Online Partial Evaluation for Shift and Reset

Kenichi Asai
Department of Information Science, Faculty of Science, Ochanomizu University
2-1-1 Otsuka Bunkyo-ku Tokyo 112-8610 Japan
asai@is.ocha.ac.jp

ABSTRACT

This paper presents an online partial evaluator for the λ-calculus with the delimited continuation constructs *shift* and *reset*. We first give the semantics of the delimited continuation constructs in two ways: one by writing a continuation passing style (CPS) interpreter and the other by transforming them into CPS. We then combine them to obtain a partial evaluator written in CPS which produces the result in CPS. By transforming this partial evaluator back into a direct style (DS) in two steps, we obtain a DS to DS partial evaluator written in DS. The correctness of the partial evaluator is not yet formally proven. The difficulty comes from the fact that the partial evaluator is written using shift and reset. The method for reasoning about such programs is not yet established. However, the development of the partial evaluator is detailed in the paper to give a degree of confidence that it behaves as we expect.

Categories and Subject Descriptors

D.1.1 [**Software**]: Programming Techniques—*applicative (functional) languages*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages —*partial evaluation*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms

Languages, Theory

Keywords

Delimited continuations, online partial evaluation, continuation-passing style (CPS), direct style (DS), CPS transformation

PEPM '02, Jan. 14-15, 2002 Portland, OR, USA

1. INTRODUCTION

It is becoming widely recognized that the ability to manipulate *delimited continuations* [6] in (functional) programming languages matters both in principle and in practice. In contrast to the unlimited continuations provided by call-with-current-continuation (or call/cc for short) in Scheme [17], delimited continuations are only captured up to an enclosing control delimiter. The first-class use of delimited continuations provides us with strong control over abstracting and composing a part of continuations without explicitly converting the whole program into a continuation passing style (CPS).

The power of delimited continuations comes from the use of direct style (DS) function applications in CPS programs. If a program without delimited continuation constructs is transformed into CPS, all the calls become tail calls. If a program contains delimited continuation constructs, on the other hand, its CPS counterpart contains non-tail calls. By manipulating delimited continuations, we obtain additional power of DS applications in the CPS counterpart of a program.

Delimited continuations are used in various places. Apart from the standard examples such as non-local jumps and non-deterministic programming [6], they are especially useful for performing non-local program transformation such as let-insertion in partial evaluation [22, 23] and translation to the A-normal form [13]. The author has also observed that it is useful for writing a reflective interpreter [1] in DS.

Partial evaluation, on the other hand, is a program transformation technique which improves efficiency of a program by performing as much computation as possible *before* all the inputs become available [16]. It is widely used, for example, in generation of compilers from interpreters [14, 16], optimization of object-oriented languages [9], and compilation of reflective languages [18]. Because of its ability to mechanically specialize general-purpose programs into specialized, more efficient programs, partial evaluation is said to achieve both the productivity and efficiency [16].

A conventional partial evaluator has not been able to handle delimited continuations. To partially evaluate a program that manipulates continuations, we had to write it in CPS. However, as delimited continuations become popular and widely used, it is preferable that a partial evaluator can manipulate programs written using delimited continuations directly. From the practical point of view, it is reported that a program written using delimited continuation constructs sometimes runs twice as fast as its CPS transformed version [23].

This paper presents a partial evaluator for a λ-calculus with the delimited continuation constructs. We first give the semantics of the delimited continuation constructs in two ways: one by writing a CPS interpreter and the other by transforming them into CPS. We then combine them to obtain a partial evaluator written in CPS which produces the result in CPS. By transforming this partial evaluator back into DS in two steps, we obtain a DS to DS partial evaluator written in DS.

The correctness of the partial evaluator is *not* yet formally proven. The difficulty comes from the fact that the partial evaluator is written using the delimited continuation constructs. Although they can be eliminated by transforming the partial evaluator into CPS, it then requires us to prove the correctness of a partial evaluator written in CPS. One of the methods for reasoning about CPS is to make correspondence to DS [20], but this requires a theory for DS terms with the delimited continuation constructs. Thus, we need to directly reason about CPS, but it seems it is not so easy. For example, it seems we cannot use the standard induction on input terms or the length of derivation. The similar observation is reported by Duba, Harper, and MacQueen [10].

Instead, what we do in the paper is to describe the development process of the partial evaluator in detail. Although informal, the critical part is natural enough to give a degree of confidence that the partial evaluator behaves as we expect. To back it up, we have implemented it in both SML/NJ and Scheme and tested various examples.

The paper is organized as follows. The delimited continuation constructs are introduced in Sect. 2 together with examples in Scheme. After the language we consider is defined in Sect. 3, three transformers (an interpreter, a CPS transformer, and a partial evaluator) from DS to CPS written in CPS are described in Sect. 4. They are then converted to transformers from DS to DS written in CPS in Sect. 5. Finally, they are converted to transformers from DS to DS written in DS in Sect. 6. Related work is in Sect. 7 and the paper concludes in Sect. 8 with future work. In Appendix A, we present in SML/NJ an implementation of the final transformers from DS to DS written in DS.

2. BRIEF INTRODUCTION TO SHIFT AND RESET

To manipulate delimited continuations, Danvy and Filinski [6] introduced two constructs: *shift* and *reset*. Let us write shift as $\xi k.\cdots$ and reset as $\langle\cdots\rangle$. Intuitively, shift takes its current continuation up to its enclosing reset and binds it to k (and discard the current continuation). For example, in the expression $1 + \langle 10 + \xi k.\, k\,(k\,100)\rangle$, k is bound to a continuation $\lambda v.\, 10 + v$. Applying it twice to 100 yields 120, the continuation $\langle 10 + \cdot\rangle$ is discarded, 120 becomes the value of $\langle 10 + \xi k.\, k\,(k\,100)\rangle$, and the final result becomes 121. The precise semantics of shift and reset is given via CPS transformation [6], which will be described in Sect. 4.2.

2.1 Example

As a running example, we present a simplified version of a string matcher shown by Danvy and Filinski [6]. Scheme is used for the presentation of examples. Given a string (represented as a list of input symbols), the matcher checks whether it matches with a given pattern. A pattern consists of a symbol, a sequence of two patterns represented as `(& p1 p2)`, or an alternative of two patterns `(+ p1 p2)`.[1] To search for a string from all the possible alternatives, we use backtracking (or non-deterministic programming) implemented by shift and reset:

```
(define (flip) (shift c (begin (c #t) (c #f))))
(define (fail) (shift c "no"))
```

The function `flip` captures the current continuation in `c` and applies it twice to search for true and false branches. The function `fail` captures and discards the current continuation indicating that the current branch has failed. They are used as follows:

```
(define (matcher p l)
  (if (symbol? p)
    (if (and (not (null? l)) (eq? p (car l)))
      (cdr l)
      (fail))
    (let ((tag (car p)))
      (cond ((eq? tag '&)
             (matcher (caddr p))
                      (matcher (cadr p) l))
            ((eq? tag '+)
             (if (flip)
               (matcher (cadr p) l)
               (matcher (caddr p) l)))
            (else (error "unknown-pattern"))))))

(define (match? p l)
  (reset (if (null? (matcher p l))
           (begin (write "yes") (newline) "no")
           (fail))))
```

If the pattern `p` matches with the prefix of input symbols `l`, the function `matcher` returns the remainder; otherwise, it fails. Thanks to `flip` and `fail`, backtracking is realized without converting whole the program into CPS. The function `matcher` is called from the function `match?`, which prints `"yes"` if the pattern is matched; otherwise it returns `"no"`.

```
> (match? '(& (+ a b) c) '(a c))
"yes"
"no"
> (match? '(& (+ a b) c) '(a b c))
"no"
```

If the pattern matches more than one way, `"yes"` is printed many times. For example,

---

[1] If we included a repetition of a pattern `(* p)`, the pattern coincides with a regular expression. We did not include it here, however, because we then need a folding mechanism to demonstrate examples in the later section.

```
> (match? '(& (+ (& a b) a) (+ (& b c) c))
          '(a b c))
"yes"
"yes"
"no"
```

3. THE LANGUAGE

The source and target language *Term* we consider is a call-by-value λ-calculus extended with shift and reset. We refer to it as DS terms. Its syntax is defined as follows:

$$M = x \mid \underline{\lambda} x.\, M \mid M \,\underline{@}\, M \mid \underline{\xi} k.\, M \mid \underline{\langle} M \underline{\rangle}$$

We also define $\lambda \mathit{Term}$ which is *Term* without shift and reset:

$$M = x \mid \underline{\lambda} x.\, M \mid M \,\underline{@}\, M$$

We sometimes refer to it as CPS terms because it is the output of CPS transformation of DS terms. Note that CPS terms here are not restricted to tail calls. We will obtain a term with non-tail calls via CPS transformation if the transformed term contains shift and reset.

The metalanguage which is used to define transformers is a (typed) call-by-value language extended with shift and reset. The core of its syntax is defined as follows:

$$M = x \mid \lambda x.\, M \mid M\, M \mid \xi k.\, M \mid \langle M \rangle$$

We use juxtaposition without @ for application. When necessary, we will use richer constructs such as case expressions, let expressions, and datatypes. Although the syntax is close to the λ-calculus, the metalanguage can be thought of as a variant of ML. In fact, all the transformers shown in this paper (including their types) are implemented in SML/NJ. Shift and reset can be implemented in SML/NJ (using a functor parameterized by the type of final answers) as shown by Filinski [11].

4. DS TO CPS TRANSFORMERS WRITTEN IN CPS

We start with DS to CPS transformers written in CPS. We will define an interpreter, a CPS transformer, and a partial evaluator. All are written in CPS, accept terms in DS, and use CPS as their internal representation.

4.1 Interpreter $\mathcal{I}_1$

Figure 1 shows an interpreter $\mathcal{I}_1$ for *Term* written in CPS where interpreted terms are represented in CPS. It accepts a DS term, an environment ρ, and a continuation κ, and returns a value represented in CPS. Given an environment ρ, $\rho[v/x]$ is the same environment as ρ except that $\rho(x) = v$.

The first three rules constitute a standard interpreter for the λ-calculus written in CPS, and need no explanation. The rule for $\underline{\xi} k.\, M$ states that the continuation κ is captured as a function $\lambda v.\, \lambda k'.\, k'\,(\kappa\,(v))$, and bound to k for the later use. When applied, the function invokes the captured continuation κ. The execution resumes to the original continuation k', after the execution of κ finishes (if it does). Namely, the captured continuation is *composable.* The identity continuation $id = \lambda x.x$ is used for the execution of the body M of $\underline{\xi} k.\, M$. Namely, the captured continuation is discarded if it is not explicitly applied in M. The use of id is in contrast to call/cc in Scheme where the value of M is passed to κ if the captured continuation is not applied in M.

$$\begin{array}{rcl}
\mathcal{I}_1 & : & \mathit{Term} \to \mathit{Env} \to \mathit{Cont} \to \mathit{CPSValue} \\
\mathit{Env} & = & \mathit{Var} \to \mathit{CPSValue} + \mathit{Error} \\
\mathit{Cont} & = & \mathit{CPSValue} \to \mathit{CPSValue} \\
\mathit{CPSValue} & = & \mathit{CPSValue} \to \mathit{Cont} \to \mathit{CPSValue}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{I}_1\, [\![x]\!]\, \rho\, \kappa & = & \kappa\,(\rho\,(x)) \\
\mathcal{I}_1\, [\![\underline{\lambda} x.\, M]\!]\, \rho\, \kappa & = & \kappa\,(\lambda v.\, \lambda k.\, \mathcal{I}_1\, [\![M]\!]\, \rho[v/x]\, k) \\
\mathcal{I}_1\, [\![M \,\underline{@}\, N]\!]\, \rho\, \kappa & = & \mathcal{I}_1\, [\![M]\!]\, \rho\, \lambda m.\, \mathcal{I}_1\, [\![N]\!]\, \rho\, \lambda n.\, m\, n\, \kappa \\
\mathcal{I}_1\, [\![\underline{\xi} k.\, M]\!]\, \rho\, \kappa & = & \mathcal{I}_1\, [\![M]\!]\, \rho[\lambda v.\, \lambda k'.\, k'\,(\kappa\,(v))/k]\, id \\
\mathcal{I}_1\, [\![\underline{\langle} M \underline{\rangle}]\!]\, \rho\, \kappa & = & \kappa\,(\mathcal{I}_1\, [\![M]\!]\, \rho\, id)
\end{array}$$

Figure 1: DS to CPS interpreter written in CPS

The rule for $\underline{\langle} M \underline{\rangle}$ installs an identity continuation id for the evaluation of M. When it finishes, the result is applied to κ *in direct style.* Since κ is not passed to $\mathcal{I}_1$, the continuation captured by some $\underline{\xi} k.\, N$ in M is up to this enclosing $\underline{\langle} M \underline{\rangle}$, and does not include the continuation κ. Namely, the continuation is *delimited* by $\underline{\langle} M \underline{\rangle}$.

Given a DS term M and an initial continuation κ_0 of type $\mathit{CPSValue} \to \mathit{CPSValue}$, the interpreter is invoked as $\mathcal{I}_1\, [\![M]\!]\, \rho_\phi\, \kappa_0$ where ρ_ϕ is an empty environment which raises an error "unbound variable" for all variables. The initial continuation κ_0 provides M with the context (or a continuation) in which M is evaluated. In a conventional CPS interpreter, κ_0 was applied only once with the final value of M. In the presence of shift, κ_0 plays more crucial role: if shift appears without enclosing reset in M, κ_0 is captured and may be used many times. For example,

$$\mathcal{I}_1\, [\![\underline{\xi} k.\, k \,\underline{@}\, (k \,\underline{@}\, \underline{\lambda} x.\, x)]\!]\, \rho_\phi\, \kappa_0 = \kappa_0\, (\kappa_0\, \lambda x.\, \lambda k.\, k\, x)$$

The interpreter $\mathcal{I}_1$ can be thought of as providing the operational semantics of DS terms. Historically, the semantics of shift and reset was originally given via a denotational semantics using two kinds of continuations [5]. In relationship to $\mathcal{I}_1$, the original definition is essentially the CPS transformed version of $\mathcal{I}_1$ where the DS applications in $\mathcal{I}_1$ are realized by the second continuation.

4.2 CPS Transformer $\mathcal{C}_1$

Figure 2 shows a CPS transformer $\mathcal{C}_1$ written in CPS which transforms DS terms into CPS. It accepts a DS term, an environment ρ, and a continuation κ, and returns the CPS counterpart of the input term.

The first three rules constitute a CPS transformer for the λ-calculus. A variable is translated to what is stored in the environment ρ. The conventional CPS transformation does not use an environment but uses the original variable itself as the CPS transformation of a variable. Here, we allow renaming of variables using an environment to make better correspondence to the partial evaluator. A lambda abstraction is transformed to a CPS term where the bound variable x is renamed to a variable $x^\diamond$. A variable with the superscript $^\diamond$ indicates that it is a fresh variable (produced by

$$
\begin{array}{rcl}
\mathcal{C}_1 & : & \mathit{Term} \to \mathit{Env}' \to \mathit{Cont}' \to \lambda\mathit{Term} \\
\mathit{Env}' & = & \mathit{Var} \to \lambda\mathit{Term} + \mathit{Error} \\
\mathit{Cont}' & = & \lambda\mathit{Term} \to \lambda\mathit{Term}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_1\,[\![x]\!]\,\rho\,\kappa & = & \kappa\,(\rho\,(x)) \\
\mathcal{C}_1\,[\![\underline{\lambda} x.\,M]\!]\,\rho\,\kappa & = & \kappa\,(\underline{\lambda} x^\diamond.\,\underline{\lambda} k^\diamond.\,\mathcal{C}_1\,[\![M]\!]\,\rho[x^\diamond/x]\,\lambda v.\,k^\diamond\,\underline{@}\,v) \\
\mathcal{C}_1\,[\![M\,\underline{@}\,N]\!]\,\rho\,\kappa & = & \mathcal{C}_1\,[\![M]\!]\,\rho\,\lambda m.\,\mathcal{C}_1\,[\![N]\!]\,\rho \\
 & & \qquad\qquad \lambda n.\,m\,\underline{@}\,n\,\underline{@}\,\underline{\lambda} t^\diamond.\,\kappa\,(t^\diamond) \\
\mathcal{C}_1\,[\![\underline{\xi} k.\,M]\!]\,\rho\,\kappa & = & \mathcal{C}_1\,[\![M]\!]\,\rho[\underline{\lambda} a^\diamond.\,\underline{\lambda} k^\diamond.\,k^\diamond\,\underline{@}\,(\kappa\,(a^\diamond))/k]\,\mathit{id} \\
\mathcal{C}_1\,[\![\underline{\langle} M\underline{\rangle}]\!]\,\rho\,\kappa & = & \kappa\,(\mathcal{C}_1\,[\![M]\!]\,\rho\,\mathit{id})
\end{array}
$$

Figure 2: CPS transformer written in CPS

gensym).[2] A trick is used to turn a dynamic continuation $k^\diamond$ into a static continuation $\lambda v.\,k^\diamond\,\underline{@}\,v$ to make all the continuations passed to $\mathcal{C}_1$ become static. It enables one-pass CPS transformation without producing the so-called *administrative* β-redexes [7].

The transformation of an application proceeds by first transforming its function and argument. Then, their results are collected to produce an application expression. The result of the application is given a fresh name $t^\diamond$, which is passed to the rest of the transformation.

The rule for $\underline{\xi} k.\,M$ is similar to the one in the interpreter. It captures the current continuation κ and binds it to k. The only difference is that the bound value is a CPS term rather than a CPS value. Because the bound value is residualized whenever k is used, it may be duplicated or discarded if k is used more than once or is not used at all. Duplication and elimination of expressions can be avoided by the standard let-insertion technique [2]:

$$
\begin{array}{rcl}
\mathcal{C}_1\,[\![\underline{\xi} k.\,M]\!]\,\rho\,\kappa & = & \underline{\mathsf{let}}\ c^\diamond = \underline{\lambda} a^\diamond.\,\underline{\lambda} k^\diamond.\,k^\diamond\,\underline{@}\,(\kappa\,(a^\diamond)) \\
 & & \underline{\mathsf{in}}\ \mathcal{C}_1\,[\![M]\!]\,\rho[c^\diamond/k]\,\mathit{id}
\end{array}
$$

where $\underline{\mathsf{let}}\ x = M\ \underline{\mathsf{in}}\ N$ is an abbreviation for $(\underline{\lambda} x.\,N)\,\underline{@}\,M$. The captured continuation $\underline{\lambda} a^\diamond.\,\underline{\lambda} k^\diamond.\,k^\diamond\,\underline{@}\,(\kappa\,(a^\diamond))$ is residualized in a let-expression once with a unique name $c^\diamond$, and the rest of the CPS transformation is performed with this new name. The rule for $\underline{\langle} M\underline{\rangle}$ goes exactly the same way as the one in the interpreter.

Given a DS term M and an initial continuation κ_0 of type $\lambda\mathit{Term} \to \lambda\mathit{Term}$, the CPS transformer is invoked as $\mathcal{C}_1\,[\![M]\!]\,\rho_\phi\,\kappa_0$. Remember that ρ_ϕ is an empty environment which raises "unbound variable" for all variables.

The CPS transformer $\mathcal{C}_1$ shown in this section is (a variant of) the CPS transformation presented by Danvy and Filinski [6], which gives the well-known definition of shift and reset. In relationship to the interpreter $\mathcal{I}_1$, $\mathcal{C}_1$ can be thought of as the generating extension [16] of $\mathcal{I}_1$.

4.3 General Transformer $\mathcal{G}_1$

By comparing Figs. 1 and 2, we notice that they have the same structure in common. The difference is in how to build the returned value. In the interpreter, the returned value is a CPS value, whereas in the CPS transformer, it is a CPS term. To be more specific, there are four places that differ:

- the type of the returned value,
- the returned value in the rule for an abstraction,
- how an application is transformed in the rule for an application, and
- how the captured continuation is constructed.

$$
\begin{array}{rcl}
\mathcal{G}_1 & : & \mathit{Term} \to \mathit{Env}_1 \to \mathit{Cont}_1 \to \boxed{\mathsf{Type}_1} \\
\mathit{Env}_1 & = & \mathit{Var} \to \boxed{\mathsf{Type}_1} + \mathit{Error} \\
\mathit{Cont}_1 & = & \boxed{\mathsf{Type}_1} \to \boxed{\mathsf{Type}_1}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{G}_1\,[\![x]\!]\,\rho\,\kappa & = & \kappa\,(\rho\,(x)) \\
\mathcal{G}_1\,[\![\underline{\lambda} x.\,M]\!]\,\rho\,\kappa & = & \kappa\,(\boxed{\mathsf{Lam}_1[\mathcal{G}_1, x, M, \rho]}) \\
\mathcal{G}_1\,[\![M\,\underline{@}\,N]\!]\,\rho\,\kappa & = & \mathcal{G}_1\,[\![M]\!]\,\rho\,\lambda m.\,\mathcal{G}_1\,[\![N]\!]\,\rho \\
 & & \qquad\qquad \lambda n.\,\boxed{\mathsf{App}_1[m, n, \kappa]} \\
\mathcal{G}_1\,[\![\underline{\xi} k.\,M]\!]\,\rho\,\kappa & = & \mathcal{G}_1\,[\![M]\!]\,\rho[\boxed{\mathsf{Cont}_1[\kappa]}/k]\,\mathit{id} \\
\mathcal{G}_1\,[\![\underline{\langle} M\underline{\rangle}]\!]\,\rho\,\kappa & = & \kappa\,(\mathcal{G}_1\,[\![M]\!]\,\rho\,\mathit{id})
\end{array}
$$

Figure 3: General transformer written in CPS

To emphasize the common structure, let us make a general transformer which can be instantiated into both the interpreter and the CPS transformer. Figure 3 shows the general transformer. It is factorized by the four parts itemized above, which are written in the figure as the four boxes: $\boxed{\mathsf{Type}_1}$, $\boxed{\mathsf{Lam}_1[\mathcal{G}_1, x, M, \rho]}$, $\boxed{\mathsf{App}_1[m, n, \kappa]}$, and $\boxed{\mathsf{Cont}_1[\kappa]}$, respectively. In the latter three, free variables are abstracted in $[\ldots]$. We will omit writing these free variables when no confusion arises.

Given a DS term M and an initial continuation κ_0 of type $\boxed{\mathsf{Type}_1} \to \boxed{\mathsf{Type}_1}$, the general transformer is invoked as $\mathcal{G}_1\,[\![M]\!]\,\rho_\phi\,\kappa_0$.

Using the general transformer, it is simple to obtain the interpreter $\mathcal{I}_1$. We replace $\mathcal{G}_1$ in Fig. 3 with $\mathcal{I}_1$ and substitute boxed parts as follows:

$$
\mathit{CPSValue}_1 = \mathit{CPSValue}_1 \to \mathit{Cont}_1 \to \mathit{CPSValue}_1
$$

$$
\begin{array}{rcl}
\boxed{\mathsf{Type}_1} & \equiv & \mathit{CPSValue}_1 \\
\boxed{\mathsf{Lam}_1[\mathcal{I}_1, x, M, \rho]} & \equiv & \lambda v.\,\lambda k.\,\mathcal{I}_1\,[\![M]\!]\,\rho[v/x]\,k \\
\boxed{\mathsf{App}_1[m, n, \kappa]} & \equiv & m\,n\,\kappa \\
\boxed{\mathsf{Cont}_1[\kappa]} & \equiv & \lambda v.\,\lambda k'.\,k'\,(\kappa\,(v))
\end{array}
$$

Likewise, we can obtain $\mathcal{C}_1$ by replacing $\mathcal{G}_1$ with $\mathcal{C}_1$ and substituting boxed parts as follows:

$$
\begin{array}{rcl}
\boxed{\mathsf{Type}_1} & \equiv & \lambda\mathit{Term} \\
\boxed{\mathsf{Lam}_1[\mathcal{C}_1, x, M, \rho]} & \equiv & \underline{\lambda} x^\diamond.\,\underline{\lambda} k^\diamond.\,\mathcal{C}_1\,[\![M]\!]\,\rho[x^\diamond/x]\,\lambda v.\,k^\diamond\,\underline{@}\,v \\
\boxed{\mathsf{App}_1[m, n, \kappa]} & \equiv & m\,\underline{@}\,n\,\underline{@}\,\underline{\lambda} t^\diamond.\,\kappa\,(t^\diamond) \\
\boxed{\mathsf{Cont}_1[\kappa]} & \equiv & \underline{\lambda} a^\diamond.\,\underline{\lambda} k^\diamond.\,k^\diamond\,\underline{@}\,(\kappa\,(a^\diamond))
\end{array}
$$

---

[2] Gabbay and Pitts [15] proposed a framework for name generation in functional languages. We have not yet investigated how we can incorporate it here.

This factorization is interesting in its own right. The general transformer can be regarded as handling all the administrative or interpretive parts required to transform the input term, while the factorized parts represent the computation that corresponds to the computation of the input term. In the interpreter $\mathcal{I}_1$, for example, there are two kinds of values: an abstraction and a continuation. They are transformed into the CPS values shown in $\boxed{\mathsf{Lam}_1}$ and $\boxed{\mathsf{Cont}_1}$, respectively. They are then applied using the rule shown in $\boxed{\mathsf{App}_1}$. All the other parts, such as decomposing evaluation of an application into evaluation of its subterms, are required only to interpret the input term. The situation is exactly the same in the CPS transformer $\mathcal{C}_1$.

4.4 Partial Evaluator $\mathcal{P}_1$

In this section, the general transformer $\mathcal{G}_1$ is instantiated into an online partial evaluator $\mathcal{P}_1$. It is written in CPS, partially evaluates the input DS term, and produces the output in CPS. The overall method is to *pair* the interpreter $\mathcal{I}_1$ and the CPS transformer $\mathcal{C}_1$.

An online partial evaluator manipulates so-called "symbolic values" [19]. It consists of a dynamic value (namely, a program text) and a static value (if available). Let us define the type $Sval_1$ of symbolic values as follows:

$$\begin{aligned} Sval_1 &= Dynamic_1 + Static_1 \\ Dynamic_1 &= \lambda Term \\ Static_1 &= \lambda Term \times CPSValue'_1 \\ CPSValue'_1 &= Sval_1 \to Cont_1 \to Sval_1 \end{aligned}$$

(Note that the definition of $CPSValue'_1$ is different from $CPSValue_1$ in the previous section because we now handle symbolic values.) $Dynamic_1$ represents a value whose static value is unknown at partial evaluation time, whereas $Static_1$ represents a known value of type $CPSValue'_1$ together with its program text. We write a symbolic value in $Dynamic_1$ or $Static_1$ as $\mathsf{dyn}(d)$ or $\mathsf{sta}(d,\ s)$, respectively, where d and s are in $\lambda Term$ and $CPSValue'_1$, respectively. We define an operator $\mathsf{take\text{-}term}(\cdot)$ (from $Sval_1$ to $\lambda Term$) which takes the program text of a given symbolic value as follows:

$$\begin{aligned} \mathsf{take\text{-}term}(\mathsf{dyn}(d)) &= d \\ \mathsf{take\text{-}term}(\mathsf{sta}(d,\ s)) &= d \end{aligned}$$

Then, the partial evaluator $\mathcal{P}_1$ can be obtained from the general interpreter by replacing $\mathcal{G}_1$ with $\mathcal{P}_1$ and substituting boxed parts as follows:

$$\begin{aligned}
&\boxed{\mathsf{Type}_1} \quad \equiv \quad Sval_1 \\
&\boxed{\mathsf{Lam}_1[\mathcal{P}_1, x, M, \rho]} \\
&\quad \equiv \quad \mathsf{sta}(\underline{\lambda} x^\diamond .\, \underline{\lambda} k^\diamond .\, \mathsf{take\text{-}term}(\mathcal{P}_1\, [\![M]\!]\, \rho[\mathsf{dyn}(x^\diamond)/x] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda v .\, \mathsf{dyn}(k^\diamond \mathbin{\underline{@}} \mathsf{take\text{-}term}(v))), \\
&\qquad\qquad \lambda v .\, \lambda k .\, \mathcal{P}_1\, [\![M]\!]\, \rho[v/x]\, k) \\
&\boxed{\mathsf{App}_1[m, n, \kappa]} \\
&\quad \equiv \quad \mathsf{case}\ m\ \mathsf{of} \\
&\qquad\qquad \mathsf{dyn}(d) \quad : \quad \mathsf{dyn}(d \mathbin{\underline{@}} \mathsf{take\text{-}term}(n) \mathbin{\underline{@}} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \underline{\lambda} t^\diamond .\, \mathsf{take\text{-}term}(\kappa\, (\mathsf{dyn}(t^\diamond)))) \\
&\qquad\qquad \mathsf{sta}(d,\ s) \quad : \quad s\, n\, \kappa \\
&\boxed{\mathsf{Cont}_1[\kappa]} \\
&\quad \equiv \quad \mathsf{sta}(\underline{\lambda} a^\diamond .\, \underline{\lambda} k^\diamond .\, k^\diamond \mathbin{\underline{@}} \mathsf{take\text{-}term}(\kappa\, (\mathsf{dyn}(a^\diamond))), \\
&\qquad\qquad \lambda v .\, \lambda k' .\, k'\, (\kappa\, (v)))
\end{aligned}$$

Although this definition seems frightening at first sight, it is essentially the mixture of $\mathcal{I}_1$ and $\mathcal{C}_1$. The $\mathcal{I}_1$ aspect appears in s of $\mathsf{sta}(d,\ s)$ and the $\mathcal{C}_1$ aspect appears in d of $\mathsf{sta}(d,\ s)$. Let us first examine the $\mathcal{I}_1$ aspect. In $\boxed{\mathsf{Lam}_1}$, a lambda abstraction is interpreted as a static symbolic value whose static part is exactly the same as $\boxed{\mathsf{Lam}_1}$ for $\mathcal{I}_1$. The same can be said for $\boxed{\mathsf{Cont}_1}$. It is interpreted as a static symbolic value whose static part is exactly the same as $\boxed{\mathsf{Cont}_1}$ for $\mathcal{I}_1$. Thus, if m in $\boxed{\mathsf{App}_1}$ has always the form $\mathsf{sta}(d,\ s)$, then $\boxed{\mathsf{App}_1}$ is also identical to $\boxed{\mathsf{App}_1}$ in $\mathcal{I}_1$, and $\mathcal{P}_1$ behaves exactly the same as $\mathcal{I}_1$.

On the other hand, the $\mathcal{C}_1$ aspect appears in the d part of $\mathsf{sta}(d,\ s)$. In $\boxed{\mathsf{Lam}_1}$, the dynamic part behaves exactly the same way as $\boxed{\mathsf{Lam}_1}$ for $\mathcal{C}_1$. Their definition is not identical because injection ($\mathsf{dyn}(\cdot)$ and $\mathsf{sta}(\cdot,\ \cdot)$) and projection ($\mathsf{take\text{-}term}(\cdot)$) between $\lambda Term$ and $Sval_1$ are required in $\mathcal{P}_1$. If we remove them, their definition is identical. The same can be said for $\boxed{\mathsf{Cont}_1}$ and the $\mathsf{dyn}(d)$ case of $\boxed{\mathsf{App}_1}$. Thus, if m in $\boxed{\mathsf{App}_1}$ has always the form $\mathsf{dyn}(d)$, then $\mathcal{P}_1$ behaves exactly the same as $\mathcal{C}_1$.

Partial evaluation is achieved by mixing $\mathcal{I}_1$ and $\mathcal{C}_1$, namely, performing application (as in $\mathcal{I}_1$) whenever possible and residualizing the application (as in $\mathcal{C}_1$) otherwise. This behavior is exactly what $\boxed{\mathsf{App}_1}$ does. If function m to be applied is known, it is applied as in $\mathcal{I}_1$. If it is unknown, on the other hand, it is residualized as in $\mathcal{C}_1$. Thus, we can *intuitively* conclude that the result of $\mathcal{P}_1$ is the CPS transformed version of the input term where possible applications are reduced at partial evaluation time.

We say 'intuitively' here because we have not formally proven a statement like: the partial evaluator obtained by this pairing preserves the semantics of input programs. Since $\mathcal{P}_1$ is written in CPS, it is hard to prove its property. It seems we cannot use the standard induction on the input terms or the length of derivation. However, although we need more formal statement here, it is quite persuasive that mixing an interpreter and a CPS transformer yields a partial evaluator.

Given a DS term M and an initial continuation κ_0 of type $Sval_1 \to Sval_1$, the partial evaluator is invoked as $\mathsf{take\text{-}term}(\mathcal{P}_1\, [\![M]\!]\, \rho_\phi\, \kappa_0)$. The outermost $\mathsf{take\text{-}term}(\cdot)$ is required to extract the output program text from the resultant symbolic value.

Since the partial evaluator $\mathcal{P}_1$ does not perform let-insertion, it may duplicate or discard expressions. In $\mathcal{C}_1$, we noted that the continuation captured by $\xi k.\, M$ will be duplicated if k is used more than once in M. The situation is the same for $\mathcal{P}_1$, too. Furthermore, $\mathcal{P}_1$ may duplicate (or discard) lambda expressions. As for an application expression, it is not duplicated nor discarded since it is given a fresh name $t^\diamond$ and residualized exactly once.

Possible duplication/elimination of lambda abstractions and captured continuations can be *partly* avoided by inserting let-expressions as follows:

$$
\begin{aligned}
&\mathcal{P}_1\,[\![\underline{\lambda}x.\,M]\!]\,\rho\,\kappa\\
&\quad= \mathsf{dyn}(\ \underline{\mathsf{let}}\ c^\diamond = \underline{\lambda}x^\diamond.\,\underline{\lambda}k^\diamond.\,\mathsf{take\text{-}term}(\mathcal{P}_1\,[\![M]\!]\,\rho[\mathsf{dyn}(x^\diamond)/x]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\lambda v.\,\mathsf{dyn}(k^\diamond\ \underline{@}\,\mathsf{take\text{-}term}(v)))\\
&\qquad\qquad\ \underline{\mathsf{in}}\ \mathsf{take\text{-}term}(\kappa\,(\mathsf{sta}(c^\diamond,\ \lambda v.\,\lambda k.\,\mathcal{P}_1\,[\![M]\!]\,\rho[v/x]\,k))))\\
&\mathcal{P}_1\,[\![\underline{\xi}k.\,M]\!]\,\rho\,\kappa\\
&\quad= \mathsf{dyn}(\ \underline{\mathsf{let}}\ c^\diamond = \underline{\lambda}a^\diamond.\,\underline{\lambda}k^\diamond.\,k^\diamond\ \underline{@}\,\mathsf{take\text{-}term}(\kappa\,(\mathsf{dyn}(a^\diamond)))\\
&\qquad\qquad\ \underline{\mathsf{in}}\ \mathsf{take\text{-}term}(\mathcal{P}_1\,[\![M]\!]\\
&\qquad\qquad\qquad\qquad\qquad\rho[\mathsf{sta}(c^\diamond,\ \lambda v.\,\lambda k'.\,k'\,(\kappa\,(v)))/k]\,id))
\end{aligned}
$$

Although injection and projection make the let-insertion somewhat complicated, the basic idea remains the same: a dynamic expression is residualized once in a let-expression with a unique name, and the further specialization is continued with this new name.

However, this solution is unsatisfactory. Remember that $\mathcal{P}_1$ is *not* written in CPS. The rule for $\langle M\rangle$ uses a DS application. Since the two new rules for $\underline{\lambda}x.\,M$ and $\underline{\xi}k.\,M$ above always return a dynamic value (*i.e.*, a let-expression), further specialization becomes impossible if they are surrounded by reset. It seems that the shift/reset-based let-insertion [22, 23] provides the satisfactory solution here, but we have not investigated it in detail yet.

4.5 Example

We now demonstrate some example execution of $\mathcal{P}_1$. These examples are executed without let-insertion, because in these examples, it simply inserts unnecessary bindings into the results. The same results can be obtained with let-insertion and a post-processing phase which removes unnecessary bindings.

Define `go1` to execute $\mathcal{P}_1$ with the empty environment and the initial identity continuation:

```
> (define (go1 exp)
    (take-term (p1 exp empty-env id)))
```

$\mathcal{P}_1$ is a CPS transformer when no reduction is possible.

```
> (go1 '(lambda (x) x))
(lambda (x) (lambda (k) (k x)))
> (go1 '(lambda (x) (lambda (y) x)))
(lambda (x) (lambda (k)
  (k (lambda (y) (lambda (k2) (k2 x))))))
```

When shift is used, the captured continuation is expanded. Since the result is in CPS, the captured continuation is always representable.

```
> (go1 '(lambda (f x) (f (shift k (k (k x))))))
(lambda (f x) (lambda (k2)
  ((f ((f x)
       (lambda (t) (k2 t))))
   (lambda (t2) (k2 t2)))))
```

Here, the captured continuation `k` is represented as `((f ·) (lambda (t) (k2 t)))` and is expanded twice. The two administrative η-redexes (`(lambda (t) (k2 t))` and `(lambda (t2) (k2 t2))`) could be removed during partial evaluation if we used more carefully designed CPS transformation [7].

Let us now partially evaluate the string matcher `match?` with respect to a known pattern `(& (+ a b) c)`:

```
> (go1 '(lambda (l) (match? '(& (+ a b) c) l)))
(lambda (l)
  (lambda (k)
    (k (begin
        (if (null? l)
          "no"
          (if (eq? 'a (car l))
            (if (null? (cdr l))
              "no"
              (if (eq? 'c (car (cdr l)))
                (if (null? (cdr (cdr l)))
                  (begin (write "yes")
                         (newline)
                         "no")
                  "no")
                "no"))
            "no"))
        (if (null? l)
          "no"
          (if (eq? 'b (car l))
            (if (null? (cdr l))
              "no"
              (if (eq? 'c (car (cdr l)))
                (if (null? (cdr (cdr l)))
                  (begin (write "yes")
                         (newline)
                         "no")
                  "no")
                "no"))
            "no"))))))
```

We observe that the backtracking is fully expanded and the two lists `(a c)` and `(b c)` are directly searched. The search for `c` is duplicated in the result because the continuation of conditionals is (duplicated and) propagated to both the branches in the current implementation. The same result can be obtained by specializing the matcher with respect to the pattern `(+ (& a c) (& b c))`. In either case, a general string matcher written using shift and reset has been successfully partially evaluated into a specialized matcher for the specific pattern.

5. DS TO DS TRANSFORMERS WRITTEN IN CPS

The transformation presented in the previous section produced the result term in CPS. However, the structure of the general transformer $\mathcal{G}_1$ does not necessarily force the result to be in CPS. With a minor change to $\mathcal{G}_1$, we can use the general transformer to produce transformers whose result is in DS (with shift and reset). In this section, we develop DS to DS transformers written in CPS.

5.1 General Transformer $\mathcal{G}_2$

The new general transformer $\mathcal{G}_2$ is the same as $\mathcal{G}_1$ except for the rule for $\langle M\rangle$ (and the replacement of subscripts from 1 to 2). See Fig. 4.

$$\begin{array}{rcl}
\mathcal{G}_2 & : & \mathit{Term} \to \mathit{Env}_2 \to \mathit{Cont}_2 \to \boxed{\mathsf{Type}_2} \\
\mathit{Env}_2 & = & \mathit{Var} \to \boxed{\mathsf{Type}_2} + \mathit{Error} \\
\mathit{Cont}_2 & = & \boxed{\mathsf{Type}_2} \to \boxed{\mathsf{Type}_2}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{G}_2\, \llbracket x \rrbracket\, \rho\, \kappa & = & \kappa\, (\rho\, (x)) \\
\mathcal{G}_2\, \llbracket \underline{\lambda} x.\, M \rrbracket\, \rho\, \kappa & = & \kappa\, (\boxed{\mathsf{Lam}_2[\mathcal{G}_2, x, M, \rho]}) \\
\mathcal{G}_2\, \llbracket M \,\underline{@}\, N \rrbracket\, \rho\, \kappa & = & \mathcal{G}_2\, \llbracket M \rrbracket\, \rho\, \lambda m.\, \mathcal{G}_2\, \llbracket N \rrbracket\, \rho \\
 & & \qquad\qquad \lambda n.\, \boxed{\mathsf{App}_2[m, n, \kappa]} \\
\mathcal{G}_2\, \llbracket \underline{\xi} k.\, M \rrbracket\, \rho\, \kappa & = & \mathcal{G}_2\, \llbracket M \rrbracket\, \rho[\boxed{\mathsf{Cont}_2[\kappa]}/k]\, \mathit{id} \\
\mathcal{G}_2\, \llbracket \underline{\langle} M \underline{\rangle} \rrbracket\, \rho\, \kappa & = & \mathsf{let}\ v = \langle \mathcal{G}_2\, \llbracket M \rrbracket\, \rho\, \mathit{id} \rangle \\
 & & \mathsf{in}\ \kappa\, (\boxed{\mathsf{Reset}_2[v]})
\end{array}$$

Figure 4: New general transformer written in CPS

There are two points to notice. One is that $\mathcal{G}_2\, \llbracket M \rrbracket\, \rho\, \mathit{id}$ in the rule for $\mathcal{G}_2\, \llbracket \underline{\langle} M \underline{\rangle} \rrbracket\, \rho\, \kappa$ is enclosed with reset. The purpose of this reset is to make the exact correspondence between non-tail calls in the transformer and delimited contexts in the input language. (See the next section for details.) The other is that a new box $\boxed{\mathsf{Reset}_2[v]}$ is introduced. The purpose of this new box is to make residualization of reset possible.

Because shift has not been used in the transformers in the previous section, the addition of reset does not have any effects for them. Thus, we could define $\mathcal{G}_2$ as the definition of the general transformer from the beginning, and derive $\mathcal{I}_1$, $\mathcal{C}_1$, and $\mathcal{P}_1$ from it (where $\boxed{\mathsf{Reset}_2[v]}$ is substituted by v).

5.2 Interpreter $\mathcal{I}_2$

Let $\mathit{DSValue}_2$ be the type of DS values:

$$\mathit{DSValue}_2 \;=\; \mathit{DSValue}_2 \to \mathit{DSValue}_2$$

Then, an interpreter $\mathcal{I}_2$ that uses DS values (rather than CPS values) as the representation of lambda terms and continuations can be obtained from the general transformer by replacing $\mathcal{G}_2$ with $\mathcal{I}_2$ and substituting the boxed parts as follows:

$$\begin{array}{rcl}
\boxed{\mathsf{Type}_2} & \equiv & \mathit{DSValue}_2 \\
\boxed{\mathsf{Lam}_2[\mathcal{I}_2, x, M, \rho]} & \equiv & \lambda v.\, \xi\kappa.\, \mathcal{I}_2\, \llbracket M \rrbracket\, \rho[v/x]\, \kappa \\
\boxed{\mathsf{App}_2[m, n, \kappa]} & \equiv & \kappa\, (m\, n) \\
\boxed{\mathsf{Cont}_2[\kappa]} & \equiv & \lambda v.\, \langle \kappa\, v \rangle \\
\boxed{\mathsf{Reset}_2[v]} & \equiv & v
\end{array}$$

It is invoked as $\langle \mathcal{I}_2\, \llbracket M \rrbracket\, \rho_\phi\, \kappa_0 \rangle$ where κ_0 has type $\mathit{DSValue}_2 \to \mathit{DSValue}_2$. The outermost reset is required to delimit the initial context.

Intuitively, boxed parts are obtained by transforming *back* to DS [4, 8] from the CPS definition of $\mathcal{I}_1$. Since a lambda term is represented as a DS value, a shift operation is used in $\boxed{\mathsf{Lam}_2}$ to obtain the current continuation which is passed to $\mathcal{I}_2$ for the evaluation of the body M. In the same way, κ is applied in DS in $\boxed{\mathsf{App}_2}$ rather than passed to the function m. Finally, the continuation κ in $\boxed{\mathsf{Cont}_2}$ is η-expanded to enclose the continuation application with reset. This insertion of reset shows that the delimited continuation is statically scoped. Without this reset, we obtain *prompt* and *control* by Felleisen [12].

The correctness of $\mathcal{I}_2$ is established by making the exact correspondence between $\mathcal{I}_1$ and $\mathcal{I}_2$. Since the non-tail call appearing in the rule for $\langle M \rangle$ in $\mathcal{G}_1$ is enclosed by reset in $\mathcal{G}_2$ (as we did in the previous section), the only non-tail call in $\mathcal{I}_2$ is in $\boxed{\mathsf{App}_2}$. There are two cases to consider. When the applied function m is a lambda closure, we can see that $\xi k.\, \cdots$ in $\boxed{\mathsf{Lam}_2}$ always captures the continuation κ which is applied non-tail-recursively in $\boxed{\mathsf{App}_2}$ because it is the only place where non-tail call is used. This behavior is exactly the same as $\mathcal{I}_1$ where the continuation κ is passed as an extra parameter. The other case is when the applied function m is a captured continuation $\lambda v.\, \langle \kappa\, v \rangle$. In this case, the non-tail-recursively applied continuation in $\boxed{\mathsf{App}_2}$ is never captured until the evaluation of κ (in $\boxed{\mathsf{Cont}_2}$) finishes, since the application of κ is enclosed by reset. Again, this behavior is exactly the same as $\mathcal{I}_1$ where the continuation k' in $\boxed{\mathsf{Cont}_2}$ of $\mathcal{I}_1$ is never captured during the evaluation of κ since it is applied in DS. We can thus conclude that the evaluation steps of $\mathcal{I}_1$ and $\mathcal{I}_2$ are exactly the same.

5.3 DS Transformer $\mathcal{D}_2$

If we change the boxed parts of the CPS transformer $\mathcal{C}_1$ back into DS, we obtain a DS transformer $\mathcal{D}_2$. (Since it is a DS transformer, we write it using $\mathcal{D}$ rather than $\mathcal{C}$.) Semantically, it is an identity function. Syntactically, however, it performs some transformations. Its definition is as follows:

$$\begin{array}{rcl}
\boxed{\mathsf{Type}_2} & \equiv & \mathit{Term} \\
\boxed{\mathsf{Lam}_2[\mathcal{D}_2, x, M, \rho]} & \equiv & \underline{\lambda} x^\diamond.\, \underline{\xi} k^\diamond.\, \mathcal{D}_2\, \llbracket M \rrbracket\, \rho[x^\diamond/x]\, \lambda v.\, k^\diamond \,\underline{@}\, v \\
\boxed{\mathsf{App}_2[m, n, \kappa]} & \equiv & \kappa\, (m \,\underline{@}\, n) \\
\boxed{\mathsf{Cont}_2[\kappa]} & \equiv & \underline{\lambda} a^\diamond.\, \underline{\langle} \kappa\, (a^\diamond) \underline{\rangle} \\
\boxed{\mathsf{Reset}_2[v]} & \equiv & \underline{\langle} v \underline{\rangle}
\end{array}$$

It is invoked as $\mathcal{D}_2\, \llbracket M \rrbracket\, \rho_\phi\, \kappa_0$ where κ_0 has type $\mathit{Term} \to \mathit{Term}$. We do not need the outermost reset here (as in $\langle \mathcal{D}_2\, \llbracket M \rrbracket\, \rho_\phi\, \kappa_0 \rangle$) since shift is not used in $\mathcal{D}_2$.

The correctness of $\mathcal{D}_2$ can be shown by mechanically transforming the DS terms appearing in the five boxes into CPS. We will then obtain $\mathcal{C}_1$. Alternatively, we can directly observe that $\mathcal{D}_2$ is the generating extension of $\mathcal{I}_2$ *written by hand* [3].

The effect of DS transformation is twofold. First, it inserts shift for all the lambda abstractions. For example, $\underline{\lambda} x.\, x$ becomes $\underline{\lambda} x.\, \underline{\xi} k.\, k \,\underline{@}\, x$. This insertion is to prepare for the continuation capturing that may occur within the body of lambda abstractions. The second effect is to expand the captured continuation. Because of the insertion of shift for every lambda abstraction, all the captured continuations become representable and are expanded in the result. Some examples are shown in Sect. 5.5.

5.4 Partial Evaluator $\mathcal{P}_2$

The partial evaluator $\mathcal{P}_2$ that produces the result in DS can be obtained by pairing $\mathcal{I}_2$ and $\mathcal{D}_2$. Let $Sval_2$ be as follows:

$$\begin{array}{rcl} Sval_2 & = & Dynamic_2 + Static_2 \\ Dynamic_2 & = & Term \\ Static_2 & = & Term \times DSValue'_2 \\ DSValue'_2 & = & Sval_2 \to Sval_2 \end{array}$$

Then, $\mathcal{P}_2$ becomes as follows:

$$\begin{array}{l} \boxed{\mathsf{Type}_2} \quad \equiv \quad Sval_2 \\ \boxed{\mathsf{Lam}_2[\mathcal{P}_2, x, M, \rho]} \\ \quad \equiv \quad \mathsf{sta}(\underline{\lambda} x^\diamond .\, \underline{\xi} k^\diamond .\, \mathsf{take\text{-}term}(\langle \mathcal{P}_2 [\![M]\!]\, \rho[\mathsf{dyn}(x^\diamond)/x] \\ \qquad\qquad\qquad\qquad\qquad\qquad \lambda v.\, \mathsf{dyn}(k^\diamond \,\underline{@}\, \mathsf{take\text{-}term}(v))\rangle), \\ \qquad\quad \lambda v.\, \xi\kappa.\, \mathcal{P}_2 [\![M]\!]\, \rho[v/x]\, \kappa) \\ \boxed{\mathsf{App}_2[m, n, \kappa]} \\ \quad \equiv \quad \mathsf{case}\ m\ \mathsf{of} \\ \qquad\quad \mathsf{dyn}(d) \quad : \quad \kappa\, (\mathsf{dyn}(d \,\underline{@}\, \mathsf{take\text{-}term}(n))) \\ \qquad\quad \mathsf{sta}(d,\, s) \quad : \quad \kappa\, (s\, n) \\ \boxed{\mathsf{Cont}_2[\kappa]} \quad \equiv \quad \mathsf{sta}(\underline{\lambda} a^\diamond .\, \underline{\langle} \mathsf{take\text{-}term}(\kappa\, (\mathsf{dyn}(a^\diamond)))\underline{\rangle}, \\ \qquad\qquad\qquad\qquad \lambda v.\, \mathsf{add\text{-}reset}(\langle \kappa\, v\rangle)) \\ \boxed{\mathsf{Reset}_2[v]} \quad \equiv \quad \mathsf{add\text{-}reset}(v) \end{array}$$

where $\mathsf{add\text{-}reset}(\cdot)$ is defined as follows:

$$\begin{array}{rcl} \mathsf{add\text{-}reset}(\mathsf{dyn}(d)) & = & \mathsf{dyn}(\underline{\langle} d \underline{\rangle}) \\ \mathsf{add\text{-}reset}(\mathsf{sta}(d,\, s)) & = & \mathsf{sta}(\underline{\langle} d \underline{\rangle},\, s) \end{array}$$

The partial evaluator is invoked as $\mathsf{take\text{-}term}(\langle \mathcal{P}_2 [\![M]\!]\, \rho_\phi\, \kappa_0\rangle)$ where κ_0 has type $Sval_2 \to Sval_2$.

Care must be taken for the residualization of reset. Since $\mathcal{D}_2$ demands the residualization of reset in $\boxed{\mathsf{Cont}_2}$ and $\boxed{\mathsf{Reset}_2}$, the dynamic parts of $\langle \kappa\, v\rangle$ in $\boxed{\mathsf{Cont}_2}$ and v in $\boxed{\mathsf{Reset}_2}$ have to be residualized with reset. The operator $\mathsf{add\text{-}reset}(\cdot)$ is introduced for this purpose.

The correctness of $\mathcal{P}_2$ relies on the correctness of the pairing technique, and has not been formally proven.

5.5 Example

Let us now see some example execution of $\mathcal{P}_2$. In contrast to $\mathcal{P}_1$, the results are obtained in DS.

```
> (define (go2 exp)
    (take-term (reset (p2 exp empty-env id))))
> (go2 '(lambda (x) x))
(lambda (x) (shift k (k x)))
> (go2 '(lambda (x) (lambda (y) x)))
(lambda (x)
  (shift k (k (lambda (y) (shift k2 (k2 x))))))
```

Shift is inserted for all the lambda abstractions. Thanks to this insertion, all the captured continuations are representable.

```
> (go2 '(lambda (f x) (f (shift k (k (k x))))))
(lambda (f x)
  (shift k2 (reset (k2 (f (reset (k2 (f x))))))))
```

Here, `k` is bound to `(k2 (f ·))` and is expanded twice into the result.

Let us now partially evaluate the string matcher `match?` with respect to a known pattern `(& (+ a b) c)`:

```
> (go2 '(lambda (l) (match? '(& (+ a b) c) l)))
(lambda (l) (shift k (k (reset
  (begin
   (reset (if (null? l)
            "no"
            (if (eq? 'a (car l))
              (if (null? (cdr l))
                "no"
                (if (eq? 'c (car (cdr l)))
                  (if (null? (cdr (cdr l)))
                    (begin (write "yes")
                           (newline)
                           "no")
                    "no")
                  "no"))
              "no")))
   (reset (if (null? l)
            "no"
            (if (eq? 'b (car l))
              (if (null? (cdr l))
                "no"
                (if (eq? 'c (car (cdr l)))
                  (if (null? (cdr (cdr l)))
                    (begin (write "yes")
                           (newline)
                           "no")
                    "no")
                  "no"))
              "no"))))))))
```

The result is the DS version of the one obtained in Sect. 4.5. The backtracking is fully expanded and the two lists `(a c)` and `(b c)` are compiled into the result.

6. DS TO DS TRANSFORMERS WRITTEN IN DS

In this section, we develop DS to DS transformers written in DS. The basic strategy is to transform the whole transformers back into DS.

6.1 General Transformer $\mathcal{G}_3$

Figure 5 shows the general DS to DS transformer written in DS. As expected, it implements shift using shift and reset using reset. The arity of $\boxed{\mathsf{App}_3}$ is decreased to two because the transformer is in DS and no continuation is used in $\boxed{\mathsf{App}_3}$. Given a DS term M and an initial continuation κ_0 of type $\boxed{\mathsf{Type}_3} \to \boxed{\mathsf{Type}_3}$, the general transformer is invoked as $\langle \kappa_0\, (\mathcal{G}_3 [\![M]\!]\, \rho_\phi)\rangle$.

6.2 Interpreter $\mathcal{I}_3$

Let $DSValue_3$ be as follows:

$$DSValue_3 \quad = \quad DSValue_3 \to DSValue_3$$

$$
\begin{array}{rcl}
\mathcal{G}_3 & : & \mathit{Term} \to \mathit{Env}_3 \to \boxed{\mathsf{Type}_3} \\
\mathit{Env}_3 & = & \mathit{Var} \to \boxed{\mathsf{Type}_3} + \mathit{Error}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{G}_3 \llbracket x \rrbracket \rho & = & \rho\,(x) \\
\mathcal{G}_3 \llbracket \underline{\lambda} x.\, M \rrbracket \rho & = & \boxed{\mathsf{Lam}_3[\mathcal{G}_3, x, M, \rho]} \\
\mathcal{G}_3 \llbracket M \mathbin{\underline{@}} N \rrbracket \rho & = & \mathsf{let}\ \ m \ = \ \mathcal{G}_3 \llbracket M \rrbracket \rho \\
 & & \qquad\ \ n \ = \ \mathcal{G}_3 \llbracket N \rrbracket \rho \\
 & & \mathsf{in}\ \boxed{\mathsf{App}_3[m, n]} \\
\mathcal{G}_3 \llbracket \underline{\xi} k.\, M \rrbracket \rho & = & \xi\kappa.\, \mathcal{G}_3 \llbracket M \rrbracket \rho[\boxed{\mathsf{Cont}_3[\kappa]}/k] \\
\mathcal{G}_3 \llbracket \underline{\langle} M \underline{\rangle} \rrbracket \rho & = & \mathsf{let}\ v = \langle \mathcal{G}_3 \llbracket M \rrbracket \rho \rangle \\
 & & \mathsf{in}\ \boxed{\mathsf{Reset}_3[v]}
\end{array}
$$

Figure 5: General transformer written in DS

Then, the interpreter $\mathcal{I}_3$ becomes:

$$
\begin{array}{rcl}
\boxed{\mathsf{Type}_3} & \equiv & \mathit{DSValue}_3 \\
\boxed{\mathsf{Lam}_3[\mathcal{I}_3, x, M, \rho]} & \equiv & \lambda v.\, \mathcal{I}_3 \llbracket M \rrbracket \rho[v/x] \\
\boxed{\mathsf{App}_3[m, n]} & \equiv & m\, n \\
\boxed{\mathsf{Cont}_3[\kappa]} & \equiv & \lambda v.\, \langle \kappa\, v \rangle \\
\boxed{\mathsf{Reset}_3[v]} & \equiv & v
\end{array}
$$

which is invoked as $\langle \kappa_0\, (\mathcal{I}_3 \llbracket M \rrbracket \rho_\phi) \rangle$ where κ_0 has type $\mathit{DSValue}_3 \to \mathit{DSValue}_3$.

The correctness of $\mathcal{I}_3$ is established by mechanically transforming it into CPS. We will then obtain $\mathcal{I}_1$ (rather than $\mathcal{I}_2$ because CPS transformation is applied for both boxed and unboxed parts).

6.3 DS Transformer $\mathcal{D}_3$

The DS transformer $\mathcal{D}_3$ becomes:

$$
\begin{array}{rcl}
\boxed{\mathsf{Type}_3} & \equiv & \mathit{Term} \\
\boxed{\mathsf{Lam}_3[\mathcal{D}_3, x, M, \rho]} & \equiv & \underline{\lambda} x^\diamond.\, \underline{\xi} k^\diamond.\, \langle k^\diamond \mathbin{\underline{@}} (\mathcal{D}_3 \llbracket M \rrbracket \rho[x^\diamond/x]) \rangle \\
\boxed{\mathsf{App}_3[m, n]} & \equiv & m \mathbin{\underline{@}} n \\
\boxed{\mathsf{Cont}_3[\kappa]} & \equiv & \underline{\lambda} a^\diamond.\, \underline{\langle} \kappa\, (a^\diamond) \underline{\rangle} \\
\boxed{\mathsf{Reset}_3[v]} & \equiv & \underline{\langle} v \underline{\rangle}
\end{array}
$$

which is invoked as $\langle \kappa_0\, (\mathcal{D}_3 \llbracket M \rrbracket \rho_\phi) \rangle$ where κ_0 has type $\mathit{Term} \to \mathit{Term}$.

The correctness of $\mathcal{D}_3$ is established by mechanically transforming it into CPS. We will then obtain $\mathcal{D}_2$ (rather than $\mathcal{C}_1$ because CPS transformation does not transform data constructors).

6.4 Partial Evaluator $\mathcal{P}_3$

Finally, the partial evaluator $\mathcal{P}_3$ can be obtained by pairing $\mathcal{I}_3$ and $\mathcal{D}_3$. Let Sval_3 be as follows:

$$
\begin{array}{rcl}
\mathit{Sval}_3 & = & \mathit{Dynamic}_3 + \mathit{Static}_3 \\
\mathit{Dynamic}_3 & = & \mathit{Term} \\
\mathit{Static}_3 & = & \mathit{Term} \times \mathit{DSValue}'_3 \\
\mathit{DSValue}'_3 & = & \mathit{Sval}_3 \to \mathit{Sval}_3
\end{array}
$$

Then, $\mathcal{P}_3$ becomes:

$$
\begin{array}{l}
\boxed{\mathsf{Type}_3} \ \equiv \ \mathit{Sval}_3 \\
\boxed{\mathsf{Lam}_3[\mathcal{P}_3, x, M, \rho]} \\
\ \equiv \ \mathsf{sta}(\underline{\lambda} x^\diamond.\, \underline{\xi} k^\diamond.\, \mathsf{take\text{-}term}(\langle \mathsf{dyn}(k^\diamond \mathbin{\underline{@}} \mathsf{take\text{-}term}(\mathcal{P}_3 \llbracket M \rrbracket \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rho[\mathsf{dyn}(x^\diamond)/x]))\rangle), \\
\qquad\qquad \lambda v.\, \mathcal{P}_3 \llbracket M \rrbracket \rho[v/x]) \\
\boxed{\mathsf{App}_3[m, n]} \ \equiv \ \mathsf{case}\ m\ \mathsf{of} \\
\qquad\qquad\qquad \mathsf{dyn}(d) \quad : \ \mathsf{dyn}(d \mathbin{\underline{@}} \mathsf{take\text{-}term}(n)) \\
\qquad\qquad\qquad \mathsf{sta}(d,\, s) \ : \ s\, n \\
\boxed{\mathsf{Cont}_3[\kappa]} \ \equiv \ \mathsf{sta}(\underline{\lambda} a^\diamond.\, \underline{\langle} \mathsf{take\text{-}term}(\kappa\, (\mathsf{dyn}(a^\diamond))) \underline{\rangle}, \\
\qquad\qquad\qquad \lambda v.\, \mathsf{add\text{-}reset}(\langle \kappa\, v \rangle)) \\
\boxed{\mathsf{Reset}_3[v]} \ \equiv \ \mathsf{add\text{-}reset}(v)
\end{array}
$$

which is invoked as $\mathsf{take\text{-}term}(\langle \kappa_0\, (\mathcal{P}_3 \llbracket M \rrbracket \rho_\phi) \rangle)$ where κ_0 has type $\mathit{Sval}_3 \to \mathit{Sval}_3$.

Again, the correctness of $\mathcal{P}_3$ relies on the correctness of the pairing technique, and has not been formally proven.

6.5 Example

Let us now see some example execution of $\mathcal{P}_3$. Although it is written in DS, its external behavior is the same as $\mathcal{P}_2$. Thus, it produces exactly the same results as $\mathcal{P}_2$:

```
> (define (go3 exp)
    (take-term (reset (p3 exp empty-env))))
> (go3 '(lambda (x) x))
(lambda (x) (shift k (k x)))
> (go3 '(lambda (x) (lambda (y) x)))
(lambda (x)
  (shift k (k (lambda (y) (shift k2 (k2 x))))))
> (go3 '(lambda (f x) (f (shift k (k (k x))))))
(lambda (f x)
  (shift k (reset (k (f (reset (k (f x))))))))
```

$\mathcal{P}_3$ produces the same result for the string matcher, too. To avoid duplication, we show the result where the pattern is somewhat different:

```
> (go3 '(lambda (l) (match? '(& a (+ b c)) l)))
(lambda (l) (shift k (k (reset
  (if (null? l)
    "no"
    (if (eq? 'a (car l))
      (begin
        (reset (if (null? (cdr l))
                 "no"
                 (if (eq? 'b (car (cdr l)))
                   (if (null? (cdr (cdr l)))
                     (begin (write "yes")
                            (newline)
                            "no")
                     "no")
                   "no")))
        (reset (if (null? (cdr l))
                 "no"
                 (if (eq? 'c (car (cdr l)))
                   (if (null? (cdr (cdr l)))
                     (begin (write "yes")
```

```
                    (newline)
                    "no")
             "no")
           "no"))))
"no"))))))
```

As another example, we have partially evaluated $\mathcal{I}_3$ with respect to a DS term. Namely, we compiled DS terms under $\mathcal{I}_3$ semantics. The effect is the same as the DS transformation. We have compiled several DS terms and obtained exactly the same results as $\mathcal{D}_3$. Namely, the interpretive overhead of $\mathcal{I}_3$ (including shift and reset) is completely reduced away via partial evaluation.

6.6 Note on DS to CPS Transformers Written in DS

We can construct DS to CPS transformers written in DS in a similar way. However, they are not particularly interesting and are not described in this paper.

7. RELATED WORK

The only work we are aware of regarding partial evaluation of continuation manipulating constructs is done by Thiemann [21]. He presented an offline partial evaluator for Scheme including call/cc. In his framework, call/cc is statically reduced if the captured continuation and the body of call/cc are both completely static. In our framework, the treatment of shift and reset is more liberal: we always reduce them. It is possible because we carefully design the partial evaluator so that the continuation is always representable. However, it comes with price. Every lambda expression in the result now contains a shift operation. If no shift is used in the body of the lambda expression, the result will always have the form $\underline{\lambda} x.\, \underline{\xi} k.\, k \,\underline{@}\, M$ where k does not occur free in M. It can be simplified to $\underline{\lambda} x.\, M$. This observation suggests us to introduce a post-processing phase that removes such unnecessary shift operations.

8. CONCLUSION AND FUTURE WORK

This paper presented an online partial evaluator for the λ-calculus with the delimited continuation constructs *shift* and *reset*. We first constructed an interpreter and a CPS transformer for DS terms with shift and reset. We then mixed them to obtain a partial evaluator. The (manual) DS transformation yielded various partial evaluators written in DS or CPS that produce output in DS or CPS. Although the formal correctness of the mixing is not yet established, the development process is detailed to give a degree of confidence that it behaves as we expect.

As future work, we want to design an offline version of our partial evaluator. It would be able to remove unnecessary shift operations introduced in lambda expressions without performing post-processing. Also, it will enable self-application, namely, the partial evaluation of a partial evaluator written using shift and reset for let-insertion. Since one of the major applications of shift and reset is let-insertion in partial evaluation, it would be interesting to try. Currently, since our partial evaluator is online, we suffer from the well-known problem of over generality [16]. How to avoid code duplication/elimination in the partial evaluator written in DS is not yet fully investigated. Our current observation is that it would require two levels of delimited continuations [6]: one for the interpretation of shift and reset and the other for let-insertion.

As another application, we are thinking of the compilation of reflective languages written in DS. Finally, the rigorous correctness proof of the mixing technique needs to be established.

9. ACKNOWLEDGEMENTS

I would like to thank Eijiro Sumii, Olivier Danvy, and anonymous referees for many helpful comments.

10. REFERENCES

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

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

[3] Bondorf, A., and D. Dussart "Improving CPS-Based Partial Evaluation: Writing Cogen by Hand," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94)*, pp. 1–9 (June 1994).

[4] Danvy, O. "Back to Direct Style," *Science of Computer Programming*, Vol. 22, pp. 183–195, Elsevier (February 1994).

[5] Danvy, O., and A. Filinski "A Functional Abstraction of Typed Contexts," Technical Report 89/12, DIKU, University of Copenhagen (July 1989).

[6] Danvy, O., and A. Filinski "Abstracting Control," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[7] Danvy, O., and A. Filinski "Representing Control, a Study of the CPS Transformation," *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. 361–391 (December 1992).

[8] Danvy, O., and J. L. Lawall "Back to Direct Style II: First-Class Continuations," *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 299–310 (June 1992).

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

[10] Duba, B. F., R. Harper, and D. MacQueen "Typing First-Class Continuations in ML," *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pp. 163–173 (January 1991).

[11] Filinski, A. "Representing Monads," *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 446–457 (January 1994).

[12] Felleisen, M. "The Theory and Practice of First-Class Prompts," *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 180–190 (January 1988).

[13] Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen "The Essence of Compiling with Continuations," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pp. 237–247 (June 1993).

[14] Futamura, Y. "Partial evaluation of computation process – an approach to a compiler-compiler," *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, (1971), reprinted in *Higher-Order and Symbolic Computation*, Vol. 12, No. 4, pp. 381–391, Kluwer Academic Publishers (December 1999).

[15] Gabbay, M., and A. Pitts "A New Approach to Abstract Syntax Involving Binders," *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pp. 214–224 (July 1999).

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

[17] Kelsey, R., W. Clinger, and J. Rees (editors) "Revised^5 Report on the Algorithmic Language Scheme," *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, pp. 7–105, Kluwer Academic Publishers (August 1998).

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

[19] Ruf, E. *Topics in Online Partial Evaluation*, Ph.D. thesis, Stanford University (March 1993). Also published as Stanford Computer Systems Laboratory technical report CSL-TR-93-563.

[20] Sabry, A., and M. Felleisen "Reasoning about Programs in Continuation-Passing Style," *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 288–298 (June 1992).

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

[22] Thiemann, P. J. "Cogen in Six Lines," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pp. 180–189 (May 1996).

[23] Thiemann, P. "Combinators for Program Generation," *Journal of Functional Programming*, Vol. 9, No. 5, pp. 483–525, Cambridge University Press (September 1999).

APPENDIX

A. SML CODE

This appendix shows the faithful SML/NJ implementation of $\mathcal{G}_3$, $\mathcal{I}_3$, $\mathcal{D}_3$, and $\mathcal{P}_3$ presented in this paper. Define the three datatypes (corresponding to *Term*, $DSValue'_3$, and $Sval_3$) as follows:

```
datatype Term     = Var of string
                  | Lam of string * Term
                  | App of Term * Term
                  | Shift of string * Term
                  | Reset of Term

datatype DSValue3 = LamVal of DSValue3 -> DSValue3

datatype Sval3    = Dyn of Term
                  | Sta of Term * (Sval3 -> Sval3)
```

The two operators take-term(·) and add-reset(·) are then defined as follows:

```
fun take_term (Dyn (d)) = d
  | take_term (Sta (d, s)) = d

fun add_reset (Dyn (d))    = Dyn (Reset (d))
  | add_reset (Sta (d, s)) = Sta (Reset (d), s)
```

A.1 General Transformer $\mathcal{G}_3$

The general transformer $\mathcal{G}_3$ is parameterized with four boxes which takes their free variables as arguments. Shift and reset are also parameterized since they have different types for $\mathcal{I}_3$, $\mathcal{D}_3$, and $\mathcal{P}_3$.

```
fun g3_gen boxLam3 boxApp3 boxCont3 boxReset3
           shift reset =
  let
    fun g3 (Var x) r = r x
      | g3 (Lam (x, M)) r = boxLam3 g3 x M r
      | g3 (App (M, N)) r =
        let
          val m = g3 M r
          val n = g3 N r
        in
          boxApp3 m n
        end
      | g3 (Shift (k, M)) r =
        shift (fn c =>
                g3 M (extend r k (boxCont3 c)))
      | g3 (Reset (M)) r =
        let
          val v = reset (fn () => g3 M r)
        in
          boxReset3 v
        end
  in
    g3
  end
```

Definition of `extend` (which extends an environment) is found in Sect. A.5.

A.2 Interpreter $\mathcal{I}_3$

Using $\mathcal{G}_3$, $\mathcal{I}_3$ is instantiated as follows. We use Filinski's implementation of shift and reset [11].

```
structure MyControl = Control (type ans = DSValue3)
open MyControl
```

The functor `Control` has the signature:

```
type ans
val reset : (unit -> ans) -> ans
val shift : (('a -> ans) -> ans ) -> 'a
```

We instantiated `ans` to `DSValue3` here. Then, $\mathcal{I}_3$ becomes:

```
val i3 =
  let
    fun boxLam3 i3 x M r =
      LamVal (fn v => i3 M (extend r x v))
    fun boxApp3 (LamVal m) n = m n
    fun boxCont3 c =
      LamVal (fn v => reset (fn () => c v))
    fun boxReset3 v = v
  in
    g3_gen boxLam3 boxApp3 boxCont3 boxReset3
           shift reset
  end

fun go_i3 M = reset (fn () => i3 M empty_env)
```

A.3 DS Transformer $\mathcal{D}_3$

Likewise, $\mathcal{G}_3$ is instantiated into $\mathcal{D}_3$ as follows:

```
structure MyControl = Control (type ans = Term)
open MyControl

val d3 =
  let
    fun boxLam3 d3 x M r =
      let
        val x' = gensym x
        val k' = gensym "k"
      in
        Lam (x', Shift (k', reset (fn () =>
          App (Var k',
               d3 M (extend r x (Var x'))))))
      end
    fun boxApp3 m n = App (m, n)
    fun boxCont3 c =
      let
        val a' = gensym "a"
      in
        Lam (a', Reset (c (Var a')))
      end
    fun boxReset3 v = Reset (v)
  in
    g3_gen boxLam3 boxApp3 boxCont3 boxReset3
           shift reset
  end

fun go_d3 M = reset (fn () => d3 M empty_env)
```

A.4 Partial Evaluator $\mathcal{P}_3$

Finally, $\mathcal{G}_3$ is instantiated into $\mathcal{P}_3$ as follows:

```
structure MyControl = Control (type ans = Sval3)
open MyControl

val p3 =
  let
    fun boxLam3 p3 x M r =
      let
        val x' = gensym x
        val k' = gensym "k"
      in
        Sta (Lam (x', Shift (k', take_term
              (reset (fn () =>
                Dyn (App (Var k',
                          take_term (p3 M
                            (extend r x
                              (Dyn (Var x')))))))))),
             fn v => p3 M (extend r x v))
      end
    fun boxApp3 m n =
      case m of
         Dyn (d) => Dyn (App (d, take_term n))
       | Sta (d, s) => s n
    fun boxCont3 c =
      let
        val a' = gensym "a"
      in
        Sta (Lam (a', Reset (take_term
                             (c (Dyn (Var a'))))),
             fn v =>
               add_reset (reset (fn () => c v)))
      end
    fun boxReset3 v = add_reset (v)
  in
    g3_gen boxLam3 boxApp3 boxCont3 boxReset3
           shift reset
  end

fun go_p3 M =
  take_term (reset (fn () => p3 M empty_env))
```

A.5 Miscellaneous Functions

```
val gensym_counter = ref 0
fun gensym var =
  (gensym_counter := !gensym_counter + 1;
   var ^ Int.toString (!gensym_counter))

exception Unbound_variable
fun empty_env x = raise Unbound_variable
fun extend env x v y = if x=y then v else env y
```