TECHNICAL REPORT NO. 611

How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible

by

Oleg Kiselyov*

March 2005

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
BLOOMINGTON, INDIANA 47405-7104

*Collaborating with Profs. Daniel P. Friedman and Amr A. Sabry on logical programming systems

Abstract

We show how to remove a dynamic `prompt` (aka `reset`) and thus to turn so-called static delimited continuation operators (`shift`/`reset`) into dynamic ones: `control`, `control0`, `shift0`. Our technique extends the continuation captured by `shift` by composing it with the previous fragments of the 'stack'. Composition of context fragments *can* be done via regular functional composition. We thus demonstrate that all the above delimited continuation operators are *macro*-expressible in term of each other — without capturing undelimited continuations and without using mutable state. Furthermore, the operators `shift`, `control`, `control0`, `shift0` are the members of a single parameterized family, and the standard CPS is sufficient to express their denotational semantics.

We give the simplest Scheme implementation of the dynamic control operators. We give a formal simulation proof that `control` realized through `shift` indeed has its standard reduction semantics.

1 Introduction

There have been proposed a number of delimited continuation operators, which are concisely surveyed in [8]. The most well-known operators are `control`/`prompt` (which were historically the first) and `shift`/`reset`. It is of considerable interest both to the implementors and the users to establish the relationship among various control operators. In one direction, it is obvious that `shift`/`reset` is macro-expressible [5] in terms of `control`/`prompt`. Whether `control`/`prompt` and similar operators (`shift0`, `splitter`, `spawn`, `cupto`, etc.) can be macro-expressed in terms of `shift`/`reset` has been for a long time an open question [1]. The so-called static delimited continuation operators `shift`/`reset` have the well-known denotational (CPS) semantics [4]. It was unknown if `control`/`prompt` and the other, so-called dynamic delimited continuation operators, have likewise CPS semantics — or they require extensions to the standard CPS, as sometimes claimed [7].

These open questions were answered by Chung-chieh Shan [8], who for the first time showed that the dynamic delimited continuation operators are macro-expressible in terms of `shift`/`reset`. This paper presents another, more uniform solution. We outline the general method of removing a dynamic `prompt`, and so reify the operation of composing frames of control stack. Contrary to the common belief [7], we do not need any special primitives for composing stack fragments other than the operation of capturing the fragments with `shift/reset`, which is expressible in standard CPS. Furthermore, we show that both static and dynamic delimited continuation operators are the members of a single family of delimited control, parameterized by simple functions specifying the composition of stack fragments. We demonstrate the simplest known Scheme implementation of `control/prompt` and other delimited control operators in terms of `shift/reset`. Finally, we formally prove that `control` implemented in terms of `shift` indeed satisfies its standard reduction semantics. Our proof method is general and can be used for proving other properties of delimited continuations.

The next section introduces notation and defines delimited continuation operators under investigation. Section 3 formulates the theorem of macro-expressivity of `control/prompt` in terms of `shift/reset`, and gives corollaries. One of them states that the delimited continuation operators are the members of the same family. We then give the Scheme implementation of the dynamic delimited continuation operators, and tests. In Section 5, we prove the macro-expressivity theorem. The proof demonstrates the composition of captured delimited continuations, which is achievable without any new primitives. We then conclude.

2 Notation

For simplicity, we will be considering delimited continuation operators in the context of untyped call-by-value lambda-calculus. Generalizations to richer call-by-value calculi are straightforward. Table 1 presents the syntax of the language. We write $\mathrm{let}\, x = e_1 \,\mathrm{in}\, e_2$ as an abbreviation for $(\lambda x.e_2)e_1$. Table 2 gives the standard small-step operational semantics of the control-free fragment of the language, by introducing the reduction relation $e_1 \;\triangleright\; e_2$ and the evaluation context $\mathcal{M}[]$. We write $e_1 \;\triangleright^*\; e_2$ for the composition of zero or more reduction steps. Intuitively, to reduce a term e, we find its representation in the form $\mathcal{M}[e_0]$ where e_0 is a redex and $e_0 \;\triangleright\; e_1$. Then $\mathcal{M}[e_0] \;\triangleright\; \mathcal{M}[e_1]$. As usual, $e[v/x]$ means a capture-free substitution of value v for variable x in expression e.

A typical proposal [8] for treating delimited continuations as first class values defines syntactic forms to set the delimiter, and to capture and reify the continuation up to the delimiter. The delimiter setting-form is commonly

$e ::=$	Expression
$\quad v$	Value
$\mid (e_1 e_2)$	Application
$\mid D[e]$	Set a delimiter
$\mid \eta f.e$	Capture and reify
$v ::=$	Value
$\quad x,\ y,\ f,\ g$	Variables
$\mid \lambda x.e$	Abstraction

Figure 1: Syntax

$\mathcal{C}[] ::=$	(Delimited) Evaluation context
$\quad [\ ]$	Hole
$\mid (\mathcal{C}[])e \mid v(\mathcal{C}[])$	Application
$\mathcal{M}[] ::=$	Arbitrary evaluation context
$\quad \mathcal{C}[] \mid \mathcal{M}[D[\mathcal{C}[]]]$	
$(\lambda x.e)v \ \triangleright\ e[v/x]$	(β-) Redex

Figure 2: Operational Semantics

called "prompt" or "reset". We will write $D[...]$ for a general prompt, and **prompt** ..., **reset** ... etc. for the delimiters of specific delimited continuation proposals. The capturing and reifying form is commonly called "shift" or "control". Likewise, we will use the notation $\eta f.e$ for a generic form and **shift** $f.e$, **control** $f.e$ etc. for a specific delimited continuation operator.

To account for delimited continuation operators, we augment the operational semantics in Table 2. The so-called "static" delimited continuation operators `shift/reset` are defined by the following reduction rules.

$$\mathcal{M}[\textbf{reset}\ v] \ \triangleright\ \mathcal{M}[v] \tag{S1}$$

$$\begin{array}{c} \mathcal{M}[\textbf{reset}\ \mathcal{C}[\textbf{shift}\ f.e]] \ \triangleright\ \mathcal{M}[\textbf{reset}\ e'] \\ \text{where } e' \ = \ \ \text{let}\, f = \lambda x.\textbf{reset}\ \mathcal{C}[x] \text{ in}\, e \end{array} \tag{S2}$$

Here, $\mathcal{C}[]$ is the context that does not cross the delimiter's (i.e., `reset`) boundary. In contrast, the context $\mathcal{M}[]$ may do so.

All these reduction rules do not handle the case when $\eta f.e$ occurs without a dynamically enclosing $D[...]$. We shall call a program that exhibits such a case to be "stuck on control." We could have followed the original Danvy and Filinski's proposal of assuming that the entire program is enclosed in a top-level $D[...]$. However, we chose to consider stuck on control expressions to be in error — as do several implementations of delimited continuations, e.g., Scheme48.

The most well-known of so-called "dynamic" delimited continuation operators (and historically the first delimited continuation operators proposed) are `control`/`prompt`:

$$\mathcal{M}[\textbf{prompt}\ v] \ \rhd\ \mathcal{M}[v] \tag{C1}$$

$$\begin{array}{c}\mathcal{M}[\textbf{prompt}\ \mathcal{C}[\textbf{control}\ f.e]] \ \rhd\ \mathcal{M}[\textbf{prompt}\ e'] \\ \text{where } e' = \ \text{let}\, f = \lambda x.\mathcal{C}[x] \text{ in}\, e\end{array} \tag{C2}$$

All various delimited continuation operators $\eta f.e$ capture the continuation up to the closest dynamically enclosing delimiter, and reify that continuation as a function. That function, bound to the variable f in the body e may be invoked in the body zero or more times. The reified delimited continuation may also be incorporated in the result of e, and invoked elsewhere in the program many times. The difference among the delimited continuation operators emerges when either the body e or the captured continuation themselves contain delimited continuation operators. The operator `shift` encloses the captured continuation in a delimiter, whereas `control` does not. Therefore, in the case of `shift`, the delimited continuation operators in $\mathcal{C}[]$ can capture only a part of that context, but no more. In contrast, in the case of `control`, delimited continuation operators embedded in $\mathcal{C}[]$ can capture continuations that span far beyond that context. Other such dynamic delimited continuation operators are possible, whose reduction rules eliminate even more delimiters:

$$\begin{array}{c}\mathcal{M}[\textbf{reset}_\textbf{0}\ \mathcal{C}[\textbf{shift}_\textbf{0}\ f.e]] \ \rhd\ \mathcal{M}[e'] \\ \text{where } e' = \ \text{let}\, f = \lambda x.\textbf{reset}_\textbf{0}\ \mathcal{C}[x] \text{ in}\, e\end{array} \tag{S2'}$$

$$\begin{array}{c}\mathcal{M}[\textbf{prompt}_\textbf{0}\ \mathcal{C}[\textbf{control}_\textbf{0}\ f.e]] \ \rhd\ \mathcal{M}[e'] \\ \text{where } e' = \ \text{let}\, f = \lambda x.\mathcal{C}[x] \text{ in}\, e\end{array} \tag{C2'}$$

Both these operators do not preserve the original delimiter, permitting therefore the body of the delimited continuation operator to capture a part of $\mathcal{M}[]$. The operator `shift0` was briefly considered in [3], whereas `control0` is equivalent to a single-prompt `cupto` [1].

3 Removing a dynamic `prompt` and expressing `control` via `shift`

We introduce a sum (discriminated union) datatype `H`, similar to Haskell's `Either` datatype:

$\mathsf{H} ::= \quad \mathsf{H}\, v_1 v_2 \,|\, \mathsf{HV}\, v$

Such a datatype is trivially realized in our calculus (cf., one of the Scheme implementations below): constructors H and HV become regular functions that yield abstractions. Slightly informally, we will call $\mathsf{H}\, v_1 v_2$ and $\mathsf{HV}\, v$ values themselves.

We introduce the following source-to-source transformation:

$$\begin{array}{l} \textbf{prompt}\; e \Longrightarrow h(\textbf{reset}\; \mathsf{HV}\, e) \\ \textbf{control}\; f.e \Longrightarrow \textbf{shift}\; f'.\mathsf{H}\,(h' \cdot f')(\lambda f.e) \end{array} \qquad \text{(PR)}$$

where the functions h and h' are defined as follows:

$$\begin{array}{rcl} h(\mathsf{H}\, f x) &=& \textbf{prompt}\; x f \\ h(\mathsf{HV}\, x) &=& x \end{array} \qquad \text{(H)}$$

$$\begin{array}{rcl} h'(\mathsf{H}\, f x) &=& \textbf{shift}\; g.\mathsf{H}\,(h' \cdot g \cdot f)x \\ h'(\mathsf{HV}\, x) &=& x \end{array} \qquad \text{(H')}$$

We also introduce the following transformation rule:

$$\mathcal{M}[\mathcal{C}[e]] \Longrightarrow \mathcal{M}[h'(\textbf{reset}\; \mathsf{HV}\,(\mathcal{C}[e]))] \equiv \mathcal{M}[\mathcal{B}[\mathcal{C}[e]]] \qquad \text{(PR1)}$$

The notation $\mathcal{B}[e]$ is a mere shorthand for $h'(\textbf{reset}\; \mathsf{HV}\, e)$. The context $\mathcal{C}[]$ in (PR1) may be empty.

It is straightforward to extend (PR) to expressions and contexts. We will write $e \stackrel{n}{\rightharpoonup} \overline{e}$ for the transformation of an expression e yielding $\overline{e}$ by applying (PR) and n-times the rule (PR1). When we apply the rule (PR1), the place to insert $\mathcal{B}[]$ is chosen non-deterministically. If the number of applications of (PR1) is irrelevant, we may write simply $e \rightharpoonup \overline{e}$. The properties of the transformation are summarized by the following Theorem.

Theorem 1. `Prompt` and `control` are macro-expressible in terms of `shift` and `reset`: applying the transformation (PR) to a program yields an observationally equivalent program. In particular, the transformations (PR) and (PR1) have the following properties:

$$\overline{v} \text{ is a value} \qquad \text{(T1)}$$

That is, values are transformed to values.

$$\mathcal{M}[e] \rightharpoonup \overline{\mathcal{M}}[\overline{e}] \tag{T2}$$

Transformations (PR) and (PR1) are context independent. This is another statement of macro-expressibility.

$$\text{if } e_1 \triangleright e_2 \text{ and } e_1 \overset{n}{\rightharpoonup} \overline{e}_1 \text{ then } \exists m.\, e_2 \overset{m}{\rightharpoonup} \overline{e}_2 \text{ and } \overline{e}_1 \triangleright^* \overline{e}_2 \tag{T3}$$

That is, the transformed code simulates reductions. In particular:

$$\text{if } e \triangleright^* v \text{ then } e \overset{0}{\rightharpoonup} \overline{e} \text{ and } \overline{e} \triangleright^* \overline{v} \tag{T4}$$

and if reductions of e diverge, so do reductions of $\overline{e}$.

3.1 One family of delimited continuation operators

Corollary 1. Both static and dynamic delimited continuation operators are the members of a single family $\mathbf{reset}_{\mathrm{hr}}\ e$ and $\mathbf{shift}_{\mathrm{hs}}\ f.e$ parameterized by functions `hr` and `hs`:

$$\begin{array}{ll} \mathbf{reset}\ e = \mathbf{reset}_{\mathrm{hr_{stop}}}\ e & \mathbf{prompt}\ e = \mathbf{reset}_{\mathrm{hr_{stop}}}\ e \\ \mathbf{shift}\ f.e = \mathbf{shift}_{\mathrm{hs_{stop}}}\ f.e & \mathbf{control}\ f.e = \mathbf{shift}_{\mathrm{hs_{prop}}}\ f.e \end{array}$$

$$\begin{array}{ll} \mathbf{reset_0}\ e = \mathbf{reset}_{\mathrm{hr_{prop}}}\ e & \mathbf{prompt_0}\ e = \mathbf{reset}_{\mathrm{hr_{prop}}}\ e \\ \mathbf{shift_0}\ f.e = \mathbf{shift}_{\mathrm{hs_{stop}}}\ f.e & \mathbf{control_0}\ f.e = \mathbf{shift}_{\mathrm{hs_{prop}}}\ f.e \end{array}$$

where $\mathbf{reset}_{\mathrm{hr}}\ e$ and $\mathbf{shift}_{\mathrm{hs}}\ f.e$ and the functions hr_{stop}, hs_{stop}, hr_{prop}, and hs_{prop} are themselves macro-expressible in terms of `shift` and `reset`:

$$\begin{array}{l} \mathbf{reset}_{\mathrm{hr}}\ e \Longrightarrow hr(\mathbf{reset}\ \mathsf{HV}\, e) \\ \mathbf{shift}_{\mathrm{hs}}\ f.e \Longrightarrow \mathbf{shift}\ f'.\mathsf{H}\,(hs \cdot f')(\lambda f.e) \end{array}$$

$$\begin{array}{rcl} hr_{stop}(\mathsf{H}\, f x) & = & \mathbf{reset}_{\mathrm{hr_{stop}}}\ x f \\ hr_{stop}(\mathsf{HV}\, x) & = & x \\ hs_{stop} & = & hr_{stop} \end{array}$$

$$\begin{array}{rcl} hr_{prop}(\mathsf{H}\, f x) & = & x f \\ hr_{prop}(\mathsf{HV}\, x) & = & x \end{array}$$

$$
\begin{aligned}
hs_{prop}(\mathsf{H}\, f x) &= \ \textbf{shift}\ g.\mathsf{H}\,(hs_{prop}\ \cdot\ g\ \cdot\ f)x \\
hs_{prop}(\mathsf{HV}\, x) &= \ x
\end{aligned}
$$

3.2 Fischer-style CPS transform of dynamic delimited continuation operators

Corollary 2. Dynamic delimited continuation operators `control`/`prompt` and the others have a simple denotational semantics and a Fischer-style [6] CPS transform, induced by the CPS transform of `shift`/`reset`.

4 Scheme implementation of the dynamic delimited continuation operators

We shall treat the sum datatype `H` as an abstract datatype with two constructors `H` and `HV` and a deconstructor `case-H`. The datatype can be realized by the following pure lambda-terms in our calculus:

```
; Constructors
(define (H a b)
  (lambda (on-h)
    (lambda (on-hv)
      ((on-h a) b))))

(define (HV v)
  (lambda (on-h)
    (lambda (on-hv)
      (on-hv v))))

; Deconstructor
(define-syntax case-H
  (syntax-rules ()
    ((case-H e
       ((f x) on-h)
       (v on-hv))
     ((e (lambda (f) (lambda (x) on-h)))
       (lambda (v) on-hv)))))
```

Alternatively, we may, potentially more efficiently, implement the `H` datatype via Scheme's indiscriminated union:

```
(define H-tag (list 'H-tag))

; Constructors
(define (H a b) (cons H-tag (cons a b)))
(define-syntax HV                            ; just the identity
  (syntax-rules ()
    ((HV v) v)))

; Deconstructor
(define-syntax case-H
  (syntax-rules ()
    ((case-H e
       ((f x) on-h)
       (v on-hv))
     (let ((val e))
       (if (and (pair? val) (eq? (car val) H-tag))
         (let ((f (cadr val)) (x (cddr val))) on-h)
         (let ((v val)) on-hv))))))
```

The family of the delimited continuation operators is implemented by the straightforward transcription into Scheme of the expressions in Corollary 1.

```
(define-syntax greset
  (syntax-rules ()
    ((greset hr e) (hr (reset (HV e))))))

(define-syntax gshift
  (syntax-rules ()
    ((gshift hs f e)
      (shift f* (H (lambda (x) (hs (f* x))) (lambda (f) e))))))

(define (hr-stop v)
  (case-H v
    ; on-h
    ((f x) (greset hr-stop (x f)))
    (v v))) ; on-hv
```

```
(define hs-stop hr-stop)

(define (hr-prop v)
  (case-H v
    ; on-h
    ((f x) (x f))
    ; on-hv
    (v v)))


(define (hs-prop v)
  (case-H v
    ; on-h
    ((f x)
      (shift g
        (H (lambda (y) (hs-prop (g (f y)))) x)))
    ; on-hv
    (v v)))
```

Using the operational semantics of `shift`/`reset`, it is easy to see that $\mathbf{reset}_{\mathrm{hr}_{\mathrm{stop}}}\ e$ is $\mathbf{reset}\ e$, and $\mathbf{shift}_{\mathrm{hs}_{\mathrm{stop}}}\ f.e$ is itself $\mathbf{shift}\ f.e$.

The operators `prompt` and `control`, in particular, have the following simple implementation:

```
(define-syntax prompt
  (syntax-rules ()
    ((prompt e) (greset hr-stop e))))

(define-syntax control
  (syntax-rules ()
    ((control f e) (gshift hs-prop f e))))
```

It is patently clear then that `control` and `prompt` are indeed *macro*-expressible in terms of `shift` and `reset`.

4.1 Tests

The tests run on Scheme48, with its built-in implementation of `shift` and `reset`.

```
(reset (let ((x (shift f (cons 'a (f '()))))) (shift g x)))
; ==> '(a)
(prompt (let ((x (control f (cons 'a (f '()))))) (control g x)))
; ==> '()

(prompt ((lambda (x) (control l 2)) (control l (+ 1 (l 0)))))
; ==> 2
(prompt (control f (cons 'a (f '()))))
; ==> '(a)
(prompt (let ((x (control f (cons 'a (f '())))))
          (control g (g x))))
; ==> '(a)
(prompt0 (prompt0 (let ((x (control0 f (cons 'a (f '())))))
                     (control0 g x))))
; ==> '()
; prompt0 is the same as reset0
(prompt0 (cons 'a (prompt0 (shift0 f (shift0 g '())))))
; ==> '()
(prompt0 (cons 'a (prompt0 (prompt0 (shift0 f (shift0 g '()))))))
; ==> '(a)

; Example by Chung-chieh Shan
(prompt (begin (display (control f (begin (f 1) (f 2))))
               (display (control f (begin (f 3) (f 4))))))
; ==> 132342344234442344442344444234444442344444442344444444234....
```

5 Proof of Theorem 1

In this section we formally prove that `control`/`prompt` operators realized in terms of `shift`/`reset`, Theorem 1, satisfy their standard reduction semantics, eqs. (C1-C2). To be precise, we prove that reductions in the translated language (`control`/`prompt` expressed via `shift`/`reset`) simulate reductions in the original language. We must stress that our proof elucidates how the context fragment compositions (implicit in the original language with `control`/`prompt`) become *explicit*, as functional compositions, in the translated language.

Properties (T1) and (T2) are obvious. Before we prove the simulation property (T3), we consider the following Lemma.

Lemma 1.

$$\mathcal{B}[v] \ \triangleright^* \ v$$

The proof is straightforward from the definition of $\mathcal{B}[e]$ as $h'(\textbf{reset}\ \mathsf{HV}\, e)$, the fact that $\mathsf{HV}\,v$ is also a value, the definition of h' in eq. (H'), and eq. (S1). The Lemma is easily generalized to arbitrary sequences of $\mathcal{B}[]$.

The notation $e_1 \ \triangleright\ e_2$ describes three different reductions: the β-reduction in Table 2 and control reductions (C1) and (C2). We shall consider all three reductions in turn.

In order for the β-reduction to apply, the expression e_1 must have the form $\mathcal{M}[(\lambda x.e_1')v]$. Thus we have:

$$\begin{array}{l} e_1 \equiv \mathcal{M}[(\lambda x.e_1')v] \\ \overset{n}{\rightharpoonup} \overline{\mathcal{M}}[(\mathcal{B}[\lambda x.\overline{e}_1'])(\mathcal{B}[\overline{v}])] \\ \{\text{ Lemma 1 }\} \\ \triangleright^* \overline{\mathcal{M}}[(\lambda x.\overline{e}_1')\overline{v}] \\ \triangleright\ \overline{\mathcal{M}}[\overline{e}_1'[\overline{v}/x]] \end{array}$$

On the other hand,

$$\begin{array}{l} e_1 \equiv \mathcal{M}[(\lambda x.e_1')v] \\ \triangleright\ \mathcal{M}[e_1'[v/x]] \\ \overset{m}{\rightharpoonup} \overline{\mathcal{M}}[\overline{e}_1'[\overline{v}/x]] \end{array}$$

for some m.

Another possible reduction is (C1). It applies when the expression in question has the form $\mathcal{M}[\textbf{prompt}\ v]$. Therefore,

$$\begin{array}{l} e_1 \equiv \mathcal{M}[\textbf{prompt}\ v] \\ \overset{n}{\rightharpoonup} \overline{\mathcal{M}}[h(\textbf{reset}\ \mathsf{HV}\,(\mathcal{B}[\overline{v}]))] \\ \{\text{ Lemma 1 }\} \\ \triangleright^* \overline{\mathcal{M}}[h(\textbf{reset}\ \mathsf{HV}\,\overline{v})] \\ \{\text{ equations (S1) and (H) }\} \\ \triangleright^* \overline{\mathcal{M}}[\overline{v}] \end{array}$$

The latter expression is obviously a transform image from $\mathcal{M}[v]$, which is the result of one-step reduction of e_1.

Lemma 2.

$$\begin{array}{l} \mathcal{B}[\mathcal{C}[\textbf{shift}\ g.\mathsf{H}\,(h' \cdot g \cdot f')(\lambda f.e)]] \ \triangleright^* \\ \textbf{shift}\ g.\mathsf{H}\,(\lambda x.h'(g(\mathcal{B}[\mathcal{C}[f'x]])))(\lambda f.e) \end{array}$$

Informally, the translation image of **control** $f.e$ can be lifted out of $\mathcal{B}[\mathcal{C}[]]$ with the latter context fragment added to the captured continuation. Indeed,

$$
\begin{array}{l}
\mathcal{B}[\mathcal{C}[\mathbf{shift}\; g.\mathsf{H}\,(h' \cdot g \cdot f')(\lambda f.e)]] \\
\{\text{ Definition of } \mathcal{B}[]\ \} \\
\equiv\; h'(\mathbf{reset}\; \mathsf{HV}\,(\mathcal{C}[\mathbf{shift}\; g.\mathsf{H}\,(h' \cdot g \cdot f')(\lambda f.e)])) \\
\{\text{ (S2), along with the fact } \mathsf{HV}\,e \text{ may be subsumed }\} \\
\{\text{ as a part of a non-reset-crossing context } \mathcal{C}[]\ \} \\
\rhd\; h'(\mathbf{reset}\; \mathrm{let}\, g = \lambda x.\mathbf{reset}\; \mathsf{HV}\,(\mathcal{C}[x])\, \mathrm{in}\, \mathsf{H}\,(h' \cdot g \cdot f')(\lambda f.e)) \\
\{\ \mathsf{H}\,ab \text{ is a value, apply (S1) and then (H') }\} \\
\rhd^*\; \mathrm{let}\, g = \lambda x.\mathbf{reset}\; \mathsf{HV}\,(\mathcal{C}[x])\, \mathrm{in}\, \mathbf{shift}\; g'.\mathsf{H}\,(h' \cdot g' \cdot h' \cdot g \cdot f')(\lambda f.e) \\
\equiv\; \mathbf{shift}\; g'.\mathsf{H}\,(\lambda x.h'(g'(h'(\mathbf{reset}\; \mathsf{HV}\,(\mathcal{C}[f'x])))))(\lambda f.e) \\
\equiv\; \mathbf{shift}\; g'.\mathsf{H}\,(\lambda x.h'(g'(\mathcal{B}[\mathcal{C}[f'x]])))(\lambda f.e)
\end{array}
$$

The last reduction to consider, (C2), applies when the expression e_1 has the form of $\mathcal{M}[\mathbf{prompt}\; \mathcal{C}[\mathbf{control}\; f.e]]$. When we transform that expression, we have to keep in mind that the transform of a non–delimiter-crossing context $\mathcal{C}[]$ may, in general, be a delimiter-crossing context: a potentially inserted $\mathcal{B}[]$ contains the delimiter, `reset`. We first consider the case, however, when the image of the context $\mathcal{C}[]$ is a non–delimiter-crossing context $\overline{\mathcal{C}}[]$. We can then write:

$$
\begin{array}{ll}
e_1 \;\equiv\; \mathcal{M}[\mathbf{prompt}\; \mathcal{C}[\mathbf{control}\; f.e]] & \\
\overset{n}{\rightharpoondown}\; \overline{\mathcal{M}}\big[h(\mathbf{reset}\; \mathsf{HV}\,(\overline{\mathcal{C}}[\mathbf{shift}\; f'.\mathsf{H}\,(h' \cdot f')(\lambda f.\overline{e})]))\big] & \\
\{\text{ (S2), along with the fact } \mathsf{HV}\,e \text{ may be subsumed }\} & \\
\{\text{ as a part of a non-reset-crossing context } \overline{\mathcal{C}}[]\ \} & \\
\rhd\; \overline{\mathcal{M}}\big[h(\mathbf{reset}\; \mathrm{let}\, f' = \lambda x.\mathbf{reset}\; \mathsf{HV}\,(\overline{\mathcal{C}}[x])\, \mathrm{in}\, \mathsf{H}\,(h' \cdot f')(\lambda f.\overline{e}))\big] & (*) \\
\{\ \mathsf{H}\,ab \text{ is a value, apply (S1) and then (H) }\} & \\
\rhd^*\; \overline{\mathcal{M}}\big[h(\mathbf{reset}\; \mathsf{HV}\,(\mathrm{let}\, f' = \lambda x.\mathbf{reset}\; \mathsf{HV}\,(\overline{\mathcal{C}}[x])\, \mathrm{in}\, (\lambda f.\overline{e})(h' \cdot f')))\big] & \\
\equiv\; \overline{\mathcal{M}}\big[h(\mathbf{reset}\; \mathsf{HV}\,(\mathrm{let}\, f = \lambda x.h'(\mathbf{reset}\; \mathsf{HV}\,(\overline{\mathcal{C}}[x]))\, \mathrm{in}\, \overline{e}))\big] & \\
\{\text{ definition of } \mathcal{B}[]\ \} & \\
\equiv\; \overline{\mathcal{M}}\big[h(\mathbf{reset}\; \mathsf{HV}\,(\mathrm{let}\, f = \lambda x.\mathcal{B}\big[\overline{\mathcal{C}}[x]\big]\;\mathrm{in}\, \overline{e}))\big] &
\end{array}
$$

On the other hand,

$$
\begin{array}{l}
e_1 \;\equiv\; \mathcal{M}[\mathbf{prompt}\; \mathcal{C}[\mathbf{control}\; f.e]] \\
\{\text{ apply (C2) }\} \\
\rhd\; \mathcal{M}[\mathbf{prompt}\; \mathrm{let}\, f = \lambda x.\mathcal{C}[x]\;\mathrm{in}\, e] \\
\overset{m}{\rightharpoondown}\; \overline{\mathcal{M}}\big[h(\mathbf{reset}\; \mathsf{HV}\,(\mathrm{let}\, f = \lambda x.\mathcal{B}\big[\overline{\mathcal{C}}[x]\big]\;\mathrm{in}\, \overline{e}))\big]
\end{array}
$$

for some m — actually, for $m = n+1$.

Finally, we consider the general case when the image of the context $\mathcal{C}[]$ does include $\mathcal{B}[]$. We can always decompose $\mathcal{C}[]$ into $\mathcal{C}_2[\mathcal{C}_1[]]$ so that

$$\mathcal{C}[] \equiv \mathcal{C}_2[\mathcal{C}_1[]] \overset{n}{\rightharpoondown} \overline{\mathcal{M}}_2\big[\mathcal{B}\big[\overline{\mathcal{C}}_1[]\big]\big] \qquad \text{(D)}$$

where either of the contexts $\mathcal{C}_1[]$ or $\mathcal{C}_2[]$ may be empty and $\mathcal{C}_2[] \overset{m}{\rightharpoondown} \overline{\mathcal{M}}_2[]$ for some m smaller than n. This equation merely expresses the fact of the insertion of $\mathcal{B}[]$. It follows then:

$$\begin{array}{l} e_1 \equiv \mathcal{M}[\textbf{prompt}\ \mathcal{C}[\textbf{control}\ f.e]] \\ \overset{n}{\rightharpoondown} \overline{\mathcal{M}}\big[h(\textbf{reset}\ \mathsf{HV}\,(\overline{\mathcal{M}}_2\big[\mathcal{B}\big[\overline{\mathcal{C}}_1[\textbf{shift}\ f'.\mathsf{H}\,(h' \cdot f')(\lambda f.\overline{e})]\big]\big]))\big] \\ \{\ \text{Lemma 2}\ \} \\ \rhd^* \overline{\mathcal{M}}\big[h(\textbf{reset}\ \mathsf{HV}\,(\overline{\mathcal{M}}_2\big[\textbf{shift}\ f'.\mathsf{H}\,(\lambda x.h'(f'(\mathcal{B}\big[\overline{\mathcal{C}}_1[x]\big])))(\lambda f.\overline{e})\big]))\big] \end{array} \qquad (**)$$

If the context $\overline{\mathcal{M}}_2[]$ does not cross the delimiter boundary — that is, contains no $\mathcal{B}[]$, we can denote it as $\overline{\mathcal{C}}_2[]$ and apply (*), obtaining

$$\overline{\mathcal{M}}\big[h(\textbf{reset}\ \mathsf{HV}\,(\text{let}\ f = \lambda x.\mathcal{B}\big[\overline{\mathcal{C}}_2\big[\mathcal{B}\big[\overline{\mathcal{C}}_1[x]\big]\big]\big]\ \text{in}\ \overline{e}))\big]$$

which is again the image of

$$\mathcal{M}[\textbf{prompt}\ \text{let}\ f = \lambda x.\mathcal{C}[x]\ \text{in}\ e]$$

keeping in mind that $\mathcal{C}[e] \equiv \mathcal{C}_2[\mathcal{C}_1[e]]$ by (D). If, however, $\overline{\mathcal{M}}_2[]$ did contain $\mathcal{B}[]$, we can apply (D) and (**) one more time. We recall that $\mathcal{C}_2[] \overset{m}{\rightharpoondown} \overline{\mathcal{M}}_2[]$ with m strictly less than n, thus our procedure must terminate.

Property (T4) is a simple consequence of (T3) and the fact $v \overset{n}{\rightharpoondown} e \rhd^* \overline{v}$, which follows from Lemma 1. We also observe that the transform of a stuck-on-control expression is itself stuck on control.

6 Conclusions

We have introduced a general method of removing a dynamic prompt, so that delimited continuation operators may capture continuations beyond that prompt. Contrary to the previously held beliefs, composing of contexts can be done by the regular functional composition. No primitives other than the operation of capturing fragments of the context with `shift`/`reset` are needed. Therefore, the standard CPS is sufficient for giving denotational semantics of the dynamic delimited continuations, and the dynamic delimited continuation operators are just as expressible as `shift`/`reset`. We thus confirm the results of [8] using a more general framework and giving the formal proof.

Our technique of removing a dynamic prompt is rather general and thus can be used for the design of many other delimited control operators, which permit capturing the continuation up to an arbitrary (not necessarily the closest one) enclosing delimiter. It becomes entirely up to the user which delimiters to skip and how the captured frames are to be composed or used separately. The essence of our technique is reifying the control effect (or the absence of it) into regular values, instances of the datatype `H`. Therefore, the user program has the ability to determine which control effects, if any, had occurred during the evaluation of a piece of code — and act accordingly. Our technique is thus reminiscent of "referring to the central authority" approach by Cartwright and Felleisen [2]. However, instead of one central authority, we have bureaucracy — a sequence of authorities. A particular authority may either fulfill a request for service, or pass it, after perhaps modifying it, up the (dynamic) chain of hierarchy.

Practically, we have shown that both static and dynamic delimited continuation operators are the members of a single family of delimited control, parameterized by simple functions specifying the composition of stack fragments. This gives rise to the simplest Scheme realization of the delimited control operators in terms of `shift/reset`.

Acknowledgments

This paper owes its existence to [8]. The inspiration of the latter cannot be over-emphasized. I am very grateful to Chung-chieh Shan for many insightful discussions. I would like to thank Amr A. Sabry for his encouragement and shepherding, and Daniel P. Friedman for valuable comments.

References

[1] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A generalization of exceptions and control in ML-like languages. In *Functional programming languages and computer architecture: 7th conference*, ed. Simon L. Peyton Jones, 12–23. New York: ACM Press.

[2] Cartwright, Robert, and Matthias Felleisen. 1994. Extensible denotational language specifications. In *TACS*, ed. Masami Hagiya and John C. Mitchell, vol. 789 of *Lecture Notes in Computer Science*, 244–272. Springer.

[3] Danvy, Olivier, and Andrzej Filinski. 1989. A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark.

[4] ———. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, 151–160. New York: ACM Press.

[5] Felleisen, Matthias. 1991. On the expressive power of programming languages. In *Science of computer programming*, vol. 17, 35–75. Preliminary version in: *Proc. European Symposium on Programming*, Lecture Notes in Computer Science, 432. Springer-Verlag (1990), 134–151.

[6] Fischer, Michael J. 1993. Lambda-calculus schemata. *Lisp and Symbolic Computation* 6(3-4):259–288.

[7] Matthias Felleisen, Daniel P. Friedman, Mitchell Wand, and Bruce F. Duba. 1988. Abstract continuations: A mathematical semantics for handling full jumps. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, 52–62. New York: ACM Press.

[8] Shan, Chung-chieh. 2004. Shift to control. In *Proceedings of the 5th workshop on scheme and functional programming*, ed. Olin Shivers and Oscar Waddell, 99–107. Technical report 600, Computer Science Department, Indiana University.