Hybrid Partial Evaluation

Amin Shali, Sidney Rosario, and William R. Cook

Department of Computer Science
University of Texas at Austin
{amshali,srosario,wcook}@cs.utexas.edu

Abstract. We present hybrid partial evaluation (HPE), 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.

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 [23]. 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 [17]. The following example illustrates how HPE can be used to optimize a naive regular expression library. The use of CT tells the compiler to instantiate the Regex at compile time. When the execute method is invoked on a runtime buffer, HPE converts the regular expression interpreter into 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.

```java
Regex regex = CT(new Regex("(a|b)*(abb|a+b)");
regex.execute(buffer);
```

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. On the other hand, the kinds of specializations performed are similar to those performed by an offline partial evaluator, since HPE does not seek to improve on the level of specialization provided by offline evaluators. In some ways, the hybrid partial evaluator is less powerful than previous approaches. HPE does not support “object lifting”, or migration of a compile-time object to runtime. HPE also checks to ensure that executing imperative code at compile time is consistent with the original semantics of the program.

Programs that do not obey these rules are rejected by the hybrid partial evaluator. For instance, a typical example of a poor binding time involves lookup of a runtime value in a compile-time hashtable.

```java
1 CommandTable t = CT(CommandTable.load("config.data"));
2 String d = System.in.readLine();
3 Command cmd = t.lookup( d );
4 cmd.process( d );
```

HPE gives a compilation error on line 3, because the compile-time object referenced by t cannot be invoked at runtime to lookup the value d. The well known fix for this problem involves rewriting the last two lines as:

```java
1 for ( Command cmd : t.getCommands() )
2 if ( cmd.matches( d ) )
3 cmd.process( d );
```

This program runs more slowly without partial evaluation, but it can be faster than the original program after the partial evaluation.

Hybrid partial evaluation rejects programs with incorrect binding times, rather than silently generating inefficient residual code. HPE also 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
Hybrid Partial Evaluation

data Primitive = null | String | Int | Bool | [v]
data v = Primitive | C:ρ | ()
type Prog = CD
data CD = class C(τ) {var τ; init{e} MD}
data mod = static | method
data MD = mod m(τ) {e}
data op = + | − | * | / | == | != | < | > | %
data e = v
  | x
  | C
  | var x = e; e
  | x := e
  | e
  | e op e
  | if e then e else e
  | while e do e
  | e.m(τ)
  | invoke(e, e, τ)
  | new C(τ)
  | CT(e, e)
  | RT(e)
  | IsCT(e)

Fig. 1. Syntax of MOOL

rewrite the program to avoid the problem. These restrictions allow developers to understand and rely on the optimizations performed by the partial evaluator.

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 [26], an existing offline partial evaluator for Java, hybrid partial evaluation generates code that is as efficient as JSpec’s residual code. HPE also specializes reflective code, which allows it to optimize a third-party pricing application [13]. Initial results show an average 6 times speedup of specialized programs.

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 [18]. 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 lists all the formal parameters of the class constructor, field declarations, and an expression which is the body of the constructor. 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 being 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.

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

$CT(e, e)$ and $RT(e)$ are expressions used to mark other 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.

2.2 Semantics

The semantics of MOOL is shown in Figure 2. All the semantics definitions in this paper are written in Haskell [16], so they are executable. Literate Haskell [19] is used to render the definitions in more conventional style. We assume that reader is familiar with monads. 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/index.php/Main/Civet

The types of environment ($\rho$) and store ($\sigma$) are shown in Figure 2. An environment, $\rho$, maps variable or field names to locations. A store, $\sigma$, maps locations to values. $EV$ is the type of the evaluation result. $\tilde{v}$ represents an abstract (approximate) value. It is used when the hybrid partial evaluator has partial information about a value. $\top$ value is also used during the partial evaluation and it represents a value for which the hybrid partial evaluator has no information. A $\bot$ 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. $\rho$ is the environment for this object, which contains the locations of its fields.

The function $E[\cdot][\cdot]$, is referred to as the “full evaluator” to distinguish it from the “partial evaluator” defined in Section 3. This function, $E[e][\rho]$, executes the
data \( EV = \nu | v | \top | \bot \)
\( \rho : x \rightarrow l \)
\( \sigma : l \rightarrow EV \)
\( \mathcal{E}[\cdot] : e \rightarrow \rho \rightarrow \text{State}(\text{Prog}, \sigma, \text{NameMap}) v \)

\[
\mathcal{E}[v] \rho = \text{return } v
\]

\[
\mathcal{E}[e_1 \ op \ e_2] \rho = \text{do}
\]
\[
\quad v_1 \leftarrow \mathcal{E}[e_1] \rho
\]
\[
\quad v_2 \leftarrow \mathcal{E}[e_2] \rho
\]
\[
\quad \text{return } op(v_1, v_2)
\]

\[
\mathcal{E}[x] \rho = \text{do}
\]
\[
\quad (\text{\_\_} \sigma, \text{\_\_}) \leftarrow \text{get}
\]
\[
\quad \text{return } \sigma(\rho(x))
\]

\[
\mathcal{E}[\text{var } x = e_1; \ e_2] \rho = \text{do}
\]
\[
\quad v \leftarrow \mathcal{E}[e_1] \rho
\]
\[
\quad [x \mapsto l] \leftarrow \text{allocate } [x \mapsto v]
\]
\[
\quad \mathcal{E}[e_2]([x \mapsto l] + \rho)
\]

\[
\mathcal{E}[x := e] \rho = \text{do}
\]
\[
\quad v \leftarrow \mathcal{E}[e] \rho
\]
\[
\quad \text{update } \rho(x) v
\]
\[
\quad \text{return } v
\]

\[
\mathcal{E}[e_1; \ e_2] \rho = \text{do}
\]
\[
\quad \mathcal{E}[e_1] \rho
\]
\[
\quad \mathcal{E}[e_2] \rho
\]

\[
\mathcal{E}[\text{if } e_1 \ then \ e_2 \ else \ e_3] \rho = \text{do}
\]
\[
\quad b \leftarrow \mathcal{E}[e_1] \rho
\]
\[
\quad \text{case } b \text{ of}
\]
\[
\quad \quad \text{True} \rightarrow \mathcal{E}[e_2] \rho
\]
\[
\quad \quad \text{False} \rightarrow \mathcal{E}[e_3] \rho
\]

\[
\mathcal{E}[\text{while } e_1 \ do \ e_2] \rho = \text{do}
\]
\[
\quad \mathcal{E}[\text{if } e_1 \ then \ (e_2; \ \text{while } e_1 \ do \ e_2) \ else \ \text{null}] \rho
\]

\[
\mathcal{E}[\text{invoke}(e, e_m, \pi)] \rho = \text{do}
\]
\[
\quad m \leftarrow \mathcal{E}[e_m] \rho
\]
\[
\quad \mathcal{E}[e.m(\pi)] \rho
\]

\[
\mathcal{E}[\text{new } C(\pi)] \rho = \text{do}
\]
\[
\quad \text{class } C(\pi) \{ \text{\_\_} \} \leftarrow \text{createClass } C
\]
\[
\quad \pi \leftarrow \text{map}(\mathcal{E}[\cdot]) \rho \pi
\]
\[
\quad [\pi \mapsto l] \leftarrow \text{allocate } [\pi \mapsto \pi]
\]
\[
\quad \rho' \leftarrow \text{allocate } [\pi \mapsto \pi]
\]
\[
\quad \mathcal{E}[e_1]([\pi \mapsto l] + \rho')
\]
\[
\quad \text{return } C; \rho'
\]

Fig. 2. Full evaluation MOOL expressions
program represented by an expression $e$ in the environment $\rho$ and returns a value attached to a state monad [29]. The state monad contains the program, a store and a NameMap. The program is in the store for consistency with the partial evaluator, which extends the program during evaluation. NameMap is a mapping of partially evaluated methods and classes to their specialized names in the program. It is only used in the partial evaluation. In Figure 2, get is a monad function which retrieves the program, the store and the NameMap in a tuple.

In evaluation of a variable declaration in Figure 2, the allocate function takes a list of name-value pairs $[x \mapsto v]$ and returns a list of name-location pairs $[x \mapsto l]$ and updates the store so that each location contains the corresponding value. The update function in the full evaluation of a variable assignment updates a location $l$ in the store with a new value.

The full evaluator assumes that all free variables are present in the environment, otherwise an error is thrown. 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. If there is type incompatibility, it reports an error. The full evaluation of if-expression starts with the evaluation of the condition expression and then full evaluator evaluates either the if or else branch, based on the value of the condition. The full evaluation of the while loop uses the full evaluation of if-expression.

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 for this object, actual parameters and the target object’s fields, $\rho'$.

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, $\bot$, and evaluates the body of the constructor and returns an object $C:\rho'$.

# 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. The known inputs are called *compile-time* values, while all other values are called *runtime* values.

The type of hybrid partial evaluator, $\mathcal{P}[\cdot]$, is shown in Figure 3. A hybrid partial evaluator, like an online one, works very much like a full evaluator. However, the environment in this case may be partial. A hybrid partial environment contains not only constant primitives and objects, but also abstract (or approximate) ones represented by $\sim v$. The result of hybrid partial evaluation is an expression accompanied with a value, which can be abstract. 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.

Partial evaluation of constants always produce abstract values. Partial evaluation of basic expressions is given in Figure 3. Binary operators, $e_1 \text{ op } e_2$, return a compile-time value if either $e_1$ or $e_2$ partially evaluate to a compile-time value, otherwise return an abstract value.

### 3.1 Variable Declaration and Assignment

Figure 3 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, $\sim v$. 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. Otherwise, it returns a residual code which contains the name of the variable along with the abstract value stored for that variable. A variable declaration may or may not be a compile-time variable. If the partial evaluated value of $e_1$ is compile-time, 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 $\text{True}$, 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 $\text{True}$ when $e$ is a compile-time value. $RT(e)$ expression marks an expression as
data PEResult = (e, EV)

\[ P[\top] :: e \to \rho \to \text{State } (\text{Prog}, \sigma, \text{NameMap}) \text{ PEResult} \]

\[ P[v] \rho = \text{return } ([v], \top) \]

\[ P[x] \rho = \text{do} \]
\[ \langle \sigma, \omega \rangle \leftarrow \text{get} \]
\[ \text{case } \sigma(\rho(x)) \text{ of} \]
\[ v \to \text{return } ([v], v) \]
\[ p \to \text{return } ([x], p) \]

\[ P[\text{var } x = e_1; e_2] \rho = \text{do} \]
\[ \langle e'_1, p_1 \rangle \leftarrow P[e_1] \rho \]
\[ [x \mapsto l] \leftarrow \text{allocate } [x \mapsto p_1] \]
\[ \langle e'_2, p_2 \rangle \leftarrow P[e_2] \rho \]
\[ \text{case } p_1 \text{ of} \]
\[ v \to \text{return } (e'_2, p_2) \]
\[ \text{else } \to \]
\[ \text{return } ([x = e'_1; e'_2], p_2) \]

\[ P[\text{CT}(e, e')] \rho = \text{do} \]
\[ v' \leftarrow E[e'] \rho \]
\[ \text{if } v' \equiv \text{True} \text{ then do} \]
\[ v \leftarrow E[e] \rho \]
\[ \text{return } ([v], v) \]
\[ \text{else } P[e] \rho \]

\[ P[\text{let } x := e] \rho = \text{do} \]
\[ \langle e', p \rangle \leftarrow P[e] \rho \]
\[ x \leftarrow e' \]
\[ \text{return } ([x := e'], p) \]

\[ P[\text{if } CT(e)] \rho = \text{do} \]
\[ \langle e', p \rangle \leftarrow P[e] \rho \]
\[ \text{case } p \text{ of} \]
\[ v \to \text{return } ([\text{True}], \text{True}) \]
\[ \text{else } \to \text{return } ([\text{False}], \text{False}) \]

\[ P[\text{RT}(e)] \rho = \text{do} \]
\[ \text{return } ([e], \top) \]

\[ P[e_1 \text{ op } e_2] \rho = \text{do} \]
\[ \langle e'_1, p_1 \rangle \leftarrow P[e_1] \rho \]
\[ \langle e'_2, p_2 \rangle \leftarrow P[e_2] \rho \]
\[ \text{case } (p_1, p_2) \text{ of} \]
\[ (v_1, v_2) \to \]
\[ \text{let } v = \text{op}(v_1, v_2) \text{ in return } ([v], v) \]
\[ (v_1, v_2) \to \]
\[ \text{let } v = \text{op}(v_1, v_2) \text{ in return } ([v], v) \]
\[ (v_1, v_2) \to \]
\[ \text{let } v = \text{op}(v_1, v_2) \text{ in return } ([v], v) \]
\[ \text{else } \to \text{return } ([v \text{ op } e'_2], \top) \]

\[ P[x := e] \rho = \text{do} \]
\[ \langle \sigma, \omega \rangle \leftarrow \text{get} \]
\[ \text{case } \sigma(\rho(x)) \text{ of} \]
\[ v \to \text{do} \]
\[ v' \leftarrow E[e] \rho \]
\[ \text{update } p(x) \leftarrow v' \]
\[ \text{return } ([x := e'], p) \]

\[ P[l : CT(e)] \rho = \text{do} \]
\[ \langle e', p \rangle \leftarrow P[e] \rho \]
\[ l \leftarrow e' \]
\[ \text{return } ([l := e'], \top) \]

\[ P[l : \perp] \rho = \text{do} \]
\[ \langle e', p \rangle \leftarrow P[e] \rho \]
\[ l \leftarrow e' \]
\[ \text{return } ([l := e'], \top) \]

\[ \text{case } p \text{ of} \]
\[ v' \to \text{do} \]
\[ \text{return } ([v'], v') \]
\[ \text{else } \to \]
\[ \text{return } ([l := e'], p) \]

Fig. 3. Partial evaluation of basic values, variables, operators, variable declarations, and assignments
Partial evaluation of control flow constructs

\[
\begin{align*}
P[e_1; e_2] \rho &= \text{do} \\
\langle e_1', p_1 \rangle &\leftarrow P[e_1] \rho \\
\langle e_2', p_2 \rangle &\leftarrow P[e_2] \rho \\
\text{return} \langle e_1'; e_2', p_2 \rangle \\
\end{align*}
\]

\[
\begin{align*}
P[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] \rho &= \text{do} \\
\langle e_1', p_1 \rangle &\leftarrow P[e_1] \rho \\
\text{case } p_1 \text{ of} \\
\text{True} &\rightarrow P[e_2] \rho \\
\text{False} &\rightarrow P[e_3] \rho \\
\text{else} &\rightarrow \text{checkStore } \rho (\text{if } e_1' \text{ then } \cdot \text{ else } \cdot) e_2 e_3 \\
\end{align*}
\]

\[
\begin{align*}
P[\text{while } e_1 \text{ do } e_2] \rho &= \text{do} \\
\langle e_1', p_1 \rangle &\leftarrow P[e_1] \rho \\
\text{case } p_1 \text{ of} \\
\text{True} &\rightarrow P[e_2]; \text{ while } e_1 \text{ do } e_2] \rho \\
\text{False} &\rightarrow P[\text{null}] \rho \\
\text{else} &\rightarrow \text{checkStore } \rho (\text{while } \cdot \text{ do } \cdot) e_1 e_2 \\
\end{align*}
\]

Fig. 4. Partial evaluation of control flow constructs

runtime. 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 4 defines hybrid partial evaluation of control flow statements. Sequences and loops are straightforward. Note that there is no guarantee that partial evaluation terminates.

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 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 5 [21]. 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.

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 [21] 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 [28, 24]. Although, this has the potential to duplicate large amounts of code.
method iftest (a) {
    var x = CT(3, True);
    var y = CT(4, True);
    if (a < x) {
        y := 2 * x;
        x := 3 + y;
    } else
        x := 5 + y;
}

Fig. 5. The problematic example of an if-expression for the partial evaluation

HPE has a pragmatic approach to this problem. The checkStore function (See Figure 4) evaluates both branches and then looks for inconsistencies in the state. If the resulting stores are different with respect to the initial environment, 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. We have found that this pragmatic approach is sufficient for many common programming idioms, as shown in Section 4.

For the example in Figure 5, 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(\(\pi\)), HPE specializes the class C if any of the parameters to the constructor call are compile-time. Class specialization is defined in Figure 6. For class specialization, the partial evaluator finds 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.
\( \mathcal{P}[\text{new } C(\overline{\text{a}}) \rho = \text{do} \]
\[
[\overline{\text{a}} \mapsto \overline{\text{p}}] \leftarrow \text{map}(\mathcal{P}[\text{\cdot}]\rho)\overline{\text{a}}
\]
\[
\text{if any isCompileTime } \overline{\text{p}} \text{ then do}
\]
\[
(\text{memoized}, C', \text{class } \overline{\text{\{f init\{e_\} \overline{\text{m}}\}}}) \leftarrow \text{findMemoClass } C\overline{\text{p}}
\]
\[
[\overline{\text{p}} \mapsto \overline{\text{t}}] \leftarrow \text{allocate } [\overline{\text{p}} \mapsto \overline{\text{p}}]
\]
\[
\overline{\text{m}} = \text{getRuntimeNames } \overline{\text{p}}[\overline{\text{p}} \mapsto \overline{\text{p}}]
\]
\[
\overline{\text{m}}' = \text{getRuntimeExprs } [\overline{\text{p}} \mapsto \overline{\text{p}}]
\]
\[
\rho' \leftarrow \text{allocate } [\overline{\text{f}} \mapsto \bot]
\]
\[
[\overline{\text{this}} \mapsto \overline{\text{l}}'] \leftarrow \text{allocate } ["\text{this}" \mapsto C';\rho']
\]
\[
\langle e', \overline{\text{e}}' \rangle = \mathcal{P}[\text{\{e\}}](\overline{\text{m}}' + [\overline{\text{t}} \mapsto \overline{\text{t}}] + \overline{\text{p}})
\]
\[
\text{when } (\neg \text{memoized}) \text{ do}
\]
\[
\overline{\text{m}}' \leftarrow \text{map}(\mathcal{M}[\text{\{this \mapsto \overline{\text{l}}'\}} + \overline{\text{p}}])\overline{\text{m}}
\]
\[
\overline{\text{f}}' \leftarrow \text{getRuntimeFields } \overline{\text{f}}\rho'
\]
\[
\text{addClass class } C'(\overline{\text{t}}\overline{\text{f}}), \langle \overline{\text{f}}' \text{ init}\{e_\} \overline{\text{m}}' \rangle\rangle
\]
\[
\text{return } \langle [\overline{\text{new } C'(\overline{\text{m}})}, C';\rho']\rangle
\]
\[
\text{else do}
\]
\[
\text{class } \overline{\text{\{f init\{\} \_\}}} \leftarrow \text{findClass } C
\]
\[
\rho' \leftarrow \text{allocate } [\overline{\text{f}} \mapsto \bot]
\]
\[
\text{return } \langle [\overline{\text{new } C(\overline{\text{a}})}], C;\rho'\rangle
\]
\[
\mathcal{M}[\text{modifier } m(\overline{\text{t}}) \{e\}]\rho = \text{do}
\]
\[
\rho' \leftarrow \text{allocate } [\overline{\text{x}} \mapsto \bot] \mid \overline{\text{x}} \leftarrow \overline{\text{t}}
\]
\[
\langle e', \overline{\text{e}}' \rangle = \mathcal{P}[\text{\{e\}}(\rho + \rho')
\]
\[
\text{return } \text{modifier } m(\overline{\text{t}}) \{e'\}
\]

Fig. 6. Partial evaluation of constructors for partial objects

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 \texttt{this} object in the store and returns an approximate object with the required residual code.

### 3.5 Method Specialization

HPE can specialize method calls \( o \). \( m(\overline{\text{t}}) \) on compile-time objects, which were introduced in Section 3.1. Since a compile-time object 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 7. 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
\[ P[e \cdot m(\pi)]\rho = \text{do} \]
\[ \langle e', p \rangle \leftarrow P[e]\rho \]
\[ \langle \tau \rightarrow p' \rangle \leftarrow \text{map}(P[\cdot]\rho) \pi \]
\text{ case } p \text{ of } C;\rho' \rightarrow \text{do} \]
\[ \text{if all isCompileTime } p \text{ then do} \]
\[ \langle e', p \rangle \leftarrow \text{findMethod } C \ m \ (\text{length } \pi) \]
\[ \langle \tau \rightarrow l \rangle \leftarrow \text{allocate } (\tau \rightarrow p') \]
\[ \text{return } (\langle e \rangle, v) \]
\text{ else } S[\pi]^{\text{static }} \rho\ C \ m \ [\bar{a} \mapsto p'] \]
\text{ C;\rho' \rightarrow } \quad \text{ -- target is an approximate object} \]
\[ \text{if any isCompileTime } p \text{ then do} \]
\[ \langle \text{this } \mapsto l' \rangle \leftarrow \text{allocate } (\text{"this" } \mapsto C;\rho') \]
\[ S[e']^{\text{method }} ([\text{this } \mapsto l'] + [\tau \mapsto l] + \rho') \ C \ m \ [\bar{a} \mapsto p'] \]
\text{ else } \text{return } (\langle e' \cdot m(\overline{a}) \rangle, \top) \]
\text{ else } \quad \text{ -- target is unknown} \]
\[ \text{if any isCompileTime } p \text{ then do} \]
\[ m' \leftarrow \text{specializeAll } \rho \ m \ [\bar{a} \mapsto p'] \]
\[ \text{let } \overline{ad} = \text{getRuntimeExprs } [\bar{a} \mapsto p'] \]
\[ \text{return } (\langle e' \cdot m'(\overline{ad}) \rangle, \top) \]
\text{ else } \quad \text{return } (\langle e' \cdot m(\overline{a}) \rangle, \top) \]
\[ P[\text{invoke}(e, e \cdot m, \overline{a})] \rho = \text{do} \]
\[ \langle e', p \rangle \leftarrow P[e]\rho \]
\text{ case } e' \text{ of } \]
\[ m \rightarrow P[e \cdot m(\pi)]\rho \]
\text{ else } \quad \text{do} \]
\[ \langle e', p' \rangle \leftarrow P[e]\rho \]
\[ \langle \tau \rightarrow p' \rangle \leftarrow \text{map}(P[\cdot]\rho) \pi \]
\text{ return } (\langle \text{invoke}(e', e \cdot m, \overline{a}) \rangle, \top) \]

Fig. 7. Partial evaluation of method calls and reflective method calls for partial objects

program, the new method must be \textit{static}. The function \( S[e]^{\text{modifier }} \rho \ C \ m \ \overline{a} \) (See Figure 8) creates a specialized version of a method. In this case the new method is marked as \textit{static}. New methods are stored in a cache, so that the same specialization of a method is not generated twice. The method specializer \( S[e]^{\text{modifier }} \rho \ C \ m \ \overline{a} \) 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 \([15, 1, 5]\). The hybrid partial evaluator uses the \textit{polyvariant specialization} strategy for program point specialization. It memoizes a call expression, \(e \cdot m(\overline{x})\), so that it can be reused from other call sites. It also memoizes object creation expressions (constructor calls). Memoization is implemented in the \textit{findMemoCall} and \textit{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 \textit{NameMap} part of the state monad.

Now consider the method call \(o \cdot m(\overline{x})\) 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 \(S\) and creates a residual instance method. This is shown in Figure 7. 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 \textit{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.

**Reflective Calls** Figure 7 also defines the partial evaluation of reflective calls. When the partial evaluation of \(e \cdot m\) results in a string value, \(m\), the name 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:

```java
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 [3]:
obj.test(arglist);
```

## 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/index.php/Main/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.

\(^1\) Civet is an animal that eats coffee beans and produces partially digested coffee berries which produce highly priced coffee.
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. [26]. 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.
- 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.

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 9 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 $2 \times 10^6$ 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 10 gives some code taken from the Pontis example. Figure 11 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 11, 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` have been turned into a normal method call. The specialized method names have been appended by a $ and a number.
class PromotionSystem {
    double calcPromotionalPrice(An_Event ev) {
        double result = ev.getListPrice();
        for (A_Promotion promotion : promotions) {
            double p = promotion.calcPromotionalPrice(ev);
            if (p < result) result = p;
        }
        return result;
    }
}

class Promotion {
    Double calcPromotionalPrice(An_Event ev) {
        Double result = null;
        if (eligibility.isEligible(ev))
            result = discounter.calcDiscountedPrice(ev);
        else result = ev.getListPrice();
        return result;
    }
}

class EligibilityByPropertyValue {
    boolean isEligible(An_Event ev) {
        boolean result = false;
        try {
            String propertyName = (String)
                ev.getClass().getMethod("get"+propertyName, null).invoke(ev, null);
            if (propertyName.contains(value)) result = true;
        } catch (Exception e) {}
        return result;
    }
}

class EligibilityByPropertyValue {
    boolean isEligible(An_Event ev) {
        boolean result = false;
        try {
            String propertyName = (String)
                ev.getClass().getMethod("get"+propertyName, null).invoke(ev, null);
            if (propertyName.contains(value)) result = true;
        } catch (Exception e) {}
        return result;
    }
}

Fig. 10. Pontis System

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.
class PromotionSystem {
  static double calcPromotionalPrice(An_Event ev) {
    double result = ev.getListPrice();
    double p = com.pontis.promotion.Promotion calcPromotionalPrice(ev);
    if (p < result) result = p;
    return result;
  }
}

class Promotion {
  static Double calcPromotionalPrice(An_Event ev) {
    Double result = null;
    if (com.pontis.eligibility.EligibilityByPropertyValue.isEligible(ev))
      result = com.pontis.discounter.PercentageDiscounter calcDiscountedPrice(ev);
    else result = ev.getListPrice();
    return result;
  }
}

class EligibilityByPropertyValue {
  static boolean isEligible(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;
  }
}

Fig. 11. Specialized Pontis System

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: \((a | b)^* (abb | (a + b))\). We also compare the run-times against that of \texttt{dk.brics.automaton} [22]. Brics Automaton is a highly tuned automaton library which claims to do fast regular expression matching.

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
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.

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.

We also specialized two more third party applications:

- File Carving: A generic program for recovering lost files by using file format descriptions and applying some heuristics.
- MJParser: A recursive-decent parser with memoization.

The File Carving and MJParser example are two real world applications written by Tijs van der Storm [2]. MJParser is a memoization based top-down parser [14] written in Java.

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.
<table>
<thead>
<tr>
<th>Example</th>
<th>NOA</th>
<th>LOC</th>
<th>NOA/LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>2</td>
<td>116</td>
<td>1.7</td>
</tr>
<tr>
<td>Romberg</td>
<td>6</td>
<td>127</td>
<td>4.7</td>
</tr>
<tr>
<td>Pipe</td>
<td>3</td>
<td>149</td>
<td>2.0</td>
</tr>
<tr>
<td>ArithInt</td>
<td>2</td>
<td>176</td>
<td>1.1</td>
</tr>
<tr>
<td>FFT</td>
<td>35</td>
<td>185</td>
<td>18.9</td>
</tr>
<tr>
<td>Visitor</td>
<td>5</td>
<td>226</td>
<td>2.2</td>
</tr>
<tr>
<td>StateMachine</td>
<td>1</td>
<td>325</td>
<td>0.3</td>
</tr>
<tr>
<td>Strategy</td>
<td>4</td>
<td>362</td>
<td>1.1</td>
</tr>
<tr>
<td>Pontis</td>
<td>2</td>
<td>938</td>
<td>0.2</td>
</tr>
<tr>
<td>MJParser</td>
<td>1</td>
<td>1245</td>
<td>2.5</td>
</tr>
<tr>
<td>File Carving</td>
<td>8</td>
<td>3651</td>
<td>0.6</td>
</tr>
</tbody>
</table>

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

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 [24]. 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 [25]. 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 [21]. 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 [24]. 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 [27, 26, 7, 20]. Schultz et al. [26] 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 [26]. 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. [17] 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’s 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 \texttt{final} and \texttt{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