# Online Type-Directed Partial Evaluation for Dynamically-Typed Languages*

Eijiro Sumii Naoki Kobayashi
{sumii, koba}@yl.is.s.u-tokyo.ac.jp
University of Tokyo

May 31, 1999†

**Abstract**

This article presents an alternative method of type-directed partial evaluation, which is simpler and more efficient than previous methods. Unlike previous methods, it is straightforwardly applicable to functional languages with various powerful type systems. As an extreme instance, this article mainly deals with a dynamically-typed functional language like Scheme. The key idea is to extend primitive value destructors such as function application and pair destruction (`car` and `cdr`), so that they generate residual code when their operands are dynamic. It unnecessitates an operation in type-directed partial evaluation called reflection, which was the major cause of complication and inefficiency in previous methods.

We formalize our method as an extension of two-level $\lambda$-calculus, and prove it correct. Furthermore, we show that our type-directed partial evaluator can be derived from a simple online syntax-directed partial evaluator with higher-order abstract syntax, by the same transformation as an offline program-generator-generator (cogen) is derived from an offline syntax-directed partial evaluator. This clarifies why type-directed partial evaluation is faster than ordinary syntax-directed partial evaluation, and implies that we can obtain more powerful partial evaluators by replacing the underlying syntax-directed partial evaluator with more sophisticated ones.

# 1 Introduction

## 1.1 Background: What is Type-Directed Partial Evaluation?

*Partial Evaluation* is program manipulation that, given a program `p` and a part of its input `s`, generates a specialized program $\mathtt{p}_\mathtt{s}$ satisfying `p @ s @ d` $= \mathtt{p}_\mathtt{s}$ `@ d` for the rest of the input `d`. (In this article, we explicitly write @ for function application, and give it higher precedence than $\lambda$.) It is closely related to *strong normalization* of a program: strong normalization of `p @ s` amounts to partial evaluation of `p` with respect to `s`.

For example, let `p` be $\lambda$`f. `$\lambda$`x. f @ (f @ x)` and `s` be $\lambda$`z. z` in $\lambda$-calculus. Then, $\mathtt{p}_\mathtt{s}$ should be $\lambda$`x. x`, which is the strong normal form of `p @ s`.

Traditional *syntax-directed partial evaluation* performs the normalization by symbolically manipulating the abstract syntax tree of a term, while *type-directed partial evaluation* [4] does it by expanding the term to a two-level term according to the type, and reducing the expanded two-level term statically and weakly. The expansion is called *reification*.

In the above example, `p @ s` can be typed as $\alpha \to \alpha$. Since the type tells us that `p @ s` is a function, we expand it to $\underline{\lambda}$`v. p @ s @ `$\underline{\mathtt{v}}$, which reduces to $\underline{\lambda}$`v. `$\underline{\mathtt{v}}$. Here, $\underline{\mathtt{v}}$ is a fresh symbol and $\underline{\lambda}$ is the syntax constructor of $\lambda$-abstraction. In Scheme [8], this procedure can be illustrated as below. Generation of the fresh symbol is omitted for brevity.

*A reformatted version of the article that appeared in *Computer Software*, vol. 17, no. 3, pp. 38–62, May 2000, Iwanami Shoten, Japan.

†Revised on September 20, 1999. Reformatted on June 18, 2000

$$
\begin{array}{lcll}
\mathit{tdpe}(v) & = & \mathcal{R}(\downarrow_\tau v) & \text{(where } \tau \text{ is a type of } v\text{)} \\
\downarrow_\alpha v & = & v & \\
\downarrow_{\sigma\to\tau} v & = & \underline{\lambda x}.\ \downarrow_\tau (v\ @\ \uparrow_\sigma \underline{x}) & \text{(where } \underline{x} \text{ is fresh)} \\
\uparrow_\alpha e & = & e & \\
\uparrow_{\sigma\to\tau} e & = & \lambda x.\ \uparrow_\tau (e\ \underline{@}\ \downarrow_\sigma x) & \text{(where } x \text{ is fresh)}
\end{array}
$$

Figure 1: Type-Directed Partial Evaluation in Simply-Typed $\lambda$-Calculus

```
> (define p (lambda (f) (lambda (x) (f (f x)))))
> (define s (lambda (z) z))
> `(lambda (v) ,((p s) 'v))
(lambda (v) v)
```

When the codomain of a function is a compound type (such as function types), type-directed partial evaluation performs reification recursively. For example, let us normalize a term `p` @ `s` of the type $\alpha \to \beta \to \alpha$, where `p` $= \lambda$`f`. $\lambda$`x`. $\lambda$`y`. `f` @ `x` @ `y` and `s` $= \lambda$`a`. $\lambda$`b`. `a`. Since the term has a function type, we expand it to $\underline{\lambda \mathtt{v}}$. `p` @ `s` @ $\underline{\mathtt{v}}$. Then, since `p` @ `s` @ $\underline{\mathtt{v}}$ also has a function type, we again expand it to $\underline{\lambda \mathtt{w}}$. `p` @ `s` @ $\underline{\mathtt{v}}$ @ $\underline{\mathtt{w}}$. As a whole, we obtain $\underline{\lambda \mathtt{v}}$. $\underline{\lambda \mathtt{w}}$. `p` @ `s` @ $\underline{\mathtt{v}}$ @ $\underline{\mathtt{w}}$, which reduces to $\underline{\lambda \mathtt{v}}$. $\underline{\lambda \mathtt{w}}$. $\underline{\mathtt{v}}$.

```
> (define p (lambda (f) (lambda (x) (lambda (y) ((f x) y)))))
> (define s (lambda (a) (lambda (b) a)))
> `(lambda (v) (lambda (w) ,(((p s) 'v) 'w)))
(lambda (v) (lambda (w) v))
```

When the *domain* of a function is a compound type, a problem arises. For example, let us try to normalize `p` @ `s` where `p` $= \lambda$`x`. $\lambda$`f`. `f` @ (`1` + `x`) and `s` $=$ `2`. It can be typed as $(\mathit{int} \to \alpha) \to \alpha$. Since the term has a function type, we want to expand it to $\underline{\lambda \mathtt{v}}$. `p` @ `s` @ $\underline{\mathtt{v}}$. If we naively do so, however, a type error happens at the specialization stage, because the symbol $\underline{\mathtt{v}}$ is applied to an integer `3`.

```
> (define p (lambda (x) (lambda (f) (f (+ 1 x)))))
> (define s 2)
> `(lambda (v) ,((p s) 'v))
Error: attempt to apply non-procedure v.
```

Danvy's original type-directed partial evaluator solves this problem by coercing the symbol $\underline{\mathtt{v}}$ to a value of the type $\mathit{int} \to \alpha$ by expanding it to a two-level term $\lambda$`a`. $\underline{\mathtt{v}}\ \underline{@}$ `a`, where the operator $\underline{@}$ is the syntax constructor of function application.

```
> `(lambda (v) ,((p s) (lambda (a) `(v ,a))))
(lambda (v) (v 3))
```

In general, when a function is applied to a symbol, the symbol is expanded to a value in the domain of the function. The expansion is called *reflection*.

As a whole, a type-directed partial evaluator for simply-typed $\lambda$-calculus looks like Figure 1, where $\mathcal{R}$, $\downarrow$ and $\uparrow$ denote static weak reduction, reification and reflection, respectively. Note that $\downarrow$ and $\uparrow$ work according to the types, hence the name of type-directed partial evaluation.

Type-directed partial evaluation is known to be much faster than traditional syntax-directed one, because it directly executes the source program instead of symbolically manipulating the abstract syntax tree.

## 1.2 Problem: Reflection Doesn't Always Work

**Reflection is difficult:** Although the above method works for the pure simply-typed $\lambda$-calculus, it doesn't straightforwardly apply to practical programming languages such as Scheme and ML [10], because it is impossible

to coerce a symbol into an arbitrary type. For example, how can we coerce a symbol into an integer? The more powerful a type system becomes, the more difficult (if not impossible) the reflection becomes. For instance, reflection over disjoint sum types and inductive types is impossible or complicated [4, 14]. An extreme case is dynamic typing. In dynamically-typed languages such as Scheme, the domain of a function is essentially undecidable at the specialization stage, so reflection is impossible.

**Reflection is inefficient:** In addition to the above problem, reflection often causes inefficiency both in specialization of a source program and in execution of the residual program, at least without post-processing such as $\eta$-reduction.

For example, let $\mathtt{p} = \lambda\mathtt{f}.\ \lambda\mathtt{x}.$ **if** *true* **then** $\mathtt{f}\ @\ \mathtt{x}$ **else** $\mathtt{f}\ @\ (\lambda\mathtt{y}.\ \mathtt{y})$ and $\mathtt{s} = \lambda\mathtt{z}.\ \mathtt{z}$. Then, $\mathtt{p}\ @\ \mathtt{s}$ can be typed as $(\alpha \to \alpha) \to (\alpha \to \alpha)$. When $\mathtt{p}\ @\ \mathtt{s}$ is reified, a fresh symbol $\underline{\mathtt{v}}$ is reflected to a function $\lambda\mathtt{a}.\ \underline{\mathtt{v}}\ @\ \mathtt{a}$, to which $\mathtt{p}\ @\ \mathtt{s}$ is applied. Then, the function $\lambda\mathtt{a}.\ \underline{\mathtt{v}}\ @\ \mathtt{a}$ is returned, which is reified to an expression $\underline{\lambda\mathtt{w}}.\ \underline{\mathtt{v}}\ @\ \underline{\mathtt{w}}$. As a result, $\mathtt{p}\ @\ \mathtt{s}$ is reified to a redundant expression $\underline{\lambda\mathtt{v}}.\ \underline{\lambda\mathtt{w}}.\ \underline{\mathtt{v}}\ @\ \underline{\mathtt{w}}$.

```
> (define p
    (lambda (f)
      (lambda (x)
        (if #t
            (f x)
            (f (lambda (y) y))))))
> (define s (lambda (z) z))
> `(lambda (v)
     ,(let* ((arg (lambda (a) `(v ,a)))
             (res ((p s) arg)))
        `(lambda (w) ,(res 'w))))
(lambda (v) (lambda (w) (v w)))
```

In this case, if we hadn't reflected $\underline{\mathtt{v}}$, we would have applied $\mathtt{p}\ @\ \mathtt{s}$ to $\underline{\mathtt{v}}$ and obtained $\underline{\mathtt{v}}$. Thus, we could have reified the source program $\mathtt{p}\ @\ \mathtt{s}$ to a much simpler residual program $\underline{\lambda\mathtt{v}}.\ \underline{\mathtt{v}}$.

```
> `(lambda (v) ,((p s) 'v))
(lambda (v) v)
```

In general, such unnecessary reflection and lengthy result occur when the type doesn't exactly express how the function actually behaves. Formally, type-directed partial evaluation produces a *long $\beta$-$\eta$-normal form* [2], which may contain $\eta$-redices. Such inexactness is inevitable in type systems with complete type inference (e.g. the Hindley-Milner type system).

For the purpose of avoiding this problem, Sheard [14] proposed a technique that he calls *lazy reflection*. However, his solution is unsatisfactory: it cannot normalize $\mathtt{p}\ @\ \mathtt{s}$ to $\underline{\lambda\mathtt{v}}.\ \underline{\mathtt{v}}$ in the above example, unless we annotate the program *by hand* with more exact type information as below ($\Lambda$ and [ ] denotes type abstraction and type application).

$$\begin{array}{l}\Lambda\alpha.\ \lambda f\colon(\forall\beta.\ \beta \to \beta).\ \lambda x\colon\alpha.\\ \quad \textbf{if}\ \mathit{true}\ \textbf{then}\ f[\alpha]\ @\ x\\ \qquad\qquad \textbf{else}\ f[\forall\gamma.\ \gamma \to \gamma]\ @\ (\Lambda\gamma.\ \lambda y\colon\gamma.\ y)\end{array}$$

In addition, reflection over disjoint sum types (such as *bool*) causes code duplication. For example, reifying a function $\lambda\mathtt{f}.\ \lambda\mathtt{x}.\ \lambda\mathtt{y}.\ \lambda\mathtt{b}.\ \lambda\mathtt{c}.\ \mathtt{f}\ @\ (\mathtt{f}\ @\ (\mathtt{f}\ @\ (\textbf{if}\ \mathtt{b}\ \mathit{xor}\ \mathtt{c}\ \textbf{then}\ \mathtt{x}\ \textbf{else}\ \mathtt{y})))$ with respect to a type $(\alpha \to \alpha) \to \alpha \to \alpha \to \mathit{bool} \to \mathit{bool} \to \alpha$ results in the residual code below. Lazy reflection doesn't solve this problem either.

$$\begin{array}{l}\lambda f.\ \lambda x.\ \lambda y.\ \lambda b.\ \lambda c.\\ \quad \textbf{if}\ b\ \textbf{then}\ (\textbf{if}\ c\ \textbf{then}\ f\ @\ (f\ @\ (f\ @\ y))\\ \qquad\qquad\qquad\qquad \textbf{else}\ f\ @\ (f\ @\ (f\ @\ x)))\\ \qquad\quad \textbf{else}\ (\textbf{if}\ c\ \textbf{then}\ f\ @\ (f\ @\ (f\ @\ x))\\ \qquad\qquad\qquad\qquad \textbf{else}\ f\ @\ (f\ @\ (f\ @\ y)))\end{array}$$

## 1.3 Solution: Avoid Reflection and Make Destructors Smart

In order to avoid the above problems, we don't perform reflection at all. Instead, we conservatively extend primitive value destructors, so that they correctly handle residual code. A similar idea has already been proposed in previous work [5, 14], but its application was rather limited, probably because reflection was still used. We exploit the idea much more extensively, and completely unnecessitate reflection.

For example, we extend function application from @ to @′, where $\mathtt{e_1}$ @′ $\mathtt{e_2} = \mathtt{e_1}$ @ $\mathtt{e_2}$ if $\mathtt{e_1}$ evaluates to a static function (e.g. $\lambda\mathtt{z}.\ \mathtt{z}$), and $\mathtt{e_1}$ @′ $\mathtt{e_2} = \mathtt{e_1}$ $\underline{@}'$ $\mathtt{e_2}$ if $\mathtt{e_1}$ evaluates to a dynamic expression (e.g. $\underline{\mathtt{v}}$). Here, we assume that we can distinguish dynamic expressions from static values during partial evaluation, as in ordinary online partial evaluation.

How does it work? In the previous example, `x` was not actually used as a function, so reflection was obviously unnecessary. However, what will happen if we try to reify, say, $\lambda\mathtt{g}.\ \mathtt{g}$ @′ $(\mathtt{1}+\mathtt{2})$? It can be typed as $(\mathit{int} \to \alpha) \to \alpha$, but that doesn't matter here. Whatever type the domain is, we just apply the function to a symbol, like $\underline{\lambda\mathtt{v}}.\ (\lambda\mathtt{g}.\ \mathtt{g}$ @′ $(\mathtt{1}+\mathtt{2}))$ @′ $\underline{\mathtt{v}}$. It reduces to $\underline{\lambda\mathtt{v}}.\ \underline{\mathtt{v}}$ @′ `3`, and then reduces to $\underline{\lambda\mathtt{v}}.\ \underline{\mathtt{v}}\ \underline{@}'$ `3`, thanks to the extended function application.

```
> (define (_@ func arg) ; implements @′
    (if (dynamic? func)
        `(_@ ,func ,arg)
        (func arg)))
> (define f (lambda (g) (_@ g (+ 1 2))))
> `(lambda (v) ,(f 'v))
(lambda (v) (_@ v 3))
```

We leave @′ instead of @ in the residual code for the purpose of multi-level specialization. When we have no more dynamic input, we can safely replace @′ with @.

## 1.4 Comparison with Previous Methods

**Applicability:** Previous methods of type-directed partial evaluation were only applicable to languages with particular static type systems, because each new kind of type required a new mechanism for reflection. For example, reflection over disjoint sum types, polymorphic types, and inductive types required partial continuation, type passing, and lazy reflection, respectively [14]. In contrast, our method is applicable to languages with various type systems including dynamic ones, provided that (1) a static value (e.g. `3`) and a dynamic expression that *evaluates* to the static value (e.g. `'(_@ (lambda (x) x) 3)`) can be merged into the same type, and (2) static values and dynamic expressions can be distinguished during partial evaluation.

**Efficiency:** Our type-directed partial evaluator generates more efficient residual code than previous ones, because it doesn't perform reflection at all. It may seem another source of inefficiency at the specialization stage that destructors must examine binding time of the operands. However, the examination was also necessary in the previous work for primitive data (e.g. integers) and lazy reflection, so the additional cost would be small. In addition, binding-time analysis on the source programs would help to reduce unnecessary tests in the destructors. For example, if $\mathtt{e_1}$ in $\mathtt{e_1}$ @′ $\mathtt{e_2}$ is known to be static, the @′ can safely be replaced with an @.

## 1.5 Relationship with Syntax-Directed Partial Evaluation

Although our type-directed partial evaluator is still directed by the types of the values to reify, it also seems like a *syntax-directed* partial evaluator embedded in the source language. Actually, it corresponds to an online version of Thiemann's cogen-based approach [16, 17] to syntax-directed partial evaluators with higher-order abstract syntax.

This observation is significant for the following two reasons.

- It clarifies why type-directed partial evaluation is faster than traditional (i.e., not cogen-based) syntax-directed one: the former involves less interpretive overhead such as syntax dispatch and environment manipulation than the latter.

$$
\begin{array}{rcccc}
t & ::= & \underline{x} & | & \underline{\lambda x.\ t} \\
 & | & t_1\ \underline{@'}\ t_2 & | & \underline{\mathbf{pair}}(t_1, t_2) \\
 & | & \underline{\mathbf{fst}'}(t) & | & \underline{\mathbf{snd}'}(t) \\
 & | & x & | & \lambda x.\ t \\
 & | & t_1\ @'\ t_2 & | & \mathbf{pair}(t_1, t_2) \\
 & | & \mathbf{fst}'(t) & | & \mathbf{snd}'(t) \\
 & | & \downarrow t & & \\
v & ::= & \underline{x} & | & \underline{\lambda x.\ v} \\
 & | & v_1\ \underline{@'}\ v_2 & | & \underline{\mathbf{pair}}(v_1, v_2) \\
 & | & \underline{\mathbf{fst}'}(v) & | & \underline{\mathbf{snd}'}(v) \\
 & | & \lambda x.\ t & | & \mathbf{pair}(v_1, v_2)
\end{array}
$$

Figure 2: Terms and Values of the Object Language

- It implies that we can combine the flexibility of syntax-directed partial evaluation and the efficiency of type-directed one, by applying the same approach to more powerful syntax-directed partial evaluators.

### 1.6 Contribution

The main contribution of our work in this article is (1) proposal and formalization of a novel method of online type-directed partial evaluation, which is simpler and more powerful than previous methods, and (2) clarification of the correspondence between type-directed partial evaluation and a cogen-based approach to syntax-directed partial evaluation with higher-order abstract syntax, which enables incorporation of both techniques.

Our method is not so useful for Gödelization [11] as it is for partial evaluation, because it requires extension of destructors (e.g. from @ to @′) or preprocessing of the source program (to replace, e.g., @ with @′). Furthermore, this article does not address issues on side effects (including type errors and non-termination) in detail, though we may incorporate existent techniques that find critical computations by program analysis and residualize them by `let` insertion ([1], for example).

### 1.7 Overview

The rest of this article is structured as follows. Section 2 formally presents the object language and our partial evaluator. Section 3 explains the derivation of our type-directed partial evaluator from an online syntax-directed partial evaluator with higher-order abstract syntax. Section 4 shows the results of our experiments and compares our method with previous methods of type-directed partial evaluation. Section 5 discusses extensions and limitations of our method, Section 6 mentions other related work, and Section 7 concludes this article. The appendix gives an implementation and examples in a subset of Scheme.

## 2 Formalization

### 2.1 The Object Language

Our object language is a variant of two-level $\lambda$-calculus with pairs. Its abstract syntax is defined in Figure 2. All the terms in the first, second, and third rows are called *dynamic*, while all the terms in the fourth, fifth, and sixth rows are called *static*. A term is called *completely dynamic* (resp. *completely static*) if all the subterms are dynamic (resp. static). An ordinary program should be completely static. Destructors (@, **fst** and **snd**) are annotated with $'$, because they are online in the sense that they are defined both for static values and for dynamic expressions. As usual, we do not care about the names of bound variables (both static and dynamic), and implicitly perform $\alpha$-conversion at any time.

We consider two reduction relations in the calculus: the *strong* reduction relation $\rightarrow_s$ and the *weak* reduction relation $\rightarrow_w$. The reduction rules are standard except for the destructors @′, **fst**′ and **snd**′. The weak reduction

$$
\begin{array}{rcll}
(\lambda x.\ t)\ @'\ v & \to_w & [v/x]t & \\
\mathbf{fst}'(\mathbf{pair}(v_1, v_2)) & \to_w & v_1 & \\
\mathbf{snd}'(\mathbf{pair}(v_1, v_2)) & \to_w & v_2 & \\
v_1\ @'\ v_2 & \to_w & v_1\ \underline{@}'\ v_2 & (\text{if } v_1 \text{ is dynamic}) \\
\mathbf{fst}'(v) & \to_w & \underline{\mathbf{fst}}'(v) & (\text{if } v \text{ is dynamic}) \\
\mathbf{snd}'(v) & \to_w & \underline{\mathbf{snd}}'(v) & (\text{if } v \text{ is dynamic})
\end{array}
$$

Figure 3: Weak Reduction Relation of the Object Language (1): Base Cases Except for $\downarrow$

$$
\begin{array}{lcll}
\downarrow(\lambda x.\ t) & \to_w & \underline{\lambda y}.\ \downarrow((\lambda x.\ t)\ @'\ \underline{y}) & (\text{where } \underline{y} \text{ is fresh}) \quad \cdots ② \\
\downarrow \mathbf{pair}(v_1, v_2) & \to_w & \underline{\mathbf{pair}}(\downarrow v_1, \downarrow v_2) & \\
\downarrow \underline{x} & \to_w & \underline{x} & \\
\downarrow(v_1\ \underline{@}'\ v_2) & \to_w & \downarrow v_1\ \underline{@}'\ \downarrow v_2 & \cdots ③ \\
\downarrow \underline{\mathbf{fst}}'(v) & \to_w & \underline{\mathbf{fst}}'(\downarrow v) & \\
\downarrow \underline{\mathbf{snd}}'(v) & \to_w & \underline{\mathbf{snd}}'(\downarrow v) &
\end{array}
$$

Figure 4: Weak Reduction Relation of the Object Language (2): Base Cases for $\downarrow$

$$
\begin{array}{rcccc}
C_w[\,] & ::= & \downarrow[\,] & | & \underline{\lambda x}.\ [\,] \\
& | & [\,]\ \underline{@}'\ t & | & t\ \underline{@}'\ [\,] \\
& | & \underline{\mathbf{pair}}([\,], t) & | & \underline{\mathbf{pair}}(t, [\,]) \\
& | & \underline{\mathbf{fst}}'([\,]) & | & \underline{\mathbf{snd}}'([\,]) \\
& | & [\,]\ @'\ t & | & t\ @'\ [\,] \\
& | & \mathbf{pair}([\,], t) & | & \mathbf{pair}(t, [\,]) \\
& | & \mathbf{fst}'([\,]) & | & \mathbf{snd}'([\,])
\end{array}
$$

$$
\frac{t_1 \to_w t_2}{C_w[t_1] \to_w C_w[t_2]} \cdots ①
$$

Figure 5: Weak Reduction Relation of the Object Language (3): Weak Reduction Context

$$
\begin{array}{rcll}
(\lambda x.\ t_1)\ @'\ t_2 & \to_s & [t_2/x]t_1 & \\
\mathbf{fst}'(\mathbf{pair}(t_1, t_2)) & \to_s & t_1 & \\
\mathbf{snd}'(\mathbf{pair}(t_1, t_2)) & \to_s & t_2 & \\
t_1\ @'\ t_2 & \to_s & t_1\ \underline{@}'\ t_2 & (\text{if } t_1 \text{ is dynamic}) \\
\mathbf{fst}'(t) & \to_s & \underline{\mathbf{fst}}'(t) & (\text{if } t \text{ is dynamic}) \\
\mathbf{snd}'(t) & \to_s & \underline{\mathbf{snd}}'(t) & (\text{if } t \text{ is dynamic})
\end{array}
$$

Figure 6: Strong Reduction Relation of the Object Language (1): Base Cases

$$C_s[\,] \quad ::= \quad C_w[\,] \quad | \quad \lambda x.\ [\,]$$

$$\frac{t_1 \to_s t_2}{C_s[t_1] \to_s C_s[t_2]}$$

Figure 7: Strong Reduction Relation of the Object Language (2): Strong Reduction Context

relation $\to_w$ is the smallest relation that satisfies the rules in Figure 3, Figure 4, and Figure 5, where $[v/x]t$ denotes the usual substitution of $v$ for free $x$'s in $t$. The strong reduction relation $\to_s$ is the smallest relation that satisfies the rules in Figure 6 and Figure 7. For example, $\lambda x.\ \lambda y.\ (\lambda z.\ z)\ @'\ (x\ @'\ y) \to_s \lambda x.\ \lambda y.\ (x\ @'\ y)$. We write $\to_w^*$ (resp. $\to_s^*$) for the reflexive transitive closure of $\to_w$ (resp. $\to_s$). We also write $t \not\to_w$ (resp. $t \not\to_s$) if there exists no such $t_0$ that $t \to_w t_0$ (resp. $t \to_s t_0$).

$\downarrow$ is the reification operator. Basically, it operates on a static value. However, it also operates on dynamic terms introduced during reduction, and reifies static values embedded in dynamic terms by $@'$ (like $\underline{x}\ @'\ (\lambda x.\ x) \to \underline{x}\ \underline{@}'\ (\lambda x.\ x)$). Alternatively, if we redefine the reduction rule for $@'$ as $t_1\ @'\ t_2 \to t_1\ \underline{@}'\ \downarrow t_2$ when $t_1$ is dynamic, we can simplify the reduction rules for $\downarrow$ as $\downarrow t \to t$ when $t$ is dynamic. Although this alternative method seems simpler and more efficient, we formalize the original method for the discussion in Section 3.

Since our object language is dynamically-typed, the reification operator is directed by dynamically examined types instead of statically given ones. It performs case analysis on the binding-time (whether static or dynamic) and the type of the value to reify, as $\downarrow$ in Figure 4 and `reify_` in the Appendix do. The latter four lines in Figure 4 may seem syntax-directed, but it is not essential because it can be omitted (as it is in the Appendix) by means of the alternative method described above. This is the reason why we call our partial evaluator online and type-directed.[1]

**Example 2.1** *Let $t = p\ @'\ s$ where $p = \lambda f.\ \lambda x.\ f\ @'\ (f\ @'\ x)$ and $s = \lambda z.\ z$. Then, strong reduction of $t$ yields $\lambda x.\ x$ and weak reduction of $\downarrow t$ yields $\underline{\lambda}\underline{y}.\ \underline{y}$ as follows.*

$$\begin{aligned}
t &= (\lambda f.\ \lambda x.\ f\ @'\ (f\ @'\ x))\ @'\ (\lambda z.\ z)\\
&\to_s \lambda x.\ (\lambda z.\ z)\ @'\ ((\lambda z.\ z)\ @'\ x)\\
&\to_s \lambda x.\ (\lambda z.\ z)\ @'\ x\\
&\to_s \lambda x.\ x
\end{aligned}$$

$$\begin{aligned}
\downarrow t &= \downarrow((\lambda f.\ \lambda x.\ f\ @'\ (f\ @'\ x))\ @'\ (\lambda z.\ z))\\
&\to_w \downarrow(\lambda x.\ (\lambda z.\ z)\ @'\ ((\lambda z.\ z)\ @'\ x))\\
&\to_w \underline{\lambda}\underline{y}.\ \downarrow((\lambda x.\ (\lambda z.\ z)\ @'\ ((\lambda z.\ z)\ @'\ x))\ @'\ \underline{y})\\
&\to_w \underline{\lambda}\underline{y}.\ \downarrow((\lambda z.\ z)\ @'\ ((\lambda z.\ z)\ @'\ \underline{y}))\\
&\to_w \underline{\lambda}\underline{y}.\ \downarrow((\lambda z.\ z)\ @'\ \underline{y})\\
&\to_w \underline{\lambda}\underline{y}.\ \downarrow\underline{y}\\
&\to_w \underline{\lambda}\underline{y}.\ \underline{y}
\end{aligned}$$

**Example 2.2** *Let $t = \lambda f.\ f\ @'\ (\mathbf{fst}'(\mathbf{pair}(f, f)))$. Then, strong reduction of $t$ yields $\lambda f.\ f\ @'\ f$ and weak reduction of $\downarrow t$ yields $\underline{\lambda}\underline{g}.\ \underline{g}\ \underline{@}'\ \underline{g}$ as follows.*

$$\begin{aligned}
t &= \lambda f.\ f\ @'\ (\mathbf{fst}'(\mathbf{pair}(f, f)))\\
&\to_s \lambda f.\ f\ @'\ f
\end{aligned}$$

[1] In addition, type-directed partial evaluation has historically been implying not only that it is directed by types, but also that it is based on reification. Our partial evaluator is type-directed in this sense as well.

$$
\begin{array}{rcl}
\mathit{erase}(\underline{x}) & = & x \\
\mathit{erase}(\underline{\lambda x.\ t}) & = & \lambda x.\ \mathit{erase}(t) \\
\mathit{erase}(t_1\ \underline{@'}\ t_2) & = & \mathit{erase}(t_1)\ @'\ \mathit{erase}(t_2) \\
\mathit{erase}(\underline{\mathbf{pair}}(t_1, t_2)) & = & \mathbf{pair}(\mathit{erase}(t_1), \mathit{erase}(t_2)) \\
\mathit{erase}(\underline{\mathbf{fst}'}(t)) & = & \mathbf{fst}'(\mathit{erase}(t)) \\
\mathit{erase}(\underline{\mathbf{snd}'}(t)) & = & \mathbf{snd}'(\mathit{erase}(t))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{erase}(x) & = & x \\
\mathit{erase}(\lambda x.\ t) & = & \lambda x.\ \mathit{erase}(t) \\
\mathit{erase}(t_1\ @'\ t_2) & = & \mathit{erase}(t_1)\ @'\ \mathit{erase}(t_2) \\
\mathit{erase}(\mathbf{pair}(t_1, t_2)) & = & \mathbf{pair}(\mathit{erase}(t_1), \mathit{erase}(t_2)) \\
\mathit{erase}(\mathbf{fst}'(t)) & = & \mathbf{fst}'(\mathit{erase}(t)) \\
\mathit{erase}(\mathbf{snd}'(t)) & = & \mathbf{snd}'(\mathit{erase}(t))
\end{array}
$$

$$
\mathit{erase}(\downarrow t) = \mathit{erase}(t)
$$

Figure 8: Definition of $\mathit{erase}(t)$

$$
\begin{array}{rcl}
\downarrow t & = & \downarrow(\lambda f.\ f\ @'\ (\mathbf{fst}'(\mathbf{pair}(f, f)))) \\
& \to_w & \underline{\lambda g}.\ \downarrow((\lambda f.\ f\ @'\ (\mathbf{fst}'(\mathbf{pair}(f, f))))\ @'\ \underline{g}) \\
& \to_w & \underline{\lambda g}.\ \downarrow(\underline{g}\ @'\ (\mathbf{fst}'(\mathbf{pair}(\underline{g}, \underline{g})))) \\
& \to_w & \underline{\lambda g}.\ \downarrow(\underline{g}\ @'\ \underline{g}) \\
& \to_w & \underline{\lambda g}.\ \downarrow(\underline{g}\ \underline{@'}\ \underline{g}) \\
& \to_w & \underline{\lambda g}.\ \downarrow\underline{g}\ \underline{@'}\ \downarrow\underline{g} \\
& \to_w & \underline{\lambda g}.\ \underline{g}\ \underline{@'}\ \downarrow\underline{g} \\
& \to_w & \underline{\lambda g}.\ \underline{g}\ \underline{@'}\ \underline{g}
\end{array}
$$

## 2.2 Correctness

Below, we show that the above system is correct in the sense that, for any completely static and closed $t$, if $\downarrow t$ weakly normalizes to $t_0$ by the call-by-value reduction, then $t_0$ is completely dynamic and $\mathit{erase}(t_0)$ (the term obtained by erasing all the underlines in $t_0$) is the strong normal form of $t$. It is formally stated in Proposition 2.5.

Since our object language is dynamically-typed, a source program may be potentially type-unsafe and cause type errors during partial evaluation. For brevity, however, we assume that every source program is type-safe, in the sense that its reduction never causes type errors (which are formally defined in Definition 2.4). It is straightforward to prevent them by ordinary type checking, or to delay them until the execution stage by making them dynamic with additional reduction rules like $v_1\ @'\ v_2 \to_w v_1\ \underline{@'}\ v_2$ if $v_1 \neq \lambda x.\ t$.

**Definition 2.3** *$\mathit{erase}(t)$ denotes the term obtained by erasing all the underlines and $\downarrow$'s in $t$. It is formally defined in Figure 8.*

**Definition 2.4** *A term $t$ is called* type-safe *if there is no such term $t_0$ that $t \to_s^* t_0$ and $t_0$ contains a subterm whose form is either $\mathbf{pair}(v_1, v_2)\ @'\ t$, $\mathbf{fst}'(\lambda x.\ t)$ or $\mathbf{snd}'(\lambda x.\ t)$.*

Any term well-typed in an ordinary static type system (e.g. simply-typed $\lambda$-calculus) is type-safe.

**Proposition 2.5** *Let $t$ be a completely static, closed, type-safe term. If $\downarrow t \to_w^* t_n \not\to_w$, then $t_n$ is completely dynamic and $erase(t) \to_s^* erase(t_n) \not\to_s$.*

This proposition means that our partial evaluator is correct in the sense that ordinary evaluation (i.e., weak reduction) of $\downarrow t$ coincides with partial evaluation (i.e., strong reduction) of $t$. Note that it does not guarantee termination of the reification. See Subsection 5.1 for a remedy.

**Example 2.6** *Let $t = \lambda f.\ f\ @'\ (\mathbf{fst}'(\mathbf{pair}(f, f)))$. Then, $\downarrow t \to_w^* \underline{\lambda g}.\ g\ \underline{@}'\ g \not\to_w$ and $erase(t) \to_s^* \lambda g.\ g\ @'\ g \not\to_s$ (cf. Example 2.2).*

Proposition 2.5 is proved below as a corollary of Theorem 2.11. Those who are not interested in the proof can skip the rest of this subsection.

**Definition 2.7** *A term is called* reifiable *if it is statically-closed (i.e., every static variable is bound by a static $\lambda$-abstraction) and it contains no subterm (including itself) whose form is either $\downarrow t$, $\underline{\lambda x}.\ t$, $\underline{\mathbf{pair}}(t_1, t_2)$, $t_s\ \underline{@}'\ t$, $\underline{\mathbf{fst}}'(t_s)$ or $\underline{\mathbf{snd}}'(t_s)$ where $t_s$ is static.*

Intuitively, a reifiable term is a term that can be an argument of $\downarrow$. Note that all completely static, closed terms are reifiable.

**Lemma 2.8** *If $t \to_w t_0$, then $erase(t) \to_s^* erase(t_0)$.*

**Proof** By induction on the derivation of $t \to_w t_0$. Note that no reduction rule introduces $\downarrow$ inside $\downarrow$ or replaces a dynamic term with a static one. □

**Lemma 2.9** *If $t$ is reifiable and $t \to_w t_0$, then $t_0$ is also reifiable.*

**Proof** By induction on the derivation of $t \to_w t_0$. □

**Lemma 2.10** *If $t$ is a reifiable, type-safe term, then $\downarrow t \to_w t_0$ for some $t_0$.*

**Proof** We prove the contraposition by induction on the structure of $t$. If $t = t_1\ @'\ t_2$ where $t_1 \not\to_w$ and $t_2 \not\to_w$, then either (1) $t_1$ is neither a static $\lambda$-abstraction nor a dynamic value, or (2) $t_2$ is not a value. In the case (1), either $t_1$ is a static pair of values and $t$ is not type-safe, or $\downarrow t_1 \not\to_w$ and $t_1$ is not type-safe by the induction hypothesis. In the case (2), $\downarrow t_2 \not\to_w$ and $t_2$ is not type-safe by the induction hypothesis. Other cases are similar or trivial. □

**Theorem 2.11** *Let $t$ be a reifiable, type-safe term. If $\downarrow t \to_w^* t_n \not\to_w$, then $t_n$ is completely dynamic and $erase(t) \to_s^* erase(t_n) \not\to_s$.*

**Proof** By induction on the length of the reduction sequence from $\downarrow t$ to $t_n$.

*Base case:* If the length is 0, then $\downarrow t = t_n \not\to_w$. Therefore, $t$ is not type-safe by Lemma 2.10.

*Induction step:* If the length is greater than 0, then $\downarrow t \to_w t_0 \to_w^* t_n \not\to_w$ for some $t_0$. We perform case analysis on the last reduction rule in the derivation of $\downarrow t \to_w t_0$.

- If the rule is ①, there exists some $t_1$ such that $\downarrow t \to_w \downarrow t_1 = t_0$ where $t \to_w t_1$. By Lemma 2.8, $erase(t) \to_s^* erase(t_1)$. By Lemma 2.9, $t_1$ is reifiable. Since $t_1$ is type-safe, $t_n$ is completely dynamic and $erase(t_1) \to_s^* erase(t_n) \not\to_s$ by the induction hypothesis. Hence $erase(t) \to_s^* erase(t_n) \not\to_s$.

- If the rule is ②, the reduction sequence should have the following form, where $\underline{y}$ is fresh and $\downarrow[\underline{y}/x]t_1 \rightarrow_w^* t_2 \not\rightarrow_w$.

$$\begin{array}{rcl} \downarrow t & = & \downarrow(\lambda x.\ t_1) \\ & \rightarrow_w & \underline{\lambda y}.\ \downarrow((\lambda x.\ t_1)\ @'\ \underline{y}) \\ & \rightarrow_w & \underline{\lambda y}.\ \downarrow[\underline{y}/x]t_1 \\ & \rightarrow_w^* & \underline{\lambda y}.\ t_2 \\ & = & t_n \\ & \not\rightarrow_w & \end{array}$$

  Since $[\underline{y}/x]t_1$ is reifiable and type-safe, $t_2$ is completely dynamic and $\mathit{erase}([\underline{y}/x]t_1) \rightarrow_s^* \mathit{erase}(t_2) \not\rightarrow_s$ by the induction hypothesis. Therefore, $t_n$ is completely dynamic and $\mathit{erase}(t)$ strongly normalizes to $\mathit{erase}(t_n)$ as follows.

$$\begin{array}{rcl} \mathit{erase}(t) & = & \lambda x.\ \mathit{erase}(t_1) \\ & = & \lambda y.\ \mathit{erase}([\underline{y}/x]t_1) \\ & \rightarrow_s^* & \lambda y.\ \mathit{erase}(t_2) \\ & = & \mathit{erase}(t_n) \\ & \not\rightarrow_s & \end{array}$$

- If the rule is ③, the reduction sequence should have the following form, where $\downarrow v_1 \rightarrow_w^* t_1 \not\rightarrow_w$ and $\downarrow v_2 \rightarrow_w^* t_2 \not\rightarrow_w$.

$$\begin{array}{rcl} \downarrow t & = & \downarrow(v_1\ \underline{@}'\ v_2) \\ & \rightarrow_w & \downarrow v_1\ \underline{@}'\ \downarrow v_2 \\ & \rightarrow_w^* & t_1\ \underline{@}'\ t_2 \\ & = & t_n \\ & \not\rightarrow_w & \end{array}$$

  Since $v_1$ and $v_2$ are reifiable and type-safe, $t_1$ and $t_2$ are completely dynamic, $\mathit{erase}(v_1) \rightarrow_s^* \mathit{erase}(t_1) \not\rightarrow_s$, and $\mathit{erase}(v_2) \rightarrow_s^* \mathit{erase}(t_2) \not\rightarrow_s$ by the induction hypothesis. Therefore, $t_n$ is completely dynamic. Furthermore, since $t$ is reifiable, $v_1$ is a dynamic term other than a dynamic $\lambda$-abstraction, and so is $t_1$. Therefore, $\mathit{erase}(t_1)$ is neither a dynamic term nor a static $\lambda$-abstraction, so $\mathit{erase}(t)$ strongly normalizes to $\mathit{erase}(t_n)$ as follows.

$$\begin{array}{rcl} \mathit{erase}(t) & = & \mathit{erase}(v_1)\ @'\ \mathit{erase}(v_2) \\ & \rightarrow_s^* & \mathit{erase}(t_1)\ @'\ \mathit{erase}(t_2) \\ & = & \mathit{erase}(t_n) \\ & \not\rightarrow_s & \end{array}$$

Other cases are similar or trivial. □

# 3 Another View: Online Cogen-Based Approach

Although our partial evaluator doesn't perform reflection at all, it still performs reification in a type-directed manner. At the same time, however, it also seems similar to online *syntax-directed* partial evaluators, in that primitive operators reconstruct themselves when their operands are dynamic. In fact, it can be seen as a combination of (1) a library to interpret a program as a generating extension, derived from an online syntax-directed partial evaluator in higher-order abstract syntax, and (2) a converter from the higher-order abstract

syntax into the first-order abstract syntax. This approach is already well-known for *offline* syntax-directed partial evaluation as a kind of so-called cogen-based approach, and found to be simpler and more efficient than traditional self-application-based approach [16, 17]. We are going to show that it is also applicable to *online* syntax-directed partial evaluation, and that the result coincides with our online type-directed partial evaluator.

## 3.1 Background

In this subsection, we briefly explain basic ideas of the cogen-based approach to online syntax-directed partial evaluation.

### 3.1.1 Higher-order Abstract Syntax

Higher-order abstract syntax [12] is a meta-programming technique that represents binding at the object level by binding at the meta level to make environment manipulation simple and efficient. For example, let us represent untyped $\lambda$-terms in ML. With first-order (i.e., not higher-order) abstract syntax, the data type can be defined as follows.

```
datatype lam = Var of string
             | Abs of string * lam
             | App of lam * lam
```

For example, $\lambda x.\ (\lambda y.\ y)\ @\ x$ can be represented as `Abs("x", App(Abs("y", Var "y"), Var "x"))`. With higher-order abstract syntax, the data type can be defined as follows.

```
datatype hlam = HAbs of hlam -> hlam
              | HApp of hlam * hlam
              | HEmbed of string
```

Then, the term above can be represented as `HAbs(fn x => HApp(HAbs(fn y => y), x))`. A converter from the higher-order abstract syntax into the first-order abstract syntax can be written as follows.

```
val seq = ref 0
fun gensym () = (seq := !seq + 1;
                 "x" ^ Int.toString(!seq))
fun htof(HAbs f) = let val s = gensym ()
                   in Abs(s, (htof o f o HEmbed) s)
                   end
  | htof(HApp(t1, t2)) = App(htof t1, htof t2)
  | htof(HEmbed s) = Var s
```

### 3.1.2 Partial Evaluation in Higher-Order Abstract Syntax

In general, it is surprisingly easy to write an evaluator for terms in higher-order abstract syntax. For example, an online syntax-directed partial evaluator for the untyped $\lambda$-terms can be written as follows.

```
fun normalize(HAbs f) = HAbs(fn x => normalize (f x))
  | normalize(HApp(t1, t2)) = let val t1' = normalize t1
                                  val t2' = normalize t2
                              in (case t1'
                                  of HAbs f => f t2'
                                  | _ => HApp(t1', t2'))
    end
  | normalize(HEmbed v) = HEmbed v
```

However, since the codomain of this partial evaluator is the higher-order abstract syntax, the result has to be converted into the first-order abstract syntax to be displayed. In fact, because of ML's weak evaluation strategy, no evaluation occurs until the conversion.

```
- (htof o normalize) (HAbs(fn x => HApp(HAbs(fn y => y), x)));
val it = Abs ("x1",Var "x1") : lam
```

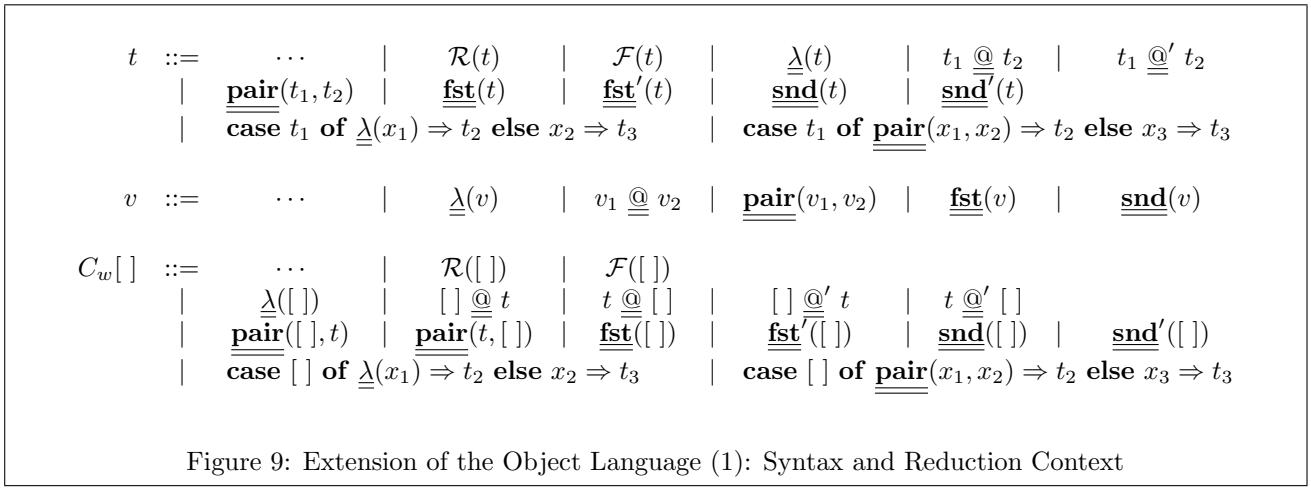


Figure 9: Extension of the Object Language (1): Syntax and Reduction Context

### 3.1.3 Composition of the Partial Evaluator and the Syntax Constructors

In the above example, we first constructed a $\lambda$-term by the syntax constructors and then destructed it by the partial evaluator. This is a waste of the memory (for storing the intermediate data structure) and the time (for traversing it by pattern matching). We can remove the overhead by composing the constructors with the destructors [13, 15].

```
fun rHAbs f = HAbs(fn x => f x)
fun rHApp(t1, t2) = let val t1' = t1
                        val t2' = t2
                    in (case t1'
                        of HAbs f => f t2'
                        | _ => HApp(t1', t2')) end
```

We can simplify these composed constructors by $\eta$-reduction and inlining.

```
val rHAbs = HAbs
fun rHApp(HAbs f, t2) = f t2
  | rHApp(t1, t2) = HApp(t1, t2)
```

This extremely simple partial evaluator works as follows.

```
- htof (rHAbs(fn x => rHApp(rHAbs(fn y => y), x)));
val it = Abs ("x1",Var "x1") : lam
```

## 3.2 Formalization

The above partial evaluator for untyped $\lambda$-terms in the higher-order abstract syntax seems very similar to our online type-directed partial evaluator for dynamically-typed languages. More specifically, the partial evaluator composed with the syntax constructors seems similar to the extended value destructors in our object language, and the converter from the higher-order abstract syntax into the first-order abstract syntax seems similar to the reification operator in our partial evaluator.

We prove this intuition by formalizing the partial evaluator, the converter, and the syntax constructors composed with the partial evaluator. For that purpose, we extend the syntax and the semantics of the object language in Subsection 2.1 as shown in Figure 9, 10, 11, and 12, where symbols with double underlines denote the syntax constructors, $\mathcal{R}$ denotes the partial evaluator, $\mathcal{F}$ denotes the converter, and $'$ denotes the composition with the partial evaluator. $C_s$ (resp. $\rightarrow_s$) is also extended accordingly to $C_w$ (resp. $\rightarrow_w$). $\underline{\underline{\lambda}}'$ and $\underline{\underline{\mathbf{pair}}}'$ are unnecessary because they are equivalent to $\underline{\underline{\lambda}}$ and $\underline{\underline{\mathbf{pair}}}$.

$$
\begin{array}{lll}
\mathcal{R}(\underline{\underline{\lambda}}(\lambda x.\ t)) & \to_w & \underline{\underline{\lambda}}(\lambda y.\ \mathcal{R}((\lambda x.\ t)\ @'\ y)) \quad \text{(where } y \text{ is fresh)} \\
\mathcal{R}(v_1\ \underline{\underline{@}}\ v_2) & \to_w & \textbf{case}\ \mathcal{R}(v_1)\ \textbf{of}\ \underline{\underline{\lambda}}(x) \Rightarrow x\ @'\ \mathcal{R}(v_2)\ \textbf{else}\ y \Rightarrow y\ \underline{\underline{@}}\ \mathcal{R}(v_2) \\
 & & \quad \text{(where } x \text{ and } y \text{ are fresh)} \\
\mathcal{R}(\underline{\underline{\textbf{pair}}}(v_1, v_2)) & \to_w & \underline{\underline{\textbf{pair}}}(\mathcal{R}(v_1), \mathcal{R}(v_2)) \\
\mathcal{R}(\underline{\underline{\textbf{fst}}}(v)) & \to_w & \textbf{case}\ \mathcal{R}(v)\ \textbf{of}\ \underline{\underline{\textbf{pair}}}(x_1, x_2) \Rightarrow x_1\ \textbf{else}\ y \Rightarrow \underline{\underline{\textbf{fst}}}(y) \quad \text{(where } x_1, x_2, \text{ and } y \text{ are fresh)} \\
\mathcal{R}(\underline{\underline{\textbf{snd}}}(v)) & \to_w & \textbf{case}\ \mathcal{R}(v)\ \textbf{of}\ \underline{\underline{\textbf{pair}}}(x_1, x_2) \Rightarrow x_2\ \textbf{else}\ y \Rightarrow \underline{\underline{\textbf{snd}}}(y) \quad \text{(where } x_1, x_2, \text{ and } y \text{ are fresh)} \\
\mathcal{R}(\underline{x}) & \to_w & \underline{x}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{case}\ \underline{\underline{\lambda}}(v)\ \textbf{of}\ \underline{\underline{\lambda}}(x) \Rightarrow t_1\ \textbf{else}\ y \Rightarrow t_2 & \to_w & [v/x]t_1 \\
\textbf{case}\ v\ \textbf{of}\ \underline{\underline{\lambda}}(x) \Rightarrow t_1\ \textbf{else}\ y \Rightarrow t_2 & \to_w & [v/y]t_2 \quad (\text{if } v \neq \underline{\underline{\lambda}}(v)) \\
\textbf{case}\ \underline{\underline{\textbf{pair}}}(v_1, v_2)\ \textbf{of}\ \underline{\underline{\textbf{pair}}}(x_1, x_2) \Rightarrow t_1\ \textbf{else}\ y \Rightarrow t_2 & \to_w & [v_1/x_1, v_2/x_2]t_1 \\
\textbf{case}\ v\ \textbf{of}\ \underline{\underline{\textbf{pair}}}(x_1, x_2) \Rightarrow t_1\ \textbf{else}\ y \Rightarrow t_2 & \to_w & [v/y]t_2 \quad (\text{if } v \neq \underline{\underline{\textbf{pair}}}(v_1, v_2))
\end{array}
$$

Figure 10: Extension of the Object Language (2): Reduction Rules for the Partial Evaluator for Terms in the Higher-Order Abstract Syntax

$$
\begin{array}{lll}
\mathcal{F}(\underline{\underline{\lambda}}(\lambda x.\ t)) & \to_w & \underline{\lambda y}.\ \mathcal{F}((\lambda x.\ t)\ @'\ \underline{y}) \quad \text{(where } \underline{y} \text{ is fresh)} \\
\mathcal{F}(v_1\ \underline{\underline{@}}\ v_2) & \to_w & \mathcal{F}(v_1)\ \underline{\underline{@}}'\ \mathcal{F}(v_2) \\
\mathcal{F}(\underline{\underline{\textbf{pair}}}(v_1, v_2)) & \to_w & \underline{\underline{\textbf{pair}}}(\mathcal{F}(v_1), \mathcal{F}(v_2)) \\
\mathcal{F}(\underline{\underline{\textbf{fst}}}(v)) & \to_w & \underline{\underline{\textbf{fst}}}'(\mathcal{F}(v)) \\
\mathcal{F}(\underline{\underline{\textbf{snd}}}(v)) & \to_w & \underline{\underline{\textbf{snd}}}'(\mathcal{F}(v)) \\
\mathcal{F}(\underline{x}) & \to_w & \underline{x}
\end{array}
$$

Figure 11: Extension of the Object Language (3): Reduction Rules for the Converter from the Higher-Order Abstract Syntax into the First-Order Abstract Syntax

$$
\begin{array}{rcll}
\underline{\underline{\lambda}}(\lambda x.\ t)\ \underline{\underline{@}}'\ v & \to_w & (\lambda x.\ t)\ @'\ v & \\
v_1\ \underline{\underline{@}}'\ v_2 & \to_w & v_1\ \underline{\underline{@}}\ v_2 & (\text{if } v_1 \neq \underline{\underline{\lambda}}(v)) \\
\underline{\underline{\textbf{fst}}}'(\underline{\underline{\textbf{pair}}}(v_1, v_2)) & \to_w & v_1 & \\
\underline{\underline{\textbf{fst}}}'(v) & \to_w & \underline{\underline{\textbf{fst}}}(v) & (\text{if } v \neq \underline{\underline{\textbf{pair}}}(v_1, v_2)) \\
\underline{\underline{\textbf{snd}}}'(\underline{\underline{\textbf{pair}}}(v_1, v_2)) & \to_w & v_2 & \\
\underline{\underline{\textbf{snd}}}'(v) & \to_w & \underline{\underline{\textbf{snd}}}(v) & (\text{if } v \neq \underline{\underline{\textbf{pair}}}(v_1, v_2))
\end{array}
$$

Figure 12: Extension of the Object Language (4): Reduction Rules for the Syntax Constructors Composed with the Partial Evaluator

$$
\begin{array}{rcll}
h^I & ::= & \underline{x} & \\
& | & x & \\
& | & \underline{\underline{\lambda}}(\lambda x.\ h^I) & \\
& | & h_1^I \ \underline{\underline{@}}\ h_2^I & \\
& | & \underline{\underline{\mathbf{pair}}}(h_1^I, h_2^I) & \\
& | & \underline{\underline{\mathbf{fst}}}(h^I) & \\
& | & \underline{\underline{\mathbf{snd}}}(h^I) & \\
& | & \underline{\underline{\lambda}}(\lambda y.\ \mathcal{R}((\lambda x.\ h^I)\ @'\ y)) & \text{(where } y \text{ is not free in } h^I\text{)} \\
h^O & ::= & \underline{x} & \\
& | & h_1^O \ \underline{\underline{@}}\ h_2^O & \text{(where } h_1^O \neq \underline{\underline{\lambda}} t\text{)} \\
& | & \underline{\underline{\mathbf{pair}}}(h_1^O, h_2^O) & \\
& | & \underline{\underline{\mathbf{fst}}}(h^O) & \text{(where } h^O \neq \underline{\underline{\mathbf{pair}}}(t_1, t_2)\text{)} \\
& | & \underline{\underline{\mathbf{snd}}}(h^O) & \text{(where } h^O \neq \underline{\underline{\mathbf{pair}}}(t_1, t_2)\text{)} \\
& | & \underline{\underline{\lambda}}(\lambda y.\ \mathcal{R}((\lambda x.\ h^I)\ @'\ y)) & \text{(where } y \text{ is not free in } h^I\text{)}
\end{array}
$$

Figure 13: Syntax of HOAS-Terms

$$
\begin{array}{lcl}
x'^{\#} & = & x \\
\underline{x}'^{\#} & = & \underline{x} \\
\underline{\underline{\lambda}}(\lambda x.\ h^I)'^{\#} & = & \underline{\underline{\lambda}}(\lambda x.\ h^{I'^{\#}}) \\
(h_1^I \ \underline{\underline{@}}\ h_2^I)'^{\#} & = & h_1^{I'^{\#}} \ \underline{\underline{@}}'\ h_2^{I'^{\#}} \\
\underline{\underline{\mathbf{pair}}}(h_1^I, h_2^I)'^{\#} & = & \underline{\underline{\mathbf{pair}}}(h_1^{I'^{\#}}, h_2^{I'^{\#}}) \\
\underline{\underline{\mathbf{fst}}}(h^I)'^{\#} & = & \underline{\underline{\mathbf{fst}}}'(h^{I'^{\#}}) \\
\underline{\underline{\mathbf{snd}}}(h^I)'^{\#} & = & \underline{\underline{\mathbf{snd}}}'(h^{I'^{\#}}) \\
\underline{\underline{\lambda}}(\lambda y.\ \mathcal{R}((\lambda x.\ h^I)\ @'\ y))'^{\#} & = & \underline{\underline{\lambda}}(\lambda x.\ h^{I'^{\#}})
\end{array}
$$

Figure 14: Definition of $h^{I'^{\#}}$

**Definition 3.1** *An* input HOAS-term *(denoted by $h^I$) and an* output HOAS-term *(denoted by $h^O$) are terms whose syntax is restricted as Figure 13.*

Intuitively, an input (resp. output) HOAS-term is a term that can be an argument (resp. a result) of $\mathcal{R}$. It is easy to see that every output HOAS-term and every statically-closed input HOAS-term are values, and every output HOAS-terms is an input HOAS-term.

**Definition 3.2** *Let $h^I$ be any input HOAS-term. $h^{I'^{\#}}$ denote a term defined in Figure 14, and Erase$^{\#}$ denote a term defined in Figure 15.*

We use this definition to show the correspondence of the reduction rules in Figure 10 to those in Figure 12 and those in Figure 3.

**Proposition 3.3** *Let $h^I$ be any statically-closed input HOAS-term where Erase$^{\#}(h^I)$ is type-safe. If $\mathcal{F}(\mathcal{R}(h^I)) \rightarrow_w^* t \not\rightarrow_w$, then $\mathcal{F}(h^{I'^{\#}}) \rightarrow_w^* t$ and $\downarrow(\mathit{Erase}^{\#}(h^I)) \rightarrow_w^* t$.*

This proposition means that the following three computations give the same result for any closed $\lambda$-expression in the higher-order abstract syntax.

$$
\begin{array}{lcl}
\mathit{Erase}^{\#}(x) & = & x \\
\mathit{Erase}^{\#}(\underline{x}) & = & \underline{x} \\
\mathit{Erase}^{\#}(\underline{\underline{\lambda}}(\lambda x.\ h^I)) & = & \lambda x.\ \mathit{Erase}^{\#}(h^I) \\
\mathit{Erase}^{\#}(h_1^I\ \underline{\underline{@}}\ h_2^I) & = & \mathit{Erase}^{\#}(h_1^I)\ @'\ \mathit{Erase}^{\#}(h_2^I) \\
\mathit{Erase}^{\#}(\underline{\underline{\mathbf{pair}}}(h_1^I, h_2^I)) & = & \mathbf{pair}(\mathit{Erase}^{\#}(h_1^I), \mathit{Erase}^{\#}(h_2^I)) \\
\mathit{Erase}^{\#}(\underline{\underline{\mathbf{fst}}}(h^I)) & = & \mathbf{fst}'(\mathit{Erase}^{\#}(h^I)) \\
\mathit{Erase}^{\#}(\underline{\underline{\mathbf{snd}}}(h^I)) & = & \mathbf{snd}'(\mathit{Erase}^{\#}(h^I)) \\
\mathit{Erase}^{\#}(\underline{\underline{\lambda}}(\lambda y.\ \mathcal{R}((\lambda x.\ h^I)\ @'\ y))) & = & \lambda x.\ \mathit{Erase}^{\#}(h^I)
\end{array}
$$

Figure 15: Definition of $\mathit{Erase}^{\#}(h^I)$

- Normalize the expression with an interpreter (e.g., `normalize`), and then convert it into the first-order abstract syntax.
- Replace the syntax constructors (e.g., `HAbs` and `HApp`) in the expression with the syntax constructors composed with the interpreter (e.g., `rHAbs` and `rHApp`), evaluate it, and then convert it into the first-order abstract syntax.
- Evaluate the expression and then reify it.

Moreover, we can confirm that the second computation is more efficient than the first one and as efficient as the third one, by closely looking at the reduction rules as follows.

- Reduction of $\mathcal{R}(h^I)$ requires case analysis on the syntax of $h^I$, while that of $h^{I'\#}$ does not. Compare Figure 10 and Figure 12.
- The reduction steps in Figure 12 (resp. Figure 11) correspond one-to-one to those in Figure 3 (resp. Figure 4).

See the example and the proof below for more details on their relationship.

Type-directed partial evaluation has been believed to be faster than syntax-directed one, because it directly executes a source program instead of symbolically manipulating the abstract syntax. On the other hand, the cogen-based approach to syntax-directed partial evaluation with higher-order abstract syntax has also been known as another way to achieve a similar result. The above observation implies that the two techniques are equivalent in online partial evaluation, which suggests that we can obtain more powerful partial evaluators by replacing the underlying syntax-directed partial evaluator with more sophisticated (e.g. let-inserting) ones.

**Example 3.4** *Let* $h^I = \underline{\underline{\lambda}}(\lambda f.\ f\ \underline{\underline{@}}\ \underline{\underline{\mathbf{fst}}}(\underline{\underline{\mathbf{pair}}}(f, f)))$. *Then,* $\mathcal{F}(\mathcal{R}(h^I)) \to_w^* \underline{\lambda} g.\ g\ \underline{@}'\ g \not\to_w$, $\mathcal{F}(h^{I'\#}) \to_w^* \underline{\lambda} g.\ g\ \underline{@}'\ g$, *and* $\downarrow(\mathit{Erase}^{\#}(h^I)) \to_w^* \underline{\lambda} g.\ g\ \underline{@}'\ g$ *as follows (cf. Example 2.2 and Example 2.6).*

$$
\begin{array}{rcl}
\mathcal{F}(\mathcal{R}(h^I)) & = & \mathcal{F}(\mathcal{R}(\underline{\underline{\lambda}}(\lambda f.\ f\ \underline{\underline{@}}\ \underline{\underline{\mathbf{fst}}}(\underline{\underline{\mathbf{pair}}}(f, f))))) \\
& \to_w^* & \underline{\lambda} g.\ \mathcal{F}(\mathcal{R}((\lambda f.\ f\ \underline{\underline{@}}\ \underline{\underline{\mathbf{fst}}}(\underline{\underline{\mathbf{pair}}}(f, f)))@'\underline{g})) \\
& \to_w & \underline{\lambda} g.\ \mathcal{F}(\mathcal{R}(g\ \underline{\underline{@}}\ \underline{\underline{\mathbf{fst}}}(\underline{\underline{\mathbf{pair}}}(\underline{g}, \underline{g})))) \\
& \to_w^* & \underline{\lambda} g.\ \mathcal{F}(g\ \underline{\underline{@}}\ \mathcal{R}(\underline{\underline{\mathbf{fst}}}(\underline{\underline{\mathbf{pair}}}(\underline{g}, \underline{g})))) \\
& \to_w^* & \underline{\lambda} g.\ \mathcal{F}(g\ \underline{\underline{@}}\ \underline{g}) \\
& \to_w^* & \underline{\lambda} g.\ g\ \underline{@}'\ g
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F}(h^{I'\#}) & = & \mathcal{F}(\underline{\underline{\lambda}}(\lambda f.\ f\ \underline{\underline{@}}'\ \underline{\underline{\mathbf{fst}}}'(\underline{\underline{\mathbf{pair}}}(f, f)))) \\
& \to_w & \underline{\lambda} g.\ \mathcal{F}((\lambda f.\ f\ \underline{\underline{@}}'\ \underline{\underline{\mathbf{fst}}}'(\underline{\underline{\mathbf{pair}}}(f, f)))\ @'\ \underline{g})
\end{array}
$$

$$
\begin{array}{lcll}
x' & = & x & \\
\underline{x}' & = & \underline{x} & \\
\underline{\underline{\lambda}}(\lambda x.\ h^I)' & = & \underline{\underline{\lambda}}(\lambda x.\ h^{I'\#}) & \\
(h_1^I\ \underline{\underline{@}}\ h_2^I)' & = & h_1^{I'}\ \underline{\underline{@}}\ h_2^{I'} & (\text{if } h_1^{I'} = v \neq \underline{\underline{\lambda}} v_0) \\
(h_1^I\ \underline{\underline{@}}\ h_2^I)' & = & h_1^{I'}\ \underline{\underline{@}}'\ h_2^{I'} & (\text{otherwise}) \\
\underline{\underline{\mathbf{pair}}}(h_1^I, h_2^I)' & = & \underline{\underline{\mathbf{pair}}}(h_1^{I'}, h_2^{I'}) & \\
\underline{\underline{\mathbf{fst}}}(h^I)' & = & \underline{\underline{\mathbf{fst}}}(h^{I'}) & (\text{if } h^{I'} = v \neq \underline{\underline{\mathbf{pair}}}(v_1, v_2)) \\
\underline{\underline{\mathbf{fst}}}(h^I)' & = & \underline{\underline{\mathbf{fst}}}'(h^{I'}) & (\text{otherwise}) \\
\underline{\underline{\mathbf{snd}}}(h^I)' & = & \underline{\underline{\mathbf{snd}}}(h^{I'}) & (\text{if } h^{I'} = v \neq \underline{\underline{\mathbf{pair}}}(v_1, v_2)) \\
\underline{\underline{\mathbf{snd}}}(h^I)' & = & \underline{\underline{\mathbf{snd}}}'(h^{I'}) & (\text{otherwise}) \\
\underline{\underline{\lambda}}(\lambda y.\ \mathcal{R}((\lambda x.\ h^I)\ @'\ y))' & = & \underline{\underline{\lambda}}(\lambda x.\ h^{I'\#}) &
\end{array}
$$

Figure 16: Definition of $h^{I'}$

$$
\begin{array}{lcll}
\mathit{Erase}(x) & = & x & \\
\mathit{Erase}(\underline{x}) & = & \underline{x} & \\
\mathit{Erase}(\underline{\underline{\lambda}}(\lambda x.\ h^I)) & = & \lambda x.\ \mathit{Erase}^{\#}(h^I) & \\
\mathit{Erase}(h_1^I\ \underline{\underline{@}}\ h_2^I) & = & \mathit{Erase}(h_1^I)\ \underline{@'}\ \mathit{Erase}(h_2^I) & (\text{if } \mathit{Erase}(h_1^I) = v \neq \lambda x.\ t) \\
\mathit{Erase}(h_1^I\ \underline{\underline{@}}\ h_2^I) & = & \mathit{Erase}(h_1^I)\ @'\ \mathit{Erase}(h_2^I) & (\text{otherwise}) \\
\mathit{Erase}(\underline{\underline{\mathbf{pair}}}(h_1^I, h_2^I)) & = & \mathbf{pair}(\mathit{Erase}(h_1^I), \mathit{Erase}(h_2^I)) & \\
\mathit{Erase}(\underline{\underline{\mathbf{fst}}}(h^I)) & = & \underline{\mathbf{fst}}'(\mathit{Erase}(h^I)) & (\text{if } \mathit{Erase}(h^I) = v \neq \mathbf{pair}(v_1, v_2)) \\
\mathit{Erase}(\underline{\underline{\mathbf{fst}}}(h^I)) & = & \mathbf{fst}'(\mathit{Erase}(h^I)) & (\text{otherwise}) \\
\mathit{Erase}(\underline{\underline{\mathbf{snd}}}(h^I)) & = & \underline{\mathbf{snd}}'(\mathit{Erase}(h^I)) & (\text{if } \mathit{Erase}(h^I) = v \neq \mathbf{pair}(v_1, v_2)) \\
\mathit{Erase}(\underline{\underline{\mathbf{snd}}}(h^I)) & = & \mathbf{snd}'(\mathit{Erase}(h^I)) & (\text{otherwise}) \\
\mathit{Erase}(\underline{\underline{\lambda}}(\lambda y.\ \mathcal{R}((\lambda x.\ h^I)\ @'\ y))) & = & \lambda x.\ \mathit{Erase}^{\#}(h^I) &
\end{array}
$$

Figure 17: Definition of $\mathit{Erase}(h^I)$

$$
\begin{array}{rcl}
 & \rightarrow_w & \underline{\lambda} g.\ \mathcal{F}(g\ \underline{\underline{@}}'\ \underline{\underline{\mathbf{fst}}}'(\underline{\underline{\mathbf{pair}}}(g, g))) \\
 & \rightarrow_w & \underline{\lambda} g.\ \mathcal{F}(g\ \underline{\underline{@}}'\ g) \\
 & \rightarrow_w^* & \underline{\lambda} g.\ g\ \underline{\underline{@}}'\ g \\
 & & \\
\downarrow(\mathit{Erase}^{\#}(h^I)) & \rightarrow_w & \downarrow(\lambda f.\ f\ @'\ \mathbf{fst}'(\mathbf{pair}(f, f))) \\
 & \rightarrow_w^* & \underline{\lambda} g.\ g\ \underline{\underline{@}}'\ g
\end{array}
$$

Proposition 3.3 is proved below as a corollary of Theorem 3.7. Those who are not interested in the proof can skip the rest of this subsection.

**Definition 3.5** *Let $h^I$ be any input HOAS-term. $h^{I'}$ denotes a term defined in Figure 16, and $\mathit{Erase}(h^I)$ denotes a term defined in Figure 17.*

We are going to use this definition to show the correspondence among the reductions of $\mathcal{F}(\mathcal{R}(h^I))$, $\mathcal{F}(h^{I'})$, and $\downarrow(\mathit{Erase}(h^I))$. It is easy to see that $h^{I'\#} \rightarrow_w^* h^{I'}$ and $\mathit{Erase}^{\#}(h^I) \rightarrow_w^* \mathit{Erase}(h^I)$ for any $h^I$, provided that $\mathit{Erase}^{\#}(h^I)$ is type-safe. In addition, $h^{O'}$ and $\mathit{Erase}(h^O)$ are values for any $h^O$. (Remember that every output

HOAS-term is an input HOAS-term.) These properties are necessary in the following proofs, which is the reason why we define $h^{I'}$ and $\mathit{Erase}(h^I)$ besides $h^{I'\#}$ and $\mathit{Erase}^{\#}(h^I)$.

**Lemma 3.6** *Let $h^I$ be any statically-closed input HOAS-term. If $\mathcal{R}(h^I) \rightarrow_w^* t \not\rightarrow_w$, then $t$ is a statically-closed output HOAS-term. Furthermore, $h^{I'} \rightarrow_w^* t'$ and $\mathit{Erase}(h^I) \rightarrow_w^* \mathit{Erase}(t)$.*

**Proof** By induction on the length of the reduction sequence from $\mathcal{R}(h^I)$ to $t$. If the length is greater than 0, we perform case analysis on the last rule in the derivation of the first reduction.

- If the rule is the second one in Figure 10, the reduction sequence should begin as follows, where $x$ and $x_0$ are fresh.

$$\begin{array}{rcl}
\mathcal{R}(h^I) & = & \mathcal{R}(h_1^I \mathbin{\underline{@}} h_2^I) \\
& \rightarrow_w & \mathbf{case}\ \mathcal{R}(h_1^I) \\
& & \mathbf{of}\ \underline{\lambda}(x) \Rightarrow x \mathbin{@'} \mathcal{R}(h_2^I) \\
& & \mathbf{else}\ x_0 \Rightarrow x_0 \mathbin{\underline{@}} \mathcal{R}(h_2^I)
\end{array}$$

  Since $h_1^I$ is a statically-closed input HOAS-term, $\mathcal{R}(h_1^I)$ weakly normalizes to a statically-closed output HOAS-term by the induction hypothesis. We again perform case analysis on the form of the output HOAS-term.

  - If $\mathcal{R}(h_1^I) \rightarrow_w^* \underline{\lambda}(\lambda y.\ \mathcal{R}((\lambda z.\ h_3^I) \mathbin{@'} y)) \not\rightarrow_w$, then the reduction sequence should continue as follows, where $y$ is fresh.

$$\begin{array}{rl}
\rightarrow_w^* & \mathbf{case}\ \underline{\lambda}(\lambda y.\ \mathcal{R}((\lambda z.\ h_3^I) \mathbin{@'} y)) \\
& \mathbf{of}\ \underline{\lambda}(x) \Rightarrow x \mathbin{@'} \mathcal{R}(h_2^I) \\
& \mathbf{else}\ x_0 \Rightarrow x_0 \mathbin{\underline{@}} \mathcal{R}(h_2^I) \\
\rightarrow_w & (\lambda y.\ \mathcal{R}((\lambda z.\ h_3^I) \mathbin{@'} y)) \mathbin{@'} \mathcal{R}(h_2^I) \\
\rightarrow_w^* & (\lambda y.\ \mathcal{R}((\lambda z.\ h_3^I) \mathbin{@'} y)) \mathbin{@'} h^O \\
\rightarrow_w & \mathcal{R}((\lambda z.\ h_3^I) \mathbin{@'} h^O) \\
\rightarrow_w & \mathcal{R}([h^O/z]h_3^I) \\
\rightarrow_w^* & t \\
\not\rightarrow_w &
\end{array}$$

    Since $h_1^I$ and $h_2^I$ are statically-closed input HOAS-terms, $\underline{\lambda}(\lambda y.\ \mathcal{R}((\lambda z.\ h_3^I) \mathbin{@'} y))$ and $h^O$ are statically-closed output HOAS-terms, $h_1^{I'} \rightarrow_w^* \underline{\lambda}(\lambda z.\ h_3^{I'\#})$, $h_2^{I'} \rightarrow_w^* h^{O'}$, $\mathit{Erase}(h_1^I) \rightarrow_w^* \lambda z.\ \mathit{Erase}^{\#}(h_3^I)$, and $\mathit{Erase}(h_2^I) \rightarrow_w^* \mathit{Erase}(h^O)$ by the induction hypothesis. Furthermore, since $[h^O/z]h_3^I$ is a statically-closed input HOAS-term, $t$ is a statically-closed output HOAS-term, $[h^{O'}/z]h_3^{I'} \rightarrow_w^* t'$, and $[\mathit{Erase}(h^O)/z]\mathit{Erase}(h_3^I) \rightarrow_w^* \mathit{Erase}(t)$ by the induction hypothesis. Therefore, $h^{I'}$ and $\mathit{Erase}(h^I)$ are weakly reducible to $t'$ and $\mathit{Erase}(t)$ as follows.

$$\begin{array}{rcl}
h^{I'} & = & (h_1^I \mathbin{\underline{@}} h_2^I)' \\
& = & h_1^{I'} \mathbin{\underline{@}'} h_2^{I'} \\
& \rightarrow_w^* & \underline{\lambda}(\lambda z.\ h_3^{I'\#}) \mathbin{\underline{@}'} h^{O'} \\
& \rightarrow_w & (\lambda z.\ h_3^{I'\#}) \mathbin{@'} h^{O'} \\
& \rightarrow_w & [h^{O'}/z]h_3^{I'\#} \\
& \rightarrow_w^* & [h^{O'}/z]h_3^{I'} \\
& \rightarrow_w^* & t'
\end{array}$$

$$
\begin{array}{rcl}
\mathit{Erase}(h^I) & = & \mathit{Erase}(h_1^I \mathbin{\underline{@}} h_2^I) \\
& = & \mathit{Erase}(h_1^I) \mathbin{@'} \mathit{Erase}(h_2^I) \\
& \to_w^* & (\lambda z.\ \mathit{Erase}^{\#}(h_3^I)) \mathbin{@'} \mathit{Erase}(h^O) \\
& \to_w & [\mathit{Erase}(h^O)/z]\mathit{Erase}^{\#}(h_3^I) \\
& \to_w^* & [\mathit{Erase}(h^O)/z]\mathit{Erase}(h_3^I) \\
& \to_w^* & \mathit{Erase}(t)
\end{array}
$$

Other cases are similar or trivial. □

**Theorem 3.7** *Let $h^I$ be any statically-closed input HOAS-term where $\mathit{Erase}(h^I)$ is type-safe. If $\mathcal{F}(\mathcal{R}(h^I)) \to_w^* t \not\to_w$, then $\mathcal{F}(h^{I'}) \to_w^* t$ and $\downarrow(\mathit{Erase}(h^I)) \to_w^* t$.*

**Proof** By induction on the length of the reduction sequence from $\mathcal{F}(\mathcal{R}(h^I))$ to $t$.

*Base Case:* If the length is 0, then $\mathcal{F}(\mathcal{R}(h^I)) = t \not\to_w$. Since $\mathcal{R}(h^I) \not\to_w$, $h^I$ should be a static variable, which contradicts with the assumption that $h^I$ is statically-closed.

*Induction Step:* If the length is greater than 0, then $\mathcal{F}(\mathcal{R}(h^I)) \to_w \mathcal{F}(t_0) \to_w^* t \not\to_w$ for some $t_0$ where $\mathcal{R}(h^I) \to_w t_0$. We perform case analysis on the last reduction rule in the derivation of $\mathcal{R}(h^I) \to_w t_0$.

- If the rule is the first one in Figure 10, then the reduction sequence of $\mathcal{F}(\mathcal{R}(h^I))$ should have the following form, where $y$ and $\underline{z}$ are fresh and $\mathcal{F}(\mathcal{R}([\underline{z}/x]h_0^I)) \to_w^* t_1 \not\to_w$.

$$
\begin{array}{rcl}
\mathcal{F}(\mathcal{R}(h^I)) & = & \mathcal{F}(\mathcal{R}(\underline{\lambda}(\lambda x.\ h_0^I))) \\
& \to_w & \mathcal{F}(\underline{\lambda}(\lambda y.\ \mathcal{R}((\lambda x.\ h_0^I) \mathbin{@'} y))) \\
& \to_w & \underline{\lambda z}.\ \mathcal{F}((\lambda y.\ \mathcal{R}((\lambda x.\ h_0^I) \mathbin{@'} y)) \mathbin{@'} \underline{z}) \\
& \to_w & \underline{\lambda z}.\ \mathcal{F}(\mathcal{R}((\lambda x.\ h_0^I) \mathbin{@'} \underline{z})) \\
& \to_w & \underline{\lambda z}.\ \mathcal{F}(\mathcal{R}([\underline{z}/x]h_0^I)) \\
& \to_w^* & \underline{\lambda z}.\ t_1 \\
& = & t
\end{array}
$$

  Since $[\underline{z}/x]h_0^I$ is a statically-closed input HOAS-term, $\mathcal{F}([\underline{z}/x]h_0^{I'}) \to_w^* t_1 \not\to_w$ and $\downarrow(\mathit{Erase}([\underline{z}/x]h_0^I)) \to_w^* t_1 \not\to_w$ by the induction hypothesis. Therefore, $\mathcal{F}(h^{I'})$ and $\downarrow(\mathit{Erase}(h^I))$ weakly reduce to $t$ as follows.

$$
\begin{array}{rcl}
\mathcal{F}(h^{I'}) & = & \mathcal{F}(\underline{\lambda}(\lambda x.\ h_0^{I'\#})) \\
& \to_w & \underline{\lambda y}.\ \mathcal{F}((\lambda x.\ h_0^{I'\#}) \mathbin{@'} \underline{y}) \\
& \to_w & \underline{\lambda y}.\ \mathcal{F}([\underline{y}/x]h_0^{I'\#}) \\
& \to_w^* & \underline{\lambda y}.\ \mathcal{F}([\underline{y}/x]h_0^{I'}) \\
& = & \underline{\lambda z}.\ \mathcal{F}([\underline{z}/x]h_0^{I'}) \\
& \to_w^* & \underline{\lambda z}.\ t_1 \\
& = & t
\end{array}
$$

$$
\begin{array}{rcl}
\downarrow(\mathit{Erase}(h^I)) & = & \downarrow(\lambda x.\ \mathit{Erase}^{\#}(h_0^I)) \\
& \to_w & \underline{\lambda y}.\ \downarrow((\lambda x.\ \mathit{Erase}^{\#}(h_0^I)) \mathbin{@'} \underline{y}) \\
& \to_w & \underline{\lambda y}.\ \downarrow(\mathit{Erase}^{\#}([\underline{y}/x]h_0^I)) \\
& \to_w^* & \underline{\lambda y}.\ \downarrow(\mathit{Erase}([\underline{y}/x]h_0^I))
\end{array}
$$

$$
\begin{array}{lll}
& \to_w & \underline{\lambda z}.\ \downarrow(\mathit{Erase}([\underline{z}/x]h_0^I)) \\
& \to_w^* & \underline{\lambda z}.\ t_1 \\
& = & t
\end{array}
$$

- If the rule is the second one in Figure 10, then the reduction sequence of $\mathcal{F}(\mathcal{R}(h^I))$ should begin as follows, where $x$ and $x_0$ are fresh.

$$
\begin{array}{lll}
\mathcal{F}(\mathcal{R}(h^I)) & = & \mathcal{F}(\mathcal{R}(h_1^I \mathbin{\underline{@}} h_2^I)) \\
& \to_w & \mathcal{F}(\mathbf{case}\ \mathcal{R}(h_1^I) \\
& & \quad \mathbf{of}\ \underline{\lambda}(x) \Rightarrow x\ @'\ \mathcal{R}(h_2^I) \\
& & \quad \mathbf{else}\ x_0 \Rightarrow x_0 \mathbin{\underline{@}} \mathcal{R}(h_2^I))
\end{array}
$$

  Because $h_1^I$ is a statically-closed input HOAS-term, $\mathcal{R}(h_1^I)$ weakly reduces to a statically-closed output HOAS-term by Lemma 3.6. We again perform case analysis on the form of the output HOAS-term.

  - If $\mathcal{R}(h_1^I) \to_w^* \underline{\lambda}(\lambda y.\ \mathcal{R}((\lambda z.\ h_0^I)\ @'\ y))$, then $h_1^{I\prime} \to_w^* \underline{\lambda}(\lambda z.\ h_0^{I\prime\#})$ and $\mathit{Erase}(h_1^I) \to_w^* \lambda z.\ \mathit{Erase}^\#(h_0^I)$ by Lemma 3.6, and the reduction sequence of $\mathcal{F}(\mathcal{R}(h^I))$ should continue as follows, where $\mathcal{R}(h_2^I) \to_w^* h^O$.

$$
\begin{array}{ll}
\to_w^* & \mathcal{F}(\mathbf{case}\ \underline{\lambda}(\lambda y.\ \mathcal{R}((\lambda z.\ h_0^I)\ @'\ y)) \\
& \quad \mathbf{of}\ \underline{\lambda}(x) \Rightarrow x\ @'\ \mathcal{R}(h_2^I) \\
& \quad \mathbf{else}\ x_0 \Rightarrow x_0 \mathbin{\underline{@}} \mathcal{R}(h_2^I)) \\
\to_w & \mathcal{F}((\lambda y.\ \mathcal{R}((\lambda z.\ h_0^I)\ @'\ y))\ @'\ \mathcal{R}(h_2^I)) \\
\to_w^* & \mathcal{F}((\lambda y.\ \mathcal{R}((\lambda z.\ h_0^I)\ @'\ y))\ @'\ h^O) \\
\to_w & \mathcal{F}(\mathcal{R}((\lambda z.\ h_0^I)\ @'\ h^O)) \\
\to_w & \mathcal{F}(\mathcal{R}([h^O/z]h_0^I)) \\
\to_w^* & t
\end{array}
$$

    Since $h_2^I$ is a statically-closed input HOAS-term, $h^O$ is a statically-closed output HOAS-term, $h_2^{I\prime} \to_w^* h^{O\prime}$, and $\mathit{Erase}(h_2^I) \to_w^* \mathit{Erase}(h^O)$ by Lemma 3.6. Furthermore, since $[h^O/z]h_0^I$ is a statically-closed input HOAS-term, $\mathcal{F}([h^{O\prime}/z]h_0^{I\prime}) \to_w^* t$ and $\downarrow([\mathit{Erase}(h^O)/z]\mathit{Erase}(h_0^I)) \to_w^* t$ by the induction hypothesis. Therefore, $\mathcal{F}(h^{I\prime})$ and $\downarrow(\mathit{Erase}(h^I))$ weakly reduce to $t$ as follows.

$$
\begin{array}{lll}
\mathcal{F}(h^{I\prime}) & = & \mathcal{F}(h_1^{I\prime} \mathbin{\underline{@}'} h_2^{I\prime}) \\
& \to_w^* & \mathcal{F}(\underline{\lambda}(\lambda z.\ h_0^{I\prime\#}) \mathbin{\underline{@}'} h^{O\prime}) \\
& \to_w & \mathcal{F}((\lambda z.\ h_0^{I\prime\#})\ @'\ h^{O\prime}) \\
& \to_w & \mathcal{F}([h^{O\prime}/z]h_0^{I\prime\#}) \\
& \to_w^* & \mathcal{F}([h^{O\prime}/z]h_0^{I\prime}) \\
& \to_w^* & t
\end{array}
$$

$$
\begin{array}{lll}
\downarrow(\mathit{Erase}(h)) & = & \downarrow(\mathit{Erase}(h_1^I)\ @'\ \mathit{Erase}(h_2^I)) \\
& \to_w^* & \downarrow((\lambda z.\ \mathit{Erase}^\#(h_0^I))\ @'\ \mathit{Erase}(h^O)) \\
& \to_w & \downarrow([\mathit{Erase}(h^O)/z]\mathit{Erase}^\#(h_0^I)) \\
& \to_w^* & \downarrow([\mathit{Erase}(h^O)/z]\mathit{Erase}(h_0^I)) \\
& \to_w^* & t
\end{array}
$$

Other cases are similar or trivial. □

Table 1: Type-Directed Partial Evaluators for Experiments

| Name | Description |
|---|---|
| **Dan96** | Danvy's totally offline type-directed partial evaluator [4], where no destructors are extended and reflection is performed eagerly for any types. |
| **She97** | Sheard's partially offline and partially online type-directed partial evaluator [14], where destructors for primitive data (such as + for integers) are extended and reflection is performed lazily for compound types (such as function types). |
| **Sum99** | Our totally online type-directed partial evaluator, where all destructors (such as @ for functions) are extended and reflection is never performed. |

Table 2: Source Programs for Experiments

| Name | Description |
|---|---|
| **id** | An identity function $\lambda \mathtt{x} : (\mathtt{int} + \mathtt{int}) \times (\mathtt{int} + \mathtt{int}).\ \mathtt{x}$ for pairs of variants of integers. This example supposes a term given a more special type than the principal type because of the context, which is probable in the Hindley-Milner type system. |
| **power** | A power function for integers, with the exponent (= 10) static. |
| **append** | An append function for lists of integers, with the first argument (= $[2, 3, 5]$) static. |
| **SK** | A combination of S/K-combinators that computes $2 \times 3$ in Church numbers. This is an example where a lot of polymorphic function application occurs at the specialization stage. |
| **format** | A string formatter, with the control parameter (that represents `"%d plus %d equals %s"`) static. |
| **int1** | An interpreter for a simple but Turing-complete imperative language, with the source program (that computes $\Sigma_{i=1}^{n} i$ for given $n$) static. An environment is represented as a finite partial function from strings to values. |
| **int2** | Similar to **int1**, but an environment is represented as a pair of a list of strings and a list of values. |

# 4 Experiments

For the purpose of assessing the merit of avoiding reflection and the demerit of extending primitive destructors, we implemented three type-directed partial evaluators (described in Table 1) and specialized seven source programs (described in Table 2). We used a variant of dynamically-typed $\lambda$-calculus with pairs, variants, integers and strings as the object language. We measured (1) the time required for specialization of the source programs with static inputs and (2) the time required for execution of the residual programs with dynamic inputs.

Although the object language is dynamically-typed, some static type information is necessary in **Dan96** and **She97**. More specifically, reification in **Dan96** needs to be guided by the static types of the residual programs, and functions in **She97** need to be annotated with the static types of the arguments. We provided those types by hand, though they may be inferred automatically.

In **Dan96**, we cannot reify a value whose type includes primitive types (such as `int`) or inductive types (such as `int list`) in negative positions, because it leads to non-termination [4]. Therefore, we abstracted out primitive operators for those types in the source programs as Danvy [4] did, so that they become dynamic. For example, in order to partially-evaluate $\lambda \mathtt{L}.\ \mathtt{tail}(1 :: \mathtt{L})$, which is typed as $\mathtt{int\ list} \to \mathtt{int\ list}$, we should abstract the `tail` out like $\lambda \mathtt{TAIL}.\ \lambda \mathtt{L}.\ \mathtt{TAIL}(1 :: \mathtt{L})$, which can be typed as $(\alpha \to \alpha) \to \alpha \to \alpha$. As a result, however, we fail to normalize $\mathtt{TAIL}(1 :: \mathtt{L})$ to `L`. Again, we performed the transformation by hand.

Table 3: The Time Required for Specialization of the Source Programs (in Microseconds)

| | **id** | **power** | **append** | **SK** | **format** | **int1** | **int2** |
|---|---|---|---|---|---|---|---|
| **Dan96** | 144.0 | 341.3 | 132.7 | 277.9 | 1,001.7 | 2,466.9 | 1,508.1 |
| **She97** | 223.9 | 382.0 | 533.7 | 3,164.3 | 730.5 | 2,592.0 | N/A |
| **Sum99** | 82.5 | 326.4 | 221.8 | 771.3 | 530.9 | 830.8 | 1,836.8 |

Table 4: The Time Required for Execution of the Residual Programs (in Microseconds)

| | **id** | **power** | **append** | **SK** | **format** | **int1** | **int2** |
|---|---|---|---|---|---|---|---|
| (No PE) | 0.54 | 33.57 | 9.48 | 97.05 | 54.76 | 401.85 | 993.05 |
| **Dan96** | 2.23 | 17.53 | 1.85 | 4.42 | 31.84 | 1,606.48 | 376.68 |
| **She97** | 1.09 | 17.39 | 1.89 | 4.25 | 30.88 | 1,516.27 | N/A |
| **Sum99** | 0.55 | 17.37 | 1.86 | 4.29 | 30.70 | 1,527.92 | 142.92 |

The type-directed partial evaluators and the residual programs are executed by Chez Scheme Version 6.0a on SunOS 5.5.1 on Sun Enterprise 4000 with UltraSPARC 168 MHz processors and 1.2 GB main memory.

The results of the experiments are shown in Table 3, Table 4, and Table 5. The time is the average of sufficiently repeated trials, excluding the overhead of an empty trial (i.e., specialization or execution of a null program). The line of "No PE" shows the time required for execution of each source program with both the static input and the dynamic input.

We rationalize the results as follows.

- In **id**, **format**, and **int1** of **Dan96** and **She97**, unnecessary reflection caused inefficiency both at the specialization stage and at the execution stage.
- In **append** and **SK** in **Dan96**, the residual programs were simply-typed (the type of `append @ [2,3,5]` was given by hand as $\alpha \to$ `unit + int` $\times$ `(unit + int` $\times$ `(unit + int` $\times \alpha$`))`) and the types accurately represented their behavior. In those cases, it is much more efficient to perform reflection once than to examine destructor operands repeatedly.
- In most source programs (especially **append** and **SK**) of **She97**, manipulation of polymorphic types and inductive types caused another inefficiency at the specialization stage.
- In **int1** of all the three methods, the residual program was much *slower* than the source program because of computation duplication. That is, an environment $\rho[s \mapsto \rho(s) + \rho(i)][i \mapsto \rho(d) + \rho(i)]$ was represented as a function:

```
λx. if x = "i"
    then (oldenv @ "d") + (oldenv @ "i")
    else if x = "s"
         then (oldenv @ "s") + (oldenv @ "i")
         else oldenv @ x
```

  so `(oldenv @ "d") + (oldenv @ "i")` and `(oldenv @ "s") + (oldenv @ "i")` were repeatedly computed. See Subsection 5.2 for a solution to this kind of problem.
- In **int2** of **Dan96**, the environments had to be completely dynamic in order for the compilation of the loop to terminate, so the environment manipulation was unnecessarily residualized like `(car (cons 0 ...))`. This is not the case for **She97** and **Sum99**, which can reify a function whose domain is an inductive type.
- In **int2** of **She97**, specialization was impossible: in order to reify partial functions on disjoint sum types (such as environment lookup), we need to catch runtime type errors, but Scheme provides no such mechanism.

Table 5: The Size of the Residual Programs as S-Expressions (in the Number of `cons` Cells)

| | **id** | **power** | **append** | **SK** | **format** | **int1** | **int2** |
|---|---|---|---|---|---|---|---|
| (No PE) | 15 | 66 | 110 | 413 | 216 | 449 | 642 |
| **Dan96** | 183 | 68 | 39 | 33 | 77 | 730 | 322 |
| **She97** | 28 | 56 | 39 | 33 | 47 | 255 | N/A |
| **Sum99** | 15 | 56 | 39 | 33 | 47 | 249 | 929 |

As a whole, **Sum99** was always comparable or faster than **Dan96** and **She97** both at the specialization stage and at the execution stage, except for **Dan96** in **append** and **SK**, whose domain is a simple type. From this fact, we expect that our method is superior to the others for realistic application programs whose input is a sophisticated data structure, at least in dynamically-typed languages.

# 5 Extensions and Limitations

## 5.1 Recursion

Theorem 2.11 does not imply termination of the reification. In fact, reification of a recursive function with a dynamic recursive variable may not terminate, because partial evaluation of a dynamic conditional expression **if** $t_1$ **then** $t_2$ **else** $t_3$ (where the condition part $t_1$ is dynamic) requires partial evaluation of both the "then" part $t_2$ and the "else" part $t_3$. In order to ease the difficulty, Danvy [5] introduced a special fixed-point operator that doesn't unfold the recursive call if a user-defined condition on the arguments (e.g. the first argument is static) is false. It is straightforward to incorporate his remedy into our method.

## 5.2 Side Effects

Side effects that should occur at the execution stage (not at the specialization stage) can be managed by `let` insertion [3]. Duplication, elimination, and reordering of code and computation can also be avoided by a similar technique [9]. It is our future work to treat more complicated side effects such as `set-car!` in Scheme.

## 5.3 Arithmetic Reduction

It is straightforward to perform arithmetic reduction by primitive operators such as + [5]. For example, we can implement the following reduction rules for $\times'$.

$$
\begin{array}{rcll}
t \times' 0 & \to & 0 & \\
0 \times' t & \to & 0 & \\
t \times' 1 & \to & t & \\
1 \times' t & \to & t & \\
t_1 \times' t_2 & \to & t & (\text{if } t_1 \times t_2 = t \text{ and both } t_1 \text{ and } t_2 \text{ are static}) \\
t_1 \times' t_2 & \to & t_1 \mathbin{\underline{\times}}' t_2 & (\text{if } t_1, t_2 \neq 0, 1 \text{ and either } t_1 \text{ or } t_2 \text{ is dynamic}) \\
\downarrow(t_1 \mathbin{\underline{\times}}' t_2) & \to & \downarrow t_1 \mathbin{\underline{\times}}' \downarrow t_2 &
\end{array}
$$

Of course, we may adopt more sophisticated reduction rules, e.g. $(t_1 \mathbin{\underline{\times}}' t_2) \times' t_3 \to t_1 \mathbin{\underline{\times}}' t$ if $t_1$ is dynamic, $t_2$ and $t_3$ are static, and $t_2 \times t_3 = t$.

## 5.4 Static Typing

Our method is applicable not only to dynamically-typed languages such as Scheme but also to statically-typed languages such as ML. In order to merge static values of a type (such as `1`) and dynamic expressions that evaluate to a value of the type (such as $\underline{\mathtt{x} + \mathtt{y}}$), we define a data type of two-level values like `datatype 'a tlv = S of 'a | D of exp` in ML. Then, we annotate (1) every value constructors (such as `fn x` $\Rightarrow$ `e` and $\mathtt{e}_1 :: \mathtt{e}_2$

in ML) with the "static" tag and (2) every value destructor (such as function application, `hd` and `tl` in ML) with the type of the value to reify. It is straightforward to provide those annotations automatically by ordinary type inference.

For example, let us partially evaluate $\lambda x.\ (\lambda y.\ y)\ @'\ x$ in ML. We represent a type as a reification operator for values of the type, as Yang [18] did. For example, we represent a type variable as an untagging function for dynamic expressions (`T`). It is undefined for static values, because we never need to reify a static value whose type is unknown, provided that the type annotation is correct.

```
- fun T (D e) = e
val T = fn : 'a tlv → exp
```

Similarly, we represent a function type whose codomain is `t` as a reification operator for functions of the type (`--> t`). We omit the domain of the function, because it is unnecessary in our method.

```
- fun --> _ (D e) = e
=   | --> t (S f) =
=     let val x = gensym ()
=     in Abs(x, reify t ((f o D o Var) x)
=     end;
val ⟶ = fn : ('a → exp) → ('b tlv → 'a) tlv → exp
```

Then, the reification operator just applies the representation of a type to a value of the type.

```
- fun reify t v : exp = t v;
val reify = fn : ('a → exp) → 'a → exp
```

Function application (`f @ (x, t)`) can be implemented as $@'$ in dynamically-typed languages, except for the binding-time tags (`S` and `D`) and the type annotation (`t`).

```
- fun (S f) @ (x, _) = f x
=   | (D e) @ (x, t) = D(App(e, reify t x));
val @ = fn : ('a → 'b tlv) tlv * ('a * ('a → exp)) → 'b tlv
```

Getting all of these together, we can normalize $\lambda x.\ (\lambda y.\ y)\ @'\ x$ to $\lambda x.\ x$ as follows.

```
- reify (--> T) (S(fn x => S(fn y => y) @ (x, T)));
val it = Abs ("x1",Var "x1") : exp
```

# 6 Related Work

In addition to Berger and Schwichtenberg's theory [2], Danvy's series of papers [3–5] and Sheard's study [14], the following work is closely related to ours.

- Mogensen [11] realized a Gödelizer in an extension of pure untyped $\lambda$-calculus. However, it won't work as a partial evaluator for realistic languages with various values, because it relies on the property that all values are functions.
- Helsen and Thiemann [7] compared Danvy's type-directed partial evaluator and the second author's cogen-based offline syntax-directed partial evaluator [16, 17], and found them similar both qualitatively and quantitatively. We showed that the two techniques are equivalent for *online* partial evaluation.
- Zhe Yang[2], independently of us, implemented a partial evaluator similar to ours in Scheme, though he did neither rigorous formalization nor comparative experiments.

[2] Personal communication, in January and March, 1999.

# 7 Conclusion

We realized an online type-directed partial evaluator for a dynamically-typed language by extending primitive value destructors and unnecessitating reflection. We also showed the correspondence between our type-directed partial evaluator and an online version of a cogen-based approach to the syntax-directed partial evaluator with higher-order abstract syntax, which suggests the possibility of more powerful partial evaluators. Our future work includes more extensive guarantee of termination and more sophisticated treatment of side effects.

# Acknowledgment

We would like to thank many people including Oliver Danvy, Peter Thiemann, Kenichi Asai, Hidehiko Masuhara, Kenjiro Taura, Zhe Yang, Morten Rhiger, and anonymous reviewers for their invaluable comments on earlier versions of this work.

# References

[1] Kenichi Asai, Hidehiko Masuhara, and Akinori Yonezawa. Partial evaluation of call-by-value $\lambda$-calculus with side-effects. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–21, 1997.

[2] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, 1991.

[3] Olivier Danvy. Pragmatic aspects of type-directed partial evaluation. Technical Report RS-96-15, Basic Research in Computer Science, 1996. Available at `http://www.brics.dk/RS/96/15/`. Appeared in Danvy, Glück and Thiemann, editors, Partial Evaluation, PE '96 Proceedings, LNCS 1110, 1996, pages 73–94.

[4] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–257, 1996.

[5] Olivier Danvy. Online type-directed partial evaluation. Technical Report RS-97-53, Basic Research in Computer Science, 1997. Available at `http://www.brics.dk/RS/97/53/`. Extended version of an article that appeared in *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, Masahiko Sato and Yoshihito Toyama, editors, World Scientific, 1998.

[6] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, 1994.

[7] Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In *4th Asian Computing Science Conference*, volume 1538 of *Lecture Notes in Computer Science*, pages 188–205. Springer-Verlag, 1998.

[8] Richard Kelsey, William Clinger, and Jonathan Rees. Revised$^5$ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Available at `http://www.brics.dk/~hosc/11-1/168705.html`. Also appears in *ACM SIGPLAN Notices*, Vol. 33, No. 9, Pages 26–76, September 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[9] Julia Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190. Springer-Verlag, 1997.

[10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[11] Torben Mogensen. Gödelisation in the untyped λ-calculus. In *Draft Proceedings of the 1999 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 19–24, 1999. Available at `http://www.brics.dk/~pepm99/Proceedings/mogensen.ps`.

[12] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

[13] Morten Rhiger. A study in higher-order programming languages. Master's thesis, University of Aarhus, Department of Computer Science, 1997. Available at `http://www.daimi.aau.dk/~mrhiger/`.

[14] Tim Sheard. A type-directed, on-line partial evaluator for a polymorphic language. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–35, 1997.

[15] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 306–313. ACM, 1995.

[16] Peter Thiemann. Cogen in six lines. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, pages 180–189, 1996.

[17] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[18] Zhe Yang. Encoding types in ML-like languages. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 289–300, 1998.

# Appendix: Implementation and Examples

Below, we show an implementation of our partial evaluator for a subset of Scheme. Be aware that it doesn't support procedures with more than one arguments.

We assume that we have already implemented partial continuation operators (`shift` and `reset`) according to Filinski ([6], Section 4 and Subsection 5.1) for `let` insertion. In fact, we don't need them — just one global mutable store that holds bindings suffices — but we use them for clarity and simplicity.

First, we define auxiliary functions for residual code as follows.

```
; we assume that programmers don't use a symbol that begins with _ or ~

; we represent dynamic variables as '~v1, '~v2, '~v3, ...
(define gensym!
  (let ((seq 0))
    (lambda ()
      (set! seq (+ 1 seq))
      (string->symbol (string-append "~v" (number->string seq))))))

(define (dynamic-symbol? x)
  (and (symbol? x)
       (char=? #\~ (string-ref (symbol->string x) 0))))

; we represent dynamic operations as '(_car pair), '(_+ 1 2), '(_@ func arg), ...
(define (dynamic-operation? x)
  (and (pair? x)
       (symbol? (car x))
       (char=? #\_ (string-ref (symbol->string (car x)) 0))))

(define (dynamic? x)
```

```
  (or (dynamic-symbol? x)
      (dynamic-operation? x)))

; we don't have to extend these operators

(define _cons cons)
(define _list list)

(define-syntax _let
  (syntax-rules ()
    ((_let x ...)
     (let x ...))))

(define-syntax _lambda
  (syntax-rules ()
    ((_lambda x ...)
     (lambda x ...))))
```

Next, we implement the reification operator.

```
(define (reify_ x)
  (cond
   ((dynamic? x)
    x)
   ((procedure? x)
    (let ((var (gensym!)))
      `(_lambda (,var) ,(reset (reify_ (x var))))))
   ((pair? x)
    `(_cons ,(reify_ (car x)) ,(reify_ (cdr x))))
   ((null? x)
    '(quote ()))
   ((symbol? x)
    `(quote ,x))
   (else
    x)))

; instruction for primitive procedures with dynamic effects
(define reifying #f)

(define (reify v)
  (set! reifying #t)
  (let ((e (reify_ v)))
    (set! reifying #f)
    e))
```

Then, we define primitive value destructors as follows.

```
(define (_@ proc args)
  (if (dynamic? proc)
      `(_@ ,proc ,(reify_ args))
      (proc args)))

(define (_apply proc args)
  (if (dynamic? proc)
      `(_apply ,proc ,(reify_ args))
```

```
      (apply proc args)))

(define (_null? pair)
  (if (dynamic? pair)
      `(_null? ,pair)
      (null? pair)))

(define (_car pair)
  (if (dynamic? pair)
      `(_car ,pair)
      (car pair)))

(define (_cdr pair)
  (if (dynamic? pair)
      `(_cdr ,pair)
      (cdr pair)))

(define (_+ x y)
  (cond
   ((and (number? x) (zero? x))
    y)
   ((and (number? y) (zero? y))
    x)
   ((or (dynamic? x) (dynamic? y))
    `(_+ ,x ,y))
   (else
    (+ x y))))

(define (_* x y)
  (cond
   ((and (number? x) (= 1 x))
    y)
   ((and (number? y) (= 1 y))
    x)
   ((and (number? x) (zero? x))
    0)
   ((and (number? y) (zero? y))
    0)
   ((or (dynamic? x) (dynamic? y))
    `(_* ,x ,y))
   (else
    (* x y))))

(define (_positive? x)
  (if (dynamic? x)
      `(_positive? ,x)
      (positive? x)))

(define (_if_ test consequent_ alternate_)
  (if (dynamic? test)
      `(_if ,test
            ,(reset (reify_ (consequent_)))
            ,(reset (reify_ (alternate_))))
```

```
      (if test (consequent_) (alternate_))))

(define-syntax _if
  (syntax-rules ()
    ((_if test consequent alternate)
     (_if_ test
           (lambda () consequent)
           (lambda () alternate)))))

(define (_fix proc)
  (lambda (arg)
    (if (dynamic? arg)
        `(_@ (_fix ,(reify_ proc)) ,(reify_ arg))
        ((proc (_fix proc)) arg))))
```

Last, we declare primitive procedures that have I/O effects at execution stage. In this implementation, we don't deal with other kinds of effects.

```
(define-syntax declare-primitive-effect
 (syntax-rules ()
  ((declare-primitive-effect stub func)
   (define stub
     (lambda arg
       (if reifying
           (shift (lambda (pk)
                    (let ((var (gensym!)))
                      `(_let ((,var (_apply stub ,(reify_ arg))))
                         ,(reset (reify_ (pk var)))))))
           (apply func arg)))))))

(declare-primitive-effect _newline newline)
(declare-primitive-effect _display display)
```

Below is an example session of the partial evaluator.

```
> (define power
    (lambda (b)
      (_fix
       (lambda (p)
         (lambda (e)
           (_if (_positive? e)
                (_* b (_@ p (_+ e -1)))
                1))))))
> ((power 2) 5)
32
> (define power-5
    (reify (lambda (b) ((power b) 5))))
> power-5
(_lambda (~v1)
  (_* ~v1 (_* ~v1 (_* ~v1 (_* ~v1 ~v1)))))
> ((eval power-5) 2)
32
> (define apnd-rev
    (lambda (y)
      (_fix
```

```
       (lambda (a)
         (lambda (x)
           (_if (_null? x)
                y
                (cons (_car x) (_@ a (_cdr x)))))))))
> (define apnd (lambda (x) (lambda (y) ((apnd-rev y) x))))
> ((apnd (list "a" "b" "c"))
         (list "d" "e"))
("a" "b" "c" "d" "e")
> (define apnd-abc
    (reify (apnd (list "a" "b" "c"))))
> apnd-abc
(_lambda (~v2)
  (_cons "a" (_cons "b" (_cons "c" ~v2))))
> ((eval apnd-abc) (list "d" "e"))
("a" "b" "c" "d" "e")
> (define (power-verbose b e)
    (_display b)
    (_display " to the power of ")
    (_display e)
    (_display " is ")
    (_display ((power b) e))
    (_display ".")
    (_newline))
> (power-verbose 2 5)
2 to the power of 5 is 32.
> (define power-verbose-5 (reify (lambda (b) (power-verbose b 5))))
> power-verbose-5
(_lambda (~v3)
 (_let ([~v4 (_apply _display (_cons ~v3 '()))])
  (_let ([~v5 (_apply _display (_cons " to the power of " '()))])
   (_let ([~v6 (_apply _display (_cons 5 '()))])
    (_let ([~v7 (_apply _display (_cons " is " '()))])
     (_let ([~v8 (_apply _display (_cons (_* ~v3 (_* ~v3 (_* ~v3 (_* ~v3 ~v3)))) '()))])
      (_let ([~v9 (_apply _display (_cons "." '()))])
       (_let ([~v10 (_apply _newline '())]) ~v10))))))))
> ((eval power-verbose-5) 2)
2 to the power of 5 is 32.
```