Integrating Partial Evaluators into Interpreters

Kenichi Asai

Department of Information Science, Faculty of Science, University of Tokyo
7-3-1 Hongo Bunkyo-ku 113-0033 Japan
asai@is.s.u-tokyo.ac.jp
"Information and Human Activity," PRESTO, JST

Abstract. This paper describes our first step towards the integration of partial evaluation into standard interpretation. The two main issues in this integration are the treatment of heap objects and side-effects. To enable specialization with respect to the heap objects that are allocated beforehand in standard evaluation, specialization is performed with respect to *the heap state at specialization time* rather than based on a program text input so far. To avoid duplication of heap objects, *direct references to heap objects* are allowed in the specialized programs, in addition to the use of the conventional let-insertion technique. Despite our modest approach on side-effects that we allow partial evaluation of only side-effect free portions of a program, the resulting system covers some typical cases, such as the use of partial evaluation during an interactive debugging session. It also enables us to specialize a (side-effect free) program with respect to cyclic data structures, which was not easy before. We formalize this integration using store semantics and state its correctness. The system is implemented in Scheme and various examples are tested. Among them, we show specialization of a ray tracing program.

1 Introduction

Given a function $f(x, y)$ and a part of its input $x = a$, partial evaluation [19] (or program specialization) produces a specialized (and usually more efficient) program $f_a(y)$ by precomputing the parts of f that depend only on the known input x. The central strategy used in partial evaluation is strong normalization: given $f(x, y)$ and $x = a$, partial evaluation builds the specialization by strongly normalizing $\lambda y.f(a, y)$. In contrast to the standard interpretation (of call-by-value functional languages such as Scheme) which evaluates higher-order programs only up to the weak head normal form, the eager behavior of partial evaluation is suited for program optimization performed before actual execution. It is also well-known that this optimization amounts to compilation when applied to interpreters [13].

The conventional partial evaluation assumes that the program text to be transformed is given as a whole. Users are asked to prepare in a file the entire program, and then partial evaluation is applied to the designated function in it. In the language like Scheme, however, users input and execute programs one

by one. In such an interactive interpreter environment, it is more desirable that users can apply partial evaluation *on the fly* during the interactive session.

As the first step towards this integration of partial evaluation and standard interpretation, this paper tries to identify the problems that arise during the integration and show possible solutions to them. The target language is a subset of Scheme. The two main issues are the treatment of heap objects and side-effects.[1] In the interpreter environment, because objects are already allocated in heap and the pointer equality (eq-ness) is important in Scheme, partial evaluation must be performed in such a way that the identity of objects are preserved. For this purpose, we propose two techniques: partial evaluation with respect to *the heap state at specialization time* and *direct reference to heap objects* in the specialized program.

The other issue is the treatment of side-effects. Side-effects are inevitable in an interactive interpreter environment. Users will redefine functions and variables, construct data structures in heap, and modify them. As for the treatment of side-effects, however, we take rather a modest approach. Rather than solving various problems caused by side-effects, we simply identify them and show how to circumvent them. This causes several (rather severe) restriction on the use of partial evaluation: the current system allows partial evaluation of only side-effect free portions of a program.

We took this approach because we have no feasible solution for dealing with side-effects in an interactive environment. The previous work [3, 26] requires a program analysis on the whole program text. Even with this restriction, however, it covers some interesting cases, such as specialization with respect to circular data structures. It would also serve as a basis for more sophisticated techniques to weaken the restriction.

As a specializer, we employ *online* partial evaluation [16, 23] that inspects the availability of values *during* specialization process. The *offline* approach where availability of values are analyzed beforehand in a separate phase (called a binding-time analysis) was not employed because it usually requires a whole program text for the global program analysis. If we made use of a separate binding-time analysis [8], we might be able to employ the offline approach as well.

We formalize the specializer in store-passing style to make special emphasis on the behavior of heap. We then state its correctness: the execution of programs with and without specialization produce a congruent result and store.

Three versions of the specializer together with the interpreter are implemented in Scheme. Various extensions are implemented, including controlling termination and residualization via user annotation (filters [7]), the multilevel specialization [14], and other language extensions. Various examples are tested on the implementation. Among them, we show a ray tracing example in this paper.

[1] By side-effects, we mean assignments (`set!`) and destructive updates (`set-car!` and `set-cdr!`). When we refer to I/O-type side-effects (such as `read` and `write`), we will explicitly say so.

The paper is organized as follows. In the next section, we describe in detail various issues that arise during the integration of partial evaluation and standard interpretation. The formal description then starts by showing the language in Sect. 3 and describing the standard interpreter in Sect. 4. The specializer is presented in detail in Sect. 5, whose correctness is mentioned in Sect. 6. The extensions employed in the implementation are discussed in Sect. 7. Specialization of ray tracing is detailed in Sect. 8. Related work is in Sect. 9 and the paper concludes in Sect. 10.

2 Issues

In this section, we describe what issues arise during the integration of partial evaluation and interpretation. We will use the following interactive session as an example:

```
> (define (nth n lst)
    (if (= n 0) (car lst) (nth (- n 1) (cdr lst))))
> (define circular-list
    (let ((aux (cons 1 (cons 2 '()))))
      (set-cdr! (cdr aux) aux)
      aux))
> (define nth2
    (pe (lambda (n) (nth n circular-list))))
```

After `nth` is defined which takes the `n`'th element of `lst`, a circular list `(1 2 1 2 ...)` is constructed. Then, we specialize `nth` with respect to the circular list. Here, `pe` initiates a partial evaluation of its argument closure.

2.1 Side-Effects before Specialization

In the above session, we used two kinds of side-effects before specialization: definition of variables (`nth` and `circular-list`) and a destructive update. Although the presence of side-effects is not preferable from the partial evaluation point of view, we allow arbitrary side-effects *before* specialization. This is because we want the integrated interpreter to be a standard interpreter if the specialization construct `pe` was not used at all. In the above example, we actually define `nth`, destructively update the cdr of `aux`, and define `circular-list`. Note that the definition of variables can be modification of variables if they were already defined beforehand.

2.2 Specialization with Respect to the Current Heap State

The decision that arbitrary side-effects are allowed before specialization does not make the later specialization process complicated. We simply perform specialization *with respect to the current heap state*.

In the conventional partial evaluators, specialization of programs containing side-effects requires a side-effect analysis [3, 26]. In particular, if we want to statically evaluate side-effects at specialization time, a partial evaluator has to be able to decide that it can be safely done at partial evaluation time (using a complex region-based binding-time analysis [26]). This complication can be avoided if we perform specialization with respect to the specialization-time heap state rather than the program text input so far in the interactive session. Whatever side-effects occur before specialization, they simply change the heap state, and when the specialization begins, the heap state is fixed (if no side-effects are used during specialization).

By taking the current heap state static, we can perform interesting partial evaluation which was not easy before: specialization with respect to a circular-list. In the above example, the value of `nth2` becomes:[2]

```
> nth2
(letrec ((f1 (lambda (n) (if (= n 0) 1 (f2 (- n 1)))))
         (f2 (lambda (n) (if (= n 0) 2 (f1 (- n 1))))))
  f1)
```

Not only the elements of `circular-list` are inlined but also its circular structure is reflected to the recursive structure of the specialized program.

2.3 Side-Effects during Specialization

Although side-effects before specialization can be handled nicely, side-effects *during* specialization is difficult. The situation is essentially the same as the conventional partial evaluation: a side-effect analysis is inevitable [3, 26]. Furthermore, because we are interested in the interactive environment, such an analysis would have to be able to cope with pre-allocated objects and interactive redefinition of programs. In this paper, we propose no solution to it but simply restrict that specialization can be applied only to pure side-effect free programs. One possible future direction would be to investigate a modular side-effect analysis, so that it can be applied to interactively input programs on the fly.

Even with this restriction, it covers some typical cases. For example, suppose a user is writing and debugging a program in a pure functional manner. He first inputs functions some of which are erroneous, executes them, finds that something was wrong, and then redefines the erroneous functions and executes them again. At every stage, the whole program does not contain any side-effects, yet the definition of functions changes as the user redefines them. In such a case, he is allowed to perform specialization at any time he wants.

2.4 Side-Effects after Specialization

In the previous subsections, we have allowed side-effects before specialization and prohibited side-effects during specialization. There is one other possibility of

[2] The definition of `f2` will be inlined into the definition of `f1` in the actual output.

side-effects: side-effects *after* specialization. Because specialization with respect to the current heap state means that the information on the current heap state is used for specialization, the modification of heap can invalidate specialization. For example, if we destructively modify `circular-list` after we obtained the above `nth2`:

```
> (set-car! circular-list 3)
```

the original program `(nth 0 circular-list)` now returns 3 but the specialized program `(nth2 0)` returns 1 because the element of the old `circular-list` is already inlined into `nth2`.

Rather than providing a sophisticated mechanism to detect which specialization is invalidated and should be respecialized, we again take rather a modest approach. We decide on two principles: (1) specialization is performed assuming that the heap state will never be modified, and (2) the future modification on heap does not have any effects on specialized programs. The first principle is reasonable. In the interactive environment, users can modify any (accessible) data structures after specialization. If we take the side-effects after specialization into account during specialization, we can perform very little specialization. We cannot even take car or cdr of a cons cell since it might be modified later.

The second principle is not very satisfactory. We could design more sophisticated techniques, such as automatic detection of invalidation and respecialization [17, 27]. In this paper, however, we do not pursue this approach further but accept the second principle as the first step towards the integration of partial evaluation and standard interpretation. One of the reasons for this decision is its simplicity and expectation that the other techniques could be built on top of it. Another reason is that invalidation is not so straightforward. Since Scheme does not provide any ways to indicate mutable data structures (in contrast to ML where only references are mutable), we have to maintain a set of *all* the data structures referenced during specialization to judge invalidation, which can be very large. This situation is in contrast to the run-time specialization [9, 21] where whole the program is given beforehand and an analysis can estimate which data structures are mutable. To keep the set small, we feel that some kind of user assistance is required — our future work.

2.5 Direct Reference to Heap Objects

The presence of pre-allocated heap objects has impact on the output of partial evaluation: care must be taken not to duplicate them. In the conventional partial evaluators, let-insertion [6] has been used to avoid code duplication. In addition to that, we need direct reference to heap objects.

Consider the following specialization where `circular-list` is already allocated in heap:

```
> (define p (pe (lambda () (cdr circular-list))))
```

The result should not produce code that reconstructs a different circular list than `circular-list`, but should refer to the second cell of `circular-list` directly.

$$
\begin{aligned}
\mathit{AExp} &:= (\mathit{Preaction}; \ldots; \mathit{Preaction}, \mathit{Var}) \\
\mathit{Preaction} &:= \mathit{Var} = \mathit{Exp} \mid \mathit{Var} \stackrel{r}{=} (\mathsf{lambda}\ (\mathit{Var}^*)\ \mathit{AExp}) \\
\mathit{Exp} &:= \mathit{Const} \mid \mathit{Var} \mid (\mathsf{if}\ \mathit{Var}\ \mathit{AExp}\ \mathit{AExp}) \mid (\mathsf{set!}\ \mathit{Var}\ \mathit{Var}) \mid \\
&\quad\ (\mathsf{lambda}\ (\mathit{Var}^*)\ \mathit{AExp}) \mid (\mathit{Var}\,\mathit{Var}^*) \mid (\mathsf{pe}\ \mathit{AExp})
\end{aligned}
$$

Fig. 1. Language

Otherwise, in the later interaction, the result of applying the specialized program may be compared its identity with the cdr of `circular-list`:

```
> (eq? (p) (cdr circular-list))
```

which should return `#t` (true). The problem here is that the cdr of `circular-list` does not have a name. To refer directly to anonymous heap objects, we assume that every heap object has a unique global name. In our example, we assume that `circular-list` was constructed something like:

```
> (define a (cons 2 '()))
> (define circular-list (cons 1 a))
> (set-cdr! a circular-list)
```

giving a global name `a` to `(cons 2 '())`. Then, we can write the result of specializing `p` as follows:

```
> p
(lambda () a)
```

In the implementation, we do not have to give a unique global name to every anonymous object explicitly. It can be thought of as the address of the object in memory (or location in the store semantics). Rather than creating a new name to each anonymous object, we will allow to write explicitly its memory address in the result of partial evaluation. In other words, we will use the memory address as a global name to the object.

3 The Language

The language we consider is a higher-order subset of Scheme expressed in A-normal form [12]. Its syntax is displayed in Fig. 1. We call an expression in A-normal form an *A-expression*. An A-expression (*AExp*) consists of a sequence of *preactions* [3] (separated by ";") and a variable. A preaction builds a let-binding (the first form) or a letrec-binding (the second form). In the latter case, the bound expression is required to be a lambda expression. For example,

$$(a = 3; f \stackrel{r}{=} (\mathsf{lambda}\ (x)\ \ldots); b = (f\ a), b)$$

is equivalent to the following Scheme expression:

(let ((a 3)) (letrec ((f (lambda (x) ...))) (let ((b (f a))) b))).

$$
\begin{aligned}
\mathcal{E} &: AExp \to Env \to Store \to (Value \times Store)\\
\mathcal{E}_p &: Exp \to Env \to Store \to (Value \times Store)\\
\rho &\in Env = Var \to Name\\
\sigma &\in Store = Name \to Value\\
& \quad Value = Const + Closure
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![(\cdot, y)]\!]\rho\,\sigma &= (\sigma(\rho(y)), \sigma)\\
\mathcal{E}[\![(x = e; P, y)]\!]\rho\,\sigma &= \textbf{let } (v_1, \sigma_1) = \mathcal{E}_p[\![e]\!]\rho\,\sigma\\
&\qquad\quad n = new(\sigma_1)\\
&\quad \textbf{in } \mathcal{E}[\![(P, y)]\!]\rho[n/x]\,\sigma_1[v_1/n]\\
\mathcal{E}[\![(x \stackrel{r}{=} \mathsf{lambda}\ (x^*)\ (P_0, x_0)); P, y)]\!]\rho\,\sigma &= \textbf{let } l = new(\sigma)\\
&\qquad\quad n = new(\sigma[\star/l])\\
&\qquad\quad clo = cl(l, x^*, (P_0, x_0), \rho[n/x])\\
&\quad \textbf{in } \mathcal{E}[\![(P, y)]\!]\rho[n/x]\,\sigma[\star/l, clo/n]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}_p[\![const]\!]\rho\,\sigma &= (const, \sigma)\\
\mathcal{E}_p[\![x]\!]\rho\,\sigma &= (\sigma(\rho(x)), \sigma)\\
\mathcal{E}_p[\![(\mathsf{if}\ x_0\ (P_1, x_1)\ (P_2, x_2))]\!]\rho\,\sigma &= \textbf{if } \sigma(\rho(x_0))\ \textbf{then } \mathcal{E}[\![(P_1, x_1)]\!]\rho\,\sigma\\
&\qquad\qquad\qquad\qquad\quad \textbf{else } \mathcal{E}[\![(P_2, x_2)]\!]\rho\,\sigma\\
\mathcal{E}_p[\![(\mathsf{set!}\ x\ y)]\!]\rho\,\sigma &= (\star, \sigma[\sigma(\rho(y))/\rho(x)])\\
\mathcal{E}_p[\![(\mathsf{lambda}\ (x^*)\ (P_0, x_0))]\!]\rho\,\sigma &= \textbf{let } l = new(\sigma)\\
&\quad \textbf{in } (cl(l, x^*, (P_0, x_0), \rho), \sigma[\star/l])\\
\mathcal{E}_p[\![(x_0\ y_0^*)]\!]\rho\,\sigma &= \textbf{let } v_y^* = \sigma(\rho(y_0^*))\\
&\quad \textbf{in case } \sigma(\rho(x_0)) \textbf{ of}\\
&\qquad prim : (prim(v_y^*), \sigma)\\
&\qquad cl(l, x^*, (P', x'), \rho') :\\
&\qquad\quad \textbf{let } l^* = new^*(\sigma)\\
&\qquad\quad \textbf{in } \mathcal{E}[\![(P', x')]\!]\rho'[l^*/x^*]\,\sigma[v_y^*/l^*]\\
\mathcal{E}_p[\![(\mathsf{pe}\ (P_0, x_0))]\!]\rho\,\sigma &= \textbf{let } (P_1, n_1, v_1, \gamma_1) = \mathcal{P}[\![(P_0, x_0)]\!]\rho\,\sigma\\
&\quad \textbf{in } \mathcal{E}[\![(P_1, n_1)]\!]\rho_{id}\,\sigma
\end{aligned}
$$

Fig. 2. Standard interpreter

The last expression (pe *AExp*) initiates partial evaluation.

Because standard Scheme programs can be converted into A-normal form in linear time [12], we will use expressions not necessarily in A-normal form for the presentation of examples.

4 Standard Interpreter

The interpreter, giving the standard operational semantics of the language, is shown in Fig. 2. It is written in store-passing style. Given an A-expression (P, y), an environment ρ, and a store σ, the interpreter $\mathcal{E}$ evaluates (P, y) under ρ and σ, and returns the final value (which is either a constant or a closure) together with the store after the evaluation.

In the figure, we use the following notations. An empty sequence of preactions is denoted by a dot $\cdot$. $new(\sigma)$ generates a name (location) which is not bound in σ. We use the word 'name' rather than 'location' because the generated names are used as the unique names of the objects allocated there. $\rho[n/x]$ is an environment identical to ρ except that $\rho(x) = n$. $\sigma[l/n]$ is similar. The constant $\star$ denotes an unspecified value. cl is a closure constructor.[3] x^* represents a sequence of variables. If $x^* = x_1 \ldots x_k$ and $l^* = l_1 \ldots l_k$, then $l^* = new^*(\sigma)$ creates k new names and assign them to l^*, $\rho(x^*)$ represents a sequence $\rho(x_1) \ldots \rho(x_k)$, $\rho[l^*/x^*]$ is equivalent to ρ except that $\rho(x_i) = l_i$ for $1 \leq i \leq k$.

The evaluation of an A-expression proceeds by evaluating each preaction in the A-expression in order. The evaluation of each expression in preactions is done by the auxiliary function $\mathcal{E}_p$. The interpreter is completely standard.[4] The only non-standard rule is the last one for the evaluation of the `pe` construct, which will be described after the specializer $\mathcal{P}$ is shown in the next section.

5 Specializer

The specializer is shown in Figs. 3 and 4. It is also written in store-passing style. Given an A-expression (P, y), an environment ρ, and a partial evaluation time store γ (*pe-store*, for short), $\mathcal{P}$ partially evaluates (P, y) under ρ and γ. The returned value is a tuple of four elements (P', n', v', γ'). P' and n' are a sequence of preactions and a variable, respectively, which constitute the specialized program. (We treat a name n' as a variable; see below.) v' is a partial evaluation time value (*pe-value*, for short) of (P, y). If (P, y) can be reduced completely, the resulting value is stored in v' and is used for further specialization. If not, v' is bound to a name. Finally, γ' is a modified pe-store.

Here, we assume that *Name* is a subset of *Var* and hence of *Exp*. Whenever we create a new name n, we also use n as a variable that refers to the object allocated at the location n. This is the same as the name creation of the conventional partial evaluation. Remember that names are created during the standard evaluation, too, when objects are allocated in heap. If such names are referred to in a specialized program, it indicates direct references to heap objects that are created before specialization. This use of names avoids duplication of pre-allocated heap objects. If we had used code that reconstruct heap objects instead, it would have duplicated them and destroyed their identity.

Like the standard evaluation, partial evaluation of an A-expression proceeds by partially evaluating each preaction in the A-expression in order. Specialization of an A-expression with an empty preaction $(\cdot, y)$ returns $(\cdot, \rho(y), \gamma(\rho(y)), \gamma)$. Here, the name $\rho(y)$ is used as a specialized program to refer directly to the object $\gamma(\rho(y))$ allocated in heap (store). This means that all the variable lookups are

[3] The first element l of a closure indicates where in memory the closure is allocated. It works as an identity when the language is extended with pointer equality.

[4] The recursion is realized in the rule for $x \stackrel{r}{=} \cdots$ by using $\rho[n/x]$ in *clo* where x is bound to n which will later be bound to the recursive closure itself.

$$
\begin{array}{rcl}
\mathcal{P} & : & \mathit{AExp} \to \mathit{Env} \to \mathit{PEStore} \to (\mathit{Preaction}^* \times \mathit{Name} \times \mathit{PEValue} \times \mathit{PEStore}) \\
\mathcal{P}_p & : & \mathit{Exp} \to \mathit{Env} \to \mathit{PEStore} \to (\mathit{Preaction}^* \times \mathit{Exp} \times \mathit{PEValue} \times \mathit{PEStore}) \\
\rho & \in & \mathit{Env} = \mathit{Var} \to \mathit{Name} \\
\gamma & \in & \mathit{PEStore} = \mathit{Name} \to \mathit{PEValue} \\
 & & \mathit{PEValue} = \mathit{Const} + \mathit{Closure} + \mathit{Name}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}[\![(\cdot, y)]\!]\rho\gamma & = & (\cdot, \rho(y), \gamma(\rho(y)), \gamma) \\
\mathcal{P}[\![(x = e; P, y)]\!]\rho\gamma & = & \textbf{let } (P_1, e_1, v_1, \gamma_1) = \mathcal{P}_p[\![e]\!]\rho\gamma \\
 & & \quad n = \mathit{new}(\gamma_1) \\
 & & \quad \gamma_1' = \textbf{if } v_1 = \mathsf{dyn} \textbf{ then } \gamma_1[n/n] \textbf{ else } \gamma_1[v_1/n] \\
 & & \quad (P_2, n_2, v_2, \gamma_2) = \mathcal{P}[\![(P, y)]\!]\rho[n/x]\gamma_1' \\
 & & \textbf{in } (P_1; n = e_1; P_2, n_2, v_2, \gamma_2) \\
\mathcal{P}[\![(x \stackrel{r}{=} (\mathsf{lambda}\ (x^*)\ (P_0, x_0)); P, y)]\!]\rho\gamma & & \\
 & = & \textbf{let } l = \mathit{new}(\gamma) \\
 & & \quad n = \mathit{new}(\gamma[\star/l]) \\
 & & \quad \mathit{clo} = \mathit{cl}(l, x^*, (P_0, x_0), \rho[n/x]) \\
 & & \quad (P_1, n_1, v_1, \gamma_1) = \mathcal{P}[\![(P, y)]\!]\rho[n/x]\gamma[\star/l, \mathit{clo}/n] \\
 & & \quad p^* = \mathit{new}^*(\gamma_1) \\
 & & \quad (P_2, n_2, v_2, \gamma_2) = \mathcal{P}[\![(P_0, x_0)]\!]\rho[n/x, p^*/x^*]\gamma_1[p^*/p^*] \\
 & & \textbf{in } (n \stackrel{r}{=} (\mathsf{lambda}\ (p^*)\ (P_2, n_2)); P_1, n_1, v_1, \gamma_2)
\end{array}
$$

Fig. 3. Partial evaluator (part 1)

done at this specialization time, and the resulting specialized program contains only the memory address.

Specialization of an A-expression (P, y) proceeds by partially evaluating all the preactions in P in order using the auxiliary function $\mathcal{P}_p$ shown in Fig. 4. $\mathcal{P}_p$ partially evaluates a given expression in a preaction and returns a tuple $(P_1, e_1, v_1, \gamma_1)$ consisting of a preaction P_1, a specialized expression e_1, a specialized value v_1, and a modified pe-store γ_1. v_1 may be a special name dyn which indicates that the result cannot be computed at partial evaluation time.

The specialization of a recursive binding includes not only creation of a recursive closure but also creation of a recursive specialized closure. It is done by specializing the body of the lambda expression with respect to the unknown parameter p^*. This can be regarded as the online realization of both static and dynamic values [2]. The creation of specialization can be postponed until it turns out that it is actually required (residualized), in a way similar to performing the backward analysis [18].

$\mathcal{P}_p$ is shown in Fig. 4. Specialization of a constant is itself. Here, the first *const* is a program text to appear in the specialized program while the second one is the value to be used in the later specialization. Specialization of a variable x returns a program text[5] $\rho(x)$ and a pe-value $\gamma(\rho(x))$. Specialization of a conditional is delegated to specialization of the then- or else-branch when the condition $\gamma(\rho(x))$ reduces to true ($\#\mathsf{t}$) or false ($\#\mathsf{f}$), respectively. If the value

[5] Or a name, a location, or a memory address.

$$
\begin{array}{rcl}
\mathcal{P} & : & AExp \to Env \to PEStore \to (Preaction^* \times Name \times PEValue \times PEStore) \\
\mathcal{P}_p & : & Exp \to Env \to PEStore \to (Preaction^* \times Exp \times PEValue \times PEStore) \\
\rho & \in & Env = Var \to Name \\
\gamma & \in & PEStore = Name \to PEValue \\
 & & PEValue = Const + Closure + Name
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}_p[\![const]\!]\rho\,\gamma & = & (\cdot, const, const, \gamma) \\
\mathcal{P}_p[\![x]\!]\rho\,\gamma & = & (\cdot, \rho(x), \gamma(\rho(x)), \gamma) \\
\mathcal{P}_p[\![(\mathsf{if}\ x_0\ (P_1, x_1)\ (P_2, x_2))]\!]\rho\,\gamma & = & \textbf{case}\ \gamma(\rho(x_0))\ \textbf{of} \\
 & & \quad \#\mathsf{t} : \mathcal{P}[\![(P_1, x_1)]\!]\rho\,\gamma \\
 & & \quad \#\mathsf{f} : \mathcal{P}[\![(P_2, x_2)]\!]\rho\,\gamma \\
 & & \quad \textbf{otherwise} : \\
 & & \qquad \textbf{let}\ (P_1', n_1, v_1, \gamma_1) = \mathcal{P}[\![(P_1, x_1)]\!]\rho\,\gamma \\
 & & \qquad\quad (P_2', n_2, v_2, \gamma_2) = \mathcal{P}[\![(P_2, x_2)]\!]\rho\,\gamma_1 \\
 & & \qquad \textbf{in}\ (\cdot, (\mathsf{if}\ \rho(x_0)\ (P_1', n_1)\ (P_2', n_2)), \mathsf{dyn}, \gamma_2) \\
\mathcal{P}_p[\![(\mathsf{lambda}\ (x^*)\ (P_0, x_0))]\!]\rho\,\gamma & = & \textbf{let}\ l = new(\gamma) \\
 & & \qquad clo = cl(l, x^*, (P_0, x_0), \rho) \\
 & & \qquad p^* = new^*(\gamma[\star/l]) \\
 & & \qquad (P_1, n_1, v_1, \gamma_1) = \\
 & & \qquad\quad \mathcal{P}[\![(P_0, x_0)]\!]\rho[p^*/x^*]\,\gamma[\star/l, p^*/p^*] \\
 & & \textbf{in}\ (\cdot, (\mathsf{lambda}\ (p^*)\ (P_1, n_1)), clo, \gamma_1) \\
\mathcal{P}_p[\![(x_0\ y_0^*)]\!]\rho\,\gamma & = & \textbf{let}\ n_x = \rho(x_0),\ v_x = \gamma(n_x) \\
 & & \qquad n_y^* = \rho(y_0^*),\ v_y^* = \gamma(n_y^*) \\
 & & \textbf{in case}\ v_x\ \textbf{of} \\
 & & \quad prim : \textbf{if}\ v_y^*\ \textit{are all known} \\
 & & \qquad\quad \textbf{then let}\ v = prim(v_y^*)\ \textbf{in}\ (\cdot, v, v, \gamma) \\
 & & \qquad\quad \textbf{else}\ (\cdot, (n_x\ n_y^*), \mathsf{dyn}, \gamma) \\
 & & \quad cl(l, x^*, (P', x'), \rho') : \\
 & & \qquad \textbf{if}\ \textit{unfold?} \\
 & & \qquad \textbf{then let}\ l^* = new^*(\gamma) \\
 & & \qquad\quad\quad (P_1, n_1, v_1, \gamma_1) = \\
 & & \qquad\quad\quad\quad \mathcal{P}[\![(P', x')]\!]\rho'[l^*/x^*]\,\gamma[v_y^*/l^*] \\
 & & \qquad\quad \textbf{in}\ (l^* = n_y^*; P_1, n_1, v_1, \gamma_1) \\
 & & \qquad \textbf{else}\ (\cdot, (n_x\ n_y^*), \mathsf{dyn}, \gamma) \\
 & & \quad \textbf{otherwise} : (\cdot, (n_x\ n_y^*), \mathsf{dyn}, \gamma) \\
\mathcal{P}_p[\![(\mathsf{pe}\ (P_0, x_0))]\!]\rho\,\gamma & = & \mathcal{P}[\![(P_0, x_0)]\!]\rho\,\gamma
\end{array}
$$

Fig. 4. Partial evaluator (part 2)

of the condition is not obtained during specialization, both the then- and else-branches are specialized and a conditional expression is returned. This is one of the places where a binding-time of a value is examined online.

Specialization of a lambda expression creates a closure and its specialized version. The former is used when the lambda expression is applied to arguments while the latter is used when it appears in the specialized program.

Specialization of application proceeds by first examining online the value of its function part. If it is *not* available at specialization time, an application expression is returned. If the function is a primitive, it computes the result if its arguments are all available, otherwise returns an application expression. If the function is a closure, one of two actions is taken: unfold the application or returns an application expression without unfolding. We can obtain more specialized results if application is unfolded. However, if we blindly unfold all the application, it is more likely that the specialization does not terminate. Here, we do not specify how to control this termination behavior of specialization. It is discussed in Sect. 7.1.

Finally, pe during specialization is simply ignored, and specialization is continued to its argument. In Sect. 7.2, we slightly change this rule to accommodate the multilevel specialization [14].

The specialization process itself is launched by executing pe during standard evaluation. Let us go back to the last rule in Fig. 2. The execution of $(\mathsf{pe}\ (P_0, x_0))$ results in the partial evaluation of (P_0, x_0), whose result (an A-expression) is then executed by the standard interpreter (under the identity environment $\rho_{id} = \lambda n.n$; see below). Although the argument (P_0, x_0) can be any A-expression, only the lambda expression $(y = (\mathsf{lambda}\ (x^*)\ (P_0, x_0)), y)$ is sensible, because otherwise the specialized program is executed only once and in this case, standard evaluation is faster than partial evaluation.

Three points are noticed here for the rule of pe. First, the same standard store σ is used for the pe-store of specialization. Namely, we specialize a program with respect to the current heap state. (We regard a standard store as a special kind of pe-store where the stored values are only constants or closures and not names.) Secondly, the final pe-store resulting from the specialization is discarded and the specialized program is executed under the original store σ. In the implementation, this means that all the objects allocated during specialization can be garbage collected. Finally, the obtained specialized program is executed under the identity environment ρ_{id}. Remember that all the environment lookups are performed during specialization and the specialized program contains only names (locations) which directly refer to heap objects. Thus, ρ_{id} is used to map these names (regarded as variables in the specialized program) to names (locations) themselves.

6 Correctness

We have defined a congruence relation between values and proved that the result of partial evaluation of any side-effect free program is congruent to the original

program in the store semantics, if the partial evaluation terminates. Details are omitted, however, for the lack of space.

7 Implementation and Extensions

We have implemented three versions[6] of the specializer together with the interpreter in Scheme. Below, we describe the extensions that we implemented.

7.1 Controlling Termination and Residualization

The specializer shown in Fig. 3 does not specify when to unfold a closure. In the implementation, we employ the user-annotation approach (filters [7]). Whenever a closure is *not* to be unfolded, a filter is attached to indicate it. Currently, filters are attached manually. This makes the implementation simpler releasing from sophisticated termination detection mechanisms. There seems to be a version of Schism that supports automatic insertion of filters.

A filter has two parts. The first part indicates when the closure is unfolded whereas the second part indicates which parameter values are used to make specialization when the closure is not unfolded. For example, consider the following power program using an accumulator parameter:

```
(define power-acc (lambda (m n acc)
  '(filter (static? n) '(- D D))
  (if (= n 0) acc (power-acc m (- n 1) (* m acc)))))
```

The first part of the filter indicates that the function is unfolded when the value of `n` is known. This reflects the fact that it is safe to unfold it when the recursion variable is known. A special construct `static?` is supported to check the availability of values at partial evaluation time. When `n` is unknown, the second part of the filter is examined and a specialized closure is created. The list `(- D D)` corresponds to the list of parameters `(m n acc)` and indicates if the value of each parameter should be propagated or not when constructing specializations. The directive `-` means "propagate" and `D` "not propagate". Even though the directive is `-`, the value is not propagated if it is unknown. Propagated parameters are inlined and disappear from the parameters of specializations. One specialization is created for each variation of the value of propagated parameters.

In the above example, the value of `m` is propagated and inlined into specialization (if its value is known) but `acc` is not. `Acc` is not propagated because otherwise infinite specializations (corresponding to infinite variations of `acc`) are created leading to non-termination of partial evaluation if both `m` and `acc` are known.

[6] One is the faithful implementation of $\mathcal{P}$ and $\mathcal{E}$, and the other two implement a store using the real memory. In the latter, let-insertion is realized via explicit continuation passing [20] or delimited continuation operators [2, 25].

7.2 Multilevel Partial Evaluation

The occurrence of pe during specialization is simply ignored in Fig. 3. If the last rule in Fig. 3 is changed to the following rule:

$$\begin{aligned}\mathcal{P}_p[\![(\mathsf{pe}\ (P_0, x_0))]\!]\rho\gamma = \ & \textbf{let}\ (P_1, n_1, v_1, \gamma_1) = \mathcal{P}[\![(P_0, x_0)]\!]\rho\gamma \\ & \textbf{in}\ (\cdot, (\mathsf{pe}\ (P_1, n_1)), \mathsf{dyn}, \gamma_1)\end{aligned}$$

we can perform the multilevel specialization [14]. We left the pe construct in the specialized program. For example,

```
(pe (lambda (x) .....(pe (lambda (y) .....)).....))
```

will first produce a closure where computation not dependent on x (nor y) is performed. Since the resulting program now keeps the pe construct:

```
(lambda (x) ...(pe (lambda (y) ...))...)
```

the inner closure is further specialized using the value of x when the outer closure is applied. If the inner closure contains computation that can be specialized significantly only after the value of x is known (and the closure is used sufficiently many times), specializing it at this time would improve the overall execution time.

The practical evaluation of this multilevel partial evaluation has not yet been done.

7.3 Language Extension

Most standard primitives and special forms are supported for standard evaluation. For partial evaluation, primitives that have no side-effects are supported by simply putting them in the initial environment. Even the primitives that test the pointer equality, such as eq?, are supported because the specialized program is congruent to the original one with respect to the store semantics. Since our partial evaluator is online, cons naturally realizes partially known data structures. Among side-effecting primitives, I/O-type primitives are supported by blindly residualizing them in the specialized program. This is justified because the order of preactions is preserved during specialization.

The use of assignments (set!) and destructive updates (set-car! and set-cdr!) during partial evaluation are currently prohibited completely. However, the restricted form of assignments and destructive updates could be supported for the case where the names (locations) of the updated variables/objects are identified (known) and their values are unknown at partial evaluation time.

In all cases, side-effects would be blindly residualized in the specialized program. Execution of side-effects during specialization requires a sophisticated program analysis [26].

8 Experiments

We have tested various examples on our implementation. We obtained expected results for the standard examples, such as specialization of a power function, Ackermann's function, and compilation of various kinds of interpreters by specialization. For the compilation of interpreters, the specializer passes Neil Jones' optimality test [19]. In this section, we show our experience on the specialization of ray tracing as a bigger example.

Ray tracing produces a realistic image of 3D objects by calculating the color of each pixel on the screen. Fig. 5 shows a sample output of ray tracing. Given a configuration of 3D objects, we trace light in the reverse direction from the viewpoint (`V-point`) to each pixel on the screen. When it intersects with an object, we collect various light that reaches that intersecting point, such as ambient light (uniform light whose intensity is the same at all places), diffuse reflection (light coming directly from the light-point (`L-point`); effective only when the point is visible from the light-point), and specular reflection (reflected light coming from the direction of mirror-reflection; computed using recursion). Here goes the psuedo code for it:

```
(define (main V-point screen L-point object)
  (foreach pixel on the screen
    ; let V-vector be the vector from V-point to the pixel
    (ray-trace V-point V-vector L-point object)))
(define (ray-trace V-point V-vector L-point object)
  ; calculate if V-vector intersects with object
  ; if it does not, return background color
  ; otherwise, return the sum of
  ; - ambient light
  ; - diffuse reflection if the intersecting point
  ;                         is visible from L-point
  ; - specular reflection (via recursion)
  ; of the intersecting point
  )
```

For each pixel on the screen, `ray-trace` is called with `L-point` and `object` being constants. Since the number of times `ray-trace` is called is proportional to the number of pixels on the screen which can be quite large, specializing `ray-trace` with respect to fixed `L-point` and `object` will have benefits.

For specialization, we made the following changes. First, we insert a filter `'(filter #f '(D D - -))` to `ray-trace`. `#f` says that `ray-trace` is never unfolded. Since the recursion structure of `ray-trace` is dependent on how objects are arranged and reflected-light runs, it is difficult to predict it. Here, we decide not to unfold at all. `'(D D - -)` means that the values of `L-point` and `object` are propagated to make specialization. Filters are inserted for other recursive functions, too. Then, we replace `ray-trace` in `main` with `ray-trace2` and include the following:

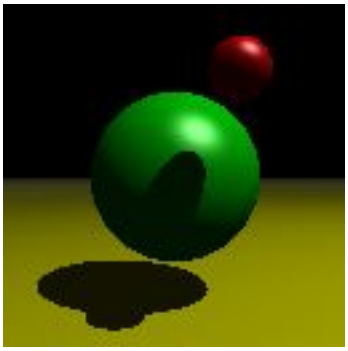

Fig. 5. A sample image (128×128 pixels)

```
(define compiled? #f)
(define compiled-ray-trace 0)
(define (ray-trace2 V-point V-vector L-point object)
  (if (not compiled?)
    (begin (set! compiled-ray-trace
             (pe (lambda (V-point V-vector)
                   (ray-trace V-point V-vector L-point object))))
           (set! compiled? #t)))
  (compiled-ray-trace V-point V-vector))
```

When `ray-trace2` is called for the first time, it calls a partial evaluator to specialize `ray-trace` and memoise it in the global variable `compiled-ray-trace`. During the specialization, `V-point` and `V-vector` are unknown since they are abstracted by `lambda` while `L-point` and `object` are bound to their values when the specializer is called. The specialization is done only once. The next time `ray-trace2` is called, it simply uses the specialization stored in `compiled-ray-trace`. The use of side-effects to store the specialized program is permitted since it is outside the `ray-trace` function and the updated variables are never referenced there.

We executed the original and specialized programs for the image in Fig. 5 with the screen size 32×32 pixels and compared their execution times. The experiment was done using MacGambit 2.2.2 on Macintosh PowerBook (PowerPC G3, 500MHz, 64MB memory for Scheme among 128MB). The specializer (together with the interpreter) is compiled by the MacGambit Compiler. The result is summarized in Table 1. The row (A) shows the execution time (in seconds) of the original program without performing specialization. (B) shows the execution time and the specialization time for the program with the changes for specialization described above. We can observe speedup of more than 16 for the main execution time excluding the specialization time. Even with the specialization time, speedup is almost the same, indicating that the specialization time is ignorable.

	specialization	execution	total
(A) orig (sec.)	-	6377.3	6377.3
(B) spec1 (sec.)	9.7	388.3	398.0
ratio (A/B)	-	16.4	16.0
(C) Trick (sec.)	21.9	347.0	368.9
ratio (A/C)	-	18.4	17.3
(D) spec2 (sec.)	4.5	547.5	552.0
ratio (A/D)	-	11.6	11.6

Table 1. Effect of specialization for `ray-trace`. The image size is 32×32 pixels. All produce the same image.

Notice that the experiment was done *interactively*: we first input the whole program and execute it; we then redefine parts of it to include the changes for specialization; finally the modified program is executed again with specialization. We will further modify the program below. This is achieved through the integration of partial evaluation into standard interpretation.

The row (C) shows the time for the program after we applied so-called "The Trick" [19]. In the calculation of diffuse reflection and specular reflection, we need a normal vector at the intersecting point. It is calculated from an object on which the intersecting point resides. Since the object is unknown until the intersecting point is known, and the intersecting point is unknown until the view-vector is known, the calculation of a normal vector cannot be done during specialization at all under the unknown view-vector. However, knowing that the object is one of the objects in the original configuration, we can make a case branch for each object in the configuration and perform some computation that depends on each object only. This replacement of unknown value with a case branch on all the possible variations is called "The Trick."

We implemented "The Trick" as follows. Suppose that we want to calculate a normal vector of `an-object` at `intersecting-point`:

```
(get-normal-vector an-object intersecting-point)
```

We first replace the above program with the following:

```
(let ((obj (bounded-variation an-object object)))
  (get-normal-vector obj intersecting-point))
```

where `object` is a (known) list of all the objects in the original configuration. Then, we add the following definition:

```
(define (bounded-variation obj lst)
  (if2 (null? (cdr lst))
       (car lst) ; the last element
       (if2 (eq? obj (car lst))
            (car lst)
            (bounded-variation obj (cdr lst)))))
```

where `if2` is a special kind of conditional described soon. The function `bounded-variation` extracts `obj` from a known `lst` and returns the object found in `lst`. From the standard evaluation point of view, the returned object is the same as the original one (assuming that `obj` is always found in `lst`). From the partial evaluation point of view, however, there is a significant difference between the input and the output: the input is unknown, but the output is known. This enables further specialization in the calculation of normal vectors.

A special form `if2` is supported in our specializer. In the standard evaluation, it is the same as `if`. During partial evaluation, it behaves like `if` except that it duplicates and propagates its continuation into the then- and else-branches. For example, `(+ (if d 1 2) 3)` where `d` is unknown is specialized into itself because the value of `(if d 1 2)` is unknown at specialization time, while specialization of `(+ (if2 d 1 2) 3)` becomes `(if d 4 5)` where the continuation "add 3" is duplicated and propagated into then- and else-branches. Duplication and propagation of continuation is easily implemented if a specializer is written in continuation-passing style or if we use delimited continuation operators [11, 20].

The use of `if2` has positive and negative effects. The positive side is more specialization. Since the result of each branch is propagated to the continuation, we can expect more specialized output if the value of each branch is useful for the continuation. The negative side is the code explosion and hence the longer specialization time. Since whole the continuation is duplicated, the code size may be doubled if no specialization could be done at all. Nested `if2` would make the code size further bigger. In fact, we tried to execute the ray tracing program where all the `if` special forms are treated as `if2`, but could not obtain the result. Since the use of `if2` is not always effective, both `if` and `if2` are supported in the implementation.

We use `if2` to make a case branch for the possible objects in the configuration. In each branch of `if2`, one of the known objects is returned and used in the continuation (where the calculation of a normal vector is specialized). The resulting program is a big case statement, each branch of which contains a specialized normal vector calculation program.

The effect of "The Trick" is shown in Table 1 (C). Because the continuation of `bounded-variation` is duplicated and executed as many times as the number of objects, the specialization time becomes larger. Once the specialization is done, however, the execution time is further reduced from the case (B).

Finally, the row (D) in Table 1 shows the specialization and execution time where only the function that calculates intersection is specialized. We changed the intersection function in the same way as we did for `ray-trace`. Since we specialize only the intersection function, the specialization time is very small, yet we achieve rather significant speedup compared to the original case. This shows that the specialization of intersection is the major factor of the speedup.

9 Related Work

Applying specialization during standard evaluation (in-place partial evaluation) is studied by Balat and Danvy [5]. They use type-directed partial evaluation [10] to achieve strong normalization. We use the standard partial evaluation without any type information to achieve the similar goal.

Currently, we do not detect when a specialized program is invalidated by side-effects after the specialization. More sophisticated mechanisms would be possible, *e.g.*, the one proposed by Volanschi *et al.* [27], where respecialization occurs when the heap objects that the specialized program depends on are modified. Hölzle *et al.* [17] describes the deoptimization technique that transforms compiled code into new versions when it is invalidated. We hope to introduce such mechanisms on top of the current framework in the future.

Although the original motivation of this work was to use partial evaluation in the interactive environment, it seems that the technique proposed here can be regarded as a foundation of run-time specialization [9, 21]. What we did in the ray tracing experiment was the specialization of a part of the program at run-time. Although our specializer is very heavy compared to the existing run-time specialization systems, it addresses one of the important characteristics of run-time specialization, namely, sharing of heap objects between standard and partial evaluation.

MetaML [24] allows one to manually separate a program into multiple stages. So to speak, it allows one to manually perform binding-time analysis. Although MetaML is not suited for partially evaluating programs interactively (since one has to annotate programs every time he wants to perform specialization), MetaML and our work have one important technique in common: the sharing of objects between standard and partial evaluation corresponds to *cross-stage persistence* in MetaML.

Specialization of ray tracing was first done by Mogensen [22]. Andersen [1] specialized an already efficient ray tracer written in C. The realistic application of specialization in graphics is also found in [15].

10 Conclusion

This paper described our first step towards the integration of partial evaluation into standard interpretation. The two main issues in this integration are the treatment of heap objects and side-effects. To enable specialization with respect to the heap objects that are allocated beforehand in standard evaluation, specialization is performed with respect to *the heap state at specialization time* rather than based on a program text input so far. To avoid duplication of heap objects, *direct references to heap objects* are allowed in the specialized programs, in addition to the use of the conventional let-insertion technique. Despite our modest approach on side-effects that we allow partial evaluation of only side-effect free portions of a program, the resulting system covers some typical cases, such as the use of partial evaluation during an interactive debugging session. It

also enables us to specialize a (side-effect free) program with respect to cyclic data structures, which was not easy before. We formalized this integration using store semantics and stated its correctness. The system is implemented in Scheme and various examples are tested. Among them, we have shown the specialization of a ray tracing program, and reported significant speedup.

In the future, we are planning to (1) develop the relationship between the preaction-based specialization and the conventional continuation-based let-insertion, (2) use the integrated specializer as a compiler for reflective languages [4], and (3) investigate the run-time specialization aspects of our system.

Acknowledgements. I would like to thank Olivier Danvy and Robert Glück for many comments and suggestions to the earlier versions of this paper.

References

1. Andersen, P. H. "Partial Evaluation Applied to Ray Tracing," *Software Engineering in Scientific Computing*, pp. 78–85, DIKU report D-289 (1996).
2. Asai, K. "Binding-Time Analysis for Both Static and Dynamic Expressions," In A. Cortesi and G. Filé, editors, *Static Analysis (LNCS 1694)*, pp. 117–133 (September 1999).
3. Asai, K., H. Masuhara, and A. Yonezawa "Partial Evaluation of Call-by-value λ-calculus with Side-effects," *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pp. 12–21 (June 1997).
4. Asai, K., S. Matsuoka, and A. Yonezawa "Duplication and Partial Evaluation — For a Better Understanding of Reflective Languages —," *Lisp and Symbolic Computation*, Vol. 9, Nos. 2/3, pp. 203–241, Kluwer Academic Publishers (May/June 1996).
5. Balat, V., and O. Danvy "Strong Normalization by Type-Directed Partial Evaluation and Run-Time Code Generation," In X. Leroy and A. Ohori editors, *Types in Compilation (LNCS 1473)* pp. 240–252 (March 1998).
6. 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).
7. 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).
8. Consel, C., P. Jouvelot, and P. Ørbæk "Separate Polyvariant Binding Time Reconstruction," Ecole des Mines, CRI Report A/261 (October 1994).
9. Consel, C., and F. Noël "A General Approach for Run-Time Specialization and its Application to C," *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 145–156 (January 1996).
10. Danvy, O. "Type-Directed Partial Evaluation," *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 242–257 (January 1996).
11. Danvy, O., and A. Filinski "Abstracting Control," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).
12. Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen "The Essence of Compiling with Continuations," *Proceedings of the ACM SIGPLAN '93 Conference on*

Programming Language Design and Implementation (PLDI), pp. 237–247 (June 1993).
13. Futamura, Y. "Partial evaluation of computation process – an approach to a compiler-compiler," *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, (1971), reprinted as *Higher-Order and Symbolic Computation*, Vol. 12, No. 4, pp. 381–391, Kluwer Academic Publishers (December 1999).
14. Glück, R., and J. Jørgensen "An Automatic Program Generator for Multi-Level Specialization," *Lisp and Symbolic Computation*, Vol. 10, No. 2, pp. 113–158, Kluwer Academic Publishers (July 1997).
15. Guenter, B., T. B. Knoblock, E. Ruf "Specializing Shaders," *Proceedings of SIGGRAPH 95 (Computer Graphics Proceedings)* pp. 343–350 (1995).
16. Hatcliff, J. "An Introduction to Online and Offline Partial Evaluation Using a Simple Flowchart Language," In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann editors, *Partial Evaluation, Practice and Theory (LNCS 1706)*, pp. 20–82 (1999).
17. Hölzle, U., C. Chambers, and D. Ungar "Debugging Optimized Code with Dynamic Deoptimization," *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, pp. 32–43 (June 1992).
18. Hornof, L., C. Consel, and J. Noyé "Effective Specialization of Realistic Programs via Use Sensitivity," In Van Hentenryck P., editor, *Static Analysis (LNCS 1302)*, pp. 63–73 (1997).
19. Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).
20. Lawall, J. L., and O. Danvy "Continuation-Based Partial Evaluation," *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 227–238 (June 1994).
21. Lee, P., and M. Leone "Optimizing ML with run-time code generation," *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pp. 137–148 (May 1996).
22. Mogensen, T. Æ. "The Application of Partial Evaluation to Ray-Tracing," Master's thesis, DIKU, University of Copenhagen (1986).
23. 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.
24. Taha, W. *Multi-Stage Programming: Its Theory and Applications*, Ph.D. thesis, Oregon Graduate Institute of Science and Technology (November 1999).
25. Thiemann, P. J. "Cogen in Six Lines," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pp. 180–189 (May 1996).
26. Thiemann, P. J., and D. Dussart "Partial Evaluation for Higher-Order Languages with State," Available at `http://www.informatik.uni-freiburg.de/~thiemann/papers/mlpe.ps.gz`.
27. Volanschi, E. N., C. Consel, G. Muller, and C. Cowan "Declarative Specialization of Object-Oriented Program," *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, pp. 286–300, (October 1997).