the store by converting all partially abstract values to $\top$. If the resulting stores (σ_1, σ_2) are different with respect to the initial environment (ρ), HPE raises an error. Otherwise, it continues with the generation of the code for the `if` and partial evaluation of the rest of the program. The same approach is used for `while` expressions, except that the store is also sanitized at the top of the loop. We have found that this pragmatic approach is sufficient for many common programming idioms, as shown in Section 4.

For the example in Figure 6, the hybrid partial evaluator starts with the environment $\{x = 3, y = 4\}$. The first branch changes the environment to $\{x = 9, y = 6\}$. The partial evaluation of the second branch results in $\{x = 9, y = 4\}$. The two branches make inconsistent changes to the environment and therefore HPE raises an error.

3.4 Class Specialization and Partial Objects

For an object creation expression, `new` $C(\overline{a})$, HPE specializes the class C if any of the parameters to the constructor call are compile-time. Class specialization is defined in Figure 7. For class specialization, the partial evaluator binds the actual parameters of the constructor in the environment. It then finds if this class with such actual parameters has been already specialized. The $findMemoClass$ returns the name of the specialized class, if there is one already, along with its class definition. Otherwise, it generates a new name and returns it with the original class definition.

When the class has not been specialized, HPE specializes the body of the constructor in an environment containing the binding for the parameters, `this` and fields. Fields are initialized to $\bot$. All the methods of the class are likewise specialized. The new class and methods are added to the program. The resulting residual code is an expression that instantiates the new class with any remaining runtime parameters. Along with the residual code, HPE returns an abstract object which has the name of the new class and the partial environment of the object.

When the class C with those actual parameters has been already specialized, the hybrid partial evaluator evaluates the body of the constructor after allocating the fields and the `this` object in the store and returns an approximate object with the required residual code.

3.5 Method Specialization

HPE can specialize method calls $o\,.\,m\,(\overline{a})$ on compile-time objects, which were introduced in Section 3.1. Since a compile-time object is never residualized, its identity and field values exist only during partial evaluation. In this case, hybrid specialization may result in *full evaluation* of the call, or create a *residual class method*.

The cases for method calls on compile-time objects are defined in Figure 8. If all the arguments to the method call are compile-time values, then the call is processed as a normal method call. If some but not all of the method arguments are compile-time values, then it must be specialized to create a new method in the residual program. Since the target object does not exist in the residual program, the new method must be $static$. The function $\mathcal{S}[\![e]\!]^{modifier}\rho o\ C\ m\ \overline{a}$ (See Figure 9) creates a specialized version of a method. In this case

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{new } C\texttt{(}\overline{a}\texttt{)}]\!]\rho o = \textbf{do}\\
\quad \langle \overline{a'}, \overline{p'}\rangle \leftarrow \text{mapM}(\mathcal{P}[\![\cdot]\!]\rho o)\ \overline{a}\\
\quad \textbf{if } \textit{any isCompileTime } \overline{p'} \textbf{ then do}\\
\quad\quad \textit{memc} \leftarrow \textit{findMemoClass } C\ \overline{p'}\\
\quad\quad \textbf{let } (z, C', \texttt{class \_(}\overline{x}\texttt{) \{}\overline{f}\ \texttt{init\{}e_c\texttt{\}}\ \overline{m}\texttt{\}}) = \textit{memc}\\
\quad\quad [\overline{x} \mapsto \overline{l}] \leftarrow \textit{allocate } [\overline{x} \mapsto \overline{p'}]\\
\quad\quad \textbf{let } \overline{x_d} = \textit{getRuntimeNames } \overline{x}\ \langle \overline{a'}, \overline{p'}\rangle\\
\quad\quad \textbf{let } \overline{a_d} = \textit{getRuntimeExprs } \langle \overline{a'}, \overline{p'}\rangle\\
\quad\quad \rho' \leftarrow \textit{allocate } [\overline{f} \mapsto \overline{\bot}]\\
\quad\quad \langle e'_c, \_\rangle \leftarrow \mathcal{P}[\![e_c]\!]([\overline{x} \mapsto \overline{l}] + \rho')(\overline{C'}{:}\rho')\\
\quad\quad \textit{when } (\neg\ z)\ (\textbf{do}\\
\quad\quad\quad \overline{m'} \leftarrow \text{mapM}(\mathcal{M}[\![\cdot]\!]\rho'(\overline{C'}{:}\rho'))\ \overline{m}\\
\quad\quad\quad \overline{f'} \leftarrow \textit{getRuntimeFields } \overline{f}\ \rho'\\
\quad\quad\quad \textit{addClass } \texttt{class } C'\texttt{(}\overline{x_d}\texttt{) \{}\overline{f'}\ \texttt{init\{}e'_c\texttt{\}}\ \overline{m'}\texttt{\}})\\
\quad\quad \textbf{return } \langle [\![\texttt{new } C'\texttt{(}\overline{a_d}\texttt{)}]\!], \overline{C'}{:}\rho'\rangle\\
\quad \textbf{else do}\\
\quad\quad \texttt{class \_(\_) \{}\overline{f}\ \texttt{init\{\_\} \_\}} \leftarrow \textit{findClass } C\\
\quad\quad \rho' \leftarrow \textit{allocate } [\overline{f} \mapsto \overline{\top}]\\
\quad\quad \textbf{return } \langle [\![\texttt{new } C\texttt{(}\overline{a'}\texttt{)}]\!], \overline{C}{:}\rho'\rangle\\
\\
\textit{findMemoClass } C\ [\overline{x_s} \mapsto \overline{v_s}] = \textbf{do}\\
\quad (p, \sigma, n) \leftarrow \textit{get} \quad \text{-- n is NameMap}\\
\quad \textbf{case } n((\ C\ [\overline{x_s} \mapsto \overline{v_s}])) \textbf{ of}\\
\quad\quad l, [\,C\,] \rightarrow \textbf{do}\\
\quad\quad\quad \textit{cdef} \leftarrow \textit{findClass } (C + \texttt{"\$"} + l)\\
\quad\quad\quad \textbf{return } \textit{True}, C + \texttt{"\$"} + l, \textit{cdef}\\
\quad\quad \textit{Nothing} \rightarrow \textbf{do}\\
\quad\quad\quad \textbf{let } l = (\textit{length } n) + 1\\
\quad\quad\quad \textit{put } (p, \sigma, (\ C\ [\overline{x_s} \mapsto \overline{v_s}], (l, [\,C\,])) : n)\\
\quad\quad\quad \textit{cdef} \leftarrow \textit{findClass } C\\
\quad\quad\quad \textbf{return } \textit{False}, C + \texttt{"\$"} + l, \textit{cdef}\\
\\
\mathcal{M}[\![\textit{modifier } m\texttt{(}\overline{x}\texttt{) \{}e\texttt{\}}]\!]\rho o = \textbf{do}\\
\quad \rho' \leftarrow \textit{allocate } [(x, \top) \mid x \leftarrow \overline{x}]\\
\quad \langle e', \_\rangle \leftarrow \mathcal{P}[\![e]\!](\rho + \rho')o\\
\quad \textbf{return } \textit{modifier } m\texttt{(}\overline{x}\texttt{) \{}e'\texttt{\}}
\end{array}
$$

Figure 7. Partial evaluation of constructors for partial objects

$$
\begin{array}{l}
\mathcal{P}[\![e\,.\,m\texttt{(}\overline{a}\texttt{)}]\!]\rho o = \textbf{do}\\
\quad \langle e', p\rangle \leftarrow \mathcal{P}[\![e]\!]\rho o\\
\quad \langle \overline{a'}, \overline{p'}\rangle \leftarrow \text{mapM}(\mathcal{P}[\![\cdot]\!]\rho o)\ \overline{a}\\
\quad \textbf{case } p \textbf{ of}\\
\quad\quad C{:}\rho' \rightarrow \textbf{do}\\
\quad\quad\quad \textbf{if } \textit{all isCompileTime } \overline{p'} \textbf{ then do}\\
\quad\quad\quad\quad \_\_\texttt{(}\overline{x}\texttt{) \{}e_b\texttt{\}} \leftarrow \textit{findMethod } C\ m\ (\textit{length } \overline{a})\\
\quad\quad\quad\quad [\overline{x} \mapsto \overline{l}] \leftarrow \textit{allocate } [\overline{x} \mapsto \overline{p'}]\\
\quad\quad\quad\quad v \leftarrow \mathcal{E}[\![e_b]\!]([\overline{x} \mapsto \overline{l}] + \rho')p\\
\quad\quad\quad\quad \textbf{return } \langle [\![v]\!], v\rangle\\
\quad\quad\quad \textbf{else}\\
\quad\quad\quad\quad \mathcal{S}[\![\,C]\!]^{\texttt{static}}\rho' \top\ C\ m\ \langle \overline{a'}, \overline{p'}\rangle\\
\quad\quad \tilde{C}{:}\rho' \rightarrow \quad \text{-- target is an approximate object}\\
\quad\quad\quad \textbf{if } \textit{any isCompileTime } \overline{p'} \textbf{ then do}\\
\quad\quad\quad\quad \mathcal{S}[\![e']\!]^{\texttt{method}}\rho' p\ C\ m\ \langle \overline{a'}, \overline{p'}\rangle\\
\quad\quad\quad \textbf{else}\\
\quad\quad\quad\quad \textbf{return } \langle [\![e'\,.\,m\texttt{(}\overline{a'}\texttt{)}]\!], \top\rangle\\
\quad\quad \textbf{else} \rightarrow \quad \text{-- target is unknown}\\
\quad\quad\quad \textbf{if } \textit{any isCompileTime } \overline{p'} \textbf{ then do}\\
\quad\quad\quad\quad m' \leftarrow \textit{specializeAll } \rho\ o\ m\ \langle \overline{a'}, \overline{p'}\rangle\\
\quad\quad\quad\quad \textbf{let } \overline{a_d} = \textit{getRuntimeExprs } \langle \overline{a'}, \overline{p'}\rangle\\
\quad\quad\quad\quad \textbf{return } \langle [\![e'\,.\,m'\texttt{(}\overline{a_d}\texttt{)}]\!], \top\rangle\\
\quad\quad\quad \textbf{else}\\
\quad\quad\quad\quad \textbf{return } \langle [\![e'\,.\,m\texttt{(}\overline{a'}\texttt{)}]\!], \top\rangle\\
\mathcal{P}[\![\texttt{invoke(}e, e_m, \overline{a}\texttt{)}]\!]\rho o = \textbf{do}\\
\quad \langle e'_m, p\rangle \leftarrow \mathcal{P}[\![e_m]\!]\rho o\\
\quad \textbf{case } e'_m \textbf{ of}\\
\quad\quad m \rightarrow \mathcal{P}[\![e\,.\,m\texttt{(}\overline{a}\texttt{)}]\!]\rho o\\
\quad\quad \textbf{else} \rightarrow \textbf{do}\\
\quad\quad\quad \langle e', p'\rangle \leftarrow \mathcal{P}[\![e]\!]\rho o\\
\quad\quad\quad \langle \overline{a'}, \overline{p'}\rangle \leftarrow \text{mapM}(\mathcal{P}[\![\cdot]\!]\rho o)\ \overline{a}\\
\quad\quad\quad \textbf{return } \langle [\![\texttt{invoke(}e', e'_m, \overline{a'}\texttt{)}]\!], \top\rangle
\end{array}
$$

Figure 8. Partial evaluation of method calls and reflective method calls for partial objects

the new method is marked as *static*. New methods are stored in a cache, so that the same specialization of a method is not generated twice. The method specializer $\mathcal{S}[\![e]\!]^{modifier}\rho o\ C\ m\ \overline{a}$ binds all the parameters in the environment and partially evaluates the method body. It then adds the method to the corresponding class and returns the residual method call expression with runtime arguments.

Program point specialization is a technique that is used to prevent the specializer from running into the infinite loop of specializing a recursive function [1, 5, 14]. The hybrid partial evaluator uses the *polyvariant specialization* [4, 8, 24] strategy for program point specialization. It memoizes a call expression, $e\,.\,m\texttt{(}\overline{a}\texttt{)}$, so that it can be reused from other call sites. It also memoizes object creation expressions (constructor calls). Memoization is implemented in the *findMemoCall* and *findMemoClass*. The partial evaluator saves the name of either method or class along with the actual parameters passed to that and the content of the store at the time of specialization. These information are stored in the *NameMap* part of the state monad.

Now consider the method call $o\,.\,m\texttt{(}\overline{a}\texttt{)}$ in which o is a partial object. The hybrid partial evaluator knows the class of a partial object. When some of the actual parameters in the method call expression are compile-time values or objects, HPE specializes using the function $\mathcal{S}$ and creates a residual *instance* method. This is shown in Figure 8. When all of the parameters are runtime values, the partial evaluator generates a residual code for the method call.

If the target of the call is not known and the partial evaluator has no information about it and some of the actual parameters are compile-time values, HPE specializes the method call. Since the class of the target is not known, all the methods in all the classes with the same name and the same number of the parameters are specialized. The *specializeAll* function finds all the methods with the same name and the same number of parameters in all the classes. It then partially evaluates each method with a copy of the store. Thereafter, it checks all the stores resulting from the partial evaluation of each method to make sure that partial evaluation of methods has not caused any inconsistency in the state.

When the target is unknown and none of the parameters are compile-time or abstract values, HPE only generates a residual code.

3.5.1 Reflective Calls

Figure 8 also defines the partial evaluation of reflective calls. When the partial evaluation of e_m results in a string value, m, the name

$$
\begin{array}{l}
\mathcal{S}[\![e]\!]^{modifier}\rho o\ C\ m\ \langle \overline{a'}, \overline{p'}\rangle = \textbf{do} \\
\quad (z, m', \_\_(\overline{x})\ \{e_b\}) \leftarrow \mathit{findMemoCall}\ C\ m\ \overline{p'} \\
\quad \textbf{let}\ \overline{a_d} = \mathit{getRuntimeExprs}\ \langle \overline{a'}, \overline{p'}\rangle \\
\quad \mathit{when}\ (\neg\ z)\ (\textbf{do} \\
\qquad [\overline{x} \mapsto \overline{l}] \leftarrow \mathit{allocate}\ [\overline{x} \mapsto \overline{p'}] \\
\qquad \langle e'_b, p\rangle \leftarrow \mathcal{P}[\![e_b]\!]([\overline{x} \mapsto \overline{l}] + \rho)o \\
\qquad \textbf{let}\ \overline{x_d} = \mathit{getRuntimeNames}\ \overline{x}\ \langle \overline{a'}, \overline{p'}\rangle \\
\qquad \mathit{addMethod}\ C\ \mathit{modifier}\ m'(\overline{x_d})\ \{e'_b\}) \\
\quad \textbf{return}\ \langle [\![e\,.\,m'(\overline{a_d})]\!], \top\rangle \\
\mathit{findMemoCall}\ C\ m\ \overline{a} = \textbf{do} \\
\quad \mathit{mdef} \leftarrow \mathit{findMethod}\ C\ m\ (\mathit{length}\ \overline{a}) \\
\quad (p, \sigma, n) \leftarrow \mathit{get} \quad \text{-- n is the NameMap} \\
\quad \textbf{case}\ n((\ m\ (\mathit{length}\ \overline{a})\ \overline{a})\ \textbf{of} \\
\qquad l, cs \rightarrow \textbf{do} \\
\qquad\quad \textbf{if}\ (\mathit{elem}\ C\ cs)\ \textbf{then} \\
\qquad\qquad \textbf{return}\ \mathit{True}, m + \texttt{"\$"} + l, \mathit{mdef} \\
\qquad\quad \textbf{else do} \\
\qquad\qquad \mathit{put}\ (p, \sigma, (\ m\ (\mathit{length}\ \overline{a})\ \overline{a}, (l, C : cs)) : n) \\
\qquad\qquad \textbf{return}\ \mathit{False}, m + \texttt{"\$"} + l, \mathit{mdef} \\
\qquad \mathit{Nothing} \rightarrow \textbf{do} \\
\qquad\quad \textbf{let}\ l = (\mathit{length}\ n) + 1 \\
\qquad\quad \mathit{put}\ (p, \sigma, (\ m\ (\mathit{length}\ \overline{a})\ \overline{a}, (l, [C])) : n) \\
\qquad\quad \textbf{return}\ \mathit{False}, m + \texttt{"\$"} + l, \mathit{mdef}
\end{array}
$$

Figure 9. Helper function for partial evaluation of method calls

of the method to be called is known at compile time. Therefore partial evaluator can specialize the method using the specialization process of a normal method call. Otherwise, when the name of the reflective method call is not known, it partially evaluates the target expression and the arguments and generates an expression for the `invoke`.

As an example, consider the following example of reflective method invocation:

```
Method m = obj.class.getMethod(name, Integer.TYPE);
m.invoke(obj, arglist);
```

if `name` is known at compile time to be `"test"` then the code above is optimized to [2]:

```
obj.test(arglist);
```

3.6 Examples

In this section we show some examples in MOOL programs and and their generated residual code. The first example is an integer exponentiation. This function works by squaring based on the fact that when n is even $x^n = x^{n/2} \times x^{n/2}$ and when n is odd $x^n = x \times x^{n-1}$. Figure 10 shows the MOOL program which implements this function. Figure 11 shows the residual code for the `power` function when the exponent has a compile-time value of 11. The `power$1` is the residual function which takes only one parameter, the base of exponentiation, and returns the 11th power of that.

The next example is a regular expression matcher program. This program is based on the idea of using derivatives of a regular expression pattern [3]. This way of constructing a regular expression matcher does not require using NFA, DFA or back-tracking. Figure 12 shows the code for matching algorithm. In this code the regular expression is a compile-time value. The input however is dynamic. The partial evaluator specializes the matching algorithm and generates a new code which has no trace of the classes and

```
1  class Main() {
2    static main(a) {
3      this.power(CT(11),a);
4    }
5    method power(n,x) {
6      var i = n;
7      var y = 1;
8      var p = x;
9      while (i > 0) {
10       if ((i % 2) = 1)
11         y := y * p;
12       i := i / 2;
13       if (i > 0)
14         p := p * p;
15     }
16     y;
17   }
18 }
```

Figure 10. Exponentiation Function

```
1  class Main() {
2    static main(a) {
3      this.power$1(a);
4    }
5    method power$1(x) {
6      var y = 1;
7      var p = x;
8      y := y * p;
9      p := p * p;
10     y := y * p;
11     p := p * p;
12     p := p * p;
13     y := y * p;
14     y;
15   }
16 }
```

Figure 11. Exponentiation Function Residual Code

function calls on the input regular expression. The residual code is shown in Figure 13.

```
1  public abstract class Regex {
2    public abstract Regex derivative(Character c);
3    public abstract boolean canBeEmpty();
4    public Set<Character> first();
5    public static boolean match(Regex e, String input) {
6      if(input.length() == 0) return e.canBeEmpty();
7      Character c = input.charAt(0);
8      for (Character ce : e.first())
9        if(c.equals(ce))
10         return match(e.derivative(ce),
11                      input.substring(1));
12     return false;
13   }
14   public static void main(String[] args) {
15     @StaticStage
16     Regex re = RegexParser.parse("(a|b)*(abb|a+b)");
17     String in = "abababababb";
18     boolean matched = match(re, in);
19     System.out.println(matched);
20   }
21 }
```

Figure 12. Regular expression matcher

```
public abstract class Regex {
  public abstract Regex derivative(Character c);
  public abstract boolean canBeEmpty();
  public Set<Character> first();
  public static void main(String[] args) {
    String in = "abababababbb";
    boolean matched = match$1000001(in);
    java.lang.System.out.println(matched);
  }
  public static boolean match$1000001(String input) {
    if(input.length() == 0) return false;
    Character c = input.charAt(0);
    if(c.equals('b'))
      return match$1000002(input.substring(1));
    if(c.equals('a'))
      return match$1000012(input.substring(1));
    return false;
  }
// ...
  public static boolean match$1000013(String input) {
    if(input.length() == 0) return true;
    Character c = input.charAt(0);
    if(c.equals('b'))
      return match$1000005(input.substring(1));
    if(c.equals('a'))
      return match$1000006(input.substring(1));
    return false;
  }
}
```

Figure 13. Regular expression matcher residual code

3.7 Discussion

Hybrid partial evaluation does not guarantee that all programs which execute correctly by themselves can be partially evaluated to produce residual code. In other words, hybrid partial evaluation can fail even if the program being analyzed is an otherwise valid program. Unfortunately, the error cases are not completely explicit in the semantic evaluation functions. One important error, which can occur anywhere, is an attempt to create residual code that contains an instantiated compile-time object. For example, the following code instantiates a hash table at compile time, but then attempts to use the hash table at runtime to lookup a runtime input string.

```
var o = CT(new HashTable());
o.put("one", 1); ...; o.put("nine", 9);
var r = o.get(readLine());
if (r != null) System.out.println(r);
```

The hybrid partial evaluator raises an error in this case, because the residual code $[\![(\mathrm{HashTable}{:}\rho)\texttt{.get(readLine())}]\!]$ is invalid, as code cannot contain an instantiated compile-time object. The partial evaluator would try to specialize the `put` method, but it cannot specialize system methods. HPE issues a compiler-error when processing the above code.

It is possible to rewrite this example to avoid the problem, by taking more advantage of compile-time information and changing when operations take place. This kind of change is known as *binding time improvement*. In this case, the trick is to iterate over the compile-time hashtable:

```
var o = CT(new HashTable());
o.put("one", "1"); ...; o.put("nine", "9");
var input = readLine();
for test : o.keys()
  if input.equals(test) then
     System.out.println( o.get(test) );
```

In this version of the program, both `o` and `test` variables are compile-time values, which are not included in the residual code. The residual code has unrolled and specialized the loop. The call to `get` only involves compile-time values, so it is specialized away:

```
var input = readLine();
if input.equals("one") then System.out.println( "1" );
if input.equals("two") then System.out.println( "2" );
...
if input.equals("nine") then System.out.println( "9" );
```

Conversely, the partial evaluator may also through errors if an expression is marked as `CT` but involves runtime data. These places are noted in Figure 4.

4. Civet: A Hybrid Partial Evaluator for Java

We have implemented a hybrid partial evaluator for the Java language, based on the semantics we explained in the previous section. This hybrid partial evaluator is called *Civet*[1]. For implementing Civet, we have extended a Java compiler written using *JastAdd Compiler Compiler* [9]. The modular structure of the JastAdd helped us easily extend the Java compiler. The Civet is about 4600 lines. It can be found at the following URL:
`http://www.cs.utexas.edu/~amshali/Civet/`

Civet currently uses annotations to specify compile-time variables rather than a special expression CT, as in semantics. In Civet, the specification is given using `@CompileTime` and `@CompileTimeIf` Java annotations. The `@CompileTimeIf(other_var)` annotation indicates a conditional situation where a variable is compile-time only if another variable with the name `other_var` is also a compile-time variable. Civet follows the closest scope rule to find the `other_var`. It generates pure Java code after partial evaluation, which makes it easier for debugging and further analysis.

There are several issues in the specialization of Java programs. One issue is in the class specialization. When Civet specializes a class constructor, it creates a new class which is a subclass of the class being specialized. It then copies the body of the super-class constructor to the subclass and then follows the semantics. The problem arises when some fields of the class are private. When the fields are private the new subclass cannot access them from within the constructor or methods.

Moreover, because partial evaluator creates a new constructor in the new generated class, it requires the original class to have a default constructor. This is because the original class might not have any constructor of the same parameters as the new specialized one. Thus, it must have at least a default constructor so as the program be able to create an object of the specialized type during runtime. In addition, a class cannot be *final* because it cannot be inherited from. These restrictions in Civet only applies to the classes which are going to be specialized.

5. Evaluation

We evaluate the performance and scalability of Civet on samples from several sources.

5.1 JSpec Suite

The JSpec test suite is created by Schultz et al. [25]. We list some of the examples from this suite with a short description:

- FFT: Fast Fourier Transform. The compile-time input for this case study is the size of radix which in our experiments are set to 16, 32 and 64.
- Romberg: This is an integration method. The compile-time input in this case study is the number of iteration which is set to 2 in our experiment.

[1] Civet is an animal that eats coffee beans and produces partially digested coffee berries which produce highly priced coffee.

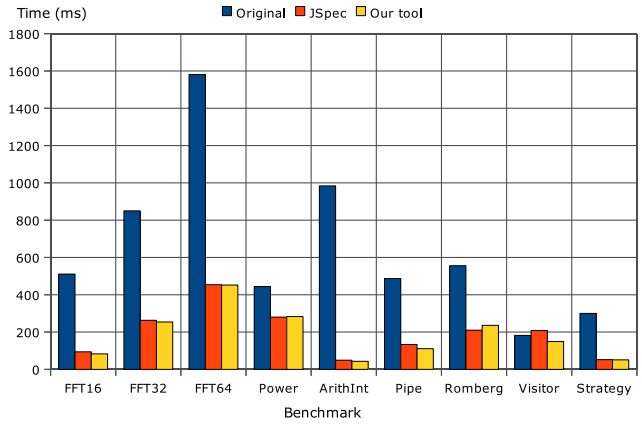

Figure 14. Time comparison between JSpec and Civet

- Power: Power function, x^n, where n is a natural number. The exponent is a compile-time value in this experiment.
- Pipe: Function composition. The composition is fixed.
- Visitor: Visitor pattern for operations on a binary tree. The choice of operations is known at compile time.
- Strategy: this is an image processing example using the strategy pattern. The specific operator is known at compile time.
- ArithInt: This case study is a simple arithmetic expression interpreter.

5.1.1 Performance

We compare the performance of Civet with JSpec on benchmarks from JSpec suite. We run Civet on the same original programs with the same set of partial inputs in order to get specialized programs. We then, run each specialized program with the rest of inputs and measure the run-times. Each benchmark is run ten times and we take the average run-time of all the ten execution to represent the final reported run-time. We run all the benchmarks on an Intel Core 2 Duo CPU P8400 2.26GHz machine with 2.8GiB of memory and running Ubuntu 10.04.

Figure 14 compares execution time between JSpec and Civet for all the case studies. Time is measured in milliseconds using the Java `currentTimeMillis()` call. This figure also shows the run-time of the original programs. Civet performs better than JSpec on all FFTs', ArithInt, Pipe, Visitor and Strategy and it performs slightly worse on the rest. The average speedup of JSpec on these examples is 5.19 and the average speedup of Civet is 5.7.

We measured the number of lines generated by Civet on different case studies. The number of lines of code of the program would increase after specialization because of method generation, loop unrolling etc. Nevertheless the effective code size, the code which is used during the execution, might be smaller. The number of lines of code increase on almost all the examples is about 1.2 to 2 times the number of lines of code in the original. On FFT examples, however, due to a lot of loop unrolling, the increase factor goes up to 7.6 on FFT64.

We compared the bytecode size of the generated programs by JSpec and Civet. The bytecode size would increase for the same reasons the lines of code would. The average bytecode size increase on these case studies for Civet is 1.37, while it is 1.39 for JSpec. Again, the effective bytecode size, the code that will be loaded into the memory, might be smaller. That is because specialization can eliminate some classes and therefore the residual program may not need to load them during runtime.

Note that we could not generate any code with JSpec because the tool is not available. We were only able to compile and run the generated code by JSpec.

5.2 ModelTalk Case Study

ModelTalk is a domain specific model driven framework [13]. It has an interpretive approach to model driven development. Since the execution is interpreter based, it is a good target for partial evaluation. We specialized a *Dynamic pricing system* called *Pontis* based on ModelTalk. The dynamic pricing system is a system for calculating the prices of different products by applying a set of *price promotions* to each of them. The promotions are known at compile time while the products are known at runtime. The run-time of the original system on a set of products for 2×10^6 iteration is $3153\ ms$, while the run-time of the specialized version of the system using Civet is about $512\ ms$. This is a factor of 6 speedup. This speedup is mainly gained by specializing the reflective method calls and turning them into normal method calls.

Figure 15 gives some code taken from the Pontis example. Figure 16 gives the specialized version of the code example. The original code has been partially evaluated with a compile-time list of *price promotions*. As shown in the Figure 16, the `calcPromotionalPrice` method call on the `promotion` object has been turned into a static method call on the `Promotion` class. In addition, the reflective method calls in `isEligible` has been turned into a normal method call. The specialized method names have been appended by a $ and a number.

```
1  class PromotionSystem {
2  ...
3  double calcPromotionalPrice(An_Event ev) {
4   double result = ev.getListPrice();
5   for (A_Promotion promotion : promotions) {
6    double p = promotion.calcPromotionalPrice(ev);
7    if (p < result) result = p;
8   }
9   return result;
10 }
11 ...
12 }
13 class Promotion {
14 ...
15 Double calcPromotionalPrice(An_Event ev) {
16  Double result = null;
17  if (eligibility.isEligible(ev))
18   result = discounter.calcDiscountedPrice(ev);
19  else result = ev.getListPrice();
20  return result;
21 }
22 ...
23 }
24 class EligibilityByPropertyValue {
25 ...
26 boolean isEligible(An_Event ev) {
27  boolean result = false;
28  try {
29   String propertyValue = (String)
30    ev.getClass().getMethod("get"+propertyName, null).
        invoke(ev, null);
31   if (propertyValue.contains(value)) result = true;
32  } catch (Exception e) {}
33  return result;
34 }
35 ...
36 }
```

Figure 15. Pontis System

```
class PromotionSystem {
...
static double calcPromotionalPrice$10508(An_Event ev) {
 double result = ev.getListPrice();
  double p = com.pontis.promotion.Promotion
               .calcPromotionalPrice$10509(ev);
  if (p < result) result = p;
 return result;
}
...
}
class Promotion {
...
static Double calcPromotionalPrice$10509(An_Event ev) {
 Double result = null;
 if (com.pontis.eligibility.EligibilityByPropertyValue
      .isEligible$10510(ev))
   result = com.pontis.discounter.PercentageDiscounter
     .calcDiscountedPrice$10511(ev);
 else result = ev.getListPrice();
 return result;
}
...
}
class EligibilityByPropertyValue {
...
static boolean isEligible$10510(An_Event ev) {
 boolean result = false;
 try {
   String propertyValue = ((com.pontis.event.
        MovieRentalEvent) ev).getDirector();
   if (propertyValue.contains("Cameron")) result = true
        ;
 } catch (Exception e) {}
 return result;
}
...
}
```

Figure 16. Specialized Pontis System

Program	Time (ms)
Original regex state machine	1189
Specialized regex state machine	573
dk.brics.automaton regex library	816

Table 1. The time comparison of regular expression matching between the state machine before and after specialization and the fast Brics Automaton

5.3 Regular Expression Case Study

The motivation behind this case study is to show the success of the partial evaluation in the optimization of general programs. This program is a pattern matching application using regular expressions. For the purpose of pattern matching of a regular expression we developed a simple and naive deterministic state machine library. This state machine library simply tests the input and makes transitions. After consuming all of the input it reports a successful match if it is in a final state.

We compare the run-time of the original state-based machine regular expression matcher with the specialized version of the state machine for detecting the occurrence of this regular expression: $(a \mid b)^*(abb \mid (a+b))$. We also compare the run-times against that of *dk.brics.automaton* [21]. Brics Automaton is a highly tuned automaton library which claims to do fast regular expression matching.

Example	NOA	LOC	NOA/LOC
Power	2	116	1.72
Romberg	6	127	4.72
Pipe	3	149	2.01
ArithInt	2	176	1.13
FFT	35	185	18.9
Visitor	5	226	2.21
StateMachine	1	325	0.30
Strategy	4	362	1.10
Pontis	2	938	0.21

Table 2. Number Of Annotations (NOA), Lines Of Code (LOC), and NOA/LOC factor for all the examples

Table 1 shows the run-time(in milliseconds) of the three programs for an input of length 10^7. Not surprisingly, the run-time of the specialized version of the state machine is less than the original state machine for the mentioned regular expression. However, the run-time of the specialized version is also less than that of Brics Automaton. This shows how partial evaluation can be used to generate efficient programs out of naive and general ones which can compete with highly tuned hand-written codes for the same functionality. For the same reason we mentioned before, we could not compare our results with that of JSpec on this case.

5.4 Scalability

There are two important aspects to scalability of hybrid partial evaluation. One is how much effort it requires to annotate the code for large programs. Second one is how much time it would take to specialize a program.

To measure the first aspect of the scalability of our method, we define and measure a factor called NOA/LOC. NOA is the number of annotations and LOC is the lines of code of the program. The NOA/LOC factor is the percentage of annotation with respect to the program size. We have listed the NOA/LOC for all the examples in Table 2. The value of this factor for all of the examples except the FFT is under %5 and their average is %1.3. This means that when using Civet, on average, we only need to annotate about %1.3 of the program regardless of the size of the program. This result is promising that we can expect almost the same constant factor of effort for even larger programs.

We investigated the reasons for high NOA/LOC factor in the FFT example. In this example there are many local and loop variables that must be tagged which increase the number of annotations. Civet is an implementation of the semantics of HPE. It is faithful to the semantics but it does not fully implement the semantics. Thus, in some cases programmer needs to specify more prior to partial evaluation. The full implementation of the semantics in Civet is left as future work.

Time scalability, on the other hand, depends on input and how much of the code is going to be affected by that input. For all the examples, the time taken to specialize was less than a second for each. We anticipate that even for larger programs with more than 100K lines of code, the time for partial evaluation would be linearly proportional to the code size.

6. Related Work

Partial evaluation has a long history. In this section we discuss the most relevant related work, specifically online partial evaluation of imperative languages, and partial evaluation of object-oriented languages.

An online partial evaluator makes decisions about what to specialize during the specialization process, while an offline partial

evaluator makes all the decisions before specialization. Ruf identifies two ways in which online partial evaluators can produce better results than offline partial evaluators [23]. On one hand, offline partial evaluators must approximate the situations that can arise at runtime, so they are not as precise as is possible in an online setting. On the other hand, they also cannot identify commonalities between situations that depend on actual values of data. Hybrid partial evaluation supports the improvements identified by Ruf, but the focus of HPE is ease of use and implementation, not better specialization. Since hybrid partial evaluation is guided by the programmer, the opportunities for specialization are likewise limited.

Hybrid partial evaluation uses an online strategy because we believe it is more direct and fits within existing compilers. The approach has some potential disadvantages. Online partial evaluation are often slower than offline partial evaluators, because they make complicated decisions at specialization time, and often repeat the same analysis [24]. However, if specialization time is a small part of the overall product development process, then specialization performance is not a major issue. Programmer's efficiency, and efficiency of the final software product are the most important factors.

Meyer presents the semantics of online partial evaluator for a Pascal-like language [20]. The language is imperative and has binary and unary operations and control flow structures, conditionals and loops. Meyer uses a continuation-passing semantics to implement state, but do not clone the continuation as suggested by Ruf [23]. Meyer has a more complex treatment of conditionals than the one given here, in which the stores produced by the two conditional branches are merged. In practice, we have not found a need for the more complex approach. Meyer provides a correctness proof of this Pascal-like language, but no practical evaluation. We leave the correctness proof of the hybrid partial evaluation as future works.

There are some works on partial evaluation of object-oriented languages such as Java [7, 19, 25, 26]. Schultz et al. [25] present a tool for automatic specialization of Java programs. Their tool is an offline partial evaluator. They show how partial evaluation can be used to reduce the overhead of object-oriented abstraction in generic programs [25]. Their tool does not support exceptions, multi-threading and reflection. Similarly, our methodology and tool do not offer anything for exceptions and multi-threading constructs yet. But we do have semantics and implementation for reflection.

Le Meur et al. [16] present a language which allows programmers to provide specifications in order to guide the partial evaluator. The specification tells the partial evaluator how to propagate the compile-time data throughout the program. The ideas behind their work and ours have similar roots. They use the programmer provided annotations to guide the offline partial evaluation of a high level language which is similar to C. They have adapted the Tempo [6] partial evaluator so that it uses the provided specifications by programmers instead of the information gathered by the binding time analyzer.

7. Conclusion

We presented a hybrid approach to partial evaluation of object-oriented languages, giving a formal definition of the technique for a miniature object-oriented language, MOOL. In MOOL, programmer must specify the compile-time expressions in programs. The hybrid partial evaluator uses the provided specification to infer what parts of the code should be specialized. Moreover, it incorporates the specification as seeds for exploiting opportunities for further specializations in other parts of the code. This hybrid approach supports method and class specialization, including specialization of partially objects. It can also convert reflective method calls into ordinary calls. However, it does not support *self-application* and therefore it can only provide the first of the three Futamura projections [11].

We described how the approach was used to build a hybrid partial evaluator for Java called Civet. While Civet is sufficient to optimize a number of real-world examples, in the current prototype some aspects of Java interfere with specialization. These include **`final`** and **`private`** modifiers on declarations. The burden of specification is light. One goal of our work is to develop techniques that can be incorporated into existing compilers. The entire Java partial evaluator took 4 person-months to build as an extension to an existing Java compiler.

The system was evaluated on a number of examples, including several Java programs written by other groups. The run-time of a small version of the Pontis dynamic pricing system, which uses model interpretation and reflection, was reduced by a factor of 6 (1/6 of the original run-time). The code generated by Civet performs as well and in some cases even better than the code generated by a state-of-the-art offline partial evaluator for Java, JSpec, which is based on Tempo [6]. Civet also handles reflection.

References

[1] Andersen, L.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis (1994), dIKU Research Report 94/19

[2] Braux, M., Noyé, J.: Towards partially evaluating reflection in java. In: PEPM '00: Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation. pp. 2–11. ACM, New York, NY, USA (1999)

[3] Brzozowski, J.A.: Derivatives of regular expressions. J. ACM 11, 481–494 (October 1964), `http://doi.acm.org/10.1145/321239.321249`

[4] Bulyonkov, M.: A theoretical approach to polyvariant mixed computation. PEMC pp. 51–64 (1988)

[5] Consel, C.: Polyvariant binding-time analysis for applicative languages. PEPM93 pp. 66–77 (1993)

[6] Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.N., Lawall, J., Noyé, J.: Tempo: specializing systems applications and beyond. ACM Comput. Surv. p. 19

[7] Dean, J., Chambers, C., Grove, D.: Identifying profitable specialization in object-oriented languages. PEPM94 pp. 85–96 (1994)

[8] Dussart, D., Bevers, E., Vlaminck, K.D.: Polyvariant constructor specialisation. PEPM95 pp. 54–65 (1995)

[9] Ekman, T., Hedin, G.: The jastadd system — modular extensible compiler construction. Sci. Comput. Program. 69(1-3), 14–26 (2007)

[10] Ershov, A.P., Ostrovski, B.N.: Controlled mixed computation and its application to systematic development of language-oriented parsers. In: The IFIP TC2/WG 2.1 Working Conference on Program specification and transformation. pp. 31–48. North-Holland Publishing Co. (1987)

[11] Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Systems, Computers, Controls 2, 45–50 (1999)

[12] Goldberg, A., Robson, D.: Smalltalk 80 : The Language. Addison-Wesley Series in Computer Science, Addison-Wesley Professional (January 1989)

[13] Hen-Tov, A., Lorenz, D.H., Schachter, L.: Modeltalk: A framework for developing domain specific executable models. CoRR abs/0906.3423 (2009)

[14] Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)

[15] Jones, S.P.: Haskell 98 Language and Libraries. Cambridge University Press (2003)

[16] Le Meur, A.F., Lawall, J.L., Consel, C.: Specialization scenarios: A pragmatic approach to declaring program specialization. Higher Order Symbol. Comput. 17(1-2), 47–92 (2004)

[17] Lindholm, T., Yellin, F.: Java(TM) Virtual Machine Specification, The (2nd Edition). Prentice Hall PTR, 2 edn. (April 1999)

[18] Löh, A.: lhs2tex. http://people.cs.uu.nl/andres/lhs2tex/

[19] Marquard, M., Steensgaard, B.: Partial Evaluation of an Object-Oriented Imperative Language. Master's thesis (April 1992)

[20] Meyer, U.: Correctness of on-line partial evaluation for a pascal-like language. Sci. Comput. Program. 34(1), 55–73 (1999)

[21] Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2010), http://www.brics.dk/automaton/

[22] Poole, J.D.: Model-driven architecture: Vision, standards and emerging technologies. In: In In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models (2001)

[23] Ruf, E., Weise, D.: Opportunities for online partial evaluation. Tech. Rep. CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA (April 1992)

[24] Ruf, E.S.: Topics in online partial evaluation. Ph.D. thesis, Stanford University, Stanford, CA, USA (1993)

[25] Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for java. ACM Trans. Program. Lang. Syst. 25(4), 452–499 (2003)

[26] Schultz, U.P.: Partial evaluation for class-based object-oriented languages. In: PADO '01: Proceedings of the Second Symposium on Programs as Data Objects. pp. 173–197. Springer-Verlag, London, UK (2001)

[27] Thiemann, P., Dussart, D.: Partial evaluation for higher-order languages with state (1996)

[28] Wadler, P.: Monads for functional programming. In: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text. pp. 24–52. Springer-Verlag, London, UK (1995)