TECHNICAL REPORT 96-04

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

Kenichi Asai Hidehiko Masuhara Akinori Yonezawa

November 6, 1996

Department of Information Science
Faculty of Science, University of Tokyo

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

TECHNICAL REPORT 96-04

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

AUTHORS
Kenichi Asai, Hidehiko Masuhara, and Akinori Yonezawa

KEY WORDS AND PHRASES
partial evaluation, side-effects, pointer equality, preaction, online specialization

ABSTRACT
We present a framework of an online partial evaluator for a call-by-value λ-calculus with destructive updates of data structures. To our knowledge, this is the first partial evaluator that can deal with full side-effects as well as pointer equality in higher-order languages. Our partial evaluator uses a side-effect analysis for handling assignment operations and then performs an online specialization using preactions. By separating mutable data structures from immutable ones using the side-effect analysis, the most accesses to structured data (e.g., cons cells) are statically reduced 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. The resulting partial evaluator is simple enough to prove its correctness. Based on the framework, we extend it to a partial evaluator for almost full Scheme and demonstrate that it is powerful enough to specialize an interpreter written using side-effects.

REPORT DATE	**WRITTEN LANGUAGE**
November 6, 1996	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
30	24

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

Kenichi Asai Hidehiko Masuhara* Akinori Yonezawa

Abstract

We present a framework of an online partial evaluator for a call-by-value λ-calculus with destructive updates of data structures. To our knowledge, this is the first partial evaluator that can deal with full side-effects as well as pointer equality in higher-order languages. Our partial evaluator uses a side-effect analysis for handling assignment operations and then performs an online specialization using preactions. By separating mutable data structures from immutable ones using the side-effect analysis, the most accesses to structured data (e.g., cons cells) are statically reduced 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. The resulting partial evaluator is simple enough to prove its correctness. Based on the framework, we extend it to a partial evaluator for almost full Scheme and demonstrate that it is powerful enough to specialize an interpreter written using side-effects.

1 Introduction

Given a program and parts of its arguments, a partial evaluator[13] produces a *specialized* program with respect to those known arguments. Since the computation that depends only on the known arguments 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[21].

Although much work has been done for partial evaluation of functional languages, partial evaluation of side-effecting constructs remains difficult. In conventional partial evaluators for functional languages, only restricted use of side-effects were allowed, and other types of side-effects were either refused or processed incorrectly. For example, Similix[4] handles only I/O operations and restricted types of variable assignments. POPE[19] provides wider use of variable assignments, but does not handle destructive updates of data structures correctly. This is a major restriction since we often use side-effects even when we program mostly in functional style.

In this paper, we present an online partial evaluator for a call-by-value λ-calculus with destructive updates of data structures. It also preserves pointer equality ('eq-ness'). This enables us to construct a partial evaluator for almost full Scheme[6]. To our knowledge, this is the first partial evaluator that can deal with these aspects correctly in higher-order languages.

The difficulties for partially evaluating programs with side-effects mainly come from mutable data structures (and mutable variables). Because partial evaluators perform non-standard

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

```
(letrec ((map (lambda (f lst)
                (if (null? lst)
                    '()
                    (cons (f (car lst))
                          (map f (cdr lst)))))))
  (map (lambda (pair)
         (set-cdr! pair (+ (car pair) (cdr pair)))
         pair)
       '((1 . 1) (2 . 2) (3 . 3))))
```

Figure 1: An Example Program

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 an expression: `(set-car! x y)` where `x` is unknown. Without assumption on the value of `x`, we have to treat all the data structures as unknown after this assignment, resulting in poor specializations.

The approach we take is to identify mutable data structures (possible values for `x` in the above example) using a side-effect analysis. 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. Since most programs modify only restricted parts of data structures, the separated mutable parts are usually small and we can obtain sufficient specializations. 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 treating side-effects exists in specialization. We cannot eliminate, duplicate, or reverse code, since it might eliminate, duplicate, or reverse side-effects. To avoid these problems, we use *preactions*[15, 16]. Without code elimination and duplication, our partial evaluator naturally preserves pointer equality ('eq-ness') of each value.

To see how our partial evaluator handles side-effects, consider a Scheme program[1] shown in Figure 1. Given a list of pairs, it updates *cdr* parts of each pair by the sum of its *car* and *cdr* parts using the higher-order function `map`. To partially evaluate this program, we first perform the side-effect analysis. It will detect that `pair` can be bound to either `(1 . 1)`, `(2 . 2)`, or `(3 . 3)`, and thus the *cdr* parts of these pairs are mutable. The partial

[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.

evaluator will residualize all accesses to these parts preserving the order of execution, and process only other parts of the program. Because our partial evaluator handles partially static data structures[18], the *car* part of `pair` is accessible. After unfolding the iteration in `map`, the result will look like this:

```
(let* ((pair1 (cons 1 1))
       (pair2 (cons 2 2))
       (pair3 (cons 3 3)))
  (set-cdr! pair1 (+ 1 (cdr pair1)))
  (set-cdr! pair2 (+ 2 (cdr pair2)))
  (set-cdr! pair3 (+ 3 (cdr pair3)))
  (list pair1 pair2 pair3))
```

All the references and updates to the mutable parts are residualized in the correct order, and other parts are completely unfolded. Since the three mutable cells are never eliminated nor duplicated, we can regard their unique names `pair1`, `pair2`, and `pair3` as the identities or locations of these pairs.

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.

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:

$$M = x \mid \eta : \lambda x.M \mid MN \mid \epsilon : consMN \mid carM \mid cdrM \mid setcarMN \mid setcdrMN \mid eqMM'NN'$$

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, $mutable_{car}$ is empty and $mutable_{cdr}$ contains cons points for underlined pairs. 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 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 During abstract interpretation, each subexpression is assigned an abstract value A_v, which has a form $\langle \tau, \varphi \rangle$, where τ is a set of cons points and φ is a set of closure points. An abstract value 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 can be either a cons cell constructed at one of ϵ_i's or a closure constructed at one of η_j's.

Abstract values are ordered by component-wise set-inclusion: $\langle \tau_1, \varphi_1 \rangle \sqsubset \langle \tau_2, \varphi_2 \rangle \Leftrightarrow (\tau_1 \subseteq \tau_2) \wedge (\varphi_1 \subseteq \varphi_2)$. 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, 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.

Two abstract environments ρ_1 and ρ_2 are ordered $\rho_1 \sqsubset \rho_2$, if $\rho_1(v) \sqsubset \rho_2(v)$ 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 environment ρ_{empty} 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, $\mathcal{AI}[\![car\,x]\!]\rho_{empty}$ represents possible values for $car\,x$, where the value of x is not yet known. Thus, the result is $(\langle \{\}, \{\} \rangle, \rho_{empty})$. In the case of $\mathcal{AI}[\![car\,x]\!]\rho$ 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). 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

$$\begin{array}{rcl} \mathcal{AI} & : & Exp \to AbsEnv \to (A_v \times AbsEnv) \\ \rho \in AbsEnv & : & Variable \to A_v \end{array}$$

$$\begin{array}{l}
\mathcal{AI}[\![x]\!]\rho = (\rho(x), \rho) \\
\mathcal{AI}[\![\eta : \lambda x.M]\!]\rho = \textbf{let } (\langle \tau, \varphi \rangle, \rho') = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad \textbf{in } (\langle \{\}, \{\eta\} \rangle, \rho' \sqcup \rho_{empty}[\langle \tau, \varphi \rangle / ran(\eta)]) \\
\mathcal{AI}[\![MN]\!]\rho = \textbf{let } (\langle \tau_1, \{\eta_1, \ldots, \eta_n\} \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![N]\!]\rho_1 \\
\qquad\qquad \textbf{in } (\bigsqcup_i \rho_2(ran(\eta_i)), \rho_2 \sqcup (\bigsqcup_{i,j} \rho_{empty}[\langle \tau_2, \varphi_2 \rangle / x^{\eta_i}])) \\
\mathcal{AI}[\![\epsilon : consMN]\!]\rho = \textbf{let } (\langle \tau_1, \varphi_1 \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![N]\!]\rho_1 \\
\qquad\qquad \textbf{in } (\langle \{\epsilon_i\}, \{\} \rangle, \rho_2 \sqcup \rho_{empty}[\langle \tau_1, \varphi_1 \rangle / car(\epsilon_i)] \sqcup \rho_{empty}[\langle \tau_2, \varphi_2 \rangle / cdr(\epsilon_i)]) \\
\mathcal{AI}[\![carM]\!]\rho = \textbf{let } (\langle \{\epsilon_1, \ldots, \epsilon_n\}, \varphi \rangle, \rho') = \mathcal{AI}[\![M]\!]\rho \textbf{ in } (\sqcup_i \rho'(car(\epsilon_i)), \rho') \\
\mathcal{AI}[\![setcarMN]\!]\rho = \textbf{let } (\langle \{\epsilon_1, \ldots, \epsilon_n\}, \varphi_1 \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![N]\!]\rho_1 \\
\qquad\qquad \textbf{in } (\langle \tau_2, \varphi_2 \rangle, \rho_2 \sqcup (\bigsqcup_i \rho_{empty}[\langle \tau_2, \varphi_2 \rangle / car(\epsilon_i)])) \\
\mathcal{AI}[\![eqMM'NN']\!]\rho = \textbf{let } (\langle \tau_1, \varphi_1 \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![M']\!]\rho_1 \\
\qquad\qquad (\langle \tau_3, \varphi_3 \rangle, \rho_3) = \mathcal{AI}[\![N]\!]\rho_2 \\
\qquad\qquad (\langle \tau_4, \varphi_4 \rangle, \rho_4) = \mathcal{AI}[\![N']\!]\rho_3 \\
\qquad\qquad \textbf{in } (\langle \tau_3 \cup \tau_4, \varphi_3 \cup \varphi_4 \rangle, \rho_4)
\end{array}$$

Figure 2: Abstract Interpretation

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.

3.3 Collecting Mutable Cells

Given an input program P, we repeatedly calculate $\mathcal{AI}[\![P]\!]\rho$ until ρ does not change, beginning with ρ_{empty}. 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 captures all the possible values for variables. Using this, we obtain $mutable_{car}$ and $mutable_{cdr}$ as follows. Let $(\langle \tau_i, \varphi_i \rangle, \rho_{fix}) = \mathcal{AI}[\![M_i]\!]\rho_{fix}$ for the i'th occurrence of $setcarM_iN_i$ in P. Since τ_i represents all the possible 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

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

Figure 3: Symbolic Value and Preaction

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 syntax category of the core language. At the codification 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*[15, 16]. Preactions hold all the operations that has to occur when the symbolic value is created.

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 becomes much complicated to codify the graph structure 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 codification 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 $car(consMN)$, 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 as in $^{\langle\langle t{:}N\rangle\rangle}M$, which is coded into $let\ t = N\ in\ M$[2].

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 $consNN$ 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: $^{\langle\langle t:N\rangle\rangle}\mathbf{pair}(t,t)$, 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 $(\lambda x.\lambda y.cons\, y\, x)MN$, where both M and N are side-effectful. Reducing the program into $consNM$ reverses the order of M and N. Instead, we save M and N as preactions in the order of their evaluation: $^{\langle\langle t_1:M,t_2:N\rangle\rangle}\mathbf{pair}(t_2,t_1)$, 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:

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

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

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

which is coded into:

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

Observe that $cdr\, x$ is reduced even though the value of x is put into preactions.

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

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 $(\lambda x.M)(consNN')$, which is specialized into $\langle\!\langle t{:}\mathbf{pair}(N,N')\rangle\!\rangle M[t/x]$. Here, the unique name t serves as the location for $consNN'$. 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 its definition (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. Note that even though all *cons* statements are residualized as preactions, they are accessible if they are immutable.

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 codification phase. (We do so in our implementation.) However, this makes the correctness proof difficult, because we cannot separate the specialization phase from the codification phase.

To partially evaluate an application MN, 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[3] (if it is not mutable). This can be seen in the next rule for *car* expressions. 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.

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

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

$$
\begin{array}{l}
\mathcal{PE}[\![x]\!]\rho\gamma = (^{\langle\langle\rangle\rangle}\rho(x), \gamma) \\
\mathcal{PE}[\![\eta : \lambda x.M]\!]\rho\gamma = \textbf{let } (^{\langle\langle P\rangle\rangle}t'', \gamma') = \mathcal{PE}[\![M]\!]\rho[\underline{t}'/x]\gamma[\textbf{var}(\underline{t}')/\underline{t}'] \\
\qquad\qquad\qquad\qquad s = \textbf{closure}(\eta, x, M, \rho, \underline{t}', ^{\langle\langle P\rangle\rangle}\, t'') \\
\qquad\qquad\qquad \textbf{in } (^{\langle\langle \underline{t}:s\rangle\rangle}\underline{t}, \gamma[s/\underline{t}]) \\
\mathcal{PE}[\![MN]\!]\rho\gamma = \textbf{let } (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\quad \textbf{in case } \gamma_2(t_1) \textbf{ of} \\
\qquad\qquad\qquad \textbf{closure}(\eta, x, M', \rho', _, _) \;:\; ^{\langle\langle P_1,P_2\rangle\rangle}\mathcal{PE}[\![M']\!]\rho'[t_2/x]\gamma_2 \\
\qquad\qquad\qquad \textbf{otherwise} \;:\; \textbf{let } s = \textbf{apply}(t_1, t_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } (^{\langle\langle P_1,P_2,\underline{t}:s\rangle\rangle}\underline{t}, \gamma_2[s/\underline{t}]) \\
\mathcal{PE}[\![\epsilon : consMN]\!]\rho\gamma = \textbf{let } (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\qquad\quad (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\qquad\quad s = \textbf{pair}(\epsilon, t_1, t_2) \\
\qquad\qquad\qquad\qquad\quad \textbf{in } (^{\langle\langle P_1,P_2,\underline{t}:s\rangle\rangle}\underline{t}, \gamma_2[s/\underline{t}]) \\
\mathcal{PE}[\![carM]\!]\rho\gamma = \textbf{let } (^{\langle\langle P\rangle\rangle}t, \gamma') = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\quad \textbf{in case } \gamma'(t) \textbf{ of} \\
\qquad\qquad\qquad\qquad \textbf{pair}(\epsilon, t_1, t_2) \;:\; \textbf{if } \epsilon \in mutable_{car} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then let } s = \textbf{car}(t) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } (^{\langle\langle P,\underline{t}':s\rangle\rangle}\underline{t}', \gamma'[s/\underline{t}']) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } (^{\langle\langle P\rangle\rangle}t_1, \gamma') \\
\qquad\qquad\qquad\qquad \textbf{otherwise} \;:\; \textbf{let } s = \textbf{car}(t) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } (^{\langle\langle P,\underline{t}':s\rangle\rangle}\underline{t}', \gamma'[s/\underline{t}']) \\
\mathcal{PE}[\![setcarMN]\!]\rho\gamma = \textbf{let } (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\qquad\quad (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\qquad\quad s = \textbf{setcar}(t_1, t_2) \\
\qquad\qquad\qquad\qquad\quad \textbf{in } (^{\langle\langle P_1,P_2,\underline{t}:s\rangle\rangle}\underline{t}, \gamma_2[s/\underline{t}]) \\
\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma = \textbf{let } (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\qquad\quad\; (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![M']\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\quad \textbf{in if } \gamma_2(t_1) \text{ and } \gamma_2(t_2) \text{ are known} \\
\qquad\qquad\qquad\qquad\qquad \textbf{then if } t_1 = t_2 \textbf{ then } ^{\langle\langle P_1,P_2\rangle\rangle}\mathcal{PE}[\![N]\!]\rho\gamma_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } ^{\langle\langle P_1,P_2\rangle\rangle}\mathcal{PE}[\![N']\!]\rho\gamma_2 \\
\qquad\qquad\qquad\qquad\qquad \textbf{else let } (^{\langle\langle P_3\rangle\rangle}t_3, \gamma_3) = \mathcal{PE}[\![N]\!]\rho\gamma_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\; (^{\langle\langle P_4\rangle\rangle}t_4, \gamma_4) = \mathcal{PE}[\![N']\!]\rho\gamma_3 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\; s = \textbf{eq}(t_1, t_2, ^{\langle\langle P_3\rangle\rangle}t_3, ^{\langle\langle P_4\rangle\rangle}t_4) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \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

$$\mathcal{PD}[\![^{\langle\langle\rangle\rangle}t]\!] = t$$
$$\mathcal{PD}[\![^{\langle\langle t_1 : s_1, \ldots\rangle\rangle}t]\!] = let\ t_1 = \mathcal{D}[\![s_1]\!]\ in\ \mathcal{PD}[\![^{\langle\langle \ldots\rangle\rangle}t]\!]$$

$$\mathcal{D}[\![\mathbf{closure}(\eta, _, _, _, t, {}^{\langle\langle P\rangle\rangle}t')]\!] = (\eta : \lambda t.\mathcal{PD}[\![^{\langle\langle P\rangle\rangle}t']\!])$$
$$\mathcal{D}[\![\mathbf{apply}(t_1, t_2)]\!] = (t_1\ t_2)$$
$$\mathcal{D}[\![\mathbf{pair}(\epsilon, t_1, t_2)]\!] = (\epsilon : cons\ t_1\ t_2)$$
$$\mathcal{D}[\![\mathbf{car}(t)]\!] = (car\ t)$$
$$\mathcal{D}[\![\mathbf{setcar}(t_1, t_2)]\!] = (setcar\ t_1\ t_2)$$
$$\mathcal{D}[\![\mathbf{eq}(t_1, t_2, {}^{\langle\langle P_3\rangle\rangle}t_3, {}^{\langle\langle P_4\rangle\rangle}t_4)]\!] = (eq\ t_1\ t_2\ \mathcal{PD}[\![^{\langle\langle P_3\rangle\rangle}t_3]\!]\ \mathcal{PD}[\![^{\langle\langle P_4\rangle\rangle}t_4]\!])$$

Figure 5: Codification

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.

4.4 Codification

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 codification 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 show the outline of the correctness proof, which states that the partial evaluator presented in the previous section preserves the store semantics of the input program. The actual proof is presented in the Appendix.

Because our partial evaluator consists of two parts (specialization $\mathcal{PE}$ and codification $\mathcal{PD}$), 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: $\mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle}t]\!]\sigma = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P\rangle\rangle}t]\!]]\!]\sigma$, 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 codify the PSval $^{\langle\langle P\rangle\rangle}t$. After that, we prove the main theorem stating that $\mathcal{PE}$ preserves the semantics: if γ respects σ, $\mathcal{E}_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$. The proof is again by induction on the number of $\mathcal{PE}$ applications required to partially evaluate M.

6 Extensions

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. Given a mechanism for handling side-effects correctly, it is not difficult to incorporate other features of functional

languages. The extensions presented below 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.

Recursion and Termination Detection Given that mutable cells are residualized correctly as unknown values, the presence of side-effects has no influence on the termination detection. We can introduce recursion and any conventional termination detection algorithm without problems. In our current implementation, we employ user annotation (or *filters*[7, 9]) to avoid non-termination.

λ-expressions with Variadic Parameters As 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*, *assq*, etc.) are provided as ordinary user-defined functions. We have not yet considered partial evaluation of *call/cc*.

Post-processing We also perform code beautification in the post-processing phase for the better readability. It includes: folding a sequence of `let` expressions into a `let*` expression and inlining expressions that are used exactly once[4].

7 Specializing an Interpreter

In this section, we show an example specialization of a larger program: an interpreter for a subset of Scheme with variable assignments `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.

[4]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 ($setcar\,xM$) 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)))))
   (eval-lambda
    (lambda (exp env)
      (let ((lambda-params (car (cdr exp)))
            (lambda-body (cdr (cdr exp))))
        (lambda args
          (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))))))
   ...)
   ; main program
   (base-eval '(lambda (x) (+ x (begin (set! x 3) x)))
              (list (cons '+ . +) (cons '- . -) ...)))
```

Figure 6: The Core Part of the Scheme Interpreter

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 environments 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.

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 same information can be obtained using other approaches. Heintze[12] deduces a set of possible values using set-based analysis for higher-order λ-calculus with updatable arrays. Landi, Ryder, and Zhang[14], Choi, Burke, and Carini[5], and Emami, Ghiya, and Hendren[11] presented interprocedural context-sensitive analyses for C-like programs, which is more accurate than our analysis. Ashley[2] showed a flow analysis for higher-order languages with side-effects, which can tune the accuracy of the analysis.

Since the information we need is mutable cells, we could also use effect analysis by Talpin and Jouvelot[22]. Based on a type system and constraint solving, it provides us with a side-effects that can occur on each location.

8.2 Partial Evaluation

There are several partial evaluators for Scheme. Among them, Similix[4] is a publicly available offline partial evaluator. Although it can deal with I/O operations and restricted 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 provides wider use of variable assignments than Similix, but does not handle destructive updates of data structures, either.

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. We also use names to simplify the codification phase considerably.

Partial evaluation of side-effects has been considered in imperative languages, such as Pascal[17], Fortran[3], 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. To improve results further by reducing assignments is our future work.

8.3 Preactions vs. Let-expressions in Similix

Preactions used in this paper have close relationship to the 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 difference arises when those 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 identities) and thus residualized as preactions. Treating them unknown results in a very poor specialization.

However, it might be possible to use let-expressions in place of preactions, if we residualized them in let-expressions and used them at the same time. We manipulate two data for one expression: its code (or name) and its value. Although it would spoil the efficiency of offline partial evaluators, it would still be possible to make an offline partial evaluator using let-expressions, which we think is interesting future work.

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. Partially static data structures are naturally realized by residualizing them as preactions and using them at the same time in the subsequent partial evaluation. The correctness of the partial evaluator is proved. Based on the framework, we have constructed a partial evaluator for almost full Scheme, which is powerful enough to specialize an interpreter written using side-effects.

References

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

[2] Ashley, J. M. "A Practical and Flexible Flow Analysis for Higher-Order Languages," *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 184–194 (January 1996).

[3] 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).

[4] 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).

[5] Choi, J. D., M. Burke, and P. Carini "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects," *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 232–245 (January 1993).

[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] Emami, M., R. Ghiya, and L. J. Hendren "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pp. 242–256 (June 1994).

[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] Landi, W., B. G. Ryder, and S. Zhang "Interprocedural Modification Side Effect Analysis With Pointer Aliasing," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pp. 56–67 (June 1993).

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

[16] 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).

[17] 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).

[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] 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).

[22] Talpin, J. P., and P. Jouvelot "Polymorphic type, region and effect inference," *Journal of Functional Programming*, Vol. 2, No. 3, pp. 245–271, Cambridge University Press (July 1992).

[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).

A Correctness of the Partial Evaluator

In this appendix, we show that the partial evaluator presented in Section 4 preserves the semantics of the input program. We first present the store semantics $\mathcal{E}$ to be preserved as well as the partial evaluator $\mathcal{PE}$. For a simpler proof, the partial evaluator we present here is slightly modified from the partial evaluator shown in Section 4. Then the correctness proof is given. Because our partial evaluator consists of two parts (specialization $\mathcal{PE}$ and codification $\mathcal{PD}$), the correctness proof is also divided into two parts: the correctness of $\mathcal{PE}$ and $\mathcal{PD}$.

A.1 Store Semantics $\mathcal{E}$

The semantics of the core language (plus *let* expression) is shown in Figure 7. It is written in the standard store-passing style. (It is based on R^4RS[6], but written in the direct style,

$$
\begin{array}{rcrcl}
 & & \mathcal{E} & : & Exp \to Env \to Store \to (Value \times Store) \\
\rho & \in & Env & : & Var \to Name \\
\sigma & \in & Store & : & Name \to Value \\
 & & Value & : & Cell \text{ or } Fun \\
 & & Cell & : & \langle ConsPoint, Name, Name \rangle \\
 & & Fun & : & \langle ClosurePoint, Name, Arg \to Store \to (Value \times Store) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![x]\!]\rho\sigma = (\sigma(\rho(x)), \sigma) \\
\mathcal{E}[\![M]\!]\rho\sigma = \textbf{let } (v, \sigma') = \mathcal{E}'[\![M]\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\hat{\underline{t}}])
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}'[\![\eta : \lambda x.M]\!]\rho\sigma = (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]\rangle, \sigma[*/\underline{t}]) \\
\mathcal{E}'[\![MN]\!]\rho\sigma = \textbf{let } (\langle \eta, t, m\rangle, \sigma_1) = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad (n, \sigma_2) = \mathcal{E}[\![N]\!]\rho\sigma_1 \\
\qquad\qquad\qquad \textbf{in } mn\sigma_2 \\
\mathcal{E}'[\![\epsilon : consMN]\!]\rho\sigma = \textbf{let } (m, \sigma_1) = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad (n, \sigma_2) = \mathcal{E}[\![N]\!]\rho\sigma_1 \\
\qquad\qquad\qquad\qquad\qquad \textbf{in } (\langle \epsilon, \underline{t_m}, \underline{t_n}\rangle, \sigma_2[m/\underline{t_m}, n/\underline{t_n}]) \\
\mathcal{E}'[\![car M]\!]\rho\sigma = \textbf{let } (\langle \epsilon, t_m, t_n\rangle, \sigma') = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad \textbf{in } (\sigma'(t_m), \sigma') \\
\mathcal{E}'[\![setcar MN]\!]\rho\sigma = \textbf{let } (\langle \epsilon, t_m, t_n\rangle, \sigma_1) = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad (n, \sigma_2) = \mathcal{E}[\![N]\!]\rho\sigma_1 \\
\qquad\qquad\qquad\qquad\qquad \textbf{in } (n, \sigma_2[n/t_m]) \\
\mathcal{E}'[\![eqMM'NN']\!]\rho\sigma = \textbf{let } (m, \sigma_1) = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (m', \sigma_2) = \mathcal{E}[\![M']\!]\rho\sigma_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in if } m = m' = \langle \epsilon, t_1, t_2\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } \mathcal{E}[\![N]\!]\rho\sigma_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else if } m = m' = \langle \eta, t_3, _\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } \mathcal{E}[\![N]\!]\rho\sigma_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathcal{E}[\![N']\!]\rho\sigma_2
\end{array}
$$

$$
\mathcal{E}[\![let\ x = M\ in\ N]\!]\rho\sigma = \textbf{let } (v, \sigma') = \mathcal{E}'[\![M]\!]\rho\sigma \textbf{ in } \mathcal{E}[\![N]\!]\rho[\underline{t'}/x]\sigma'[v/\underline{t'}]
$$

Figure 7: Standard Semantics of the Core Language

rather than the continuation passing style.) Given an expression, the semantic function $\mathcal{E}$ evaluates it under an environment ρ and a store σ, and returns a result value together with a possibly modified store.

To make the correctness proof simple, we store all the values of subexpressions of an input program into the store, as shown in the rule for $\mathcal{E}[\![M]\!]\rho\sigma$ (the second rule in Figure 7). We introduce a fresh hatted name $\underline{\hat{t}}$ and bind it the value of M. The hatted names correspond to the names introduced in preactions. Because hatted names are freshly introduced and will never be accessed afterwards, they can be safely ignored during standard evaluation.

Denotable values in the core language is either a λ-abstraction or a cons cell. They are respectively represented as a triplet of the form $\langle \eta, t, \lambda x \ldots \rangle$ or $\langle \epsilon, t_m, t_n \rangle$, where t, t_m, and t_n are *names* or *locations*.

A function value $\langle \eta, t, \lambda x \ldots \rangle$ indicates that a λ-expression with a closure point η is allocated at the location t. The equality of λ-abstractions are judged by the equality of their allocated locations. Since the name t in $\langle \eta, t, \lambda x \ldots \rangle$ is used only for the equality of functions, the value of t in a store σ is irrelevant and will never be looked up. In the rule for λ-expressions, we explicitly indicate this by binding $\sigma(t)$ to $*$.

A cons cell $\langle \epsilon, t_m, t_n \rangle$ is constructed by a *cons* expression with a cons point ϵ. It holds locations for its *car* and *cdr* part. Two cons cells $\langle \epsilon, t_m, t_n \rangle$ and $\langle \epsilon', t'_m, t'_n \rangle$ are equal when $t_m = t'_m$ and $t_n = t'_n$. (In this case, ϵ and ϵ' are always the same.)

As was in the rules for $\mathcal{PE}$, names are underlined when they are intended to be fresh names. Fresh names are introduced in three ways (excluding hatted names mentioned above): when a λ-expression is allocated ($\underline{t}$ in the rule for λ-expression), when a λ-abstraction is applied and its bound variable is allocated ($\underline{t'}$ in the rule for λ-expression), and when a cons cell is allocated ($\underline{t_m}$ and $\underline{t_n}$ in the rule for cons cells). We use '_' when we do not care about its value. We say a name is *mutable* if it is updated by *setcar* or *setcdr*. Otherwise, it is *immutable*.

A.2 The Partial Evaluator $\mathcal{PE}$

We re-show the partial evaluator $\mathcal{PE}$ in Figure 8. It is modified from the original partial evaluator presented in Figure 4 in three ways to simplify the correctness proof. First, we renamed fresh names $\underline{t}$ introduced in preactions to fresh hatted names $\underline{\hat{t}}$. These names corresponds to the hatted names in standard semantics. The second change is in the rule for *car*. When $\gamma'(t)$ is a known pair: $\mathbf{pair}(\epsilon, t_1, t_2)$ and $\epsilon \notin \mathit{mutable}_{car}$, we say $\mathbf{let}\ s = \mathbf{car}(t)\ \mathbf{in}\ ({}^{\langle\langle P, \underline{\hat{t'}} : s \rangle\rangle} t_1, \gamma'[s/\underline{\hat{t'}}])$ instead of $({}^{\langle\langle P \rangle\rangle} t_1, \gamma')$. This is because all subexpressions are bound to hatted names in the standard semantics. We can safely ignore $\underline{\hat{t'}} : s$ and $[s/\underline{\hat{t'}}]$, since $\mathbf{car}(t)$ has no side-effects and $\underline{\hat{t'}}$ is fresh, which means that it will never be accessed afterwards. Thirdly, we expand the abbreviation ${}^{\langle\langle P \rangle\rangle}\mathcal{PE}[\![M]\!]\rho\sigma$ in the rules for applications and *eq*. Here, we introduce hatted names for the same reason mentioned above for *car*. We say $({}^{\langle\langle P_1, P_2, \underline{\hat{t}} : s \rangle\rangle} t_3, \gamma_3[s/\underline{\hat{t}}])$ instead of $({}^{\langle\langle P_1, P_2, P_3 \rangle\rangle} t_3, \gamma_3)$. The semantics of these two are the same except for the allocation of $\underline{\hat{t}}$, because the evaluation of $\underline{\hat{t}} : s$ (where $s = \mathbf{apply}(t_1, t_2)$ for the application rule) makes the same side-effects as P_3.

We note that γ plays a role only when its bound value is a known Sval. The places where γ is used are the rules for applications, *car*, and *eq*. In all cases, γ is used to see if its value is known. When the value is a pair, its component might be accessed. When it is a closure, it might be applied.

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

$$
\begin{array}{l}
\mathcal{PE}[\![x]\!]\rho\gamma = (^{\langle\langle\rangle\rangle}\rho(x), \gamma) \\
\mathcal{PE}[\![\eta : \lambda x.M]\!]\rho\gamma = \mathbf{let}\ (^{\langle\langle P\rangle\rangle}t'', \gamma') = \mathcal{PE}[\![M]\!]\rho[\underline{t}'/x]\gamma[\mathbf{var}(\underline{t}')/\underline{t}'] \\
\qquad\qquad\qquad\qquad s = \mathbf{closure}(\eta, x, M, \rho, \underline{t}', {}^{\langle\langle P\rangle\rangle}\, t'') \\
\qquad\qquad\qquad\quad \mathbf{in}\ (^{\langle\langle \underline{t}:s\rangle\rangle}\hat{\underline{t}}, \gamma[s/\hat{\underline{t}}]) \\
\mathcal{PE}[\![MN]\!]\rho\gamma = \mathbf{let}\ (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\quad (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\quad \mathbf{in\ case}\ \gamma_2(t_1)\ \mathbf{of} \\
\qquad\qquad\qquad \mathbf{closure}(\eta, x, M', \rho', _, _) \ : \ \boxed{\begin{array}{l}\mathbf{let}\ (^{\langle\langle P_3\rangle\rangle}t_3, \gamma_3) = \mathcal{PE}[\![M']\!]\rho'[t_2/x]\gamma_2 \\ \qquad s = \mathbf{apply}(t_1, t_2) \\ \mathbf{in}\ (^{\langle\langle P_1, P_2, \underline{t}:s\rangle\rangle}t_3, \gamma_3[s/\hat{\underline{t}}])\end{array}} \\
\qquad\qquad\qquad \mathbf{otherwise} \qquad\qquad : \ \begin{array}{l}\mathbf{let}\ s = \mathbf{apply}(t_1, t_2) \\ \mathbf{in}\ (^{\langle\langle P_1, P_2, \underline{t}:s\rangle\rangle}\hat{\underline{t}}, \gamma_2[s/\hat{\underline{t}}])\end{array}
\end{array}
$$

Figure 8: Modified Partial Evaluator (Part 1)

A.3 Semantics of PSval $\mathcal{E}_{PS}$ and the Correctness of $\mathcal{PD}$

In this section, we prove the correctness of the codification function $\mathcal{PD}$. The correctness of the specialization function $\mathcal{PE}$ is proved in the next section. The correctness of the codification function $\mathcal{PD}$ is proved with respect to the store semantics $\mathcal{E}$ and the semantics of PSval $\mathcal{E}_{PS}$. Here, we define $\mathcal{E}_{PS}$ in such a way that the semantics of PSval is given by the semantics of its codified program. Figure 9 shows the definition of $\mathcal{E}_{PS}$, which is quite similar to the store semantics.

Theorem 1 (Correctness of $\mathcal{PD}$) $\mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle}t]\!]\sigma = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P\rangle\rangle}t]\!]]\!]\rho_{id}\sigma$.

Here, ρ_{id} is an identity environment $\lambda t.t$. Because codified programs use freshly introduced names (or locations) as variables, we do not need environments. The theorem says that if $^{\langle\langle P\rangle\rangle}t$ evaluates to v and modifies σ to σ', then its codified program evaluates to the same v and modifies σ to the same σ' (modulo renaming). The proof is by induction on the depth of $\mathcal{PD}$ application required to codify $^{\langle\langle P\rangle\rangle}t$.

Proof: By induction on the depth of $\mathcal{PD}$ application.

(Base Case) Since $\mathcal{PD}$ is applied only once, the PSval has the form $^{\langle\langle\rangle\rangle}t$.

$$rhs = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle\rangle\rangle}t]\!]]\!]\rho_{id}\sigma = \mathcal{E}[\![t]\!]\rho_{id}\sigma = (\sigma(\rho_{id}(t)), \sigma) = (\sigma(t), \sigma) = \mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle}t]\!]\sigma = lhs$$

(Induction Step) Since $\mathcal{PD}$ is applied more than once, the PSval has a sequence of pre-action whose length is more than one. Let $\langle\langle P\rangle\rangle = \langle\langle t_1 : s_1, P'\rangle\rangle$.

$$lhs \quad = \quad \mathcal{E}_{PS}[\![^{\langle\langle t_1:s_1, P'\rangle\rangle}t]\!]\sigma$$

$$
\begin{array}{l}
\mathcal{PE}[\![\epsilon : consMN]\!]\rho\gamma = \textbf{let } (^{\langle\!\langle P_1 \rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\quad (^{\langle\!\langle P_2 \rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\quad s = \textbf{pair}(\epsilon, t_1, t_2) \\
\qquad\qquad\qquad\qquad \textbf{in } (^{\langle\!\langle P_1, P_2, \underline{\hat{t}} : s \rangle\!\rangle} \underline{\hat{t}}, \gamma_2[s/\underline{\hat{t}}]) \\
\mathcal{PE}[\![carM]\!]\rho\gamma = \textbf{let } (^{\langle\!\langle P \rangle\!\rangle} t, \gamma') = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad \textbf{in case } \gamma'(t) \textbf{ of} \\
\qquad\qquad\qquad\qquad \textbf{pair}(\epsilon, t_1, t_2) \quad : \quad \textbf{if } \epsilon \in mutable_{car} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then let } s = \textbf{car}(t) \textbf{ in } (^{\langle\!\langle P, \underline{\hat{t}'} : s \rangle\!\rangle} \underline{\hat{t}'}, \gamma'[s/\underline{\hat{t}'}]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \boxed{\textbf{let } s = \textbf{car}(t) \textbf{ in } (^{\langle\!\langle P, \underline{\hat{t}'} : s \rangle\!\rangle} t_1, \gamma'[s/\underline{\hat{t}'}])} \\
\qquad\qquad\qquad\qquad \textbf{otherwise} \quad : \quad \textbf{let } s = \textbf{car}(t) \textbf{ in } (^{\langle\!\langle P, \underline{\hat{t}'} : s \rangle\!\rangle} \underline{\hat{t}'}, \gamma'[s/\underline{\hat{t}'}]) \\
\mathcal{PE}[\![setcarMN]\!]\rho\gamma = \textbf{let } (^{\langle\!\langle P_1 \rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\quad (^{\langle\!\langle P_2 \rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\quad s = \textbf{setcar}(t_1, t_2) \\
\qquad\qquad\qquad\qquad \textbf{in } (^{\langle\!\langle P_1, P_2, \underline{\hat{t}} : s \rangle\!\rangle} \underline{\hat{t}}, \gamma_2[s/\underline{\hat{t}}]) \\
\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma = \textbf{let } (^{\langle\!\langle P_1 \rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\quad\ (^{\langle\!\langle P_2 \rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}[\![M']\!]\rho\gamma_1 \\
\qquad\qquad\qquad\quad \textbf{in if } \gamma_2(t_1) \text{ and } \gamma_2(t_2) \text{ are known} \\
\qquad\qquad\qquad\qquad \textbf{then if } t_1 = t_2 \textbf{ then } \boxed{\begin{array}{l}\textbf{let } (^{\langle\!\langle P_3 \rangle\!\rangle} t_3, \gamma_3) = \mathcal{PE}[\![N]\!]\rho\gamma_2 \\ \qquad s = \textbf{eq}(t_1, t_2, ^{\langle\!\langle P_3 \rangle\!\rangle} t_3, _) \\ \textbf{in } (^{\langle\!\langle P_1, P_2, \underline{\hat{t}} : s \rangle\!\rangle} t_3, \gamma_3[s/\underline{\hat{t}}])\end{array}} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \boxed{\begin{array}{l}\textbf{let } (^{\langle\!\langle P_4 \rangle\!\rangle} t_4, \gamma_4) = \mathcal{PE}[\![N']\!]\rho\gamma_2 \\ \qquad s = \textbf{eq}(t_1, t_2, _, ^{\langle\!\langle P_4 \rangle\!\rangle} t_4) \\ \textbf{in } (^{\langle\!\langle P_1, P_2, \underline{\hat{t}} : s \rangle\!\rangle} t_4, \gamma_3[s/\underline{\hat{t}}])\end{array}} \\
\qquad\qquad\qquad\qquad \textbf{else let } (^{\langle\!\langle P_3 \rangle\!\rangle} t_3, \gamma_3) = \mathcal{PE}[\![N]\!]\rho\gamma_2 \\
\qquad\qquad\qquad\qquad\qquad\quad\ (^{\langle\!\langle P_4 \rangle\!\rangle} t_4, \gamma_4) = \mathcal{PE}[\![N']\!]\rho\gamma_3 \\
\qquad\qquad\qquad\qquad\qquad\quad\ s = \textbf{eq}(t_1, t_2, ^{\langle\!\langle P_3 \rangle\!\rangle} t_3, ^{\langle\!\langle P_4 \rangle\!\rangle} t_4) \\
\qquad\qquad\qquad\qquad\quad \textbf{in } (^{\langle\!\langle P_1, P_2, \underline{\hat{t}} : s \rangle\!\rangle} \underline{\hat{t}}, \gamma_4[s/\underline{\hat{t}}])
\end{array}
$$

Figure 8: Modified Partial Evaluator (Part 2)

$$
\begin{array}{l}
\mathcal{E}_{PS} : \mathit{PSval} \to \mathit{Store} \to (\mathit{Value} \times \mathit{Store}) \\
\mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle} t]\!]\sigma = (\sigma(t), \sigma) \\
\mathcal{E}_{PS}[\![^{\langle\langle t_1 : s_1, \ldots\rangle\rangle} t]\!]\sigma = \textbf{let } (v', \sigma') = \mathcal{E}_S[\![s_1]\!]\sigma \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle \ldots\rangle\rangle} t]\!]\sigma'[v'/t_1] \\
\\
\mathcal{E}_S : \mathit{Sval} \to \mathit{Store} \to (\mathit{Value} \times \mathit{Store}) \\
\mathcal{E}_S[\![\textbf{closure}(\eta, _, _, _, t', ^{\langle\langle P\rangle\rangle} t'')]\!]\sigma = (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle} t'']\!]\sigma'[x'/t']\rangle, \sigma[*/\underline{t}]) \\
\mathcal{E}_S[\![\textbf{apply}(t_1, t_2)]\!]\sigma = \textbf{let } \langle \eta, t, m\rangle = \sigma(t_1) \textbf{ in } m\sigma(t_2)\sigma \\
\mathcal{E}_S[\![\textbf{pair}(\epsilon, t_1, t_2)]\!]\sigma = (\langle \epsilon, \underline{t_m}, \underline{t_n}\rangle, \sigma[\sigma(t_1)/\underline{t_m}, \sigma(t_2)/\underline{t_n}]) \\
\mathcal{E}_S[\![\textbf{car}(t)]\!]\sigma = \textbf{let } \langle \epsilon, t_m, t_n\rangle = \sigma(t) \textbf{ in } (\sigma(t_m), \sigma) \\
\mathcal{E}_S[\![\textbf{setcar}(t_1, t_2)]\!]\sigma = \textbf{let } \langle \epsilon, t_m, t_n\rangle = \sigma(t_1) \textbf{ in } (\sigma(t_2), \sigma[\sigma(t_2)/t_m]) \\
\mathcal{E}_S[\![\textbf{eq}(t_1, t_2, ^{\langle\langle P_3\rangle\rangle} t_3, ^{\langle\langle P_4\rangle\rangle} t_4)]\!]\sigma = \textbf{if } \sigma(t_1) = \sigma(t_2) = \langle \epsilon, t_m, t_n\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } \mathcal{E}_{PS}[\![^{\langle\langle P_3\rangle\rangle} t_3]\!]\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else if } \sigma(t_1) = \sigma(t_2) = \langle \eta, t', _\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } \mathcal{E}_{PS}[\![^{\langle\langle P_3\rangle\rangle} t_3]\!]\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathcal{E}_{PS}[\![^{\langle\langle P_4\rangle\rangle} t_4]\!]\sigma
\end{array}
$$

Figure 9: Semantics of PSval $\mathcal{E}_{PS}$

$$
\begin{array}{rcl}
 & = & \textbf{let } (v', \sigma') = \mathcal{E}_S[\![s_1]\!]\sigma \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle P'\rangle\rangle} t]\!]\sigma'[v'/t_1] \\
\mathit{rhs} & = & \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle t_1 : s_1, P'\rangle\rangle} t]\!]]\!]\rho_{id}\sigma \\
 & = & \mathcal{E}[\![\mathit{let } t_1 = \mathcal{D}[\![s_1]\!] \mathit{ in } \mathcal{PD}[\![^{\langle\langle P'\rangle\rangle} t]\!]]\!]\rho_{id}\sigma \\
 & = & \textbf{let } (v'', \sigma'') = \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \textbf{ in } \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P'\rangle\rangle} t]\!]]\!]\rho_{id}[\underline{t'}/t_1]\sigma''[v''/\underline{t'}] \\
 & = & \textbf{let } (v'', \sigma'') = \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \textbf{ in } \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P'\rangle\rangle} t]\!]]\!]\rho_{id}\sigma''[v''/t_1]
\end{array}
$$

Since t_1 is chosen freshly at specialization time, we can take $\underline{t'}$ as t_1, keeping the environment to be ρ_{id}. If we can prove $\mathcal{E}_S[\![s_1]\!]\sigma = \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma$ (and hence $v' = v''$ and $\sigma' = \sigma''$), we are done:

$$
\begin{array}{rcl}
\mathit{rhs} & = & \textbf{let } (v', \sigma') = \mathcal{E}_S[\![s_1]\!]\sigma \textbf{ in } \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P'\rangle\rangle} t]\!]]\!]\rho_{id}\sigma'[v'/t_1] \\
 & = & \textbf{let } (v', \sigma') = \mathcal{E}_S[\![s_1]\!]\sigma \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle P'\rangle\rangle} t]\!]\sigma'[v'/t_1] \\
 & = & \mathit{lhs}
\end{array}
$$

using the induction hypothesis. Thus, we now prove $\mathcal{E}_S[\![s_1]\!]\sigma = \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma$ by the case analysis of s_1.

(Case for $s_1 = \textbf{closure}(\eta, _, _, _, t', ^{\langle\langle P''\rangle\rangle} t'')$)

$$
\begin{array}{rcl}
\mathit{lhs} & = & \mathcal{E}_S[\![s_1]\!]\sigma \\
 & = & (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![^{\langle\langle P''\rangle\rangle} t'']\!]\sigma'[x'/t']\rangle, \sigma[*/\underline{t}]) \\
\mathit{rhs} & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\
 & = & \mathcal{E}'[\![\eta : \lambda t'.\mathcal{PD}[\![^{\langle\langle P''\rangle\rangle} t'']\!]]\!]\rho_{id}\sigma \\
 & = & (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P''\rangle\rangle} t'']\!]]\!]\rho_{id}[\underline{t'}/t']\sigma'[x'/\underline{t'}]\rangle, \sigma[*/\underline{t}]) \\
 & = & (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P''\rangle\rangle} t'']\!]]\!]\rho_{id}\sigma'[x'/t']\rangle, \sigma[*/\underline{t}])
\end{array}
$$

Here, we take the same $\underline{t}$ as we took for *lhs*. We also choose $\underline{t'}$ to be t' to keep the environment ρ_{id}. From the induction hypothesis, $\mathcal{E}_{PS}[\![^{\langle\!\langle P'' \rangle\!\rangle} t'']\!]\sigma'[x'/t'] = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\!\langle P'' \rangle\!\rangle} t'']\!]]\!]\rho_{id}\sigma'[x'/t']$, and thus $lhs = rhs$.

(Case for $s_1 = \mathbf{apply}(t'_1, t'_2)$)

$$\begin{array}{rcl} lhs & = & \mathcal{E}_S[\![s_1]\!]\sigma \\ & = & \mathbf{let}\ \langle \eta, t, m \rangle = \sigma(t'_1)\ \mathbf{in}\ m\sigma(t'_2)\sigma \\ rhs & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\ & = & \mathcal{E}'[\![t'_1 t'_2]\!]\rho_{id}\sigma \\ & = & \mathbf{let}\ (\langle \eta, t, m \rangle, \sigma_1) = \mathcal{E}[\![t'_1]\!]\rho_{id}\sigma \\ & & \quad (n, \sigma_2) = \mathcal{E}[\![t'_2]\!]\rho_{id}\sigma_1 \\ & & \mathbf{in}\ mn\sigma_2 \end{array}$$

Because both t'_1 and t'_2 are variables, we have $\mathcal{E}[\![t'_1]\!]\rho_{id}\sigma = (\sigma(t'_1), \sigma)$, $\mathcal{E}[\![t'_2]\!]\rho_{id}\sigma = (\sigma(t'_2), \sigma)$, $\sigma = \sigma_1 = \sigma_2$, and $n = \sigma(t'_2)$. Thus, $lhs = rhs$.

(Case for $s_1 = \mathbf{pair}(\epsilon, t'_1, t'_2)$)

$$\begin{array}{rcl} lhs & = & \mathcal{E}_S[\![s_1]\!]\sigma \\ & = & (\langle \epsilon, t_m, t_n \rangle, \sigma[\sigma(t'_1)/\underline{t_m}, \sigma(t'_2)/\underline{t_n}]) \\ rhs & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\ & = & \mathcal{E}'[\![\epsilon : cons\ t'_1 t'_2]\!]\rho_{id}\sigma \\ & = & \mathbf{let}\ (m, \sigma_1) = \mathcal{E}[\![t'_1]\!]\rho_{id}\sigma \\ & & \quad (n, \sigma_2) = \mathcal{E}[\![t'_2]\!]\rho_{id}\sigma_1 \\ & & \mathbf{in}\ (\langle \epsilon, t_m, t_n \rangle, \sigma_2[m/\underline{t_m}, n/\underline{t_n}]) \end{array}$$

Here, we take the same $\underline{t_m}$ and $\underline{t_n}$ as we took for *lhs*. Because both t'_1 and t'_2 are variables, we have $\mathcal{E}[\![t'_1]\!]\rho_{id}\sigma = (\sigma(t'_1), \sigma)$, $\mathcal{E}[\![t'_2]\!]\rho_{id}\sigma = (\sigma(t'_2), \sigma)$, $\sigma = \sigma_1 = \sigma_2$, $m = \sigma(t'_1)$, and $n = \sigma(t'_2)$. Thus, $lhs = rhs$.

(Case for $s_1 = \mathbf{car}(t')$)

$$\begin{array}{rcl} lhs & = & \mathcal{E}_S[\![s_1]\!]\sigma \\ & = & \mathbf{let}\ \langle \epsilon, t_m, t_n \rangle = \sigma(t')\ \mathbf{in}\ (\sigma(t_m), \sigma) \\ rhs & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\ & = & \mathcal{E}'[\![car\ t']\!]\rho_{id}\sigma \\ & = & \mathbf{let}\ (\langle \epsilon, t_m, t_n \rangle, \sigma') = \mathcal{E}[\![t']\!]\rho_{id}\sigma\ \mathbf{in}\ (\sigma'(t_m), \sigma') \end{array}$$

Because t' is a variable, we have $\mathcal{E}[\![t']\!]\rho_{id}\sigma = (\sigma(t'), \sigma)$ and $\sigma' = \sigma$. Thus, $lhs = rhs$.

(Case for $s_1 = \mathbf{setcar}(t'_1, t'_2)$)

$$\begin{array}{rcl} lhs & = & \mathcal{E}_S[\![s_1]\!]\sigma \\ & = & \mathbf{let}\ \langle \epsilon, t_m, t_n \rangle = \sigma(t'_1)\ \mathbf{in}\ (\sigma(t'_2), \sigma[\sigma(t'_2)/t_m]) \\ rhs & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \end{array}$$

$$
\begin{array}{rcl}
 & = & \mathcal{E}'[\![setcar\ t'_1\ t'_2]\!]\rho_{id}\sigma \\
 & = & \textbf{let } (\langle \epsilon, t_m, t_n\rangle, \sigma_1) = \mathcal{E}[\![t'_1]\!]\rho_{id}\sigma \\
 & & \quad\ (n, \sigma_2) = \mathcal{E}[\![t'_2]\!]\rho_{id}\sigma_1 \\
 & & \textbf{in } (n, \sigma_2[n/t_m])
\end{array}
$$

Because both t'_1 and t'_2 are variables, we have $\mathcal{E}[\![t'_1]\!]\rho_{id}\sigma = (\sigma(t'_1), \sigma)$, $\mathcal{E}[\![t'_2]\!]\rho_{id}\sigma = (\sigma(t'_2), \sigma)$, $\sigma = \sigma_1 = \sigma_2$, and $n = \sigma(t'_2)$. Thus, $lhs = rhs$.

(Case for $s_1 = \mathbf{eq}(t'_1, t'_2, {}^{\langle\langle P'_3\rangle\rangle}t'_3, {}^{\langle\langle P'_4\rangle\rangle}t'_4)$)

$$
\begin{array}{rcl}
lhs & = & \mathcal{E}_S[\![s_1]\!]\sigma \\
 & = & \textbf{if } \sigma(t'_1) = \sigma(t'_2) = \langle \epsilon, t_m, t_n\rangle \\
 & & \textbf{then } \mathcal{E}_{PS}[\![{}^{\langle\langle P'_3\rangle\rangle}t'_3]\!]\sigma \\
 & & \textbf{else if } \sigma(t'_1) = \sigma(t'_2) = \langle \eta, t', _\rangle \\
 & & \quad \textbf{then } \mathcal{E}_{PS}[\![{}^{\langle\langle P'_3\rangle\rangle}t'_3]\!]\sigma \\
 & & \quad \textbf{else } \mathcal{E}_{PS}[\![{}^{\langle\langle P'_4\rangle\rangle}t'_4]\!]\sigma \\
rhs & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\
 & = & \mathcal{E}'[\![eq\ t'_1\ t'_2\ \mathcal{PD}[\![{}^{\langle\langle P'_3\rangle\rangle}t'_3]\!]\ \mathcal{PD}[\![{}^{\langle\langle P'_3\rangle\rangle}t'_3]\!]]\!]\rho_{id}\sigma \\
 & = & \textbf{let } (m, \sigma_1) = \mathcal{E}[\![t'_1]\!]\rho_{id}\sigma \\
 & & \quad\ (m', \sigma_2) = \mathcal{E}[\![t'_2]\!]\rho_{id}\sigma_1 \\
 & & \textbf{in if } m = m' = \langle \epsilon, t_m, t_n\rangle \\
 & & \quad \textbf{then } \mathcal{E}[\![\mathcal{PD}[\![{}^{\langle\langle P'_3\rangle\rangle}t'_3]\!]]\!]\rho_{id}\sigma_2 \\
 & & \quad \textbf{else if } m = m' = \langle \eta, t', _\rangle \\
 & & \quad\quad \textbf{then } \mathcal{E}[\![\mathcal{PD}[\![{}^{\langle\langle P'_3\rangle\rangle}t'_3]\!]]\!]\rho_{id}\sigma_2 \\
 & & \quad\quad \textbf{else } \mathcal{E}[\![\mathcal{PD}[\![{}^{\langle\langle P'_4\rangle\rangle}t'_4]\!]]\!]\rho_{id}\sigma_2
\end{array}
$$

Because both t'_1 and t'_2 are variables, we have $\mathcal{E}[\![t'_1]\!]\rho_{id}\sigma = (\sigma(t'_1), \sigma)$, $\mathcal{E}[\![t'_2]\!]\rho_{id}\sigma = (\sigma(t'_2), \sigma)$, $\sigma = \sigma_1 = \sigma_2$, $m = \sigma(t'_1)$, and $m' = \sigma(t'_2)$. From the induction hypotheses, we have $\mathcal{E}_{PS}[\![{}^{\langle\langle P'_3\rangle\rangle}t'_3]\!]\sigma = \mathcal{E}[\![\mathcal{PD}[\![{}^{\langle\langle P'_3\rangle\rangle}t'_3]\!]]\!]\rho_{id}\sigma$ and $\mathcal{E}_{PS}[\![{}^{\langle\langle P'_4\rangle\rangle}t'_4]\!]\sigma = \mathcal{E}[\![\mathcal{PD}[\![{}^{\langle\langle P'_4\rangle\rangle}t'_4]\!]]\!]\rho_{id}\sigma$. Thus, $lhs = rhs$. □

A.4 Correctness of the Partial Evaluator $\mathcal{PE}$

We now prove the correctness of the partial evaluator $\mathcal{PE}$. To prove the correctness of $\mathcal{PE}$, we need three lemmas. The first lemma gives us the way to obtain the semantics of composite PSval from the semantics of component PSvals. The proof is almost trivial.

Lemma 1 *If $\mathcal{E}_{PS}[\![{}^{\langle\langle P_1\rangle\rangle}t_1]\!]\sigma = (v_1, \sigma_1)$ and $\mathcal{E}_{PS}[\![{}^{\langle\langle P_2\rangle\rangle}t_2]\!]\sigma_1 = (v_2, \sigma_2)$, then $\mathcal{E}_{PS}[\![{}^{\langle\langle P_1, P_2\rangle\rangle}t_2]\!]\sigma = (v_2, \sigma_2)$.*

Proof: By induction on the length of P_1.

(Base Case) P_1 is a null sequence. From

$$(v_1, \sigma_1) = \mathcal{E}_{PS}[\![{}^{\langle\langle P_1\rangle\rangle}t_1]\!]\sigma = \mathcal{E}_{PS}[\![{}^{\langle\langle\rangle\rangle}t_1]\!]\sigma = (\sigma(t_1), \sigma),$$

we have $\sigma_1 = \sigma$ and $v_1 = \sigma(t_1)$. Then,

$$\mathcal{E}_{PS}[\![{}^{\langle\langle P_1, P_2\rangle\rangle}t_2]\!]\sigma = \mathcal{E}_{PS}[\![{}^{\langle\langle P_2\rangle\rangle}t_2]\!]\sigma = \mathcal{E}_{PS}[\![{}^{\langle\langle P_2\rangle\rangle}t_2]\!]\sigma_1 = (v_2, \sigma_2).$$

(Induction Step) P_1 is a sequence of at least length one. Let $\langle\langle P_1 \rangle\rangle = \langle\langle t : s, P_1' \rangle\rangle$. Assuming that $(v', \sigma') = \mathcal{E}_S[\![t : s]\!]\sigma$,

$$(v_1, \sigma_1) = \mathcal{E}_{PS}[\![^{\langle\langle P_1 \rangle\rangle} t_1]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle t:s, P_1' \rangle\rangle} t_1]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle P_1' \rangle\rangle} t_1]\!]\sigma'[v'/t].$$

Since $\mathcal{E}_{PS}[\![^{\langle\langle P_2 \rangle\rangle} t_2]\!]\sigma_1 = (v_2, \sigma_2)$, we have $\mathcal{E}_{PS}[\![^{\langle\langle P_1', P_2 \rangle\rangle} t_2]\!]\sigma'[v'/t] = (v_2, \sigma_2)$ from the induction hypothesis. Then,

$$\mathcal{E}_{PS}[\![^{\langle\langle P_1, P_2 \rangle\rangle} t_2]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle t:s, P_1', P_2 \rangle\rangle} t_2]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle P_1', P_2 \rangle\rangle} t_2]\!]\sigma'[v'/t] = (v_2, \sigma_2).$$

□

The second lemma is counter-intuitive in the presence of side-effects:

Lemma 2 *The names that appear in a PSval are immutable.*

Because accesses to mutable cells are forbidden, mutable names cannot appear in PSval.

Proof: The name t in a PSval $^{\langle\langle P \rangle\rangle} t'$ is mutable if t is t_m and is updated in the $\mathcal{E}_S$ rule for **setcar** (or **setcdr**) during the evaluation of $\mathcal{E}_{PS}[\![^{\langle\langle P \rangle\rangle} t']\!]\sigma$. Because t_m in the rule for **setcar** can be created only in the rule for **pair** freshly, the only possibility that the mutable name t_m appears in a PSval is by accessing it during partial evaluation. However, in the $\mathcal{PE}$ rule for *car*, pairs are accessed only when their *car* parts are immutable, and thus mutable names cannot appear in PSvals. □

The last lemma relates names to its values:

Lemma 3 *If* $\mathcal{E}_{PS}[\![^{\langle\langle P \rangle\rangle} t]\!]\sigma = (v, \sigma')$, *then* $\sigma'(t) = v$.

Proof: Formally, the proof is by induction on the length of P. However, we can easily see that the evaluation of $\mathcal{E}_{PS}[\![^{\langle\langle P \rangle\rangle} t]\!]\sigma$ will eventually encounter $\mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle} t]\!]\sigma'$, which returns $(\sigma'(t), \sigma')$. Thus, $\sigma'(t) = v$. □

To relate a store σ and a pe-store γ, we introduce a relation 'respect' between them.

Definition 1 *A pe-store* γ respects *a store* σ, *if the following holds for all* $t \in domain(\gamma)$.

- *if* $\gamma(t) = \mathbf{pair}(\epsilon, t_1, t_2)$, *then* $\sigma(t) = \langle \epsilon, t_m, t_n \rangle$, $\sigma(t_1) = \sigma(t_m)$, *and* $\sigma(t_2) = \sigma(t_n)$, *or* $\sigma(t)$ *is undefined.*
- *if* $\gamma(t) = \mathbf{closure}(\eta, x, M, \rho, _, _)$, *then* $\sigma(t) = \langle \eta, t'', \lambda x'.\lambda\sigma'.\mathcal{E}[\![M]\!]\rho[t'/x]\sigma'[x'/t'] \rangle$, *or* $\sigma(t)$ *is undefined.*

The relation 'respect' only requires that if $\gamma(t)$ is known, then $\sigma(t)$ has the same value. Since γ plays a role only when its bound value is known, we do not need to mention the other cases where $\gamma(t)$ is unknown. The reason for having the phrase "or $\sigma(t)$ is undefined" is that we have conditionals. Because the partial evaluator eagerly evaluates both branches of conditionals, some names are defined only in γ and not in σ.

We now show the correctness of $\mathcal{PE}$.

Theorem 2 (Correctness of $\mathcal{PE}$) *If* γ *respects* σ, $\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$.

Here, $\mathcal{E}'_{PS}$ takes the PSval part of the $\mathcal{PE}$'s return value: $\mathcal{E}'_{PS}[\![(^{\langle\langle P \rangle\rangle} t, \gamma)]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle P \rangle\rangle} t]\!]\sigma$. The theorem says that if M evaluates under ρ and σ to v and modifies σ to σ', then the result of partially evaluating M under the same ρ and γ evaluates to the same v and modifies σ to the same σ' (modulo renaming). The proof is by induction on the depth of $\mathcal{PE}$ application required to partially evaluate M.

Proof: By induction on the depth of $\mathcal{PE}$ application. In the proof, we prove simultaneously the following: if $\mathcal{PE}[\![M]\!]\rho\gamma = ({}^{\langle\langle P\rangle\rangle}t, \gamma')$ and $\mathcal{E}[\![M]\!]\rho\sigma = (v, \sigma')$, then γ' respects σ'.

(Base Case) Since $\mathcal{PE}$ is applied only once, M is a variable. Assume that γ respects σ.

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![x]\!]\rho\gamma]\!]\sigma = \mathcal{E}'_{PS}[\![({}^{\langle\langle\rangle\rangle}\rho(x), \gamma)]\!]\sigma = \mathcal{E}_{PS}[\![{}^{\langle\langle\rangle\rangle}\rho(x)]\!]\sigma = (\sigma(\rho(x)), \sigma) = \mathcal{E}[\![x]\!]\rho\sigma$$

As $\gamma' = \gamma$ and $\sigma' = \sigma$, γ' respects σ'.

(Induction Step) Case analysis on the form of M. It is a good idea to read from the case for $\epsilon : consMN$, since it gives a typical proof.

(Case for $\eta : \lambda x.M$)

$$\begin{aligned}
rhs &= \mathcal{E}[\![\eta : \lambda x.M]\!]\rho\sigma \\
&= \textbf{let } (v, \sigma') = \mathcal{E}'[\![\eta : \lambda x.M]\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\hat{\underline{t}}]) \\
&= \textbf{let } (v, \sigma') = (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]\rangle, \sigma[*/\underline{t}]) \textbf{ in } (v, \sigma'[v/\hat{\underline{t}}]) \\
&= \textbf{let } v = \langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]\rangle \textbf{ in } (v, \sigma[*/\underline{t}, v/\hat{\underline{t}}])
\end{aligned}$$

Let $\mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}] = (m, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho[\underline{t'}/x]\gamma[\textbf{var}(\underline{t'})/\underline{t'}] = ({}^{\langle\langle P\rangle\rangle}t'', \gamma_1)$. Because σ' is a store at the application of this closure, it is an extension of σ. From lemma 2, we have $\sigma(t) = \sigma'(t)$ for all $t \in domain(\gamma)$, and thus, $\gamma[\textbf{var}(\underline{t'})/\underline{t'}]$ respects $\sigma'[x'/\underline{t'}]$. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho[\underline{t'}/x]\gamma[\textbf{var}(\underline{t'})/\underline{t'}]]\!]\sigma'[x'/\underline{t'}] = \mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}] = (m, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t'') = m$.

$$\begin{aligned}
lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![\eta : \lambda x.M]\!]\rho\gamma]\!]\sigma \\
&= \mathcal{E}_{PS}[\![{}^{\langle\langle \hat{\underline{t}}:\textbf{closure}(\eta, x, M, \rho, \underline{t'}, {}^{\langle\langle P\rangle\rangle}t'')\rangle\rangle}\hat{\underline{t}}]\!]\sigma \\
&= \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\textbf{closure}(\eta, x, M, \rho, \underline{t'}, {}^{\langle\langle P\rangle\rangle}t'')]\!]\sigma \textbf{ in } \mathcal{E}_{PS}[\![{}^{\langle\langle\rangle\rangle}\hat{\underline{t}}]\!]\sigma'[v/\hat{\underline{t}}] \\
&= \textbf{let } (v, \sigma') = (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}]\rangle, \sigma[*/\underline{t}]) \textbf{ in } (v, \sigma'[v/\hat{\underline{t}}]) \\
&= \textbf{let } v = \langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}]\rangle \textbf{ in } (v, \sigma[*/\underline{t}, v/\hat{\underline{t}}])
\end{aligned}$$

where we take the same $\underline{t}$ and $\hat{\underline{t}}$ as chosen in *rhs*. Using the induction hypothesis, we have $lhs = rhs$.

The pe-store returned form $\mathcal{PE}[\![\eta : \lambda x.M]\!]\rho\gamma$ is $\gamma[\textbf{closure}(\eta, x, M, \rho, \underline{t'}, {}^{\langle\langle P\rangle\rangle}t'')/\hat{\underline{t}}]$. It respects $\sigma[*/\underline{t}, v/\hat{\underline{t}}] = \sigma[*/\underline{t}, \langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}]\rangle/\hat{\underline{t}}]$, since γ respects σ and

$$\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}] = \mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho[\underline{t'}/x]\gamma[\textbf{var}(\underline{t'})/\underline{t'}]]\!]\sigma'[x'/\underline{t'}] = \mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]$$

from the induction hypothesis.

(Case for MN) Let $\mathcal{E}[\![M]\!]\rho\sigma = (\langle \eta, t, m\rangle, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = ({}^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![^{\langle\langle P_1\rangle\rangle}t_1]\!]\sigma = (\langle\eta, t, m\rangle, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = \langle\eta, t, m\rangle$.

In the same way, let $\mathcal{E}[\![N]\!]\rho\sigma_1 = (n, \sigma_2)$ and $\mathcal{PE}[\![N]\!]\rho\gamma_1 = (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2)$. Since γ_1 respects σ_1, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N]\!]\rho\gamma_1]\!]\sigma_1 = \mathcal{E}[\![N]\!]\rho\sigma_1$$

we have $\mathcal{E}_{PS}[\![^{\langle\langle P_2\rangle\rangle}t_2]\!]\sigma_1 = (n, \sigma_2)$ and γ_2 respects σ_2. From lemma 3, we have $\sigma_2(t_2) = n$.

$$\begin{aligned} rhs &= \mathcal{E}[\![MN]\!]\rho\sigma \\ &= \textbf{let } (v, \sigma') = \mathcal{E}'[\![MN]\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = mn\sigma_2 \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \end{aligned}$$

Since t_1 is fresh and not defined in P_2, we have $\sigma_2(t_1) = \sigma_1(t_1) = \langle\eta, t, m\rangle$. Since γ_2 respects σ_2, if $\gamma_2(t_1)$ has the form: $\textbf{closure}(\eta, x, M', \rho', _, _)$, this means that $m = \lambda x'.\lambda\sigma'.\mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma'[x'/\underline{t'}]$.

$$\begin{aligned} rhs &= \textbf{let } (v, \sigma') = mn\sigma_2 \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = \mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma_2[n/\underline{t'}] \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \end{aligned}$$

In this case, we use another induction hypothesis. Let $\mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma_2[n/\underline{t'}] = (m', \sigma_3)$ and $\mathcal{PE}[\![M']\!]\rho'[\underline{t'}/x]\gamma_2 = (^{\langle\langle P_3\rangle\rangle}t_3, \gamma_3)$. Since γ_2 respects $\sigma_2[n/\underline{t'}]$, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M']\!]\rho'[\underline{t'}/x]\gamma[n/\underline{t'}]]\!]\sigma_2[n/\underline{t'}] = \mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma_2[n/\underline{t'}]$$

$\mathcal{E}_{PS}[\![^{\langle\langle P_3\rangle\rangle}t_3]\!]\sigma_3 = (m', \sigma_3)$ and γ_3 respects σ_3. From lemma 3, we have $\sigma_3(t_3) = m'$.

$$\begin{aligned} rhs &= \textbf{let } (v, \sigma') = \mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma_2[n/\underline{t'}] \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = (m', \sigma_3) \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= (m', \sigma_3[m'/\underline{\hat{t}}]) \\ lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![MN]\!]\rho\gamma]\!]\sigma \\ &= \mathcal{E}_{PS}[\![^{\langle\langle P_1, P_2, \underline{\hat{t}}:\textbf{apply}(t_1,t_2)\rangle\rangle}t_3]\!]\sigma \\ \mathcal{E}_{PS}[\![^{\langle\langle \underline{\hat{t}}:\textbf{apply}(t_1,t_2)\rangle\rangle}t_3]\!]\sigma_2 &= \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\textbf{apply}(t_1, t_2)]\!]\sigma_2 \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle}t_3]\!]\sigma'[v/\underline{\hat{t}}] \\ &= \textbf{let } (v, \sigma') = m\sigma_2(t_2)\sigma_2 \textbf{ in } (\sigma'[v/\underline{\hat{t}}](t_3), \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = mn\sigma_2 \textbf{ in } (\sigma'[v/\underline{\hat{t}}](t_3), \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = (m', \sigma_3) \textbf{ in } (\sigma'[v/\underline{\hat{t}}](t_3), \sigma'[v/\underline{\hat{t}}]) \\ &= (\sigma_3[m'/\underline{\hat{t}}](t_3), \sigma_3[m'/\underline{\hat{t}}]) \\ &= (m', \sigma_3[m'/\underline{\hat{t}}]) \end{aligned}$$

where we take the same $\underline{\hat{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have $lhs = rhs$.

The pe-store returned form $\mathcal{PE}[\![MN]\!]\rho\gamma$ in this case is $\gamma_3[\textbf{apply}(t_1, t_2)/\underline{\hat{t}}]$. It respects $\sigma_3[m'/\underline{\hat{t}}]$, since γ_3 respects σ_3.

If $\gamma_2(t_1)$ is not a **closure** Sval, it means that $\gamma_2(t_1)$ is unknown code. In this case,

$$\begin{aligned} lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![MN]\!]\rho\gamma]\!]\sigma \\ &= \mathcal{E}_{PS}[\![^{\langle\langle P_1, P_2, \underline{\hat{t}}:\textbf{apply}(t_1,t_2)\rangle\rangle}\underline{\hat{t}}]\!]\sigma \\ \mathcal{E}_{PS}[\![^{\langle\langle \underline{\hat{t}}:\textbf{apply}(t_1,t_2)\rangle\rangle}\underline{\hat{t}}]\!]\sigma_2 &= \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\textbf{apply}(t_1, t_2)]\!]\sigma_2 \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle}\underline{\hat{t}}]\!]\sigma'[v/\underline{\hat{t}}] \\ &= \textbf{let } (v, \sigma') = m\sigma_2(t_2)\sigma_2 \textbf{ in } (\sigma'[v/\underline{\hat{t}}](\underline{\hat{t}}), \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = mn\sigma_2 \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \end{aligned}$$

where we take the same $\hat{\underline{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have *lhs* = *rhs*.

The pe-store returned form $\mathcal{PE}[\![MN]\!]\rho\gamma$ in this case is $\gamma_2[\mathbf{apply}(t_1, t_2)/\hat{\underline{t}}]$. It respects $\sigma'[v/\hat{\underline{t}}]$, since γ_2 respects σ_2 and σ' is an extension of σ_2, where $\sigma'(t) = \sigma_2(t)$ for all $t \in domain(\gamma_2)$ from lemma 2.

(Case for $\epsilon : consMN$) Let $\mathcal{E}[\![M]\!]\rho\sigma = (m, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = ({}^{\langle\!\langle P_1 \rangle\!\rangle}t_1, \gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\!\langle P_1 \rangle\!\rangle}t_1]\!]\sigma = (m, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = m$.

In the same way, let $\mathcal{E}[\![N]\!]\rho\sigma_1 = (n, \sigma_2)$ and $\mathcal{PE}[\![N]\!]\rho\gamma_1 = ({}^{\langle\!\langle P_2 \rangle\!\rangle}t_2, \gamma_2)$. Since γ_1 respects σ_1, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N]\!]\rho\gamma_1]\!]\sigma_1 = \mathcal{E}[\![N]\!]\rho\sigma_1$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\!\langle P_2 \rangle\!\rangle}t_2]\!]\sigma_1 = (n, \sigma_2)$ and γ_2 respects σ_2. From lemma 3, we have $\sigma_2(t_2) = n$.

$$\begin{array}{rcl}
rhs & = & \mathcal{E}[\![\epsilon : consMN]\!]\rho\sigma \\
 & = & \mathbf{let}\ (v, \sigma') = \mathcal{E}'[\![\epsilon : consMN]\!]\rho\sigma\ \mathbf{in}\ (v, \sigma'[v/\hat{\underline{t}}]) \\
 & = & \mathbf{let}\ v = \langle \epsilon, \underline{t_m}, \underline{t_n} \rangle\ \mathbf{in}\ (v, \sigma_2[m/\underline{t_m}, n/\underline{t_n}, v/\hat{\underline{t}}]) \\
lhs & = & \mathcal{E}'_{PS}[\![\mathcal{PE}[\![\epsilon : consMN]\!]\rho\gamma]\!]\sigma \\
 & = & \mathcal{E}_{PS}[\![{}^{\langle\!\langle P_1, P_2, \hat{\underline{t}}:\mathbf{pair}(\epsilon, t_1, t_2) \rangle\!\rangle}\hat{\underline{t}}]\!]\sigma
\end{array}$$

Since t_1 is fresh and not defined in P_2, we have $\sigma_2(t_1) = \sigma_1(t_1) = m$, and thus

$$\begin{array}{rcl}
\mathcal{E}_{PS}[\![{}^{\langle\!\langle \hat{\underline{t}}:\mathbf{pair}(\epsilon, t_1, t_2) \rangle\!\rangle}\hat{\underline{t}}]\!]\sigma_2 & = & \mathbf{let}\ (v, \sigma') = \mathcal{E}_S[\![\mathbf{pair}(\epsilon, t_1, t_2)]\!]\sigma_2\ \mathbf{in}\ \mathcal{E}_{PS}[\![{}^{\langle\!\langle \rangle\!\rangle}\hat{\underline{t}}]\!]\sigma'[v/\hat{\underline{t}}] \\
 & = & \mathbf{let}\ (v, \sigma') = (\langle \epsilon, \underline{t_m}, \underline{t_n} \rangle, \sigma_2[\sigma_2(t_1)/\underline{t_m}, \sigma_2(t_2)/\underline{t_n}])\ \mathbf{in}\ (v, \sigma'[v/\hat{\underline{t}}]) \\
 & = & \mathbf{let}\ v = \langle \epsilon, \underline{t_m}, \underline{t_n} \rangle\ \mathbf{in}\ (v, \sigma_2[\sigma_2(t_1)/\underline{t_m}, \sigma_2(t_2)/\underline{t_n}, v/\hat{\underline{t}}]) \\
 & = & \mathbf{let}\ v = \langle \epsilon, \underline{t_m}, \underline{t_n} \rangle\ \mathbf{in}\ (v, \sigma_2[m/\underline{t_m}, n/\underline{t_n}, v/\hat{\underline{t}}])
\end{array}$$

where we take the same $\underline{t_m}$, $\underline{t_n}$, and $\hat{\underline{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have *lhs* = *rhs*.

The pe-store returned form $\mathcal{PE}[\![\epsilon : consMN]\!]\rho\gamma$ is $\gamma_2[\mathbf{pair}(\epsilon, t_1, t_2)/\hat{\underline{t}}]$. Because $\sigma_2(t_1) = m$ and $\sigma_2(t_2) = n$, and γ_2 respects σ_2, it respects $\sigma_2[m/\underline{t_m}, n/\underline{t_n}, \langle \epsilon, \underline{t_m}, \underline{t_n} \rangle/\hat{\underline{t}}]$.

(Case for $carM$) Let $\mathcal{E}[\![M]\!]\rho\sigma = (\langle \epsilon, t_m, t_n \rangle, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = ({}^{\langle\!\langle P_1 \rangle\!\rangle}t_1, \gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\!\langle P_1 \rangle\!\rangle}t_1]\!]\sigma = (\langle \epsilon, t_m, t_n \rangle, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = \langle \epsilon, t_m, t_n \rangle$.

$$\begin{array}{rcl}
rhs & = & \mathcal{E}[\![carM]\!]\rho\sigma \\
 & = & \mathbf{let}\ (v, \sigma') = \mathcal{E}'[\![carM]\!]\rho\sigma\ \mathbf{in}\ (v, \sigma'[v/\hat{\underline{t}}]) \\
 & = & (\sigma_1(t_m), \sigma_1[\sigma_1(t_m)/\hat{\underline{t}}])
\end{array}$$

Since γ respects σ and $\sigma_1(t_1) = \langle \epsilon, t_m, t_n \rangle$, if $\gamma_1(t_1)$ is a **pair** Sval, it has the form: $\mathbf{pair}(\epsilon, t_m, t_n)$. In this case, when $\epsilon \in mutable_{car}$,

$$\begin{aligned} lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![carM]\!]\rho\gamma]\!]\sigma \\ &= \mathcal{E}_{PS}[\![^{\langle\!\langle P_1, \underline{\hat{t}}:\mathbf{car}(t_1)\rangle\!\rangle}\underline{\hat{t}}]\!]\sigma \end{aligned}$$

$$\begin{aligned} \mathcal{E}_{PS}[\![^{\langle\!\langle \underline{\hat{t}}:\mathbf{car}(t_1)\rangle\!\rangle}\underline{\hat{t}}]\!]\sigma &= \mathbf{let}\ (v, \sigma') = \mathcal{E}_S[\![\mathbf{car}(t_1)]\!]\sigma_1\ \mathbf{in}\ \mathcal{E}_{PS}[\![^{\langle\!\langle\rangle\!\rangle}\underline{\hat{t}}]\!]\sigma'[v/\underline{\hat{t}}] \\ &= \mathbf{let}\ (v, \sigma') = (\sigma_1(t_m), \sigma_1)\ \mathbf{in}\ (v, \sigma'[v/\underline{\hat{t}}]) \\ &= (\sigma_1(t_m), \sigma_1[\sigma_1(t_m)/\underline{\hat{t}}]) \end{aligned}$$

where we take the same $\underline{\hat{t}}$ as chosen in *rhs*. Using lemma 1, we have $lhs = rhs$.

When $\epsilon \notin mutable_{car}$,

$$\begin{aligned} lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![carM]\!]\rho\gamma]\!]\sigma \\ &= \mathcal{E}_{PS}[\![^{\langle\!\langle P_1, \underline{\hat{t}}:\mathbf{car}(t_1)\rangle\!\rangle}t_m]\!]\sigma \end{aligned}$$

$$\begin{aligned} \mathcal{E}_{PS}[\![^{\langle\!\langle \underline{\hat{t}}:\mathbf{car}(t_1)\rangle\!\rangle}t_m]\!]\sigma &= \mathbf{let}\ (v, \sigma') = \mathcal{E}_S[\![\mathbf{car}(t_1)]\!]\sigma_1\ \mathbf{in}\ \mathcal{E}_{PS}[\![^{\langle\!\langle\rangle\!\rangle}t_m]\!]\sigma'[v/\underline{\hat{t}}] \\ &= \mathbf{let}\ (v, \sigma') = (\sigma_1(t_m), \sigma_1)\ \mathbf{in}\ (\sigma'[v/\underline{\hat{t}}](t_m), \sigma'[v/\underline{\hat{t}}]) \\ &= (\sigma_1(t_m), \sigma_1[\sigma_1(t_m)/\underline{\hat{t}}]) \end{aligned}$$

where we take the same $\underline{\hat{t}}$ as chosen in *rhs*. Using lemma 1, we have $lhs = rhs$.

If $\gamma_1(t_1)$ is not a **pair** Sval, it means that $\gamma_1(t_1)$ is unknown code. In this case, we can prove in the same way as the above case where $\epsilon \in mutable_{car}$.

The pe-store returned form $\mathcal{PE}[\![carM]\!]\rho\gamma$ is $\gamma_1[\mathbf{car}(t_1)/\underline{\hat{t}}]$. It respects $\sigma_1[\sigma_1(t_m)/\underline{\hat{t}}]$ since γ_1 respects σ_1.

(Case for $setcarMN$) Let $\mathcal{E}[\![M]\!]\rho\sigma = (\langle \epsilon, t_m, t_n \rangle, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = (^{\langle\!\langle P_1 \rangle\!\rangle}t_1, \gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![^{\langle\!\langle P_1 \rangle\!\rangle}t_1]\!]\sigma = (\langle \epsilon, t_m, t_n \rangle, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = \langle \epsilon, t_m, t_n \rangle$.

In the same way, let $\mathcal{E}[\![N]\!]\rho\sigma_1 = (n, \sigma_2)$ and $\mathcal{PE}[\![N]\!]\rho\gamma_1 = (^{\langle\!\langle P_2 \rangle\!\rangle}t_2, \gamma_2)$. Since γ_1 respects σ_1, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N]\!]\rho\gamma_1]\!]\sigma_1 = \mathcal{E}[\![N]\!]\rho\sigma_1$$

we have $\mathcal{E}_{PS}[\![^{\langle\!\langle P_2 \rangle\!\rangle}t_2]\!]\sigma_1 = (n, \sigma_2)$ and γ_2 respects σ_2. From lemma 3, we have $\sigma_2(t_2) = n$.

$$\begin{aligned} rhs &= \mathcal{E}[\![setcarMN]\!]\rho\sigma \\ &= \mathbf{let}\ (v, \sigma') = \mathcal{E}'[\![setcarMN]\!]\rho\sigma\ \mathbf{in}\ (v, \sigma'[v/\underline{\hat{t}}]) \\ &= (n, \sigma_2[n/t_m, n/\underline{\hat{t}}]) \\ lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![setcarMN]\!]\rho\gamma]\!]\sigma \\ &= \mathcal{E}_{PS}[\![^{\langle\!\langle P_1, P_2, \underline{\hat{t}}:\mathbf{setcar}(t_1,t_2)\rangle\!\rangle}\underline{\hat{t}}]\!]\sigma \end{aligned}$$

Since t_1 is fresh and not defined in P_2, we have $\sigma_2(t_1) = \sigma_1(t_1) = \langle \epsilon, t_m, t_n \rangle$, and thus

$$\begin{array}{rcl}
\mathcal{E}_{PS}[\![^{\langle\langle \underline{\hat{t}}:\mathbf{setcar}(t_1,t_2)\rangle\rangle}\underline{\hat{t}}]\!]\sigma_2 & = & \textbf{let } (v,\sigma') = \mathcal{E}_S[\![\mathbf{setcar}(t_1,t_2)]\!]\sigma_2 \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle}\underline{\hat{t}}]\!]\sigma'[v/\underline{\hat{t}}] \\
& = & \textbf{let } (v,\sigma') = (\sigma_2(t_2), \sigma_2[\sigma_2(t_2)/t_m]) \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\
& = & (\sigma_2(t_2), \sigma_2[\sigma_2(t_2)/t_m, \sigma_2(t_2)/\underline{\hat{t}}]) \\
& = & (n, \sigma_2[n/t_m, n/\underline{\hat{t}}])
\end{array}$$

where we take the same $\underline{\hat{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have *lhs* = *rhs*.

The pe-store returned form $\mathcal{PE}[\![setcarMN]\!]\rho\gamma$ is $\gamma_2[\mathbf{setcar}(t_1,t_2)/\underline{\hat{t}}]$. It respects $\sigma_2[n/t_m, n/\underline{\hat{t}}]$, since γ_2 respects σ_2.

(Case for $eqMM'NN'$) Let $\mathcal{E}[\![M]\!]\rho\sigma = (m, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![^{\langle\langle P_1\rangle\rangle}t_1]\!]\sigma = (m, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = m$.

In the same way, let $\mathcal{E}[\![M']\!]\rho\sigma_1 = (m', \sigma_2)$ and $\mathcal{PE}[\![M']\!]\rho\gamma_1 = (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2)$. Since γ_1 respects σ_1, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M']\!]\rho\gamma_1]\!]\sigma_1 = \mathcal{E}[\![M']\!]\rho\sigma_1$$

we have $\mathcal{E}_{PS}[\![^{\langle\langle P_2\rangle\rangle}t_2]\!]\sigma_1 = (m', \sigma_2)$ and γ_2 respects σ_2. From lemma 3, we have $\sigma_2(t_2) = m'$.

Since t_1 is fresh and not defined in P_2. Suppose that both $\gamma_2(t_1)$ and $\gamma_2(t_2)$ are known and $t_1 = t_2$. Since γ_2 respects σ_2, this means that $m = m' = \langle \epsilon, t'_1, t'_2 \rangle$ or $m = m' = \langle \eta, t'_3, _ \rangle$. In both cases, we use the third induction hypothesis. Let $\mathcal{E}[\![N]\!]\rho\sigma_2 = (n, \sigma_3)$ and $\mathcal{PE}[\![N]\!]\rho\gamma_2 = (^{\langle\langle P_3\rangle\rangle}t_3, \gamma_3)$. Since γ_2 respects σ_2, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N]\!]\rho\gamma_2]\!]\sigma_2 = \mathcal{E}[\![N]\!]\rho\sigma_2$$

we have $\mathcal{E}_{PS}[\![^{\langle\langle P_3\rangle\rangle}t_3]\!]\sigma_2 = (n, \sigma_3)$ and γ_3 respects σ_3. From lemma 3, we have $\sigma_3(t_3) = n$.

$$\begin{array}{rcl}
rhs & = & \mathcal{E}[\![eqMM'NN']\!]\rho\sigma \\
& = & \textbf{let } (v,\sigma') = \mathcal{E}'[\![eqMM'NN']\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\
& = & \textbf{let } (v,\sigma') = \mathcal{E}[\![N]\!]\rho\sigma_2 \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\
& = & (n, \sigma_3[n/\underline{\hat{t}}]) \\
lhs & = & \mathcal{E}'_{PS}[\![\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma]\!]\sigma \\
& = & \mathcal{E}_{PS}[\![^{\langle\langle P_1,P_2,\underline{\hat{t}}:\mathbf{eq}(t_1,t_2,^{\langle\langle P_3\rangle\rangle}t_3,_)\rangle\rangle}t_3]\!]\sigma \\
\mathcal{E}_{PS}[\![^{\langle\langle \underline{\hat{t}}:\mathbf{eq}(t_1,t_2,^{\langle\langle P_3\rangle\rangle}t_3,_)\rangle\rangle}t_3]\!]\sigma_2 & = & \textbf{let } (v,\sigma') = \mathcal{E}_S[\![\mathbf{eq}(t_1,t_2,^{\langle\langle P_3\rangle\rangle}t_3,_)]\!]\sigma_2 \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle}t_3]\!]\sigma'[v/\underline{\hat{t}}] \\
& = & \textbf{let } (v,\sigma') = \mathcal{E}_{PS}[\![^{\langle\langle P_3\rangle\rangle}t_3]\!]\sigma_2 \textbf{ in } (\sigma'[v/\underline{\hat{t}}](t_3), \sigma'[v/\underline{\hat{t}}]) \\
& = & (\sigma_3[n/\underline{\hat{t}}](t_3), \sigma_3[n/\underline{\hat{t}}]) \\
& = & (n, \sigma_3[n/\underline{\hat{t}}])
\end{array}$$

where we take the same $\underline{\hat{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have *lhs* = *rhs*.

The pe-store returned form $\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma$ is $\gamma_3[\mathbf{eq}(t_1,t_2,^{\langle\langle P_3\rangle\rangle}\, t_3, _)/\underline{\hat{t}}]$. It respects $\sigma_3[n/\underline{\hat{t}}]$, since γ_3 respects σ_3.

The case where both $\gamma_2(t_1)$ and $\gamma_2(t_2)$ are known but $t_1 \neq t_2$ can be proved similarly. When one (or both) of $\gamma_2(t_1)$ and $\gamma_2(t_2)$ are unknown, we use the fourth hypothesis as well

as the third one presented above. Let $\mathcal{E}[\![N']\!]\rho\sigma_2 = (n', \sigma_4)$ and $\mathcal{PE}[\![N']\!]\rho\gamma_3 = (^{\langle\langle P_4\rangle\rangle}t_4, \gamma_4)$. Because γ_2 respects σ_2 and $\sigma_2(t)$ is undefined for all $t \in domain(\gamma_3)$ and $t \notin domain(\gamma_2)$, γ_3 respects σ_2. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N']\!]\rho\gamma_3]\!]\sigma_2 = \mathcal{E}[\![N']\!]\rho\sigma_2$$

we have $\mathcal{E}_{PS}[\![^{\langle\langle P_4\rangle\rangle}t_4]\!]\sigma_2 = (n', \sigma_4)$ and γ_4 respects σ_4. From lemma 3, we have $\sigma_4(t_4) = n'$.

$$\begin{array}{rcl}
rhs & = & \mathcal{E}[\![eqMM'NN']\!]\rho\sigma \\
 & = & \textbf{let } (v, \sigma') = \mathcal{E}'[\![eqMM'NN']\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\
lhs & = & \mathcal{E}'_{PS}[\![\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma]\!]\sigma \\
 & = & \mathcal{E}_{PS}[\![^{\langle\langle P_1, P_2, \underline{\hat{t}}:\mathbf{eq}(t_1, t_2, ^{\langle\langle P_3\rangle\rangle}t_3, ^{\langle\langle P_4\rangle\rangle}t_4)\rangle\rangle}\underline{\hat{t}}]\!]\sigma \\
\mathcal{E}_{PS}[\![^{\langle\langle \underline{\hat{t}}:\mathbf{eq}(t_1, t_2, ^{\langle\langle P_3\rangle\rangle}t_3, ^{\langle\langle P_4\rangle\rangle}t_4)\rangle\rangle}\underline{\hat{t}}]\!]\sigma_2 & = & \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\mathbf{eq}(t_1, t_2, ^{\langle\langle P_3\rangle\rangle}t_3, ^{\langle\langle P_4\rangle\rangle}t_4)]\!]\sigma_2 \\
 & & \textbf{in } \mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle}\underline{\hat{t}}]\!]\sigma'[v/\underline{\hat{t}}] \\
 & = & \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\mathbf{eq}(t_1, t_2, ^{\langle\langle P_3\rangle\rangle}t_3, ^{\langle\langle P_4\rangle\rangle}t_4)]\!]\sigma_2 \\
 & & \textbf{in } (v, \sigma'[v/\underline{\hat{t}}])
\end{array}$$

Because $\sigma_2(t_1) = m$ and $\sigma_2(t_2) = m'$, if $m = m'$, then

$$\begin{array}{rcl}
rhs & = & \textbf{let } (v, \sigma') = \mathcal{E}'[\![eqMM'NN']\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\
 & = & (n, \sigma_3[n/\underline{\hat{t}}]) \\
lhs & = & \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\mathbf{eq}(t_1, t_2, ^{\langle\langle P_3\rangle\rangle}t_3, ^{\langle\langle P_4\rangle\rangle}t_4)]\!]\sigma_2 \\
 & & \textbf{in } (v, \sigma'[v/\underline{\hat{t}}]) \\
 & = & (n, \sigma_3[n/\underline{\hat{t}}])
\end{array}$$

where we take the same $\underline{\hat{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have $lhs = rhs$.

The pe-store returned form $\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma$ in this case is $\gamma_4[\mathbf{eq}(t_1, t_2, ^{\langle\langle P_3\rangle\rangle}t_3, ^{\langle\langle P_4\rangle\rangle}t_4)/\underline{\hat{t}}]$. It respects $\sigma_3[n/\underline{\hat{t}}]$, since γ_3 respects σ_3 and $\sigma_3(t)$ is undefined for all $t \in domain(\gamma_4)$ and $t \notin domain(\gamma_3)$,

The case for $m \neq m'$ can be proved similarly. □