

# Towards Type-Directed Partial Evaluation for Shift and Reset

Kanae Tsushima  
Ochanomizu University  
tsushima.kanae@is.ocha.ac.jp

Kenichi Asai  
Ochanomizu University  
asai@is.ocha.ac.jp

## Abstract

*This paper reports on the ongoing work to extend type-directed partial evaluation (TDPE) to cope with delimited-control operators, `shift` and `reset`. By examining call-by-value TDPE of CPS expressions and transforming it back to direct style, we obtain call-by-value TDPE for `shift` and `reset`. The preliminary experiments show that the `printf` function written with `shift` and `reset` has been successfully partially evaluated.*

## 1 Introduction

Type-directed partial evaluation (TDPE), proposed by Danvy [3], is an instance of normalization by evaluation. Given an input term, it produces partially evaluated term by systematically  $\eta$ -expanding the input term and then reducing it to a normal form. Since TDPE does not touch the structure of the input term but executes it directly, the partial evaluation process is extremely fast. Because of this efficiency, TDPE is an attractive tool for optimizing programs.

Control operators, on the other hand, are used to manipulate flow of control, or continuations, without transforming a program into continuation-passing style (CPS). Among various control operators, this paper deals with `shift` and `reset` [5], which are used to capture delimited continuations. Because of their close relationship to CPS and clear semantics, they are used in various places, such as non-deterministic programming [5], let-insertion in partial evaluation [10], and writing a typed `printf` function [2].

In this paper, we aim at extending the traditional TDPE to cope with the delimited-control operators, `shift` and `reset`, thereby enlarging applicability of

TDPE. We do so by first observing how CPS expressions are treated in the conventional call-by-value TDPE. We then transform it back to direct style to obtain TDPE for `shift` and `reset`.

Although the proof of correctness is not established yet, the preliminary experiments show that the `printf` function written with `shift/reset` can be successfully partially evaluated in this framework. We hope that the present work paves a way towards type-directed partial evaluation for `shift/reset`.

In the next section, we introduce `shift/reset`. After reviewing Danvy's TDPE in Section 3 and TDPE for call by value in Section 4, we present TDPE for `shift/reset` in Section 5. The paper concludes in Section 6.

## 2 Shift/reset

The control operators `shift` and `reset` are used to capture delimited continuations. Like `call/cc` in Scheme, `shift` captures the current continuation, but unlike `call/cc`, it captures the continuation only up to the enclosing `reset`.

In the concrete syntax, `(shift k M)` captures the current continuation, binds it to `k`, and executes `M` with an empty continuation; `(reset M)` executes `M` with an empty continuation. For example,

```
(+ 1 (reset (+ 2 (shift k (k (k 3))))))  
~> (+ 1 (reset (+ 2 (+ 2 3))))  
~> (+ 1 (reset 7))  
~> (+ 1 7)  
~> 8
```

The continuation captured at the first line is `(+ 2 [])`, because the `shift` expression is enclosed by `reset`. The continuation is used twice in the body of the `shift` expression. When the body of `reset` is executed to a value, it becomes the value of the `reset` expression. So, `(reset n)` becomes `n`.

$$\begin{array}{lcl}
t \in \text{Type} & ::= & b \mid t_1 \rightarrow t_2 \mid t_1 \times t_2 \\
v \in \text{Value} & ::= & c \mid x \mid \overline{\lambda x.v} \mid v_0 \overline{\text{@}} v_1 \mid \\
& & \underline{\text{pair}}(v_1, v_2) \mid \underline{\text{fst}} v \mid \underline{\text{snd}} v \\
e \in \text{Expr} & ::= & c \mid x \mid \underline{\lambda x.e} \mid e_0 \underline{\text{@}} e_1 \\
& & \underline{\text{pair}}(e_1, e_2) \mid \underline{\text{fst}} e \mid \underline{\text{snd}} e \\
\text{reify} & = & \lambda t. \lambda v : t. \downarrow^t v \\
& & : \text{Type} \rightarrow \text{Value} \rightarrow \text{TLT} \\
\downarrow^b v & = & v \\
\downarrow^{t_1 \rightarrow t_2} v & = & \underline{\lambda x_1^\diamond}. \downarrow^{t_2} (v \overline{\text{@}} \uparrow_{t_1} x_1^\diamond) \\
\downarrow^{t_1 \times t_2} v & = & \underline{\text{pair}}(\downarrow^{t_1} \underline{\text{fst}} v, \downarrow^{t_2} \underline{\text{snd}} v) \\
\text{reflect} & = & \lambda t. \lambda e : t. \uparrow_t e \\
& & : \text{Type} \rightarrow \text{Expr} \rightarrow \text{TLT} \\
\uparrow_b e & = & e \\
\uparrow_{t_1 \rightarrow t_2} e & = & \overline{\lambda v_1}. \uparrow_{t_2} (e \underline{\text{@}} \downarrow^{t_1} v_1) \\
\uparrow_{t_1 \times t_2} e & = & \underline{\text{pair}}(\uparrow_{t_1} \underline{\text{fst}} e, \uparrow_{t_2} \underline{\text{snd}} e) \\
\text{residualize} & = & \text{statically-reduce} \circ \text{reify} \\
& & : \text{Type} \rightarrow \text{Value} \rightarrow \text{Expr}
\end{array}$$

**Figure 1. Danvy’s original Type-Directed Partial Evaluation for the  $\lambda$ -calculus [3]**

The exact semantics of `shift` and `reset` is defined using CPS transformation [5].

### 3 Type-Directed Partial Evaluation for the $\lambda$ -calculus

Partial evaluation [8] simplifies a program as much as possible by executing known (or static) parts of the program to cut-down their execution time. Type-directed partial evaluation (TDPE) falls into the same category but differs from the standard partial evaluation in two points. First, TDPE is directed by the structure of the type of expressions, hence the name “type-directed” partial evaluation. Second, it does not manipulate structure of expressions. Instead, it receives a compiled representation of a program and produces the result by actually executing it. Thus, the partial evaluation process is much faster than the standard interpreter-based partial evaluation. This is analogous to compiled execution being faster than interpreted execution.

#### 3.1 Syntax

Danvy’s type-directed partial evaluation for the  $\lambda$ -calculus extended with constants ( $c$ ) and pairs is

shown in Figure 1 [3]. We assume that TDPE is expressed in a call-by-value language with datatype constructors, such as Scheme.

A type is either a base type ( $b$ ), a function type ( $t_1 \rightarrow t_2$ ), or a product type ( $t_1 \times t_2$ ). The overline means static. In the context of TDPE, it is a compiled representation whose internal structure cannot be inspected. On the other hand, the underline means dynamic. It is an expression represented by datatype constructors. TLT is the domain of two-level terms. It contains Expr and Value. A variable with a superscript  $\diamond$  represents a fresh variable.

#### 3.2 Reify and Reflect

TDPE process is driven by reify and reflect operators. The downarrow ( $\downarrow^t$ ) is called reify. Directed by the type  $t$ , it transforms a compiled representation into datatype.

For example, suppose that we have a compiled representation  $f$  of a function whose type is  $b \rightarrow b$ . Since it has type  $b \rightarrow b$ , we know that  $f$  has a form  $\underline{\lambda x_1.M}$  for some  $M$ . To obtain its body  $M$ , we apply  $f$  to  $x_1$ , a newly created dynamic variable. If  $f$  has a completely polymorphic type, we know that  $f$  will never inspect the value of  $x_1$  and thus  $x_1$  can be safely applied to  $f$ . We can then obtain the body of  $f$ , in this case  $x_1$ , to obtain  $\underline{\lambda x_1.x_1}$ .

The uparrow ( $\uparrow_t$ ) is called reflect. It does the opposite: it transforms the datatype representation into a compiled executable value.

Using reify, TDPE first transforms a simply-typed, completely static input into a dynamic term. It then reduces all the static terms that appear during reify (and reflect called therein).

#### 3.3 Example

Let us consider  $(\overline{\lambda x. \overline{\lambda f. f \overline{\text{@}} ((\overline{\lambda a. a}) \overline{\text{@}} x)})$  whose type is given by  $t_1 \rightarrow ((t_1 \rightarrow t_2) \rightarrow t_2)$ . TDPE for this term is shown in Figure 2. Since all its subexpressions are overlined, it is a completely static or compiled term.

By mechanically applying  $\downarrow^{t_1 \rightarrow ((t_1 \rightarrow t_2) \rightarrow t_2)}$  to  $(\overline{\lambda x. \overline{\lambda f. f \overline{\text{@}} ((\overline{\lambda a. a}) \overline{\text{@}} x)})$ , we obtain a two-level term:

$$\underline{\lambda x_1. \underline{\lambda x_2. (\overline{\lambda x. \overline{\lambda f. f \overline{\text{@}} ((\overline{\lambda a. a}) \overline{\text{@}} x)}) \overline{\text{@}} x_1 \overline{\text{@}} \overline{\lambda v_1. x_2 \overline{\text{@}} v_1}}$$

Since the original term has type  $t_1 \rightarrow ((t_1 \rightarrow t_2) \rightarrow t_2)$ , we know that it has a form  $\underline{\lambda x_1. \underline{\lambda x_2. M}}$  for some  $M$ . To obtain  $M$ , we apply the original term to two arguments:  $x_1$  and  $\overline{\lambda v_1. x_2 \overline{\text{@}} v_1}$ . Because

$$\begin{aligned}
& \downarrow^{t_1 \rightarrow ((t_1 \rightarrow t_2) \rightarrow t_2)} (\bar{\lambda}x. \bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@}x))) \\
= & \underline{\lambda}x_1. \downarrow^{(t_1 \rightarrow t_2) \rightarrow t_2} ((\bar{\lambda}x. \bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@}x))) \bar{\@} (\uparrow_{t_1} x_1)) \\
= & \underline{\lambda}x_1. \downarrow^{(t_1 \rightarrow t_2) \rightarrow t_2} ((\bar{\lambda}x. \bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@}x))) \bar{\@} x_1) \\
= & \underline{\lambda}x_1. \underline{\lambda}x_2. \downarrow^{t_2} (((\bar{\lambda}x. \bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@}x))) \bar{\@} x_1) \bar{\@} (\uparrow_{t_1 \rightarrow t_2} x_2)) \\
= & \underline{\lambda}x_1. \underline{\lambda}x_2. \downarrow^{t_2} ((\bar{\lambda}x. \bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@}x))) \bar{\@} x_1) \bar{\@} (\bar{\lambda}v_1. \uparrow_{t_2} (x_2 \bar{\@} (\downarrow^{t_1} v_1))) \\
= & \underline{\lambda}x_1. \underline{\lambda}x_2. \downarrow^{t_2} ((\bar{\lambda}x. \bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@}x))) \bar{\@} x_1) \bar{\@} (\bar{\lambda}v_1. \uparrow_{t_2} (x_2 \bar{\@} v_1)) \\
= & \underline{\lambda}x_1. \underline{\lambda}x_2. \downarrow^{t_2} ((\bar{\lambda}x. \bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@}x))) \bar{\@} x_1) \bar{\@} (\bar{\lambda}v_1. (x_2 \bar{\@} v_1)) \\
= & \underline{\lambda}x_1. \underline{\lambda}x_2. ((\bar{\lambda}x. \bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@}x))) \bar{\@} x_1) \bar{\@} (\bar{\lambda}v_1. (x_2 \bar{\@} v_1)) \\
\rightsquigarrow & \underline{\lambda}x_1. \underline{\lambda}x_2. (\bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@} x_1))) \bar{\@} (\bar{\lambda}v_1. (x_2 \bar{\@} v_1)) \\
\rightsquigarrow & \underline{\lambda}x_1. \underline{\lambda}x_2. (\bar{\lambda}v_1. (x_2 \bar{\@} v_1)) \bar{\@} ((\bar{\lambda}a.a) \bar{\@} x_1) \\
\rightsquigarrow & \underline{\lambda}x_1. \underline{\lambda}x_2. (\bar{\lambda}v_1. (x_2 \bar{\@} v_1)) \bar{\@} x_1 \\
\rightsquigarrow & \underline{\lambda}x_1. \underline{\lambda}x_2. x_2 \bar{\@} x_1
\end{aligned}$$

**Figure 2. Example: TDPE of  $(\bar{\lambda}x. \bar{\lambda}f. (f \bar{\@} ((\bar{\lambda}a.a) \bar{\@}x)))$  of type  $t_1 \rightarrow ((t_1 \rightarrow t_2) \rightarrow t_2)$**

the type of  $x_2$  is  $t_1 \rightarrow t_2$ , we do not pass  $x_2$  itself, but turn it into a static term using `reflect` and pass the resulting two-level  $\eta$ -expanded term  $\bar{\lambda}v_1. x_2 \bar{\@} v_1$ . It will enable  $f$  to use its second argument as an executable function inside  $f$ .

By executing all the static parts of the resulting term, we obtain the residual term  $\underline{\lambda}x_1. \underline{\lambda}x_2. x_2 \bar{\@} x_1$ , which is a partially evaluated term of the original term. Notice that the redex  $(\bar{\lambda}a.a) \bar{\@} x$  in the original term under lambda has been evaluated.

## 4 Type-Directed Partial Evaluation for the Call-by-value $\lambda$ -calculus

TDPE in Section 3 is for call by name. For example, it evaluates  $\bar{\lambda}x. ((\bar{\lambda}y. 1) \bar{\@} (x \bar{\@} 2))$  of type  $(\text{int} \rightarrow \alpha) \rightarrow \text{int}$  to  $(\underline{\lambda}x. 1)$ , discarding the application  $(x \bar{\@} 2)$ . However, we want to consider `shift/reset` in the call-by-value setting.

To introduce TDPE for call by value, we show Filinski’s implementation [7], where all the monads are realized using Scheme’s native effects, namely, exception, `gensym`, and state-based let-insertion [9].

### 4.1 Residualization

Syntax for call-by-value TDPE is the same as Figure 1. `Reify` and `reflect` for call by value are shown in Figure 3. The standard way to realize call-by-value TDPE is to introduce let-expressions to avoid discarding dynamic application. Here, we use state-based let-insertion [9].

As a state, we maintain a stack *wrapstack* of lists of dynamic applications to be residualized. It holds

pairs of a variable and an expression that have to be residualized in let-expressions. To residualize an expression  $M$ , we write  $[M]$  which stores  $M$  in the *wrapstack*, and returns a fresh variable  $g^\diamond$  attached to  $M$ . The collected pairs are turned into a sequence of let-expressions using *wrap*, which are placed immediately under the enclosing dynamic abstraction. Every time a dynamic abstraction is created, a new empty list is pushed onto *wrapstack* using *init* to preserve bindings of enclosing dynamic abstraction and collect the bindings for the current abstraction.

Using these operators, call-by-value TDPE is realized by residualizing dynamic application  $e \bar{\@} \downarrow^{t_1} v_1$  in  $\uparrow_{t_1 \rightarrow t_2} e$  in a let-expression, which will be placed under dynamic abstraction using *wrap* in  $\downarrow^{t_1 \rightarrow t_2} v$ . Notice that if we ignore *init*, *wrap*, and  $[..]$ , `reify` and `reflect` keep good symmetry as in Danvy’s TDPE.

### 4.2 Example

Using call-by-value TDPE, the example shown at the beginning of this section,  $\bar{\lambda}x. ((\bar{\lambda}y. 1) \bar{\@} (x \bar{\@} 2))$  of type  $(\text{int} \rightarrow \alpha) \rightarrow \text{int}$ , is partially evaluated to  $\underline{\lambda}x. \text{let } y = x \bar{\@} 2 \text{ in } 1$ . The application  $x \bar{\@} 2$  is not discarded but is residualized in a let-expression.

## 5 Towards Type-Directed Partial Evaluation for Shift and Reset

### 5.1 Syntax

The target language and its type are shown in Figure 4. A function type in the presence of

$\begin{aligned} \text{reify} &= \lambda t. \lambda v : t. \downarrow^t v \\ &: \text{Type} \rightarrow \text{Value} \rightarrow \text{TLT} \\ \downarrow^b v &= v \\ \downarrow^{t_1 \rightarrow t_2} v &= \underline{\lambda} x_1^\diamond. \text{init}; \text{wrap}(\downarrow^{t_2} (v \overline{\text{@}} \uparrow_{t_1} x_1^\diamond)) \\ \downarrow^{t_1 \times t_2} v &= \underline{\text{pair}}(\downarrow^{t_1} \underline{\text{fst}} v, \downarrow^{t_2} \underline{\text{snd}} v) \end{aligned}$	$\begin{aligned} \text{init} &= \text{wrapstack} \leftarrow () :: \text{wrapstack} \\ \text{wrap}(e) &= \text{let } f :: \text{rest} = \text{wrapstack} \text{ in} \\ &\quad \text{wrapstack} \leftarrow \text{rest}; \\ &\quad \text{wrap-gen}(f, e) \end{aligned}$
$\begin{aligned} \text{reflect} &= \lambda t. \lambda e : t. \uparrow_t e \\ &: \text{Type} \rightarrow \text{Expr} \rightarrow \text{TLT} \\ \uparrow_b e &= e \\ \uparrow_{t_1 \rightarrow t_2} e &= \overline{\lambda} v_1. \uparrow_{t_2} [e \overline{\text{@}} \downarrow^{t_1} v_1] \\ \uparrow_{t_1 \times t_2} e &= \underline{\text{pair}}(\uparrow_{t_1} \underline{\text{fst}} e, \uparrow_{t_2} \underline{\text{snd}} e) \end{aligned}$	$\begin{aligned} [e] &= \text{let } f :: \text{rest} = \text{wrapstack} \text{ in} \\ &\quad ((g^\diamond, e) :: f) :: \text{rest}; \\ &\quad g^\diamond \end{aligned}$
$\begin{aligned} \text{residualize} &= \text{statically-reduce} \circ \text{reify} \\ &: \text{Type} \rightarrow \text{Value} \rightarrow \text{Expr} \end{aligned}$	$\begin{aligned} \text{wrap-gen}([], \text{body}) &= \text{body} \\ \text{wrap-gen}(\text{lst}, \text{body}) &= \text{let } (l, r) :: \text{rest} = \text{lst} \text{ in} \\ &\quad \text{wrap-gen}(\text{rest}, \underline{\langle \text{let } l = r \text{ in } \text{body} \rangle}) \end{aligned}$

**Figure 3. Type-Directed Partial Evaluation for the call-by-value  $\lambda$ -calculus**

$t \in \text{Type}$	::= $b \mid t_1/\alpha \rightarrow t_2/\beta \mid t_1 \times t_2$
$v \in \text{Value}$	::= $c \mid x \mid \overline{\lambda} x. v \mid v_0 \overline{\text{@}} v_1 \mid \overline{\langle v \rangle} \mid \overline{\text{Sk}}. v$ $\underline{\text{pair}}(v_1, v_2) \mid \underline{\text{fst}} v \mid \underline{\text{snd}} v$
$e \in \text{Expr}$	::= $c \mid x \mid \underline{\lambda} x. e \mid e_0 \overline{\text{@}} e_1 \mid \underline{\langle e \rangle} \mid \underline{\text{Sk}}. e$ $\underline{\text{pair}}(e_1, e_2) \mid \underline{\text{fst}} e \mid \underline{\text{snd}} e$

**Figure 4.  $\lambda$ -calculus with shift/reset**

**shift/reset** has the shape  $t_1/\alpha \rightarrow t_2/\beta$ . Here,  $t_1$  is the type of the argument and  $t_2$  is the type of the returned expression. In addition,  $\alpha$  and  $\beta$  are the answer types of the context before and after the function is called, respectively. The type of an expression containing **shift/reset** is formally defined by Danvy and Filinski [4].

The terms of the language contain the standard  $\lambda$ -calculus plus **shift/reset** expressions. The term  $\text{Sk}. v$  represents (**shift k v**), whereas  $\langle v \rangle$  represents (**reset v**). We assume that the input term is completely static and is typed according to the type system by Danvy and Filinski.

## 5.2 Type-Directed Partial Evaluation for CPS Expressions

Before introducing **shift/reset** into TDPE, let us consider how CPS expressions are evaluated by the conventional TDPE.

By CPS transformation, a function type  $t_1/\alpha \rightarrow t_2/\beta$  corresponds to  $(t_1 \times (t_2 \rightarrow \alpha)) \rightarrow \beta$  in uncur-

ried form. Thus, we can observe how CPS expressions are handled by considering  $\downarrow^{(t_1 \times (t_2 \rightarrow \alpha)) \rightarrow \beta} v$  and  $\uparrow_{(t_1 \times (t_2 \rightarrow \alpha)) \rightarrow \beta} e$  in the traditional call-by-value TDPE:

$$\begin{aligned} \downarrow^{t_1 \times (t_2 \rightarrow \alpha) \rightarrow \beta} v &= \\ &\quad \underline{\lambda} p_1^\diamond. \text{init}; \text{wrap}(\downarrow^\beta (v \overline{\text{@}} \overline{\text{pair}}(\uparrow_{t_1} \underline{\text{fst}} p_1^\diamond, \\ &\quad \overline{\lambda} v_1. \uparrow_\alpha [( \underline{\text{snd}} p_1^\diamond \overline{\text{@}} \downarrow^{t_2} v_1 ]))) \\ \uparrow_{t_1 \times (t_2 \rightarrow \alpha) \rightarrow \beta} e &= \\ &\quad \overline{\lambda} p_1. \uparrow_\beta [(e \overline{\text{@}} \underline{\text{pair}}(\downarrow^{t_1} \underline{\text{fst}} p_1, \\ &\quad \underline{\lambda} x_1^\diamond. \text{init}; \text{wrap}(\downarrow^\alpha (\underline{\text{snd}} p_1 \overline{\text{@}} \uparrow_{t_2} x_1^\diamond)))] \end{aligned}$$

In the above two expressions, the outermost abstractions ( $\underline{\lambda} p_1^\diamond \dots$ ) and ( $\overline{\lambda} p_1 \dots$ ) receive an argument of type pair. By introducing a pattern match construct for a pair, we can rewrite the above expressions as follows:

$$\begin{aligned} \downarrow^{t_1 \times (t_2 \rightarrow \alpha) \rightarrow \beta} v &= \\ &\quad \underline{\lambda}(x_1^\diamond, k_1^\diamond). \text{init}; \text{wrap}(\downarrow^\beta (v \overline{\text{@}} \overline{\text{pair}}(\uparrow_{t_1} x_1^\diamond, \\ &\quad \overline{\lambda} v_1. \uparrow_\alpha [(k_1^\diamond \overline{\text{@}} \downarrow^{t_2} v_1)]))) \\ \uparrow_{t_1 \times (t_2 \rightarrow \alpha) \rightarrow \beta} e &= \\ &\quad \overline{\lambda}(v_1, k_1). \uparrow_\beta [(e \overline{\text{@}} \underline{\text{pair}}(\downarrow^{t_1} v_1, \\ &\quad \underline{\lambda} x_1^\diamond. \text{init}; \text{wrap}(\downarrow^\alpha (k_1 \overline{\text{@}} \uparrow_{t_2} x_1^\diamond)))] \end{aligned}$$

Using this observation, we consider type-directed partial evaluation for **shift/reset**.

## 5.3 Residualization

In this section, we consider type-directed partial evaluation for **shift/reset**. Essentially, we trans-

$$\begin{aligned}
\text{reify} &= \lambda t. \lambda v : t. \downarrow^t v \\
&: \text{Type} \rightarrow \text{Expr} \rightarrow \text{TLT} \\
\downarrow^b v &= v \\
\downarrow^{t_1/\alpha \rightarrow t_2/\beta} v &= \underline{\lambda} x_1^\diamond. \underline{\mathcal{S}} k_1^\diamond. \underline{\text{init}}; \underline{\text{wrap}}(\downarrow^\beta \overline{\text{let}} v_1 = v \overline{\text{@}} \uparrow_{t_1} x_1^\diamond \overline{\text{in}} (\uparrow_\alpha [k_1^\diamond \overline{\text{@}} \downarrow^{t_2} v_1])) \\
\downarrow^{t_1 \times t_2} v &= \underline{\text{pair}}(\downarrow^{t_1} \underline{\text{fst}} v, \downarrow^{t_2} \underline{\text{snd}} v) \\
\\
\text{reflect} &= \lambda t. \lambda e : t. \uparrow_t e \\
&: \text{Type} \rightarrow \text{Value} \rightarrow \text{TLT} \\
\uparrow_b e &= e \\
\uparrow_{t_1/\alpha \rightarrow t_2/\beta} e &= \overline{\lambda} v_1. \overline{\mathcal{S}} k_1. \uparrow^\beta [(\underline{\text{let}} x_1^\diamond = (e \overline{\text{@}} \downarrow_{t_1} v_1) \underline{\text{in}} (\underline{\text{init}}; \underline{\text{wrap}}(\downarrow_\alpha (k_1 \overline{\text{@}} \uparrow^{t_2} x_1^\diamond)))] \\
\uparrow^{t_1 \times t_2} e &= \underline{\text{pair}}(\uparrow_{t_1} \underline{\text{fst}} e, \uparrow_{t_2} \underline{\text{snd}} e)
\end{aligned}$$

**Figure 5. Type-Directed Partial Evaluation for shift/reset**

form the TDPE of CPS expressions back to direct style.

Let us consider `reify`. In CPS,  $\underline{\lambda}(x_1^\diamond, k_1^\diamond)$  receives an argument and the continuation. In direct style,  $x_1^\diamond$  corresponds to receiving an argument and  $k_1^\diamond$  capturing the continuation. Thus, we obtain  $\underline{\lambda} x_1^\diamond. \underline{\mathcal{S}} k_1^\diamond. M$  where  $M$  is the direct-style counterpart of the body.

To obtain  $M$ , we first observe that  $v$  is applied to  $\uparrow_{t_1} x_1^\diamond$  with the continuation  $\overline{\lambda} v_1. \uparrow_\alpha [(k_1^\diamond \overline{\text{@}} \downarrow^{t_2} v_1)]$ . In direct style, it corresponds to executing  $v \overline{\text{@}} \uparrow_{t_1} x_1^\diamond$  in the *context*  $\overline{\lambda} v_1. \uparrow_\alpha [(k_1^\diamond \overline{\text{@}} \downarrow^{t_2} v_1)]$ . Thus, the overall structure of  $\downarrow^{t_1/\alpha \rightarrow t_2/\beta} v$  appears to become:

$$\begin{aligned}
&\underline{\lambda} x_1^\diamond. \underline{\mathcal{S}} k_1^\diamond. \underline{\text{init}}; \\
&\underline{\text{wrap}}(\downarrow^\beta (\overline{\lambda} v_1. \uparrow_\alpha [(k_1^\diamond \overline{\text{@}} \downarrow^{t_2} v_1)]) \overline{\text{@}} (v \overline{\text{@}} \uparrow_{t_1} x_1^\diamond))
\end{aligned}$$

However, it does not work because it does not delimit the context. Remember that  $v$  is a static term that may contain a static `shift` expression. If  $\downarrow^{t_1/\alpha \rightarrow t_2/\beta} v$  were defined as above, the static `shift` in  $v$  would have captured whole the rest of the computation. Instead, it needs to capture only the context  $(\overline{\lambda} v_1. \uparrow_\alpha [(k_1^\diamond \overline{\text{@}} \downarrow^{t_2} v_1)])$ . To delimit the context captured by  $v$ , we define  $\downarrow^{t_1/\alpha \rightarrow t_2/\beta} v$  as follows:

$$\begin{aligned}
&\underline{\lambda} x_1^\diamond. \underline{\mathcal{S}} k_1^\diamond. \underline{\text{init}}; \\
&\underline{\text{wrap}}(\downarrow^\beta \overline{(\overline{\lambda} v_1. \uparrow_\alpha [k_1^\diamond \overline{\text{@}} \downarrow^{t_2} v_1]) \overline{\text{@}} (v \overline{\text{@}} \uparrow_{t_1} x_1^\diamond)})
\end{aligned}$$

Similarly, we can define `reflect`  $\uparrow_{t_1/\alpha \rightarrow t_2/\beta} e$  as:

$$\begin{aligned}
&\overline{\lambda} v_1. \overline{\mathcal{S}} k_1. \\
&\uparrow^\beta [(\underline{\lambda} x_1^\diamond. \underline{\text{init}}; \underline{\text{wrap}}(\downarrow_\alpha (k_1 \overline{\text{@}} \uparrow^{t_2} x_1^\diamond))) \\
&\quad \overline{\text{@}} (e \overline{\text{@}} \downarrow_{t_1} v_1)]
\end{aligned}$$

By rewriting static and dynamic  $\beta$ -redexes in the above definitions with static and dynamic `let`-expressions, we obtain TDPE for `shift/reset`

shown in Figure 5. Here, we assumed that the underlying language has (static) `shift` and `reset` operators [6].

If we ignore `init`, `wrap`, and `[...]`, as before, `reify` and `reflect` are defined in a symmetrical way.

## 5.4 Implementation in Scheme

Figure 6 shows an implementation of type-directed partial evaluation for `shift/reset` in Scheme. It is a straightforward translation of Figure 5 into Scheme. The main function `residualize` receives an expression and its type, and returns a result of TDPE.

## 5.5 Example

Some examples are shown in Figure 7. An auxiliary function `parse` in Figure 7 is a parser for types. It transforms, for example, a type of the form `'((A * B) / C -> D / E)` into a constructor form `(make-func (make-prod (make-base 'A) (make-base 'B)) (make-base 'C) (make-base 'D) (make-base 'E))`.

The first example in Figure 7 is TDPE of  $(\overline{\lambda} x. x)$ , and the second example is of  $\overline{\lambda} x. (\overline{\lambda} f. f) \overline{\text{@}} x$ . These two examples have the same result:

```

(lambda (x) (shift k
                (reset (let ((y (k x)))
                        y))))

```

Since the original term has type `A / B -> A / B`, we assume that it has a form  $\underline{\lambda} x_1. \underline{\mathcal{S}} k_2. M$  for some  $M$ . To prepare for static `shift` operations that may occur in the body of abstraction, a function is always residualized in this form. This is

the same for the traditional partial evaluation for `shift/reset` [1].

The final result is equivalent to  $\lambda x_1. x_1$ . Removing unnecessary `shift` operators can be done in a simple post-processing.

The third example is  $\overline{\lambda} f. \overline{f} \overline{\text{S}} k. k \overline{\text{S}} (k \overline{\text{S}} (1 + 2))$ . When we apply this function to  $f$ , the `shift` operation in the body captures a continuation  $f \overline{\text{S}} \square$  and the outer continuation when the function is applied. By naming the outer continuation  $h$ , it can be represented as  $\langle h \overline{\text{S}} (f \overline{\text{S}} \square) \rangle$  (or  $\langle \text{let } g^\diamond = f \overline{\text{S}} \square \text{ in } \langle h g^\diamond \rangle \rangle$ ). The continuation is used twice:  $k \overline{\text{S}} (k \overline{\text{S}} (1 + 2))$ . In the result, we can observe (after a few  $\eta$ -reduction) that the continuation is actually expanded twice. Notice that the original `shift` is reduced and replaced by another `shift` placed under dynamic abstraction.

The last example deals with the (typed) `printf` function [2] written with `shift/reset` (to achieve dependent behavior of `printf`). Imagine that `f` is a function that appends two strings given as a pair. Then, the expression `(reset (f (cons (% str) (f (cons " is " (% str))))))` behaves the same as `sprintf "%s is %s"` in C. Since `% str` occurs twice, the expression expects two arguments and returns a string. Thus, its type becomes `string -> string -> string`, which is represented as a type shown in Figure 7 (where `S` is regarded as `string` and `A` as an answer type).

The result of TDPE is also shown in Figure 7. Although it is somewhat complicated, all the uses of `shift` in the result is trivial: after  $\eta$ -reducing let-expressions, we see that captured continuations are all applied at the tail position. By removing all the unnecessary operations, we obtain the following:

```
(lambda (f) (lambda (x) (lambda (y)
  (reset (let ((g3 (f (cons " is " y))))
    (reset (f (cons x g3))))))))
```

The result is a function that receives necessary number of arguments and returns a string, which is the expected behavior of `sprintf "%s is %s"`.

## 5.6 A Note on Shift/reset-based let-insertion

In the previous sections, we have used the state-based let-insertion. It is well-known that let-insertion can also be implemented using `shift/reset` [10]. To use `shift/reset`-based let-insertion, we replace  $\text{wrap}(e)$  with  $\overline{\langle e \rangle}$  and  $[e]$  with  $\overline{\text{S}} k. \langle \text{let } x^\diamond = e \text{ in } k \overline{\text{S}} x^\diamond \rangle$ . We can then define call-

by-value TDPE for the  $\lambda$ -calculus.

However, if we try to introduce `shift/reset`, we encounter a problem. The context for `shift/reset` in the object language interferes with the context for let-insertion.

For example, in  $\downarrow^{t_1/\alpha \rightarrow t_2/\beta} v$ , we want to residualize the application  $k_1 \overline{\text{S}} \downarrow^{t_2} v$  at the `wrap` operation, but since  $k_2 \overline{\text{S}} \downarrow^{t_2} v$  is already enclosed by another static `reset`, the application would not be residualized at the `wrap` operation, but at the innermost static `reset`.

To use `shift/reset`-based let-insertion, it appears that we need `shift2` [4].

## 6 Conclusion

In this paper, we introduced `shift/reset` to type-directed partial evaluation, showed the implementation and demonstrated some examples.

Preliminary experimental results show that `shift` and `reset` are handled correctly in the framework. We are currently working on how to establish its correctness.

## Acknowledgment

We would like to thank Andrzej Filinski for explanation on his TLCA paper, and anonymous reviewers for constructive comments. This work is partly supported by JSPS Grant-in-Aid for Scientific Research (B) 21300005.

## References

- [1] Asai, K. “Logical Relations for Call-by-value Delimited Continuations,” *Trends in Functional Programming (TFP 2005)*, Vol. 6, pp. 63–78, Intellect (2007).
- [2] Asai, K. “On Typing Delimited Continuations: Three New Solutions to the Printf Problem,” To appear in *Higher-Order and Symbolic Computation*, 17 pages (2009).
- [3] Danvy, O. “Type-Directed Partial Evaluation,” *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 242–257 (January 1996).
- [4] Danvy, O., and A. Filinski “A Functional Abstraction of Typed Contexts,” Technical Report 89/12, DIKU, University of Copenhagen (July 1989).

```

(define-struct base (base-type))
(define-struct func (domain bef range aft)) ; domain / bef -> range / aft
(define-struct prod (type1 type2)) ; type1 * type2

(define wrap-stack '())

(define (init)
  (set! wrap-stack (cons '() wrap-stack)))

(define (wrap e)
  (let ([lst (car wrap-stack)])
    (set! wrap-stack (cdr wrap-stack))
    (wrap-gen lst e)))

(define (add e)
  (let ([g (gensym)])
    (set! wrap-stack (cons (cons (list g e) (car wrap-stack))
                          (cdr wrap-stack)))
    g))

(define (wrap-gen list body)
  (cond [(null? list) body]
        [else (wrap-gen (cdr list)
                        '(reset (let ((,(car (car list)) ,(car (cdr (car list))))) ,body)))]))

(define (residualize v t)
  (letrec
    ([reify (lambda (t v)
              (cond [(base? t) v]
                    [(func? t) (let ([t1 (func-domain t)] [t2 (func-range t)]
                                      [bef (func-bef t)] [aft (func-aft t)]
                                      [x1 (gensym)] [k1 (gensym)])
                                '(lambda (,x1)
                                  (shift ,k1 ,(begin (init) (wrap (reify aft (reset
                                                                    (let ([v1 (v (reflect t1 x1))])
                                                                    (reflect bef (add '(,k1 ,(reify t2 v1)))))))))))]
                    [(prod? t) (let ([t1 (prod-type1 t)] [t2 (prod-type2 t)])
                                '(cons ,(reify t1 (car v)) ,(reify t2 (cdr v)))]))]
        [reflect (lambda (t e)
                   (cond [(base? t) e]
                         [(func? t) (let ([t1 (func-domain t)] [t2 (func-range t)]
                                           [bef (func-bef t)] [aft (func-aft t)] [x1 (gensym)])
                                       (lambda (v1)
                                         (shift k1 (reflect aft (add '(reset
                                                                           (let ([,x1 (,e ,(reify t1 v1))])
                                                                           ,(begin (init)
                                                                                   (wrap (reify bef (k1 (reflect t2 x1)))))))))))]
                         [(prod? t) (let ([t1 (prod-type1 t)] [t2 (prod-type2 t)])
                                       (cons (reflect t1 '(car ,e)) (reflect t2 '(cdr ,e)))]))]
                   (reify t v))])
    (reify (lambda (t v)
            (cond [(base? t) v]
                  [(func? t) (let ([t1 (func-domain t)] [t2 (func-range t)]
                                    [bef (func-bef t)] [aft (func-aft t)] [x1 (gensym)])
                              (lambda (v1)
                                (shift k1 (reflect aft (add '(reset
                                                                (let ([,x1 (,e ,(reify t1 v1))])
                                                                ,(begin (init)
                                                                        (wrap (reify bef (k1 (reflect t2 x1)))))))))))]
                  [(prod? t) (let ([t1 (prod-type1 t)] [t2 (prod-type2 t)])
                              (cons (reflect t1 '(car ,e)) (reflect t2 '(cdr ,e)))]))]
            (reify t v))])
  (reify t v))

```

**Figure 6. Type-Directed Partial Evaluation for shift/reset in Scheme**

```

> (residualize (lambda (x) x) (parse '(A / B -> A / B)))
(lambda (x) (shift k
  (reset (let ((y (k x)))
    y))))

> (residualize (lambda (x) ((lambda (f) f) x)) (parse '(A / B -> A / B)))
(lambda (x) (shift k
  (reset (let ((y (k x)))
    y))))

> (residualize (lambda (f) (f (shift k (k (k (+ 1 2))))))
  (parse '((I / A -> B / I) / A -> B / I)))
(lambda (f) (shift h
  (reset (let ((g3 (reset (let ((g1 (f 3))) (reset (let ((g2 (h g1))) g2))))
    (reset (let ((g6 (reset (let ((g4 (f g3))) (reset (let ((g5 (h g4))) g5))))
      g6))))))

; (printf)
> (define (str x) x)
> (define (% to_str) (shift k (lambda (x) (k (to_str x)))))
> (residualize (lambda (f) (reset (f (cons (% str) (f (cons " is " (% str)))))))
  (parse '((S * S) / A -> S / A) / A -> (S / A -> (S / A -> S / A) / A) / A))
(lambda (f) (shift k
  (reset (let ((g9 (k
    (lambda (x) (shift k1
      (reset (let ((g8 (k1
        (lambda (y) (shift k2
          (reset (let ((g6 (reset (let ((g3 (f (cons " is " y)))
            (reset (let ((g5
              (reset (let ((g4 (f (cons x g3))) g4)))
                g5))))))
          (reset (let ((g7 (k2 g6)))
            g7))))))))))
    g8))))))
    g9))))

```

**Figure 7. Example: TDPE for shift/reset in Scheme**

- [5] Danvy, O., and A. Filinski “Abstracting Control,” *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).
- [6] Filinski, A. “Representing Monads,” *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 446–457 (January 1994).
- [7] Filinski, A. “Normalization by Evaluation for the Computational Lambda-Calculus,” In S. Abramsky, editor, *Typed Lambda Calculi and Applications (LNCS 2044)*, pp. 151–165 (May 2001).
- [8] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).
- [9] Sumii, E., and N. Kobayashi “A Hybrid Approach to Online and Offline Partial Evaluation,” *Higher-Order and Symbolic Computation*, Vol. 14, Nos. 2/3, pp. 101–142, Kluwer Academic Publishers (2001).
- [10] Thiemann, P. J. “Cogen in Six Lines,” *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP’96)*, pp. 180–189 (May 1996).