TECHNICAL REPORT 95-10

Partial Evaluator as a Compiler for Reflective Languages

Kenichi Asai Hidehiko Masuhara Satoshi Matsuoka
Akinori Yonezawa

December 13, 1995

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TECHNICAL REPORT 95-10

TITLE
Partial Evaluator as a Compiler for Reflective Languages

AUTHORS
Kenichi Asai, Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa

KEY WORDS AND PHRASES
partial evaluation, reflection, preaction, I/O-type side-effects

ABSTRACT
This paper presents an online partial evaluator with a mechanism to handle I/O-type side-effects using *preactions*, and reports our experiment of using the partial evaluator as a compiler for the reflective language *Black* we are designing. Black is a Scheme-based reflective language, which allows user programs to access and modify its metalevel interpreter (or the language semantics) from within the same language framework. Because the semantics may change during computation, it is impossible to compile using a conventional Scheme compiler. To cope with this flexibility, we implemented an online partial evaluator, and specialized a Scheme meta-circular interpreter with respect to a modified interpreter to obtain an efficient version of the modified interpreter. The resulting interpreter turns out to be quite efficient in that it is almost identical to the original Scheme interpreter, except that it correctly reflects the modification made by users. In fact, we got more: by supplying a specific user program, we obtained a compiled program under the modified language semantics.

REPORT DATE	**WRITTEN LANGUAGE**
December 13, 1995	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
14	18

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

DISTRIBUTION STATEMENT
First issue 35 copies.

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku Tokyo, 113 Japan

Partial Evaluator as a Compiler for Reflective Languages

Kenichi Asai Hidehiko Masuhara* Satoshi Matsuoka† Akinori Yonezawa

Abstract

This paper presents an online partial evaluator with a mechanism to handle I/O-type side-effects using *preactions*, and reports our experiment of using the partial evaluator as a compiler for the reflective language *Black* we are designing. Black is a Scheme-based reflective language, which allows user programs to access and modify its metalevel interpreter (or the language semantics) from within the same language framework. Because the semantics may change during computation, it is impossible to compile using a conventional Scheme compiler. To cope with this flexibility, we implemented an online partial evaluator, and specialized a Scheme meta-circular interpreter with respect to a modified interpreter to obtain an efficient version of the modified interpreter. The resulting interpreter turns out to be quite efficient in that it is almost identical to the original Scheme interpreter, except that it correctly reflects the modification made by users. In fact, we got more: by supplying a specific user program, we obtained a compiled program under the modified language semantics.

1 Introduction

This paper presents an online partial evaluator with a mechanism to handle I/O-type side-effects using *preactions*[14], and reports our experiment of using the partial evaluator as a compiler for our reflective language called *Black*[2]. Black is a Scheme-based reflective language, which allows user programs to access and modify its metalevel interpreter (or the language semantics) from within the same language framework. For example, suppose that we want to change the language system so that it prints traces, i.e., it prints expressions whenever it evaluates them. To achieve this with a conventional Scheme interpreter, we would have to fully understand its low-level implementation (which is usually written in lower level languages), and carefully modify appropriate places in the implementation code. Reflective languages substantially simplify such modifications by representing a high-level abstraction view of the interpreter within the same language framework, relieving the user from unnecessary details of the low-level implementation.

The structure of the metalevel interpreter in Black is given by an ordinary meta-circular interpreter[1]. With clear knowledge of how the metalevel interpreter is constructed, we can easily program the above trace example in Black as follows:

```
0-1> (exec-at-metalevel
       (let ((old-eval eval))
          (set! eval
                (lambda (exp env)
```

*Department of Graphics and Computer Science, College of Arts and Sciences, University of Tokyo
†Department of Information Engineering, Faculty of Engineering, University of Tokyo

```
                    (write 'trace:)
                    (write exp)
                    (newline)
                    (old-eval exp env)))))
```

`Exec-at-metalevel` is a special construct in Black: the body of the construct is evaluated at the metalevel, installing a 'wrapper' to the metalevel evaluator in this case. After this program is executed, traces are displayed every time the metalevel interpreter executes the `eval` function:

```
0-2> (+ 3 4)
trace:(+ 3 4)
trace:+
trace:3
trace:4
0-2: 7
```

The performance of conventional reflective languages such as 3-LISP[8] and $\mathcal{I}_R$[11], however, suffers substantially due to such flexibility.

Under straightforward implementation schemes, user programs would be initially interpreted by an efficient directly executable metalevel interpreter. However, an interpreted metalevel interpreter would be lazily created once a user program executes `exec-at-metalevel` to perform modifications at metalevel. Henceforth, user programs are interpreted by the created metalevel interpreter, which is interpreted by a directly executable *metametalevel* interpreter. Due to this 'double' interpretation, the overall execution becomes quite inefficient.

The direct execution technique, employed in Brown[9, 17] and Blond[7], cannot be used here, because Black allows user programs to *modify* the metalevel interpreter itself. Direct execution of metalevel interpreter makes modification of interpreters impossible.

One way to efficiently execute reflective programs that are allowed to modify its metalevel interpreter is to enumerate the changes users are likely to perform and prefabricate an efficient customized metalevel interpreter for each. This method, however, severely restricts the flexibility of reflective languages. Instead, we take a more general approach: to cope with every change users may make, we employ a partial evaluator as a compiler that can account for the modified metalevel interpreter. We faithfully follow the first Futamura projection[10]. This method might seem to be overly general, in that residual code would preserve too much of the 'double' interpretation structure above, resulting only in minor performance gains. But according to our experiment, the obtained interpreter turns out to be quite efficient, and runs more than 30 times faster than the double interpretation. It is almost identical to the standard Scheme interpreter, except that it correctly reflects the modification made by users and contains error-handling routines, which were originally in the specialized interpreter.

Figure 1 summarizes the overall scenario. Initially, a user program `P` is interpreted by a directly executed metalevel interpreter $\mathcal{I}_1$[1] (a). (The squares and ovals in the figure represent directly executed code and interpreted code, respectively.) If the user program executes an `exec-at-metalevel` construct, a metametalevel interpreter $\mathcal{I}_2$ is lazily created (b) and the body of `exec-at-metalevel` is executed at the metalevel, possibly modifying the metalevel interpreter $\mathtt{I}_1$ to obtain $\mathtt{I}'_1$ (c). As a result, the baselevel user program is inefficiently interpreted by two interpreters $\mathcal{I}_2$ and $\mathtt{I}'_1$. Using a partial evaluator, we obtain an efficient directly

[1]We use a calligraphic font (such as $\mathcal{I}$) to denote a directly executing program, and a typewriter font (such as `I`) to denote a program that is being interpreted by a metalevel interpreter, respectively.

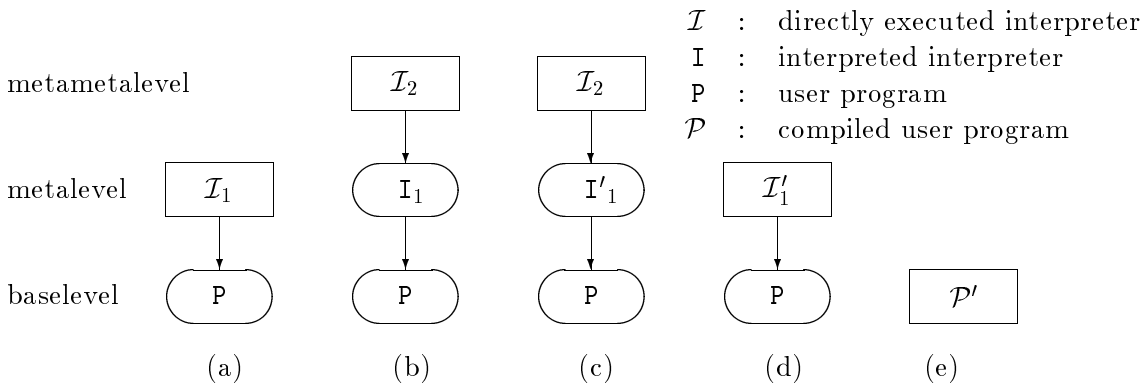

Figure 1: Overall Scenario

executed interpreter $\mathcal{I}'_1$, which is almost identical to the original directly executed interpreter $\mathcal{I}_1$, except that it correctly reflects the modification made by the user program (d).

We could conceive further use of the partial evaluator: we could obtain a compiled user program $\mathcal{P}'$ (e) by partially evaluating the modified interpreter $\mathcal{I}'_1$ with respect to the user program `P`. Currently, we do not directly do so, because the partial evaluator cannot handle `letrec` forms (as opposed to recursive `define` forms) which are output from the first partial evaluation. Instead, we obtain a compiled user program $\mathcal{P}'$ (e) directly from (c) by partially evaluating the metametalevel interpreter $\mathcal{I}_2$ with respect to the modified metalevel interpreter `I'`$_1$ *and* the baselevel user program `P`. The obtained program $\mathcal{P}'$ is a compiled program under the modified language semantics.

The partial evaluator we have designed is an online[15], polyvariant, higher order partial evaluator, which allows partially static values and I/O-type side-effects. We choose online instead of offline, because of its simplicity and power. Symbolic values and graph representations[18] are used to solve the code duplication/elimination problem. I/O-type side-effects are supported by attaching *preactions*[14] to symbolic values. We use filters[5] (user annotations) to make unfold/residualize decisions, rather than employing automatic termination mechanisms, since it is quite difficult to devise a sufficiently strong termination detection mechanism and such mechanisms does not seem to fit very well in the online setting.

This paper is organized as follows: the next section presents the input language for our partial evaluator. The partial evaluator itself is described in Section 3. Section 4 illustrates an example of compiling a modified interpreter using the partial evaluator. Section 5 discusses some related work, and Section 6 concludes the paper.

2 The Language to be Partially Evaluated

The language our partial evaluator can handle is a subset of Scheme. Figure 2 shows the syntax. Programs are a set of `define` forms followed by an expression. An expression is either a constant, a variable, a lambda closure, a conditional, a function application, a sequence, or a `known?` special form, which will be explained below. A filter is attached to each lambda closure to indicate when to unfold/residualize. If the body of a filter evaluates to `'unfold`, the lambda closure is unfolded, whereas if it evaluates to a list of boolean values, the closure is residualized. Each boolean value exactly corresponds to each argument to the closure, and indicate whether or not the value of the argument is propagated on specialization.

⟨program⟩	::=	`(define (`⟨var⟩ ...`)` ⟨filter⟩ ⟨exp⟩)* ⟨`exp`⟩	
⟨filter⟩	::=	`(filter` ⟨exp⟩`)`	
⟨exp⟩	::=	const	constants
	|	⟨var⟩	variables
	|	`(lambda (`⟨var⟩ ...`)` ⟨filter⟩ ⟨exp⟩`)`	lambda closures
	|	`(if` ⟨exp⟩ ⟨exp⟩ ⟨exp⟩`)`	conditionals
	|	`(begin` ⟨exp⟩ ...`)`	sequences
	|	`(`⟨exp⟩ ...`)`	applications
	|	`(known?` ⟨var⟩`)`	`known?` special forms

Figure 2: Syntax of the Input Language (abridged)

`Known?` special forms are used exclusively in filters and evaluates to *true* if its argument is a known value at partial evaluation time. The following `power` program shows the typical use of filters:

```
(define (power1 m n acc)
  (filter (if (known? n) 'unfold '(#t #f #f)))
  (if (= n 0)
      acc
      (power1 m (- n 1) (* m acc))))
(define (power m n)
  (filter 'unfold)
  (power1 m n 1))
```

Since `power` is not recursive, it can always be safely unfolded. As for `power1`, if the value of `n` is known at partial evaluation time, it can be completely unfolded. Thus, partially evaluating `(power m 3)` will yield `(* m (* m (* m 1)))`. When the value of `n` is unknown, however, `power1` needs to be residualized to avoid non-termination. In such a case, the value of `m` can be safely propagated to obtain more efficient residualized code: e.g., specializing `(power 3 n)` will yield

```
(letrec ((power1 (lambda (n acc)
                   (if (= n 0)
                       acc
                       (power1 (- n 1) (* 3 acc))))))
   (power1 n 1))
```

in which the value of `m` is inlined in the residualized code. The value of `acc`, however, cannot be propagated, because it may lead to infinite residualizations; that is, if the value of `m` is known, the partial evaluator will create infinite versions of residualized code for `power1`, each corresponding to the value 1, m, m^2, m^3, ... for `acc`.

3 The Partial Evaluator

An online partial evaluator can be regarded as an interpreter that can handle unknown values. For known input values, it behaves just the same way as an interpreter. When it encounters unknown values, it creates code which computes its result at runtime. To treat both values and code uniformly, we use *symbolic values*[18].

	Symbolic Value	Dumped Form
Known Value	*const*(*value*)	*value*
	pair(*a*, *b*)	`(cons` *a b*`)`
	closure(*params*, *body*, *env*)	`(lambda` *params body*`)`
Unknown Value	*unknown*(*var*)	*var*
	if(*p*, *t*, *e*)	`(if` *p t e*`)`
	application(*f*, *a*, ...)	(*f a* ...)
	letrec(*t*, *body*, *a*, ...)	`(letrec ((`*t* `(lambda (`*a* ...)*body*`)))` `(t` *a* `...))`

Figure 3: Symbolic Values (without Preactions) and their Dumped Form

3.1 Symbolic Values

The symbolic values we employ consist of values or pieces of code, together with an ordered sequence of *preactions*[14]. Value are used in normal interpretation, while pieces of code are used for constructing residual code. Preactions are used for pinning down possibly dangling interactions of I/O-type side-effects. Intuitively, we treat the value of `(begin (write e1) (newline) e2)` as `e2` with preactions $\langle\langle$`(write e1)`, `(newline)`$\rangle\rangle$. We denote this by: $\langle\langle$`(write e1)`,`(newline)`$\rangle\rangle$`e2`.

Figure 3 lists all the symbolic values (without preactions). *Const*, *pair*, and *closure* are known values, which represent a constant, a pair, and a lambda closure, respectively. The other four values are unknown values, which become pieces of code in the output. The *unknown* represents an unknown variable. *Letrec* appears only as the result of residualization. The figure also shows how these symbolic values are dumped into code. Note that since parts of symbolic values may be shared, `let` forms are inserted into the dumped code if necessary to express such sharing.

3.2 The Partial Evaluator with Preactions

Figure 4 shows the core of our partial evaluator $\mathcal{PE}$. It takes an expression, an environment, and a cache which holds specializations that have been previously constructed, and returns a symbolic value. When the expression is either a constant, a variable, or a lambda closure, $\mathcal{PE}$ returns the constant itself, a value stored in the environment, or a lambda closure, respectively, coupled with an empty preaction $\langle\langle\rangle\rangle$. If the expression is a conditional, it first executes its predicate-part to determine its subsequent behavior: if the value is *true* or *false* (with possible preactions, written as P), it evaluates the then-part or the else-part, respectively, and returns the result with the preaction of the predicate-part. In a case where the predicate value is unknown, $\mathcal{PE}$ constructs *if* code that evaluates *if* at runtime. Observe how the preaction of the predicate-part propagates out of the `if` expression. This enables $\mathcal{PE}$ to reduce, for example, `(if (begin (write e1) #t) e2 e3)` into `(begin (write e1) e2)`.

When the expression is a `begin` form, $\mathcal{PE}$ evaluates its body and returns the last value with all the previous values as preactions. Here, concatenation of two preactions is defined in an obvious way to preserve the order of I/O-type side-effects:

$$\langle\langle a_1, \ldots, a_n \rangle\rangle \cdot \langle\langle b_1, \ldots, b_m \rangle\rangle = \langle\langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle\rangle.$$

Finally, function applications are performed by first gathering all the preactions of the function and the arguments, and then calling the auxiliary function $\mathcal{A}$.

$$
\begin{array}{rcl}
\mathcal{PE} & : & Exp \to Env \to Cache \to Sval \\
\rho \in \quad Env & : & Var \to Sval \\
c \in \quad Cache & : & Var \to (Sval \times Sval^*) \\
\mathcal{A} & : & (Sval \times Sval^* \times Cache) \to Sval
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{PE}[\![con]\!]\rho c & = & {}^{\langle\langle\rangle\rangle} const(con) \\
\mathcal{PE}[\![var]\!]\rho c & = & {}^{\langle\langle\rangle\rangle} \rho(var) \\
\mathcal{PE}[\![\texttt{(lambda } params\ body\texttt{)}]\!]\rho c & = & {}^{\langle\langle\rangle\rangle} closure(params, body, \rho) \\
\mathcal{PE}[\![\texttt{(if } p\ t\ e\texttt{)}]\!]\rho c & = & \text{case } \mathcal{PE}[\![p]\!]\rho c \text{ of} \\
 & & \quad {}^{P} true \ : \ {}^{P}\mathcal{PE}[\![t]\!]\rho c \\
 & & \quad {}^{P} false \ : \ {}^{P}\mathcal{PE}[\![e]\!]\rho c \\
 & & \quad {}^{P} p \ : \ {}^{P} if(p, \mathcal{PE}[\![t]\!]\rho c, \mathcal{PE}[\![e]\!]\rho c) \\
\mathcal{PE}[\![\texttt{(begin } a_1 \ldots a_n\texttt{)}]\!]\rho c & = & \text{let} \quad {}^{A_1} a_1' = \mathcal{PE}[\![a_1]\!]\rho c \\
 & & \qquad \ldots \\
 & & \qquad {}^{A_n} a_n' = \mathcal{PE}[\![a_n]\!]\rho c \\
 & & \text{in} \quad {}^{A_1 \cdot \langle\langle a_1' \rangle\rangle \cdots A_{n-1} \cdot \langle\langle a_{n-1}' \rangle\rangle \cdot A_n} a_n' \\
\mathcal{PE}[\![\texttt{(}f\ a_1 \ldots a_n\texttt{)}]\!]\rho c & = & \text{let} \quad {}^{F} f' = \mathcal{PE}[\![f]\!]\rho c \\
 & & \qquad {}^{A_1} a_1' = \mathcal{PE}[\![a_1]\!]\rho c \\
 & & \qquad \ldots \\
 & & \qquad {}^{A_n} a_n' = \mathcal{PE}[\![a_n]\!]\rho c \\
 & & \text{in} \quad {}^{F \cdot A_1 \cdots A_n} \mathcal{A}(f', a_1', \ldots, a_n', c)
\end{array}
$$

Figure 4: The Core of the Partial Evaluator

Notice that the rules in the figure preserve the order of execution. In each of these rules, the returned code will be executed in the same order as the original expression. This is a crucial property in the presence of I/O-type side-effects, because without this, the resulting output could be in a wrong order.

3.3 Function Application Rules with Preactions

The function $\mathcal{A}$, shown in Figure 5, receives an operator, arguments, a cache, and returns a symbolic value that represents the result of the application. Partial evaluation of expressions other than application results in tree-like construction of symbolic values; i.e., there are no sharings of substructures neither between different symbolic values, nor within a symbolic value. On the other hand, partial evaluation of function application causes sharing of symbolic values to occur, where preactions play an important role.

Because application of primitives or lambda closures might not use all their arguments, a naive implementation of $\mathcal{A}$ could cause code elimination. Two possible cases are: (1) when an operator is a data constructor (in our case, *cons*), since no selector functions might be applied to the data, and (2) when it is a lambda closure, since formal parameters might never be used in the body of the closure. In such cases, we place their arguments into the preaction of the result, so that they will not be inadvertently eliminated.

When *cons* is applied to symbolic values a_1 and a_2, $\mathcal{A}$ returns a pair consisting of a_1 and a_2, together with a preaction $\langle\langle a_1, a_2 \rangle\rangle$. In the returned value, the occurrences of a_1 in the preaction and within the pair constructor are shared; this holds as well for a_2. (See Figure 6.)

$$\mathcal{A}(cons, a_1, a_2, c) = {}^{\langle\langle a_1, a_2 \rangle\rangle} pair(a_1,\ a_2)$$

$$\mathcal{A}(f = closure(x_1, \ldots, x_n, body, \rho), a_1, \ldots, a_n, c) = \begin{cases} {}^{\langle\langle a_1, \ldots, a_n \rangle\rangle} \mathcal{PE}[\![body]\!]\rho[a_1/x_1, \ldots, a_n/x_n]c \\ \qquad\qquad \text{if filter evaluates to } \texttt{'unfold} \\ {}^{\langle\langle\rangle\rangle} application(t', a_1, \ldots, a_n) \\ \qquad\qquad \text{if cache hits : } c(t') = (f, a'_1, \ldots, a'_n) \\ {}^{\langle\langle a_1, \ldots, a_n \rangle\rangle} letrec(t, \mathcal{PE}[\![body]\!]\rho[a'_1/x_1, \ldots, a'_n/x_n]c', a_1, \ldots, a_n) \\ \qquad\qquad \text{otherwise} \end{cases}$$

$$\text{where} \quad c' = c[(f, a'_1, \ldots, a'_n)/t]$$

$$a'_i = \begin{cases} a_i & \text{if to be propagated} \\ unknown(x_i) & \text{otherwise} \end{cases}$$

$$t = \text{a fresh variable}$$

$$\mathcal{A}(prim, a_1, \ldots, a_n, c) = \begin{cases} {}^{\langle\langle\rangle\rangle} prim(a_1, \ldots, a_n) & \text{if all known} \\ {}^{\langle\langle\rangle\rangle} application(prim, a_1, \ldots, a_n) & \text{otherwise} \end{cases}$$

Figure 5: Function $\mathcal{A}$

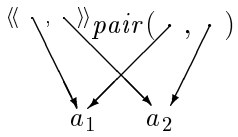

Figure 6: Symbolic Value Returned by Applying *cons*

For example, suppose we have the following expression:

```
(cdr (cons (if p (write 1) (write 2)) 3))
```

where `p` is an unknown variable. To specialize this expression, $\mathcal{PE}$ first constructs a symbolic value: ${}^{\langle\langle if(p,(write\ 1),(write\ 2)),3\rangle\rangle} pair(if(p,\ (write\ 1),\ (write\ 2)),\ 3)$. Since both arguments are put into the preaction, they correctly propagate out of the application of *cdr*. The result would be ${}^{\langle\langle if(p,(write\ 1),(write\ 2)),3\rangle\rangle} 3$ or, in textual form:

```
(begin (if p (write 1) (write 2)) 3 3).
```

(The redundant 3 in the result is eliminated by simple post-processing in our partial evaluator.)

Note that if the arguments were not placed in the preaction, the result would become incorrect. Without preactions, $\mathcal{PE}$ will extract 3 out of $pair(...)$ discarding $if(...)$. The result will display nothing, whereas the original program will display either 1 or 2 according to the dynamic value of `p`.

It might initially seem that this technique would have the drawback of code being duplicated, if both of the arguments are eventually used in the original program. Fortunately, this is not the case, since they are *shared*; instead of being duplicated, they are collected to form a `let` form by finding the least upper bound of shared values[18]. Thus, specializing

```
(car (cons (if p (write 1) (write 2)) 3))
```

will yield

```
(let ((t1 (if p (write 1) (write 2)))) (begin t1 t1))
```

which is reduced to `(if p (write 1) (write 2))` via post-processing.

The situation is exactly the same for applying lambda closures, because formal parameters may never be used in the body of a closure. We put the arguments of a lambda closure into its preaction as well, so that they will not be eliminated.

For a lambda closure, we also need to decide if it should be unfolded or residualized. This is determined by evaluating filters[5] . Filters allow us to avoid incorporating complex termination detection algorithm, maintaining the simplicity of online partial evaluation. If the filter evaluates to `'unfold`, the lambda closure is unfolded: the body of the lambda closure is partially evaluated in the extended environment. On the other hand, when the filter evaluates to a list of boolean values, the lambda closure is residualized: a cache is first looked-up to determine whether the same lambda closure has already been residualized with the same set of arguments. If so, a piece of code to call this specialization is returned. When the cache misses, residualized *letrec* code is constructed, with the body of the lambda closure being evaluated under the extended environment plus the cache that contains the current residualization. Here, the environment is extended only for those arguments whose value the filter indicates that they should be propagated—the arguments for which the filter indicates not to propagate are discarded, and become completely unknown in the extended environment.

Finally, application of primitive functions (other than *cons*) is done by actually applying the primitive to its arguments if both the function and the arguments are all known, and otherwise constructing a pieces of code for application. Specifically, application of unknown functions always returns application code.

3.4 Post-Processing

Here, we briefly mention the post-processing, performed by our partial evaluator. After an output symbolic value is obtained, the partial evaluator first specializes residualized lambda closures (which are left unspecialized) on completely unknown arguments. Then, the resulting symbolic value is dumped into code, preserving sharings using `let` forms. Finally, the obtained code is beautified by repeatedly applying post-processing rules, such as:

- `(begin` *const exp* ...`)` $\rightarrow$ `(begin` *exp* ...`)`
- `(begin` *var exp* ...`)` $\rightarrow$ `(begin` *exp* ...`)`
- `(begin` *exp*`)` $\rightarrow$ *exp*
- If a variable bound by a `let` form is used only once, inline.

4 Specializing an Interpreter on a Modified Interpreter

In this section, we experiment the power of our online partial evaluator through compiling a modified interpreter. Let the interpreter to be partially evaluated be $\mathcal{I}_2$ and the modified interpreter $\mathtt{I'}_1$. We will specialize $\mathcal{I}_2(\mathtt{I'}_1)$ to obtain $\mathcal{I}'_1$, which is a directly executable version of the modified interpreter.

4.1 Interpreter to be Specialized

Figure 7 shows the overview of $\mathcal{I}_2$. It is a standard continuation-passing style metacircular interpreter. Filters are attached to interpreter functions `base-eval`[2], `eval-begin`, `eval-list`, `eval-lambda`, and some other environment manipulation functions, which are not shown here[3]. The filters usually have a common form, stating that if the expression is known, then unfold, otherwise residualize without propagating any of the values of arguments. Interpreter functions are thereby unfolded only when the expression to be interpreted is known at partial evaluation time.

The filter in `eval-lambda` is somewhat different from others: this filter avoids infinite unfolding in the case where the *interpreted* program contains recursion. Even though the interpreter is unfolded only when expressions are known, the partial evaluation process may still unfold infinitely. For example, suppose we are specializing $\mathcal{I}_2$ with respect to a factorial program, i.e., $\mathcal{PE}(\mathcal{I}_2(\texttt{(fac n)}))$ for some unknown `n`. Because the expression is known to be `(fac n)`, the partial evaluator unfolds it and evaluates the body of `fac`. Eventually, it encounters `fac` again in the body of `fac`. Because the expression is still known to be `fac`, however, it is unfolded again, leading to infinite unfolding.

Since we employed the filter approach rather than automatic detection approach for termination, we again take the same approach here to avoid such non-termination: we attach filters to interpreted programs themselves. For example, given a recursive definition of `fac` in interpreted programs:

```
(define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
```

we attach a filter in the following way:

```
(define (fac n)
  (filter (if (known? n) 'unfold '(#f)))
  (if (= n 0) 1 (* n (fac (- n 1))))).
```

(Note that $\mathcal{I}_2$ in Figure 7 supports the `known?` special form.)

`Eval-lambda` uses this filter to avoid non-termination. It returns a metalevel lambda closure, which corresponds to a closure in a baselevel interpreted program. When it is applied to arguments during partial evaluation, the baselevel filter is used to form a filter of the metalevel closure; i.e., the filter in the body of the baselevel closure is extracted and evaluated to make the unfold/residualize decision of the metalevel closure.

4.2 Modified Interpreter

Figure 8 shows the modified interpreter $\texttt{I}'_1$, on which $\mathcal{I}_2$ is specialized. Since interpreted programs now need filters, $\texttt{I}'_1$ is very similar to $\mathcal{I}_2$, including filter descriptions. The last `let` expression of $\texttt{I}'_1$ specifies user modification: it is essentially equivalent to the trace example presented in Section 1.

The `set!` statement in $\texttt{I}'_1$ is handled by `eval-set!` in $\mathcal{I}_2$, in which `set!` is realized by an assignment to a global variable. This does not cause a problem, because the program we are treating contains only unconditional toplevel `set!`. (For more elaborate treatment of assignments, see our forthcoming paper[3].)

[2] We use the name `base-eval` rather than `eval` to distinguish it from the Scheme `eval` function.

[3] The code for interpreting `exec-at-metalevel` is not shown here, either. Because `exec-at-metalevel` involves metalevel interactions, we have to treat it separately[2].

```
(define (base-eval exp env cont)
  (filter (if (known? exp) 'unfold '(#f #f #f)))
  (cond ((number? exp)           (cont exp))
        ((symbol? exp)           (eval-var exp env cont))
        ((eq? (car exp) 'quote)  (eval-quote exp env cont))
        ((eq? (car exp) 'if)     (eval-if exp env cont))
        ((eq? (car exp) 'define) (eval-define exp env cont))
        ((eq? (car exp) 'set!)   (eval-set! exp env cont))
        ((eq? (car exp) 'lambda) (eval-lambda exp env cont))
        ((eq? (car exp) 'begin)  (eval-begin (cdr exp) env cont))
        ((eq? (car exp) 'known?) (eval-known? (car (cdr exp)) env cont))
        (else
         (eval-list exp env
                    (lambda (l) (base-apply (car l) (cdr l) env cont))))))
(define (eval-var exp env cont) ...)
(define (eval-quote exp env cont) (cont (car (cdr exp))))
(define (eval-if exp env cont) ...)
(define (eval-define exp env cont) ...)
(define (eval-set! exp env cont) ...)
(define (eval-lambda exp env cont)
  (let ((lambda-body (cdr (cdr exp)))
        (lambda-params (car (cdr exp))))
     (cont (lambda operand
             (filter (base-eval (car (cdr (car lambda-body)))
                                (extend env lambda-params operand)
                                (lambda (directive) directive)))
             (eval-begin (cdr lambda-body)
                         (extend env lambda-params operand)
                         (lambda (x) x))))))
(define (eval-begin body env cont)
  (filter (if (known? body) 'unfold '(#f #f #f)))
  (cond ((null? body) '())
        ((null? (cdr body))
         (base-eval (car body) env cont))
        (else
         (base-eval (car body) env
                    (lambda (x) (eval-begin (cdr body) env cont))))))
(define (eval-known? exp env cont) ...)
(define (eval-list exp env cont)
  (filter (if (known? exp) 'unfold '(#f #f #f)))
  (if (null? exp)
      (cont '())
      (base-eval (car exp) env
                 (lambda (val1)
                    (eval-list (cdr exp) env
                               (lambda (val2)
                                  (cont (cons val1 val2)))))))))
(define (base-apply operator operand env cont)
  (if (procedure? operator)
      (cont (scheme-apply operator operand)))
      (error (list 'Not 'a 'function: operator)))
```

Figure 7: The Interpreter to be Partially Evaluated $\mathcal{I}_2$

```
(define (base-eval exp env cont) ...)
...
(define (base-apply operator operand env cont) ...)
```

(Up to here is identical to $\mathcal{I}_2$ (Figure 7))

```
(let ((old-eval base-eval))                ; modify base-eval to print traces
  (set! base-eval (lambda (exp env cont)
                    (write exp)
                    (old-eval exp env cont))))
base-eval                                  ; returns the modified interpreter
```

Figure 8: The Modified Interpreter $\mathtt{I}'_1$

4.3 The Result

Figure 9 shows the actual result of partial evaluation using our partial evaluator. (We have renamed some bound variables for better readability.) The output value is a lambda closure, which, when applied, prints its first argument, and then calls `base-eval` defined in `letrec`. `Base-eval` has almost the same structure as the original `base-eval`, but it precedes every call to `base-eval` with `write` expressions, correctly reflecting the user modification. In fact, according to our preliminary experiment, it runs more than 30 times faster than the 'double' interpretation. We could even *compile* the modified interpreter into directly executable code under the underlying Scheme compiler.

Another point to notice is that error-handling code is inlined into the residualized code. If the continuation argument is not a function, it reports an error[4]. These error-handling routines were originally in $\mathcal{I}_2$, but were inherited to the resulting modified interpreter via the partial evaluation.

To proceed further, we now specialize $\mathcal{I}_2$ with respect to $\mathtt{I}'_1$ *and* a baselevel user program: i.e., we specialize $\mathcal{I}_2(\mathtt{I}'_1(\mathtt{P}))$. We replace the last `base-eval` in Figure 8 with

```
(base-eval '(+ 3 4) init-env (lambda (x) x)).
```

That is, instead of returning the modified `base-eval`, we use it directly to evaluate `'(+ 3 4)`.

The result of this partial evaluation is as expected:

```
(begin (write (list '+ 3 4)) (write '+) (write 3) (write 4) 7).
```

The behavior is identical to the execution under the 'doubly interpreted' reflective Black in Section 1, but entirely specialized. That is to say, user program is successfully compiled under the modified language semantics.

5 Related Work

There seems to be only a few papers that propose the use of partial evaluation for efficient implementation of reflective languages. Danvy[6] pointed out the similarity between a reflec-

[4]Notice that the continuation can take non-function values, if the user erroneously modifies the metalevel interpreter.

```
(lambda (exp1 env1 cont1)
  (write exp1)                                                    ; write
  (letrec ((base-eval
; the definition of base-eval
(lambda (exp2 env2 cont2)
  (cond ((number? exp2)
         (if (procedure? cont2)                                   ; error handling
             (cont2 exp2)
             (list 'not 'a 'function: cont2)))
        ((symbol? exp2) ...)
        ((eq? (car exp2) 'quote) ...)
        ((eq? (car exp2) 'if)
         (let ((pred-part (car (cdr exp2))))
           (write pred-part)                                      ; write
           (base-eval pred-part env2
                      (lambda (pred)
                         (if pred
                             (let ((then-part (car (cdr (cdr exp2)))))
                                (write then-part)                 ; write
                                (base-eval then-part env2 cont2))
                             (let ((else-part (car (cdr (cdr (cdr exp2))))))
                                (write else-part)                 ; write
                                (base-eval else-part env2 cont2)))))))
        ...
        (else ; function application
         (letrec ((eval-list (lambda (exp3 env3 cont3)
                    (if (null? exp3)
                        (if (procedure? cont3)                    ; error handling
                            (cont3 '())
                            (list 'not 'a 'function: cont3))
                        (let ((t103 (car exp3)))
                           (write t103)                           ; write
                           (base-eval t103 env3 (lambda (val1)
                            (eval-list (cdr exp3) env3 (lambda (val2)
                             (if (procedure? cont3)               ; error handling
                                 (cont3 (cons val1 val2))
                                 (list 'not 'a 'function: cont3)))))))))))
           (eval-list exp2 env2
                      (lambda (l)
                         (let ((t109 (car l)))
                            (if (procedure? t109)
                                (if (procedure? cont2)            ; error handling
                                    (cont2 (apply t109 (cdr l)))
                                    (list 'not 'a 'function: cont2))
                                (list 'not 'a 'function: t109)))))))))))
; end of the definition of base-eval
  (base-eval exp1 env1 cont1)))
```

Figure 9: The Result of Specializing $\mathcal{I}_2(\mathtt{I'}_1)$

tive tower and the Futamura projection[10]. This is in contrast to our approach, in which we use a partial evaluator as a tool for efficiency in reflective languages.

The *partial compilation* technique[13] used in the implementation of the reflective concurrent object-oriented language ABCL/R2 takes a similar approach to ours using partial evaluators, but it is strictly limited. In ABCL/R2, the expression patterns that can be modified are statically predetermined to compile-out unmodifiable parts. As a result, some expressions are *always* interpreted even if the metalevel interpreter is not modified at all. Their approach, although faster than 'full' interpretation, proves to be slower than the direct (non-interpretive) execution by an order of magnitude.

Kiczales et al.[12] designed an 'open compiler' based on the Meta Object Protocol (MOP) approach. By allowing users to control the compilation process using MOP, they avoid using a general partial evaluator for compilation. They are advantageous in controlling language properties that are *static* at compile-time (e.g., to modify the run-time representation of a data structure), while our partial evaluation based compilation is suitable for modifying language properties that depends on *run-time* conditions. This is because the partial evaluator unfolds the user extensions that can be determined at compile-time, and residualize the others, while in the MOP approach, such residualization is not generally supported. We also provide the user with a concise and high-level view of the language implementation by exposing the entire interpreter, instead of 'spreading out' the implementation among multiple metaobjects.

The reflective language Refci[16] allows user programs to redefine a metalevel interpreter, which is divided into two pieces, called *prelim* and *dispatch*. By restricting user modifications to only these two parts, it achieves good performance. Through employing a partial evaluation technique, we obtain efficient interpreters without restricting possible modifications.

The partial evaluator we designed is based on Fuse[15]. It is an online partial evaluator with a sophisticated automatic termination detection mechanism. We did not employ the automatic termination detection approach, however, because it complicates the partial evaluator considerably and it is not clear whether recursions on both metalevel and baselevel programs can be automatically detected.

Similix[4] is an offline partial evaluator, which can treat both I/O-type side-effects and assignments to global variables. However, we did not use Similix as our compiler, because it seemed rather difficult to identify which parts of a program were static and which were dynamic to control the behavior of Similix. The online approach, with our simple preaction mechanism for I/O-type side-effects, gave us sufficient control, with which interpreters were compiled successfully.

6 Conclusion

We presented an online partial evaluator with a new mechanism to handle I/O-type side-effects using *preactions*, and reported our experiment of using the partial evaluator as a compiler for reflective languages. Although the approach taken here is quite general, the result shows that the partial evaluator is powerful enough to obtain compiled interpreters for reflective languages. Moreover, we have succeeded in compiling user programs under modified language semantics. We are now undergoing work to actually incorporate the partial evaluator as a compiler for both Black and ABCL/R2 systems.

Acknowledgements

We thank Erik Ruf for his various comments on partial evaluation. The first author thanks Charles Consel, Olivier Danvy, Peter Lee, and Catherine Copetas for the most enjoyable summer school they held in CMU, July 1994.

References

[1] Abelson, H., and G. J. Sussman with J. Sussman *Structure and Interpretation of Computer Programs*, Cambridge: MIT Press (1985).

[2] Asai, K., S. Matsuoka, and A. Yonezawa "Duplication and Partial Evaluation — For a Better Understanding of Reflective Languages —," *Lisp and Symbolic Computation* (to appear).

[3] Asai, K., H. Masuhara, and A. Yonezawa "Toward Partial Evaluation of Functional Programs with Side-effects," Submitted to the Dagstuhl Seminar 9607 for Partial Evaluation.

[4] Bondorf, A. *Similix 5.0 Manual*, DIKU, University of Copenhagen, (May 1993).

[5] Consel, C., "New Insights into Partial Evaluation: the SCHISM Experiment," *Lecture Notes in Computer Science* 300, pp. 236–246 (1988).

[6] Danvy, O. "Across the bridge between reflection and partial evaluation," In D. Djørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pp. 83–116, North-Holland (1988).

[7] Danvy, O., and K. Malmkjær "Intensions and Extensions in a Reflective Tower," *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pp. 327–341 (July 1988).

[8] des Rivières, J., and B. C. Smith "The Implementation of Procedurally Reflective Languages," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 331–347 (August 1984).

[9] Friedman, D. P., and M. Wand "Reification: Reflection without Metaphysics," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 348–355 (August 1984).

[10] Futamura, Y. "Partial evaluation of computation process – an approach to a compiler-compiler," *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, (1971).

[11] Jefferson, S., and D. P. Friedman "A Simple Reflective Interpreter," *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pp. 24–35 (November 1992).

[12] Kiczales, G., J. Lamping, and A. Mendhekar "What A Metaobject Protocol Based Compiler Can Do For Lisp," Xerox PARC technical report (January 1994).

[13] Masuhara, H., S. Matsuoka, T. Watanabe, and A. Yonezawa "Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently," *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, pp. 127–145, Vancouver, (October 1992).

[14] Masuhara, H. et al. "A simple mechanism to handle I/O-type side-effects in on-line partial evaluators," in preparation.

[15] Ruf, E. *Topics in Online Partial Evaluation*, Ph.D. thesis, Stanford University (March 1993). Also published as Stanford Computer Systems Laboratory technical report CSL-TR-93-563.

[16] Simmons II, J. W., S. Jefferson, and D. P. Friedman "Language Extension Via First-class Interpreters," Indiana University Technical Report No. 362, (September 1992).

[17] Wand, M., and D. P. Friedman "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower," *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, pp. 298–307 (August 1986).

[18] Weise, D., R. Conybeare, E. Ruf, and S. Seligman "Automatic Online Partial Evaluation," In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pp. 165–191, Cambridge, (August 1991).