Partial Evaluation of Call-by-value λ-calculus with Side-effects

Kenichi Asai Hidehiko Masuhara[*] Akinori Yonezawa

Department of Information Science, Faculty of Science,
University of Tokyo
7-3-1 Hongo Bunkyo-ku 113 Japan
{asai, masuhara, yonezawa}@is.s.u-tokyo.ac.jp

Abstract

We present a framework of an online partial evaluator for a call-by-value λ-calculus with destructive updates of data structures. It properly and correctly specializes expressions that contain side-effects, while preserving pointer equality, which is an important property for programs using updates. Our partial evaluator uses a side-effect analysis to extract immutable data structures and then performs an online specialization using preactions. Once mutable and immutable data structures are separated, partial evaluation is done in such a way that accesses to immutable ones are performed at specialization time, while accesses to mutable ones are residualized. For the correct residualization of side-effecting operations, preactions are used to solve various issues, including code elimination, code duplication, and execution order preservation. The preaction mechanism also enables us to reduce expressions that were residualized when the conventional let-expression approach of Similix was used. The resulting partial evaluator is simple enough to prove its correctness. Based on the framework, we have constructed a partial evaluator for Scheme, which is powerful enough to specialize fairly complicated programs with side-effects, such as an interpreter.

1 Introduction

Partial evaluation[13] is a program transformation technique which, given a program and parts of its arguments, produces a *specialized* program with respect to those known arguments. Since computation that depends only on the known arguments (and constants) is performed at partial evaluation time, the resulting program usually runs faster at runtime than the original one. Partial evaluation is widely used not only as a general optimization tool[10] but also as various generators such as compiler generators and parser generators[22].

Although much work has been done on partial evaluation of functional languages, partial evaluation of side-effecting constructs remains difficult. In conventional partial evaluators for functional languages, only limited kinds of side-effects are allowed; other types are either merely refused or incorrectly processed. For example, Similix[5] handles only I/O operations and limited types of variable assignments. POPE[19] accepts richer kinds of variable assignments, but does not handle destructive updates of data structures correctly. This is a severe restriction since we often use side-effects even in mostly functional programs.

In this paper, we present an online partial evaluator for a call-by-value λ-calculus with destructive updates of data structures. It has the following novel characteristics:

- It correctly deals with full side-effects (including destructive updates of data structures).
- Pointer equality ('eq-ness') is preserved for all data structures.

These features are realized by a side-effect analysis and a systematic use of *preactions*[14, 15]. The preaction mechanism also enables us to reduce more expressions than the conventional let-expression approach used in Similix. Based on this framework, we have created a partial evaluator for Scheme[6], which accepts almost all features of Scheme.

Why side-effects are difficult to handle? The difficulties for partially evaluating programs with side-effects mainly come from mutable data structures (and mutable variables). Because partial evaluators perform non-standard eager evaluation, we need knowledge on the runtime value of data structures. In pure functional languages, the property of referential transparency assures that any data structure created at partial evaluation time has the same value throughout the specialization (and at runtime). With side-effects, this assumption does not hold any more, because assignment operations in residual code might modify the value of data structures.

The most conservative approach to handle mutable data structures is to treat all data structures as mutable and residualize them. This is unacceptable in practice, however, since the actually updated parts are usually small and the results become too conservative. Another approach would be to find mutable data structures during specialization. However, this does not work well either in the presence of first-class pointers. For example, consider partial evaluation of expressions that contain: `(set-car! x y)` where `x` is unknown. Without knowing the possible values for `x`, we have to treat all the data structures as unknown after this assignment, resulting in poor specialization.

Our approach is to identify mutable data structures (possible values for `x` in the above example) using a side-effect analysis. It extracts functional parts from programs with side-effects. Once mutable and immutable data structures are separated, partial evaluation is done in such a way that accesses to immutable ones are performed at specialization time, while accesses to mutable ones are residualized.

The separation is effective in practice, because most functional programs modify only restricted parts of data structures, and thus the separated mutable parts are usually small. Most parts are classified

---

[*] Department of Graphics and Computer Science, College of Arts and Sciences, University of Tokyo

```
(lambda (x y z)
  (letrec ((inc (lambda (node visit)               ; returns visited nodes
                  (cond ((null? node) visit)
                        ((memq node visit) visit)          ; equality test
                        (else
                         (set-car! node (+ (car node) 1))         ; update
                         (inc (car (cdr node))
                              (inc (cdr (cdr node))
                                   (cons node visit))))))))
    (let* ((make-node (lambda (value left right)
                        (cons value (cons left right))))
           (node1 (make-node x '() '()))
           (node2 (make-node y node1 '()))
           (node3 (make-node z node1 node2)))       ; node1 is shared !!!
      (inc node3 '())
      node3)))
```

Figure 1: An Example Program

as immutable and we can achieve sufficient specialization. For example, consider an environment of interpreters for imperative languages, which is realized by an association list and destructive updates on it. The mutable parts are usually only the value slots of the environment and all other parts (including the overall structure of the environment) are immutable. When specializing this interpreter, we can unfold all the environment accesses, removing interpretation overhead completely.

Another difficulty for handling side-effects exists in residualization. We cannot eliminate, duplicate, or reverse code, since it might eliminate, duplicate, or reverse side-effects. These problems are avoided by the use of *preactions*[14, 15]. Without code elimination and duplication, our partial evaluator naturally preserves pointer equality of each value. This is an important property which has been ignored in the conventional partial evaluators for pure functional languages.

The preaction mechanism also enables us to treat data structures as both dynamic and static, i.e., we can use the value of data structures while residualizing them at the same time. This is a crucial property in reducing data structures whose identity has to be preserved. In conventional let-expression approach used in Similix, such data structures were made dynamic and put into let-expressions. However, this is an overly general approach for languages with pointer equality (such as Scheme), since *all* data structures have their identity and thus have to be residualized as dynamic expressions. The preaction mechanism enables us to residualize them without making them dynamic.

Example To see how our partial evaluator handles side-effects, consider a Scheme program[1] shown in Figure 1. Given a DAG (directly acyclic graph) made by `make-node`, it increments the number associated to each node by one using a destructive update operation `set-car!`. To avoid multiple increment on shared nodes, visited nodes are recorded in the argument `visit`, and checked via the predicate `eq?`[2]. To partially evaluate this program, we first perform the side-effect analysis. It will detect that the argument `node` of `inc` can be bound to cons cells constructed at the underlined `cons` statement in `make-node`. Because the *car* part of `node` is destructively updated in `inc`, the *car* parts of the cells that are constructed at this underlined `cons` statement are mutable. The partial evaluator will residualize all accesses to these parts preserving the order of execution, and process other parts of the program. Because our partial evaluator reduces partially static data structures[17, 18], the *cdr* part of `node` is still accessible. After unfolding the iteration in `inc` (with a definition of `memq`), the result will look like this:

```
(lambda (x y z)
  (let* ((node1 (cons x (cons '() '())))
         (node2 (cons y (cons node1 '())))
         (node3 (cons z (cons node1 node2))))
    (set-car! node3 (+ (car node3) 1))
    (set-car! node2 (+ (car node2) 1))
    (set-car! node1 (+ (car node1) 1))
    node3))
```

All the references and updates to the mutable parts are residualized in the correct order, and other parts are completely unfolded. The names `node1`, `node2`, and `node3` are required to hold the identity (for pointer equality) of the three mutable cells. Notice here that even though these mutable cells declare their identity and are residualized in let-expressions, their static elements are reduced at specialization time. We can observe that `inc` is unfolded for the *cdr* part of `node3`, although `node3` itself is residualized in a let-expression. This distinguishes itself from the let-expression of Similix, where once an expression is residualized in a let-expression, it becomes completely dynamic. The use of preaction enables us to reduce such dynamic data structure that has static elements in it.

Outline of the Paper Towards a partial evaluator for Scheme, we first define a simple core language (Section 2) to concentrate on the treatment of side-effects. After presenting the side-effect analysis in Section 3, the partial evaluator for the core language is presented in Section 4, whose correctness is discussed in Section 5. We then describe extensions of the core language to more realistic languages in Section 6. Using this partial evaluator, we have succeeded to partially evaluate an interpreter written with side-effects with respect to a side-effecting program (Section 7). Section 8 discusses related work and the paper concludes in Section 9.

---

[1] In this paper, we handle partial evaluation of closed terms only. When we want to specialize $f(x, y)$ with respect to $x = a$, we specialize $\lambda y.f(a, y)$ instead.

[2] `Eq?` is used in `memq`, whose definition is omitted here.

2 The Core Language

The core language we use is a call-by-value λ-calculus with mutable cons cells. The syntax is given as follows:

$$
\begin{aligned}
M \quad = \quad & x \mid \eta : \lambda x.M \mid M N \mid \epsilon : cons M N \mid car M \mid cdr M \\
& \mid setcar M N \mid setcdr M N \mid eq M M' N N'
\end{aligned}
$$

An expression is either a variable, a λ-expression, an application, or one of six primitive expressions: *Cons* constructs a pair, whose elements are accessed by *car* and *cdr*. *Setcar* and *setcdr* destructively update an element of a pair. *Eq* is used to test the 'eq-ness' of values. It first evaluates its first two arguments to see if they are allocated in the same location. If they are, *eq* evaluates the third argument, discarding the fourth. Otherwise, it evaluates the fourth, discarding the third.

Each λ-expression and *cons* expression is attached with a unique label η and ϵ called a *closure point* and a *cons point*[8], respectively. These are used in the side-effect analysis in Section 3.

We assume that all bound variables are distinct. We write x^{η} when we explicitly show that the variable x is bound by a λ-expression labeled with η.

3 Side-effect Analysis

The goal of this section is to obtain two sets: *mutable*$_{car}$ and *mutable*$_{cdr}$, which contain cons points for mutable cells. In the partial evaluator presented in the next section, they are used to residualize accesses to mutable cells. If a cons point ϵ is in *mutable*$_{car}$, it means that the *car* part of the cons cells constructed by $\epsilon : cons$ might be modified by *setcar*. For the program in Figure 1, *mutable*$_{car}$ contains a cons point for the underlined `cons` expression and *mutable*$_{cdr}$ is empty. Since it is difficult to obtain the smallest *mutable*$_{car}$ and *mutable*$_{cdr}$, we obtain their safe approximation using an abstract-interpretation based approach similar to the Consel's[8].

The analysis consists of two stages. At the first stage, we collect all possible values (cons points and closure points) for each subexpression of a given input program. This is the central part of the analysis, which is achieved by the fixed-point iteration of abstract interpretation. It is a higher-order, interprocedural, context-insensitive, 0CFA[21] analysis.

Given the possible values for each subexpression, *mutable*$_{car}$ and *mutable*$_{cdr}$ are obtained fairly easily at the second stage. For each occurrence of $setcar M N$ in the program, we add the possible cons points for M to *mutable*$_{car}$. *Mutable*$_{cdr}$ is obtained similarly.

3.1 Notation

Abstract Value To identify mutable cells in higher-order programs, we abstract a runtime value to a set of cons points and closure points during abstract interpretation, ignoring other values, such as numbers and symbols. An abstract value A_v has a form $\langle \tau, \varphi \rangle$, where τ is a set of cons points and φ is a set of closure points. It represents the possible values for expressions. When an expression M has an abstract value $\langle \{\epsilon_1, \ldots, \epsilon_n\}, \{\eta_1, \ldots, \eta_m\} \rangle$, it means that M's value can be a cons cell constructed at one of ϵ_i's or a closure constructed at one of η_j's[3].

Abstract values are ordered by component-wise set-inclusion: $\langle \tau_1, \varphi_1 \rangle \sqsubseteq \langle \tau_2, \varphi_2 \rangle \Leftrightarrow (\tau_1 \subseteq \tau_2) \wedge (\varphi_1 \subseteq \varphi_2)$. The least upper bound of two abstract values is defined by component-wise set-union: $\langle \tau_1, \varphi_1 \rangle \sqcup \langle \tau_2, \varphi_2 \rangle = \langle \tau_1 \cup \tau_2, \varphi_1 \cup \varphi_2 \rangle$.

Abstract Environment The fixed-point iteration of abstract interpretation proceeds by updating an *abstract environment* ρ, a total function which maps variables to their abstract values. It has a functionality: $Variable \rightarrow A_v$, where *Variable* includes all variables appearing in the input program as well as three special variables: $car(\epsilon)$, $cdr(\epsilon)$, and $ran(\eta)$. $Car(\epsilon)$ and $cdr(\epsilon)$ are variables representing the *car* part and the *cdr* part of ϵ, respectively, while $ran(\eta)$ represents the range of η. If $\rho(car(\epsilon)) = \langle \{\epsilon_1, \ldots, \epsilon_n\}, \{\eta_1, \ldots, \eta_m\} \rangle$, it means that the *car* part of a cons cell constructed at ϵ can be a cons cell constructed at one of ϵ_i's or a closure constructed at one of η_j's. Likewise, $\rho(ran(\eta))$ represents the possible return value when a closure constructed at η is applied.

Two abstract environments ρ_1 and ρ_2 are ordered $\rho_1 \sqsubseteq \rho_2$, if $\rho_1(v) \sqsubseteq \rho_2(v)$ for all variables v. The least upper bound is similarly defined: $\rho_1 \sqcup \rho_2 = \lambda v.\rho_1(v) \sqcup \rho_2(v)$. The initial (empty) environment ρ_ϕ is defined by $\lambda v.\langle \{\}, \{\} \rangle$.

3.2 The Abstract Interpretation

The rules for the abstract interpretation is shown in Figure 2. Given an expression M and the current abstract environment ρ, the abstract interpreter $\mathcal{AI}$ returns an abstract value for M and a possibly modified environment. For example, $\mathcal{AI}\llbracket car\ x \rrbracket \rho_\phi$ represents possible values for $car\, x$, where the value of x is not yet known. Thus, the result is $(\langle \{\}, \{\} \rangle, \rho_\phi)$. In the case of $\mathcal{AI}\llbracket car\ x \rrbracket \rho$ where $\rho(x) = \langle \{\epsilon\}, \varphi \rangle$ and $\rho(car(\epsilon)) = \langle \tau', \varphi' \rangle$, $\mathcal{AI}$ deduces that $car\, x$ can have the value $\langle \tau', \varphi' \rangle$, giving the result $(\langle \tau', \varphi' \rangle, \rho)$.

The rules in Figure 2 are fairly straightforward. We briefly sketch them here. The value of a variable is defined in ρ. The value of λ-expression is its attached closure point. Meanwhile, we analyze the body of the λ-expression under ρ to enlarge the environment and include the information on the range of the λ-expression (possible return values when the λ-expression is applied)[4]. This information is used in the next rule for applications. The value of an application is a union of its operators' range. The operand information is propagated to operators by adding it to the operators' bound variables. (This information is used in the next iteration when operators are analyzed.) Here, x^{η_i} is a parameter of the closure labeled with η_i. The rule for *cons* returns a cons point, leaving the abstract values of its *car* and *cdr* parts in the abstract environment. The rule for *car* collects the *car* part of its argument. The rule for *setcar* adds the possibility that the *car* part of its first argument may have the value of its second argument. Finally, the value of *eq* is a union of its then and else branches.

The analysis is interprocedural, because the values of operands are propagated to the body of operators. It is context-insensitive, because the possible value of a bound variable is collected in one abstract value regardless of where it is originated from. It is 0CFA[21], because single closure point abstracts all the lambda closures created at that closure point.

3.3 Collecting Mutable Cells

Given an input program P, we repeatedly calculate $\mathcal{AI}\llbracket P \rrbracket \rho$ until ρ does not change, beginning with ρ_ϕ. Because abstract environments have only finite number of variations and operations on them are all monotonic, the environment will eventually reach a fixed-point ρ_{fix}, which can be shown to capture all the possible values for variables. Using this, we obtain *mutable*$_{car}$ and *mutable*$_{cdr}$ as follows. Let $(\langle \tau_i, \varphi_i \rangle, \rho_{\mathit{fix}}) = \mathcal{AI}\llbracket M_i \rrbracket \rho_{\mathit{fix}}$ for the i'th occurrence of $setcar M_i N_i$ in P. Since τ_i represents all the possible

---

[3] Or values other than cons cells or closures, which are not of our interest.

[4] The returned environment here is $\rho' \sqcup \rho_\phi[\langle \tau, \varphi \rangle / ran(\eta)]$ rather than $\rho'[\langle \tau, \varphi \rangle / ran(\eta)]$, because we *add* $\langle \tau, \varphi \rangle$ to $\rho'(ran(\eta))$ rather than *update* it to $\langle \tau, \varphi \rangle$ ignoring the former value of $\rho'(ran(\eta))$.

$$
\begin{array}{rcl}
\mathcal{AI} & : & Exp \rightarrow AbsEnv \rightarrow (A_v \times AbsEnv) \\
\rho \in AbsEnv & : & Variable \rightarrow A_v \\
\\
\mathcal{AI}[\![x]\!]\rho & = & (\rho(x), \rho) \\
\mathcal{AI}[\![\eta : \lambda x.M]\!]\rho & = & \mathbf{let}\ (\langle \tau, \varphi \rangle, \rho') = \mathcal{AI}[\![M]\!]\rho \\
 & & \mathbf{in}\ (\langle \{\}, \{\eta\} \rangle, \rho' \sqcup \rho_\phi[\langle \tau, \varphi \rangle / ran(\eta)]) \\
\mathcal{AI}[\![M N]\!]\rho & = & \mathbf{let}\ (\langle \tau_1, \{\eta_1, \ldots, \eta_n\} \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
 & & \quad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![N]\!]\rho_1 \\
 & & \mathbf{in}\ (\bigsqcup_i \rho_2(ran(\eta_i)), \rho_2 \sqcup (\bigsqcup_i \rho_\phi[\langle \tau_2, \varphi_2 \rangle / x^{\eta_i}])) \\
\mathcal{AI}[\![\epsilon : cons M N]\!]\rho & = & \mathbf{let}\ (\langle \tau_1, \varphi_1 \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
 & & \quad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![N]\!]\rho_1 \\
 & & \mathbf{in}\ (\langle \{\epsilon\}, \{\} \rangle, \rho_2 \sqcup \rho_\phi[\langle \tau_1, \varphi_1 \rangle / car(\epsilon)] \sqcup \rho_\phi[\langle \tau_2, \varphi_2 \rangle / cdr(\epsilon)]) \\
\mathcal{AI}[\![car M]\!]\rho & = & \mathbf{let}\ (\langle \{\epsilon_1, \ldots, \epsilon_n\}, \varphi \rangle, \rho') = \mathcal{AI}[\![M]\!]\rho\ \mathbf{in}\ (\bigsqcup_i \rho'(car(\epsilon_i)), \rho') \\
\mathcal{AI}[\![setcar M N]\!]\rho & = & \mathbf{let}\ (\langle \{\epsilon_1, \ldots, \epsilon_n\}, \varphi_1 \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
 & & \quad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![N]\!]\rho_1 \\
 & & \mathbf{in}\ (\langle \tau_2, \varphi_2 \rangle, \rho_2 \sqcup (\bigsqcup_i \rho_\phi[\langle \tau_2, \varphi_2 \rangle / car(\epsilon_i)])) \\
\mathcal{AI}[\![eq M M' N N']\!]\rho & = & \mathbf{let}\ (\langle \tau_1, \varphi_1 \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
 & & \quad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![M']\!]\rho_1 \\
 & & \quad (\langle \tau_3, \varphi_3 \rangle, \rho_3) = \mathcal{AI}[\![N]\!]\rho_2 \\
 & & \quad (\langle \tau_4, \varphi_4 \rangle, \rho_4) = \mathcal{AI}[\![N']\!]\rho_3 \\
 & & \mathbf{in}\ (\langle \tau_3 \cup \tau_4, \varphi_3 \cup \varphi_4 \rangle, \rho_4)
\end{array}
$$

Figure 2: Abstract Interpretation

cons points for M_i, $mutable_{car}$ is a union of these cons points: $mutable_{car}=\bigcup_i \tau_i$. $Mutable_{cdr}$ is obtained similarly.

4 Partial Evaluation

We now present a partial evaluator for the core language. It is online, and interprets input programs recursively as interpreters do. When it encounters a primitive application, it takes one of two actions: reduce or residualize. If all the arguments to the primitive are known values, it reduces the application and returns a result. If some of the arguments are unknown, it returns an expression that performs the primitive application at runtime.

In this paper, we do not address termination issues of partial evaluation, because our main concern here is the treatment of side-effects. Our partial evaluation mechanism can employ standard termination mechanisms such as the one used in Fuse[20].

4.1 Symbolic Value and Preaction

To distinguish a value from code, online partial evaluators use symbolic values (Sval). The symbolic values we use are shown in Figure 3. There are nine Svals, each corresponding to the nine syntactic category of the core language. At the code generation phase (Section 4.4), they are translated into the corresponding piece of code. Among them, Svals for closures and pairs are *known* Svals: closures are unfolded when applied and pairs are referenced by *car* and *cdr*.

In the presence of side-effects, several issues that did not appear in pure functional languages become important, such as code elimination, code duplication, and the order of execution. To handle these issues, we introduce PSval, a symbolic value with a sequence of *preactions*[14, 15]. Preactions hold all the operations that has to occur when the symbolic value is created.

$$
\begin{array}{rcl}
Sval & = & \mathsf{var}(Var) \\
 & | & \mathsf{closure}(ClosurePoint, Param, Body, \\
 & & \qquad Env, Name, PSval) \\
 & | & \mathsf{apply}(Name, Name) \\
 & | & \mathsf{pair}(ConsPoint, Name, Name) \\
 & | & \mathsf{car}(Name) \\
 & | & \mathsf{cdr}(Name) \\
 & | & \mathsf{setcar}(Name, Name) \\
 & | & \mathsf{setcdr}(Name, Name) \\
 & | & \mathsf{eq}(Name, Name, PSval, PSval) \\
Preaction & = & Name : Sval \\
PSval & = & {}^{\langle\!\langle Preaction^* \rangle\!\rangle} Name
\end{array}
$$

Figure 3: Symbolic Value and Preaction

In the conventional online partial evaluators such as Fuse[20], code duplication was avoided by using the graph structure of symbolic values. Although the graph structure is useful to separate the specialization process from the code duplication problem, it is complicated to translate the graph structure into concrete code preserving the sharing relationship between symbolic values. Rather than using the graph structure, we introduce indirection and use *names* to achieve the same effect. Readers can regard them as 'locations' of Svals. Then, a sequence of preactions acts like a store during partial evaluation. Names not only simplify the code generation phase, but also make the structure of the whole partial evaluator much clearer, which enables us to prove its correctness.

4.2 Basic Strategies

Before presenting the partial evaluator, we demonstrate with examples how preactions are used to handle the above-mentioned issues, as well as how they handle side-effects and preserve eq-ness of values. It is instructive to know that a sequence of preactions will be coded as a sequence of *let* expressions.

Code Elimination Consider a program[5] $car(cons M N)$, where N is a side-effectful expression. Reducing this program into M is incorrect because the side-effect that has to occur in N disappears. Instead, we save it as a preaction using a fresh name t as in ${}^{\langle\!\langle t:N \rangle\!\rangle}M$, which is coded into $let\ t = N\ in\ M$[6].

Code Duplication Code duplication occurs when a bound variable is used more than once. For example, assuming that N is side-effectful, reducing $(\lambda x.cons\ x\ x)N$ into $cons N N$ is incorrect because the side-effect in N occurs twice. In such a case, we save N as a preaction with a fresh name t and refer to it by its name: ${}^{\langle\!\langle t:N \rangle\!\rangle}\mathsf{pair}(t, t)$, which is coded into $let\ t = N\ in\ cons\, t\, t$.

Order of Execution The order of side-effects has to be preserved. Consider a program $(\lambda x.\lambda y.cons\ y\ x)M N$, where both M and N are side-effectful. Reducing the program into $cons N M$ reverses the order of M and N. Instead, we save M and N as preactions in the order of their evaluation: ${}^{\langle\!\langle t_1:M, t_2:N \rangle\!\rangle}\mathsf{pair}(t_2, t_1)$, which is coded into $let\ t_1 = M\ in\ let\ t_2 = N\ in\ cons\, t_2\, t_1$. Notice that a sequence of preactions holds a kind of execution history.

Handling of Side-effects Side-effectful operations are handled by residualizing all definitions, references, and updates of mutable cells in the correct order. For example, consider a program:

$$\begin{array}{l} let\ x = cons M N \\ in\ let\ y = setcar\ x\ M' \\ \quad in\ cons(cdr\ x)(car\ x). \end{array}$$

Since we know from the analysis in Section 3 that the *car* part of x is mutable, its definition $(cons M N)$, reference $(car\ x)$, and update $(setcar\ x\ M')$ are residualized as preactions. The partial evaluation of the above program becomes:

$${}^{\langle\!\langle t_1:M, t_2:N, t_3:\mathsf{pair}(t_1,t_2), t_4:\mathsf{setcar}(t_3,M'), t_5:\mathsf{car}(t_3) \rangle\!\rangle}\mathsf{pair}(t_2, t_5),$$

which is coded into:

$$\begin{array}{l} let\ t_1 = M \\ in\ let\ t_2 = N \\ \quad in\ let\ t_3 = cons\, t_1\, t_2 \\ \qquad in\ let\ t_4 = setcar\ t_3\ M' \\ \qquad\quad in\ let\ t_5 = car\ t_3 \\ \qquad\qquad in\ cons\, t_2\, t_5. \end{array}$$

Observe that $cdr\ x$ is reduced even though the value of x (i.e. $\mathsf{pair}(t_1, t_2)$) is put into preactions. Although $\mathsf{pair}(t_1, t_2)$ is given a name t_3 and residualized, it is not a dynamic value. This is in contrast to the conventional let-expression used in Similix, where once an expression is residualized in a let-expression, it is completely dynamic, and the information on its static parts are lost.

Preserving Eq-ness of Values In pure functional languages, it was enough to test value equality. In impure languages, we also have to handle pointer equality (eq-ness). It is intuitively understood that eq-ness is preserved in the absence of code elimination and duplication. We can also see this by regarding a sequence of preactions as a *store*. For example, consider a program $(\lambda x.M)(cons N N')$, which is specialized into ${}^{\langle\!\langle t:\mathsf{pair}(N,N') \rangle\!\rangle}M[t/x]$. Here, the unique name t serves as the location for $cons N N'$. To test the eq-ness of cons cells, we can compare them by their names. The preservation of eq-ness is formally shown by proving that the partial evaluator does not change the store semantics of input programs, which is discussed in Section 5.

4.3 The Partial Evaluator

We now present the partial evaluator for the core language (Figure 4). Given an expression, an environment ρ, and a *pe-store* γ, the partial evaluator $\mathcal{PE}$ returns a PSval together with a (possibly) modified pe-store. A pe-store maps names introduced so far to their bound Svals.

The rules in Figure 4 are written in such a way that all Svals created are saved as preactions to prevent code elimination. The underlined names (such as $\underline{t}$) denote fresh names. Side-effects on mutable cells are treated by residualizing their definitions (in the *cons* rule) and updates (in the *setcar* rule). As for references (in the *car* rule), we lookup $mutable_{car}$ to see whether it is safe to access the cell at specialization time.

Let us see the rules in detail. A variable is translated into its bound name. A λ-expression is partially evaluated to a closure Sval. A fresh name $\underline{t}$ is given for the closure which is used to test its eq-ness. When constructing a closure Sval, we eagerly partially evaluate the body of the λ-expression. This is because the closure might be left residual in the residual program, in which case we want to create a specialized version of the closure. It is possible to defer the partial evaluation of body expressions to the code generation phase. (We do so in our implementation.) However, this makes the correctness proof difficult, because we cannot separate the specialization phase from the code generation phase.

To partially evaluate an application $M N$, we first partially evaluate M and N. If M evaluates to a known closure, it is unfolded. Here, we write ${}^{\langle\!\langle P \rangle\!\rangle}\mathcal{PE}[\![M]\!]\rho\gamma$ to mean $({}^{\langle\!\langle P,P' \rangle\!\rangle}t, \gamma')$ where $({}^{\langle\!\langle P' \rangle\!\rangle}t, \gamma') = \mathcal{PE}[\![M]\!]\rho\gamma$. Observe that preactions from M and N are propagated to the result without disturbing the order of execution. If M evaluates to unknown code, on the other hand, the application is residualized.

The partial evaluation of a *cons* expression proceeds similarly. After partially evaluating its arguments, a pair Sval with a fresh name $\underline{t}$ is constructed. Since *cons* is non-strict on its arguments during partial evaluation, we can always construct a pair. Although the pair Sval is given a name $\underline{t}$, it does not mean that the pair becomes an unknown value. Instead, it is residualized with the name $\underline{t}$ and used at the same time in the subsequent partial evaluation[7] (if it is not mutable). Observe that the pair Sval constructed here is also put into the pe-store for the later use[8].

For a *car* expression, its argument is first partially evaluated to see if it is a known pair. If it is *not*, residual code to take its *car* part is constructed. Otherwise, the *car* part is basically extracted and returned. However, because the *car* part might be mutable and prohibited from accessing, we lookup $mutable_{car}$. If it turns out to be mutable, we construct code rather than returning its *car* part.

The rule for *setcar* is simple. It residualizes a piece of code to perform *setcar* at runtime. Finally, partial evaluation of *eq* is done by first partially evaluating its first two arguments. When they are both known Svals, one of two branches is selected according to whether their locations are the same. When one (or both) of the arguments are unknown, both branches are specialized to construct *eq* code.

---

[5] In this subsection, we omit cons points for better readability.

[6] We denote $let\ t = N\ in\ M$ as a syntactic sugar for $(\lambda t.M)N$.

[7] That is, it is "both reduced and left residual"[24].

[8] Although all cons cells are residualized to declare their identity, they do not necessarily appear in the residual program. Post-processing phase will remove those preactions that will never be referenced during the runtime evaluation of the residual program.

$$\begin{array}{rcrcl} & & \mathcal{PE} & : & \mathit{Exp} \to \mathit{Env} \to \mathit{PEStore} \to (\mathit{PSval} \times \mathit{PEStore}) \\ \rho & \in & \mathit{Env} & : & \mathit{Variable} \to \mathit{Name} \\ \gamma & \in & \mathit{PEStore} & : & \mathit{Name} \to \mathit{Sval} \end{array}$$

$$\mathcal{PE}\llbracket x \rrbracket \rho\gamma \;=\; (^{\langle\!\langle\rangle\!\rangle}\rho(x), \gamma)$$

$$\begin{array}{rcl} \mathcal{PE}\llbracket \eta : \lambda x.M \rrbracket \rho\gamma & = & \textbf{let } (^{\langle\!\langle P\rangle\!\rangle} t'', \gamma') = \mathcal{PE}\llbracket M \rrbracket \rho[\underline{t'}/x]\gamma[\mathsf{var}(\underline{t'})/\underline{t'}] \\ & & \qquad s = \mathsf{closure}(\eta, x, M, \rho, \underline{t'}, {}^{\langle\!\langle P\rangle\!\rangle} t'') \\ & & \textbf{in } (^{\langle\!\langle \underline{t}:s\rangle\!\rangle}\underline{t}, \gamma[s/\underline{t}]) \end{array}$$

$$\begin{array}{rcl} \mathcal{PE}\llbracket M N \rrbracket \rho\gamma & = & \textbf{let } (^{\langle\!\langle P_1\rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}\llbracket M \rrbracket \rho\gamma \\ & & \qquad (^{\langle\!\langle P_2\rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}\llbracket N \rrbracket \rho\gamma_1 \\ & & \textbf{in case } \gamma_2(t_1) \textbf{ of} \\ & & \qquad \begin{array}{lcl} \mathsf{closure}(\eta, x, M', \rho', \_, \_) & : & {}^{\langle\!\langle P_1, P_2\rangle\!\rangle}\mathcal{PE}\llbracket M' \rrbracket \rho'[t_2/x]\gamma_2 \\ \textbf{otherwise} & : & \textbf{let } s = \mathsf{apply}(t_1, t_2) \\ & & \textbf{in } (^{\langle\!\langle P_1, P_2, \underline{t}:s\rangle\!\rangle}\underline{t}, \gamma_2[s/\underline{t}]) \end{array} \end{array}$$

$$\begin{array}{rcl} \mathcal{PE}\llbracket \epsilon : \mathit{cons} M N \rrbracket \rho\gamma & = & \textbf{let } (^{\langle\!\langle P_1\rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}\llbracket M \rrbracket \rho\gamma \\ & & \qquad (^{\langle\!\langle P_2\rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}\llbracket N \rrbracket \rho\gamma_1 \\ & & \qquad s = \mathsf{pair}(\epsilon, t_1, t_2) \\ & & \textbf{in } (^{\langle\!\langle P_1, P_2, \underline{t}:s\rangle\!\rangle}\underline{t}, \gamma_2[s/\underline{t}]) \end{array}$$

$$\begin{array}{rcl} \mathcal{PE}\llbracket \mathit{car} M \rrbracket \rho\gamma & = & \textbf{let } (^{\langle\!\langle P\rangle\!\rangle} t, \gamma') = \mathcal{PE}\llbracket M \rrbracket \rho\gamma \\ & & \textbf{in case } \gamma'(t) \textbf{ of} \\ & & \qquad \begin{array}{lcl} \mathsf{pair}(\epsilon, t_1, t_2) & : & \textbf{if } \epsilon \in \mathit{mutable}_{\mathit{car}} \\ & & \textbf{then let } s = \mathsf{car}(t) \\ & & \qquad \textbf{in } (^{\langle\!\langle P, \underline{t'}:s\rangle\!\rangle}\underline{t'}, \gamma'[s/\underline{t'}]) \\ & & \textbf{else } (^{\langle\!\langle P\rangle\!\rangle} t_1, \gamma') \\ \textbf{otherwise} & : & \textbf{let } s = \mathsf{car}(t) \\ & & \textbf{in } (^{\langle\!\langle P, \underline{t'}:s\rangle\!\rangle}\underline{t'}, \gamma'[s/\underline{t'}]) \end{array} \end{array}$$

$$\begin{array}{rcl} \mathcal{PE}\llbracket \mathit{setcar} M N \rrbracket \rho\gamma & = & \textbf{let } (^{\langle\!\langle P_1\rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}\llbracket M \rrbracket \rho\gamma \\ & & \qquad (^{\langle\!\langle P_2\rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}\llbracket N \rrbracket \rho\gamma_1 \\ & & \qquad s = \mathsf{setcar}(t_1, t_2) \\ & & \textbf{in } (^{\langle\!\langle P_1, P_2, \underline{t}:s\rangle\!\rangle}\underline{t}, \gamma_2[s/\underline{t}]) \end{array}$$

$$\begin{array}{rcl} \mathcal{PE}\llbracket \mathit{eq} M M' N N' \rrbracket \rho\gamma & = & \textbf{let } (^{\langle\!\langle P_1\rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}\llbracket M \rrbracket \rho\gamma \\ & & \qquad (^{\langle\!\langle P_2\rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}\llbracket M' \rrbracket \rho\gamma_1 \\ & & \textbf{in if } \gamma_2(t_1) \text{ and } \gamma_2(t_2) \text{ are known} \\ & & \quad \textbf{then if } t_1 = t_2 \textbf{ then } {}^{\langle\!\langle P_1, P_2\rangle\!\rangle}\mathcal{PE}\llbracket N \rrbracket \rho\gamma_2 \\ & & \qquad\qquad\qquad\quad \textbf{else } {}^{\langle\!\langle P_1, P_2\rangle\!\rangle}\mathcal{PE}\llbracket N' \rrbracket \rho\gamma_2 \\ & & \quad \textbf{else let } (^{\langle\!\langle P_3\rangle\!\rangle} t_3, \gamma_3) = \mathcal{PE}\llbracket N \rrbracket \rho\gamma_2 \\ & & \qquad\qquad (^{\langle\!\langle P_4\rangle\!\rangle} t_4, \gamma_4) = \mathcal{PE}\llbracket N' \rrbracket \rho\gamma_3 \\ & & \qquad\qquad s = \mathsf{eq}(t_1, t_2, {}^{\langle\!\langle P_3\rangle\!\rangle} t_3, {}^{\langle\!\langle P_4\rangle\!\rangle} t_4) \\ & & \qquad \textbf{in } (^{\langle\!\langle P_1, P_2, \underline{t}:s\rangle\!\rangle}\underline{t}, \gamma_4[s/\underline{t}]) \end{array}$$

Figure 4: Partial Evaluator

$$
\begin{array}{rcl}
\mathcal{PD}\llbracket^{\langle\langle\rangle\rangle}t\rrbracket & = & t \\
\mathcal{PD}\llbracket^{\langle\langle t_1:s_1,\ldots\rangle\rangle}t\rrbracket & = & \mathit{let}\ t_1 = \mathcal{D}\llbracket s_1\rrbracket\ \mathit{in}\ \mathcal{PD}\llbracket^{\langle\langle\ldots\rangle\rangle}t\rrbracket
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{D}\llbracket\mathsf{closure}(\eta,\_,\_,\_,t,{}^{\langle\langle P\rangle\rangle}t')\rrbracket & = & (\eta : \lambda t.\mathcal{PD}\llbracket^{\langle\langle P\rangle\rangle}t'\rrbracket) \\
\mathcal{D}\llbracket\mathsf{apply}(t_1,t_2)\rrbracket & = & (t_1\ t_2) \\
\mathcal{D}\llbracket\mathsf{pair}(\epsilon,t_1,t_2)\rrbracket & = & (\epsilon : \mathit{cons}\ t_1\ t_2) \\
\mathcal{D}\llbracket\mathsf{car}(t)\rrbracket & = & (\mathit{car}\ t) \\
\mathcal{D}\llbracket\mathsf{setcar}(t_1,t_2)\rrbracket & = & (\mathit{setcar}\ t_1\ t_2) \\
\mathcal{D}\llbracket\mathsf{eq}(t_1,t_2,{}^{\langle\langle P_3\rangle\rangle}t_3,{}^{\langle\langle P_4\rangle\rangle}t_4)\rrbracket & = & (\mathit{eq}\ t_1\ t_2\ \mathcal{PD}\llbracket^{\langle\langle P_3\rangle\rangle}t_3\rrbracket \\
 & & \qquad \mathcal{PD}\llbracket^{\langle\langle P_4\rangle\rangle}t_4\rrbracket)
\end{array}
$$

Figure 5: Code Genaration

4.4 Code Genaration

Since $\mathcal{PE}$ produces its output as a PSval, we have to convert it into the core language to complete the partial evaluation. Thanks to the use of names, the code generation is straightforward as shown in Figure 5. We only note that the name clash will never occur, because all names are uniquely chosen at the specialization phase.

5 Correctness of the Partial Evaluator

In this section, we will briefly see that the partial evaluator presented in the previous section preserves the store semantics of the input program. Although we have already completed the proof, we only outline it here due to the lack of space. The details are presented in the technical report version[2] of this paper.

Because our partial evaluator consists of two parts (specialization $\mathcal{PE}$ and code generation $\mathcal{PD}$), the correctness is also divided into two parts: the correctness of $\mathcal{PD}$ and $\mathcal{PE}$. As for $\mathcal{PD}$, we first define the semantics of PSval $\mathcal{E}_{PS}$ in such a way that the semantics of PSval is given by the semantics of its codified program. Then, we prove that $\mathcal{PD}$ actually preserves the semantics: $\mathcal{E}_{PS}\llbracket^{\langle\langle P\rangle\rangle}t\rrbracket\sigma = \mathcal{E}\llbracket\mathcal{PD}\llbracket^{\langle\langle P\rangle\rangle}t\rrbracket\rrbracket\sigma$, where $\mathcal{E}$ is a standard store semantics of the core language. The proof is straightforward and done by induction on the number of $\mathcal{PD}$ applications required to transform the PSval ${}^{\langle\langle P\rangle\rangle}t$ into code. After that, we prove the main theorem stating that $\mathcal{PE}$ preserves the semantics: given the sound $\mathit{mutable}_{car}$ and $\mathit{mutable}_{cdr}$, if γ respects σ, $\mathcal{E}_{PS}\llbracket\mathcal{PE}\llbracket M\rrbracket\rho\gamma\rrbracket\sigma = \mathcal{E}\llbracket M\rrbracket\rho\sigma$. The proof is again by induction on the number of $\mathcal{PE}$ applications required to partially evaluate M.

6 Extensions and Limitations

So far, we have considered the partial evaluation of the core language. In this section, we describe extensions to cope with more realistic languages such as Scheme, as wel as limitations our partial evaluator currently has. Given a mechanism for handling side-effects correctly, it is not difficult to incorporate other features of functional languages. The extensions presented below (other than the last two) are actually implemented in our Scheme partial evaluator, which can now deal with almost all features of Scheme.

Variable Assignments and I/O Assignments to variables can be handled by translating them to updates on data structures. Instead of storing a value v to a mutable variable x, we store a cell containing the value v to x. References and updates to x are translated to references and updates to the cell, respectively. Alternatively, assignments to variables can be handled by finding mutable variables and residualizing their definitions, references, and updates. This process parallels to the treatment of mutable cells and is realized almost the same as mutable cells. Since mutable variables can be found by scanning the input program once, the analysis phase for finding mutable variables becomes much simpler. In our current implementation, the latter approach is taken.

Input/output operations are handled by simply residualizing them as preactions[15].

Recursion and Termination Detection Given that mutable cells are completely separated by the side-effect analysis, the presence of side-effects has no influence on the termination detection nor on the folding mechanism. All the mutable parts are hidden from accesses and are correctly residualized by the preaction mechanism. We can introduce recursion and any conventional termination detection algorithm for pure functional languages, treating side-effecting operations as ordinary dynamic expressions. In our current implementation, we employ user annotation (or *filters*[7, 9]) to avoid non-termination.

λ-expressions with Variadic Parameters As was pointed out by Thiemann[23], it is easy to handle variadic parameters when pairs are treated as partially static data structures[18]. By explicitly supplying cons points for the implicitly introduced cons cells for variadic parameters, we can support variadic parameters including destructive updates to those cons points. We can also support a higher-order primitive *apply* through extending the side-effect analysis to cope with function application caused by *apply*.

Constants and Other Primitives Constants and first-order primitives are handled straightforwardly. Higher-order primitives other than *apply* (e.g., *map* and *assq*) are provided as ordinary user-defined functions. We have not yet considered partial evaluation of *call/cc*.

Post-processing The post-processing phase removes those preactions that will not referenced during the runtime evaluation of the residual program. It also performs additional local transformation for the better readability. It includes: folding a sequence of `let` expressions into a `let*` expression, inlining expressions that are used exactly once[9], etc.

No Code Sharing We have not considered code sharing at all. The result of partial evaluation sometimes becomes quite large because of the repeated residualization of the same functions. In the presence of side-effects, it is not obvious if a previously residualized function can be re-used later, because some of the referred cells might be modified. In our case, however, references to mutable cells are never performed and their values are not used during partial evaluation. This means that residualized functions are independent of the values of mutable cells. Thus, we expect that we can use the conventional mechanism for code sharing for pure functional languages.

No External Cons and Closure Points Our partial evaluator assumes that all the cons points and closure points are available at partial evaluation time. If some of them are not present, it may produce a wrong result. For example, to partially evaluate the

---

[9] Care must be taken if it is harmless to reverse the order of execution. In particular, the order of references to mutable cells (e.g., *car x*) and updates on them (*setcar x M*) cannot be reversed.

```
(letrec
  ((base-eval
    (lambda (exp env)
      (cond ((number? exp) exp)
            ((symbol? exp) (eval-var exp env))
            ((eq? (car exp) 'quote) (eval-quote exp env))
            ((eq? (car exp) 'set!) (eval-set! exp env))
            ((eq? (car exp) 'lambda) (eval-lambda exp env))
            ...
            (else (eval-application exp env)))))
   (eval-set!
    (lambda (exp env)
      (let ((var (car (cdr exp)))
            (body (car (cdr (cdr exp)))))
        (set-cdr! (assq var env) (base-eval body env)))))                 ; side-effects
   (eval-lambda
    (lambda (exp env)
      (let ((lambda-params (car (cdr exp)))
            (lambda-body (cdr (cdr exp))))
        (lambda args                            ; variadic parameter for lambda closures
          (base-eval lambda-body (extend env lambda-params args))))))
   (extend
    (lambda (env params args)
      (if (null? params)
          env
          (cons (cons (car params) (car args))
                (extend env (cdr params) (cdr args))))))
   ...)
   (base-eval '(lambda (x) (+ x (begin (set! x 3) x)))                    ; main program
              (list (cons '+ +) (cons '- -) ...)))                         ;  environment
```

Figure 6: The Core Part of the Scheme Interpreter

following program:

$$\begin{aligned}&\lambda f.\, let\ a = cons\, 1\, 2\\ &\quad in\ let\ b = f\ a\\ &\qquad in\ car\ a,\end{aligned}$$

the side-effect analysis will determine that a is immutable. Thus, our specializer will reduce $car\ a$, producing the following result:

$$\begin{aligned}&\lambda f.\, let\ a = cons\, 1\, 2\\ &\quad in\ let\ b = f\ a\\ &\qquad in\ 1.\end{aligned}$$

However, the result is not correct if the program is applied to $\lambda x. setcar\ x\ 3$. The problem can be avoided by treating unknown parameters conservatively as "any value", as done by Shivers[21].

7 Specializing an Interpreter

In this section, we show an example specialization of a larger program: an interpreter for a subset of Scheme with a variable assignment operation `set!`. The core part is shown in Figure 6. The main function `base-eval` dispatches on the expression to be evaluated as usual. Among various special forms, we show evaluator functions for `set!` and `lambda`. `Set!` statements in interpreted programs are interpreted as `set-cdr!` operations to the environment, which is implemented as a list of bindings (an association list). This is the only place where side-effects are used in the program. λ-expressions are realized by λ-closures with a variadic parameter.

Assume that the initial environment is defined as:

```
(list (cons '+ +) (cons '- -) ...).
```

To partially evaluate this interpreter with respect to a program:

```
'(lambda (x) (+ x (begin (set! x 3) x)))
```

and the initial environment, the side-effect analysis will first find that the *cdr* parts of the bindings in the environment (underlined `cons` expressions in the figure) are mutable. Since these are the only mutable parts, the shape of the environment is known and the partial evaluator can unfold all the environment accesses, completely removing the interpretation process. The result of partially evaluating the interpreter (after renaming) thus becomes:

```
(let ((pair1 (cons '+ +)))
  (lambda args
    (let* ((pair2 (cons 'x (car args)))
           (operator-+ (cdr pair1))
           (g-270 (cdr pair2)))
      (set-cdr! pair2 3)
      (operator-+ g-270 (cdr pair2)))))
```

We can observe that all the environment accesses are unfolded and only the pairs for `+` (`pair1`) and `x` (`pair2`) residualize. The `set!` statement in the original interpreted program is compiled into a single `set-cdr!` statement. We have succeeded in compiling side-effecting programs using an interpreter written with side-effects.

The performance of the partial evaluator is reasonable. To analyze and partially evaluate the above program (the version which supports seven special forms including `letrec` and 27 primitives), it takes less than one second on Chez Scheme on a SUN SS20 workstation (HiperSPARC 150MHz) with 128 Mbyte of memory.

Currently, we are trying to use our partial evaluator as a compiler for the reflective language Black[3]. Since updates on cons cells are used in environment manipulation in Black, the ability to handle updates as well as pointer equality is essential. The primary experiment shows that it is powerful enough to collapse multi-level meta-circular interpreters, which enables us to compile a program under modified semantics.

8 Related Work

8.1 Side-effect Analysis

The side-effect analysis we have used is based on Consel's binding time analysis[8], which introduces the notion of cons points and closure points. We have extended it to return possible values for expressions rather than their binding times. The resulting analysis becomes quite similar to other existing analyses such as a set-based analysis[12]. In fact, the same information can be obtained by using the set-based analysis, which can deduce a set of possible values for higher-order λ-calculus with updatable arrays. Using a set-based-analysis, we expect to have complexity benefit over abstract-interpretation based approach.

8.2 Partial Evaluation

There are several partial evaluators for Scheme. Among them, Similix[5] is a publicly available offline partial evaluator. Although it can deal with I/O operations and limited types of variable assignments, it does not handle destructive updates of data structures. Schism[9] is also an offline partial evaluator for Scheme, but does not address the partial evaluation of side-effects. POPE[19] is an online partial evaluator for Scheme, which is also publicly available. It accepts richer kinds of variable assignments than Similix, but does not handle destructive updates of data structures correctly.

Fuse[20] is an online partial evaluator, which our partial evaluator was originally based on. It uses graph structure to avoid code duplication. Using preactions, we extended it to handle side-effects, which Fuse does not handle[10]. We also use names to simplify the code generation phase considerably.

Partial evaluation of side-effects has been considered in imperative languages, such as Pascal[16], Fortran[4], and C[1]. Among them, C-MIX is a powerful partial evaluator that can deal with pointers. It incorporates various analyses (e.g., a pointer analysis and a data-flow analysis) to obtain side-effect information. We performed a similar analysis for higher-order languages where local closures are allowed. Currently, we do not reduce assignments because it does not improve the result very much in functional languages. In imperative languages, most computation is done by assignments to local variables, and thus reducing them is essential. In functional languages, because this type of computation is realized without using assignments, we can still obtain sufficient specializations without reducing them.

Recently, Dussart and Thiemann[11] developed a partial evaluator for ML with a reference type that can statically reduce local static side-effects. In this respect, their partial evaluator is more powerful than ours. Because mutable data are syntactically clear in ML programs, they do not need a side-effect analysis. However, since their specializer is based on Similix's let-expressions, it would be difficult to reduce programs where the preservation of pointer equality is essential, such as Scheme programs. It is interesting to see if it is possible to incorporate their technique in our framework.

8.3 Preactions

Preactions used in this paper have close relationship to the CPS-based let-expression technique used in Similix. When unknown expressions are encountered in Similix, let-expressions are constructed to residualize them. Since names bound to them are used in the subsequent partial evaluation, problems such as code elimination and code duplication are avoided in the same way as preactions are used. The order of side-effects are preserved by incorporating let-expressions to CPS-based specialization.

Added to these features realized by the CPS-based let-expression technique, the preaction mechanism achieves the propagation of information on residualized expressions. This difference becomes important when residualized expressions are *used* in the subsequent partial evaluation. When preactions are used, they are still accessible, enabling further specialization. If let-expressions are used, on the other hand, they become unknown and prohibited from accessing. Remember that λ-expressions and *cons* expressions have side-effects on the store (to declare their identity) and thus residualized as preactions. Treating them unknown results in a very poor specialization. The preaction mechanism enables us to treat expressions as both dynamic and static.

In our earlier work[15], preactions are used to residualize I/O operations. Here, we have extended it to cope with destructive assignments of data structures, as well as partially static data structures. In Mogensen's original paper on partially static data structures[17], he mentioned a use of *field* to avoid the code duplication problem. The preaction mechanism can be regarded as its generalization where residualized expressions are not necessarily dynamic ones.

9 Conclusion

We have presented a framework of an online partial evaluator for a call-by-value λ-calculus that deals with destructive updates of data structures and preserves pointer equality. The two key techniques used here are the side-effect analysis and preactions. The side-effect analysis is used to separate immutable cells from mutable ones. Since mutable parts are usually small in functional languages, it enables us to reduce most accesses to structured data at partial evaluation time. For the correct residualization of side-effecting operations, preactions are used to solve various issues, such as code elimination, code duplication, and the order of execution. Furthermore, it enables us to treat expressions as both static and dynamic. Partially static data structures are naturally realized by residualizing them as preactions and using them at the same time in the subsequent partial evaluation. This makes more expressions to be reduced at specialization time that were residualized and treated as dynamic when the conventional let-expression was used. The correctness of the partial evaluator is proven. Based on the framework, we have constructed a partial evaluator for Scheme, which is powerful enough to specialize an interpreter written using side-effects.

Acknowledgements

We received many helpful comments from Olivier Danvy, Peter Thiemann, and Naoki Kobayashi for the earlier version of this paper. Comments from anonymous referees improved this paper in various ways.

---

[10] There seems to be a version of Fuse that handles side-effects[25]. However, actual treatment of side-effects is not clear.

References

[1] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*, Ph.D. thesis, DIKU, University of Copenhagen (May 1994).

[2] Asai, K., H. Masuhara, and A. Yonezawa “Partial Evaluation of Call-by-value λ-calculus with Side-effects,” Technical report of Department of Information Science, University of Tokyo, 96-04 (November 1996).

[3] Asai, K., S. Matsuoka, and A. Yonezawa “Duplication and Partial Evaluation — For a Better Understanding of Reflective Languages —,” *Lisp and Symbolic Computation*, Vol. 9, Nos. 2/3, pp. 203–241, Kluwer Academic Publishers (May/June 1996).

[4] Baier, R., R. Glück, and R. Zöchling “Partial Evaluation of Numerical Programs in Fortran,” *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM ’94)*, pp. 119–132 (June 1994).

[5] Bondorf, A., and O. Danvy “Automatic autoprojection of recursive equations with global variables and abstract data types," *Science of Computer Programming*, Vol. 16, pp. 151–195, Elsevier (1991).

[6] Clinger, W., and Rees, J. (editors) “Revised^4 Report on the Algorithmic Language Scheme", *LISP Pointers*, Vol. IV, No. 3, pp. 1–55 (July-September 1991).

[7] Consel, C. “New Insights into Partial Evaluation: the SCHISM Experiment," *ESOP’88, 2nd European Symposium on Programming (LNCS 300)*, pp. 236–246 (March 1988).

[8] Consel, C. “Binding Time Analysis for Higher Order Untyped Functional Languages," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 264–272 (June 1990).

[9] Consel, C. “A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages,” *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM’93)*, pp. 145–154 (June 1993).

[10] Dean, J., C. Chambers, and D. Grove “Identifying Profitable Specialization in Object-Oriented Languages," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM ’94)*, pp. 85–96 (June 1994).

[11] Dussart, D., and P. Thiemann “Partial Evaluation for Higher-Order Languages with State," Submitted for publication (1996).

[12] Heintze, N. “Set-Based Analysis of ML Programs," *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 306–317 (June 1994).

[13] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

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

[15] Masuhara, H., S. Matsuoka, K. Asai, and A. Yonezawa “Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation,” *Tenth Annual Conference of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ’95)*, pp. 300–315, (October 1995).

[16] Meyer, U. “Techniques for partial evaluation of imperative languages,” *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM ’91)*, pp. 94–105 (June 1991).

[17] Mogensen, T. Æ. “Partially Static Structures in a Self-Applicable Partial Evaluator," In D. Bjørner, A. P. Ershov, and N. D. Jones editors, *Partial Evaluation and Mixed Computation*, Elsevier Science Publishers, pp. 325–347 (1988).

[18] Mogensen, T. Æ. “Separating Binding Times in Language Specifications," *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA’89)*, pp. 12–25 (September 1989).

[19] Ørbæk, P. “POPE: An On-line Partial Evaluator," available from `ftp://ftp.daimi.aau.dk/pub/empl/poe/pope.ps.gz` (June 1994).

[20] 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.

[21] Shivers, O. “Control Flow Analysis in Scheme," *Proceedings of the ACM SIGPLAN ’88 Conference on Programming Language Design and Implementation (PLDI)*, pp. 164–174 (June 1988).

[22] Sperber, M., and P. Thiemann “The Essence of LR Parsing," *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM’95)*, pp. 146–155 (June 1995).

[23] Thiemann, P. “Towards Partial Evaluation of Full Scheme," *Proceedings of Reflection’96*, pp. 105–115 (April 1996).

[24] 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 (August 1991).

[25] Weise, D., and S. Seligman “Accelerating Object-Oriented Simulation via Automatic Program Specialization," Technical Report of Computer Systems Laboratory, Stanford University, CSL-TR-92-519 (also FUSE Memo 92-10), (April 1992).