BRICS RS-05-10 Biernacki & Danvy: A Simple Proof of a Folklore Theorem about Delimited Control

**BRICS**

**Basic Research in Computer Science**

# A Simple Proof of a Folklore Theorem about Delimited Control

**Dariusz Biernacki**
**Olivier Danvy**

**BRICS Report Series** **RS-05-10**

**ISSN 0909-0878** **March 2005**

**See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:**

**BRICS**
**Department of Computer Science**
**University of Aarhus**
**Ny Munkegade, building 540**
**DK–8000 Aarhus C**
**Denmark**

| | |
|---|---|
| **Telephone:** | **+45 8942 3360** |
| **Telefax:** | **+45 8942 3255** |
| **Internet:** | **BRICS@brics.dk** |

**BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:**

`http://www.brics.dk`
`ftp://ftp.brics.dk`
**This document in subdirectory** `RS/05/10/`

# A Simple Proof of a Folklore Theorem about Delimited Control

Dariusz Biernacki and Olivier Danvy
BRICS*
Department of Computer Science
University of Aarhus†

March 18, 2005

**Abstract**

We formalize and prove the folklore theorem that the static delimited-control operators `shift` and `reset` can be simulated in terms of the dynamic delimited-control operators `control` and `prompt`. The proof is based on small-step operational semantics.

**Keywords**

Delimited continuations, abstract machines.

*Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.
†IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: {`dabi`,`danvy`}`@brics.dk`

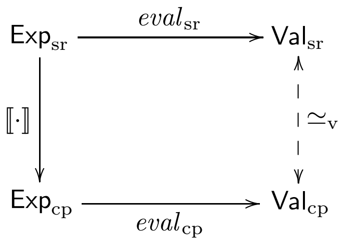
# Contents



# List of Figures

# 1 Introduction

The recent upsurge of interest in delimited continuations [1, 5, 8, 12] seems to take it for granted that dynamic delimited continuations can simulate static delimited continuations by delimiting the context of their resumption. And indeed this property has been mentioned early in the literature about delimited continuations [4, Section 5]. We are, however, not aware of any proof of this folklore theorem, and our goal here is to provide such a proof. To this end, we present two abstract machines—one for static delimited continuations as provided by the control operators `shift` and `reset` [4] and inducing a partial evaluation function $eval_{\mathrm{sr}}$, and one for dynamic delimited continuations as provided by the control operators `control` and `prompt` [7] and inducing a partial evaluation function $eval_{\mathrm{cp}}$—and one compositional mapping $[\![\cdot]\!]$ from programs using `shift` and `reset` to programs using `control` and `prompt`. We then prove that the following diagram commutes:




where the value equivalence $\simeq_{\mathrm{v}}$, for ground values, is defined as equality.

# 2 The formalization

Figures 1 and 2 display two abstract machines, one for the $\lambda$-calculus extended with `shift` and `reset`, and one for the $\lambda$-calculus extended with `control` and `prompt`. These two machines only differ in the application of captured contexts.

In the source syntax, we distinguish between $\lambda$-bound variables ($x$) and `shift`- or `control`-bound variables ($k$). We use the same meta-variables ($e$, $n$, $i$, $x$, $k$, $v$, $\rho$, $C_1$ and $C_2$) ranging over the components of the two abstract machines whenever it does not lead to ambiguity. Programs are closed terms.

## 2.1 A definitional abstract machine for `shift` and `reset`

In our earlier work [2], we derived a definitional abstract machine for `shift` and `reset` by defunctionalizing the continuation and meta-continuation of Danvy and Filinski's definitional evaluator [4]. This definitional abstract machine is displayed in Figure 1; it is a straightforward extension of Felleisen et al.'s CEK machine [6] with a meta-context. The source language is the untyped $\lambda$-calculus extended with integers, the successor function, `shift` (noted $\mathcal{S}$), and `reset` (noted $\langle\cdot\rangle$).

- Terms and identifiers: $e ::= \ulcorner n \urcorner \mid i \mid \lambda x.e \mid e_0\, e_1 \mid succ\ e \mid \langle e \rangle \mid \mathcal{S}k.e$
  $i ::= x \mid k$
- Values (integers, closures, and captured contexts): $v ::= n \mid [x,\, e,\, \rho] \mid C_1$
- Environments: $\rho ::= \rho_{mt} \mid \rho\{i \mapsto v\}$
- Contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((e,\rho),\ C_1) \mid \mathsf{FUN}\,(v,\ C_1) \mid \mathsf{SUCC}\,(C_1)$
- Meta-contexts: $C_2 ::= \mathsf{nil} \mid C_1 :: C_2$
- Initial transition, transition rules, and final transition:

| | | |
|---|---|---|
| $e$ | $\Rightarrow_{sr}$ | $\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{eval}$ |
| $\langle \ulcorner n \urcorner,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{sr}$ | $\langle C_1,\, n,\, C_2\rangle_{cont_1}$ |
| $\langle i,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{sr}$ | $\langle C_1,\, \rho(i),\, C_2\rangle_{cont_1}$ |
| $\langle \lambda x.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{sr}$ | $\langle C_1,\, [x,\, e,\, \rho],\, C_2\rangle_{cont_1}$ |
| $\langle e_0\, e_1,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{sr}$ | $\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1,\rho),\ C_1),\, C_2\rangle_{eval}$ |
| $\langle succ\ e,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{sr}$ | $\langle e,\, \rho,\, \mathsf{SUCC}\,(C_1),\, C_2\rangle_{eval}$ |
| $\langle \langle e \rangle,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{sr}$ | $\langle e,\, \rho,\, \mathsf{END},\, C_1 :: C_2\rangle_{eval}$ |
| $\langle \mathcal{S}k.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{sr}$ | $\langle e,\, \rho\{k \mapsto C_1\},\, \mathsf{END},\, C_2\rangle_{eval}$ |
| $\langle \mathsf{END},\, v,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{sr}$ | $\langle C_2,\, v\rangle_{cont_2}$ |
| $\langle \mathsf{ARG}\,((e,\rho),\ C_1),\, v,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{sr}$ | $\langle e,\, \rho,\, \mathsf{FUN}\,(v,\ C_1),\, C_2\rangle_{eval}$ |
| $\langle \mathsf{FUN}\,([x,\, e,\, \rho],\ C_1),\, v,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{sr}$ | $\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{eval}$ |
| $\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{sr}$ | $\langle C_1',\, v,\, C_1 :: C_2\rangle_{cont_1}$ |
| $\langle \mathsf{SUCC}\,(C_1),\, n,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{sr}$ | $\langle C_1,\, n+1,\, C_2\rangle_{cont_1}$ |
| $\langle C_1 :: C_2,\, v\rangle_{cont_2}$ | $\Rightarrow_{sr}$ | $\langle C_1,\, v,\, C_2\rangle_{cont_1}$ |
| $\langle \mathsf{nil},\, v\rangle_{cont_2}$ | $\Rightarrow_{sr}$ | $v$ |

Figure 1: A definitional abstract machine for `shift` and `reset`

**Definition 1.** *The partial evaluation function* $eval_{\rm sr}$ *mapping programs to values is defined as follows:* $eval_{\rm sr}(e) = v$ *if and only if* $\langle e, \rho_{mt}, \mathsf{END}, \mathsf{nil}\rangle_{eval} \Rightarrow^+_{sr} \langle \mathsf{nil}, v\rangle_{cont_2}$.

N.B.: $\rho$ is a partial function mapping identifiers to values, $\rho_{mt}$ is the empty environment, i.e., a function with an empty domain; and $\rho\{i \mapsto v\}$ is defined as follows:

$$(\rho\{i \mapsto v\})(i') = \begin{cases} v & \textit{if } i' = i \\ (\rho \setminus \{i\})(i') & \textit{otherwise} \end{cases}$$

where $\rho \setminus \{i\}$ denotes the restriction of $\rho$ to its domain excluding $i$.

## 2.2 A definitional abstract machine for `control` and `prompt`

In our earlier work [2], we also showed how to modify the abstract machine for `shift` and `reset` to obtain a definitional abstract machine for `control` and `prompt` [7]. This abstract machine is displayed in Figure 2. The source language is the $\lambda$-calculus extended with integers, the successor function, `control` (noted $\mathcal{F}$) and `prompt` (noted $\#$).

**Definition 2.** *The partial evaluation function* $eval_{\rm cp}$ *mapping programs to values is defined as follows:* $eval_{\rm cp}(e) = v$ *if and only if* $\langle e, \rho_{mt}, \mathsf{END}, \mathsf{nil}\rangle_{eval} \Rightarrow^+_{cp} \langle \mathsf{nil}, v\rangle_{cont_2}$.

## 2.3 Static vs. dynamic delimited continuations

In Figure 1, `shift` and `reset` are said to be *static* because the application of a delimited continuation (represented as a captured context) does not depend on the current context. It is implemented by pushing the current context on the stack of contexts and installing the captured context as the new current context, as shown by the following transition:

$$\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2\rangle_{cont_1} \ \Rightarrow_{sr}\ \langle C_1',\, v,\, C_1 :: C_2\rangle_{cont_1}$$

A subsequent `shift` operation will therefore capture the remainder of the reinstated context, statically.

In Figure 2, `control` and `prompt` are said to be *dynamic* because the application of a delimited continuation (also represented as a captured context) depends on the current context. It is implemented by concatenating the captured context to the current context, as shown by the following transition:

$$\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2\rangle_{cont_1} \ \Rightarrow_{cp}\ \langle C_1' \star C_1,\, v,\, C_2\rangle_{cont_1}$$

A subsequent `control` operation will therefore capture the remainder of the reinstated context together with the then-current context, dynamically.

- Terms and identifiers: $e ::= \ulcorner n \urcorner \mid i \mid \lambda x.e \mid e_0\, e_1 \mid succ\ e \mid \#e \mid \mathcal{F}k.e$
  $i ::= x \mid k$
- Values (integers, closures, and captured contexts): $v ::= n \mid [x,\, e,\, \rho] \mid C_1$
- Environments: $\rho ::= \rho_{mt} \mid \rho\{i \mapsto v\}$
- Contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((e,\rho),\ C_1) \mid \mathsf{FUN}\,(v,\ C_1) \mid \mathsf{SUCC}\,(C_1)$
- Concatenation of contexts:

$$\begin{array}{rcl} \mathsf{END} \star C_1' & \stackrel{\text{def}}{=} & C_1' \\ (\mathsf{ARG}\,((e,\rho),\ C_1)) \star C_1' & \stackrel{\text{def}}{=} & \mathsf{ARG}\,((e,\rho),\ C_1 \star C_1') \\ (\mathsf{FUN}\,(v,\ C_1)) \star C_1' & \stackrel{\text{def}}{=} & \mathsf{FUN}\,(v,\ C_1 \star C_1') \\ (\mathsf{SUCC}\,(C_1)) \star C_1' & \stackrel{\text{def}}{=} & \mathsf{SUCC}\,(C_1 \star C_1') \end{array}$$

- Meta-contexts: $C_2 ::= \mathsf{nil} \mid C_1 :: C_2$
- Initial transition, transition rules, and final transition:

| | | |
|---|---|---|
| $e$ | $\Rightarrow_{cp}$ | $\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{eval}$ |
| $\langle \ulcorner n \urcorner,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{cp}$ | $\langle C_1,\, n,\, C_2\rangle_{cont_1}$ |
| $\langle i,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{cp}$ | $\langle C_1,\, \rho(i),\, C_2\rangle_{cont_1}$ |
| $\langle \lambda x.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{cp}$ | $\langle C_1,\, [x,\, e,\, \rho],\, C_2\rangle_{cont_1}$ |
| $\langle e_0\, e_1,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{cp}$ | $\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1,\rho),\ C_1),\, C_2\rangle_{eval}$ |
| $\langle succ\ e,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{cp}$ | $\langle e,\, \rho,\, \mathsf{SUCC}\,(C_1),\, C_2\rangle_{eval}$ |
| $\langle \#e,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{cp}$ | $\langle e,\, \rho,\, \mathsf{END},\, C_1 :: C_2\rangle_{eval}$ |
| $\langle \mathcal{F}k.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$ | $\Rightarrow_{cp}$ | $\langle e,\, \rho\{k \mapsto C_1\},\, \mathsf{END},\, C_2\rangle_{eval}$ |
| $\langle \mathsf{END},\, v,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{cp}$ | $\langle C_2,\, v\rangle_{cont_2}$ |
| $\langle \mathsf{ARG}\,((e,\rho),\ C_1),\, v,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{cp}$ | $\langle e,\, \rho,\, \mathsf{FUN}\,(v,\ C_1),\, C_2\rangle_{eval}$ |
| $\langle \mathsf{FUN}\,([x,\, e,\, \rho],\ C_1),\, v,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{cp}$ | $\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{eval}$ |
| $\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{cp}$ | $\langle C_1' \star C_1,\, v,\, C_2\rangle_{cont_1}$ |
| $\langle \mathsf{SUCC}\,(C_1),\, n,\, C_2\rangle_{cont_1}$ | $\Rightarrow_{cp}$ | $\langle C_1,\, n+1,\, C_2\rangle_{cont_1}$ |
| $\langle C_1 :: C_2,\, v\rangle_{cont_2}$ | $\Rightarrow_{cp}$ | $\langle C_1,\, v,\, C_2\rangle_{cont_1}$ |
| $\langle \mathsf{nil},\, v\rangle_{cont_2}$ | $\Rightarrow_{cp}$ | $v$ |

Figure 2: A definitional abstract machine for `control` and `prompt`

The two abstract machines differ only in this single transition. Because of this single transition, programs using `shift` and `reset` are compatible with the traditional notion of continuation-passing style [2, 4, 11] whereas programs using `control` and `prompt` give rise to a more complex notion of continuation-passing style that threads a dynamic state [3, 5, 12]. This difference in the semantics of `shift` and `control` also induces distinct computational behaviors. For example, using `call-with-current-delimited-continuation` (instead of `shift` or `control`) and `delimit-continuation` (instead of `reset` or `prompt`), let us consider the following function that traverses a given list and returns another list;[1] this function is written in the syntax of Scheme [9]:

```
(define traverse
  (lambda (xs)
    (letrec ([visit
              (lambda (xs)
                (if (null? xs)
                    '()
                    (visit (call-with-current-delimited-continuation
                             (lambda (k)
                               (cons (car xs) (k (cdr xs))))))))])
      (delimit-continuation
        (lambda ()
          (visit xs))))))
```

- The function <u>copies</u> its input list if `shift` and `reset` are used instead of `call-with- current-delimited-continuation` and `delimit-continuation`. The reason why is that reinstating a `shift`-abstracted context keeps it distinct from the current context. Here, `shift` successively abstracts a delimited context that solely consists in the call to `visit`. Intuitively, this delimited context reads as follows:

  ```
  (lambda (v)
    (delimit-continuation
      (lambda ()
        (visit v))))
  ```

- The function <u>reverses</u> its input list if `control` and `prompt` are used instead of `call-with-current-delimited-continuation` and `delimit-continuation`. The reason why is that reinstating a `control`-abstracted context grafts it to the current context. Here, `control` successively abstracts a context that consists in the call to `visit` followed by the construction of a reversed prefix of the input list. Intuitively, when the input list is `(1 2 3)`, the successive contexts read as follows:

  ```
  (lambda (v) (visit v))
  (lambda (v) (cons 1 (visit v))
  (lambda (v) (cons 2 (cons 1 (visit v))))
  ```

[1] This example is due to Biernacka, Biernacki, and Danvy (March 2005).

**Programming folklore.** *To obtain the effect of* `shift` *and* `reset` *using* `control` *and* `prompt`*, one should replace every occurrence of a shift-bound variable $k$ by its $\eta$-expanded and delimited version $\lambda x.\#(k\, x)$. (As a $\beta$-optimization, every application of $k$ to a simple expression $e$ can be replaced by $\#(k\, e)$.)*

And indeed, replacing

```
(cons (car xs) (k (cdr xs)))
```

by

```
(cons (car xs) (delimit-continuation
                 (lambda ()
                   (k (cdr xs)))))
```

in the definition of `traverse` above makes it copy its input list, no matter whether `shift` and `reset` or `control` and `prompt` are used.

We formalize the replacement above with the following compositional translation from the language with `shift` and `reset` to the language with `control` and `prompt`.

**Definition 3.** *The translation $[\![\cdot]\!]$ is defined as follows:*

$$
\begin{array}{rcl}
[\![\ulcorner n \urcorner]\!] & = & \ulcorner n \urcorner \\
[\![x]\!] & = & x \\
[\![k]\!] & = & \lambda x.\#(k\, x), \text{ where } x \text{ is fresh} \\
[\![\lambda x.e]\!] & = & \lambda x.[\![e]\!] \\
[\![e_0\, e_1]\!] & = & [\![e_0]\!]\, [\![e_1]\!] \\
[\![\langle e \rangle]\!] & = & \#[\![e]\!] \\
[\![\mathcal{S}k.e]\!] & = & \mathcal{F}k.[\![e]\!]
\end{array}
$$

In the next section, we prove that for any program $e$, $\mathit{eval}_{\mathrm{sr}}\,(e)$ and $\mathit{eval}_{\mathrm{cp}}\,([\![e]\!])$ are observationally equivalent and, in particular, equal for ground values.

# 3 The folklore theorem and its formal proof

We first define an auxiliary abstract machine for `control` and `prompt` that implements the application of an $\eta$-expanded and delimited continuation in one step. By construction, this auxiliary abstract machine is equivalent to the definitional one of Figure 2. We then show that the auxiliary machine operates in lock step with the definitional abstract machine of Figure 1. To this end, we define a family of relations between the abstract machine for `shift` and `reset` and the auxiliary abstract machine. The folklore theorem follows.

## 3.1 An auxiliary abstract machine for `control` and `prompt`

**Definition 4.** *The auxiliary abstract machine for* `control` *and* `prompt` *is defined as follows:*

*(1) All the components, including configurations $\delta$, of the auxiliary abstract machine are identical to the components of the definitional abstract machine of Figure 2.*

*(2) The transitions of the auxiliary abstract machine, denoted $\delta \Rightarrow_{aux} \delta'$, are defined as follows:*

- *if* $\delta = \langle \mathsf{FUN}\,([x,\, \#(k\, x),\, \rho],\, C_1),\, v,\, C_2\rangle_{cont_1}$ *then* $\delta' = \langle C_1',\, v,\, C_1 :: C_2\rangle_{cont_1}$, *where* $C_1' = \rho(k)$;
- *otherwise,* $\delta'$ *is the configuration such that* $\delta \Rightarrow_{cp} \delta'$, *if it exists.*

*(3) The partial evaluation function* $eval_{aux}$ *is defined in the usual way:* $eval_{aux}\,(e) = v$ *if and only if* $\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{eval} \Rightarrow^{+}_{aux} \langle \mathsf{nil},\, v\rangle_{cont_2}$.

The following lemma shows that the definitional abstract machine for `control` and `prompt` simulates the single step of the auxiliary abstract machine in several steps.

**Lemma 1.** *For all* $v$, $C_1$, $C_1'$ *and* $C_2$,

$$\langle \mathsf{FUN}\,([x,\, \#(k\, x),\, \rho],\, C_1),\, v,\, C_2\rangle_{cont_1} \Rightarrow^{+}_{cp} \langle C_1',\, v,\, C_1 :: C_2\rangle_{cont_1},\ \text{where } C_1' = \rho(k).$$

*Proof.* From the definition of the abstract machine for `control` and `prompt` in Figure 2:

$$\begin{array}{rl}
\langle \mathsf{FUN}\,([x,\, \#(k\, x),\, \rho],\, C_1),\, v,\, C_2\rangle_{cont_1} & \Rightarrow_{cp} \\
\langle \#(k\, x),\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{eval} & \Rightarrow_{cp} \\
\langle k\, x,\, \rho\{x \mapsto v\},\, \mathsf{END},\, C_1 :: C_2\rangle_{eval} & \Rightarrow_{cp} \\
\langle k,\, \rho\{x \mapsto v\},\, \mathsf{ARG}\,((x, \rho\{x \mapsto v\}),\, \mathsf{END}),\, C_1 :: C_2\rangle_{eval} & \Rightarrow_{cp} \\
\langle \mathsf{ARG}\,((x, \rho\{x \mapsto v\}),\, \mathsf{END}),\, C_1',\, C_1 :: C_2\rangle_{cont_1} & \Rightarrow_{cp} \\
\langle x,\, \rho\{x \mapsto v\},\, \mathsf{FUN}\,(C_1',\, \mathsf{END}),\, C_1 :: C_2\rangle_{eval} & \Rightarrow_{cp} \\
\langle \mathsf{FUN}\,(C_1',\, \mathsf{END}),\, v,\, C_1 :: C_2\rangle_{cont_1} & \Rightarrow_{cp} \\
\langle C_1',\, v,\, C_1 :: C_2\rangle_{cont_1} &
\end{array}$$

□

**Proposition 1.** *For any program* $e$ *and for any value* $v$, $eval_{\mathrm{cp}}\,(e) = v$ *if and only if* $eval_{aux}\,(e) = v$.

*Proof.* Follows directly from Definition 4 and Lemma 1. □

## 3.2 A family of relations

We now define a family of relations between the abstract machine for `shift` and `reset` and the auxiliary abstract machine for `control` and `prompt`. To distinguish between the two machines, as a diacritical convention [10], we annotate the components of the machine for `shift` and `reset` with a tilde.

**Definition 5.** *The relations between the components of the abstract machine for* `shift` *and* `reset` *and the auxiliary abstract machine for* `control` *and* `prompt` *are defined as follows:*

(1) *Terms:* $\widetilde{e} \simeq_{\mathrm{e}} e$ *iff* $[\![\widetilde{e}]\!] = e$

(2) *Values:*

   (a) $\widetilde{n} \simeq_{\mathrm{v}} n$ *iff* $\widetilde{n} = n$

   (b) $[\widetilde{x},\, \widetilde{e},\, \widetilde{\rho}] \simeq_{\mathrm{v}} [x,\, e,\, \rho]$ *iff* $\widetilde{x} = x$, $\widetilde{e} \simeq_{\mathrm{e}} e$ *and* $\widetilde{\rho} \simeq_{\mathrm{env}} \rho$

   (c) $\widetilde{C_1} \simeq_{\mathrm{v}} [x,\, \#(k\, x),\, \rho]$ *iff* $\widetilde{C_1} \simeq_{\mathrm{c}} \rho(k)$

   (d) $\widetilde{C_1} \simeq_{\mathrm{v}} C_1$ *iff* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$

(3) *Environments:*

   (a) $\widetilde{\rho_{mt}} \simeq_{\mathrm{env}} \rho_{mt}$

   (b) $\widetilde{\rho}\{i \mapsto \widetilde{v}\} \simeq_{\mathrm{env}} \rho\{i \mapsto v\}$ *iff* $\widetilde{v} \simeq_{\mathrm{v}} v$ *and* $\widetilde{\rho} \setminus \{i\} \simeq_{\mathrm{env}} \rho \setminus \{i\}$.

(4) *Contexts:*

   (a) $\widetilde{\mathsf{END}} \simeq_{\mathrm{c}} \mathsf{END}$

   (b) $\widetilde{\mathsf{ARG}}\,((\widetilde{e}, \widetilde{\rho}),\, \widetilde{C_1}) \simeq_{\mathrm{c}} \mathsf{ARG}\,((e, \rho),\, C_1)$ *iff* $\widetilde{e} \simeq_{\mathrm{e}} e$, $\widetilde{\rho} \simeq_{\mathrm{env}} \rho$, *and* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$

   (c) $\widetilde{\mathsf{FUN}}\,(\widetilde{v},\, \widetilde{C_1}) \simeq_{\mathrm{c}} \mathsf{FUN}\,(v,\, C_1)$ *iff* $\widetilde{v} \simeq_{\mathrm{v}} v$ *and* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$

   (d) $\widetilde{\mathsf{SUCC}}\,(\widetilde{C_1}) \simeq_{\mathrm{c}} \mathsf{SUCC}\,(C_1)$ *iff* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$

(5) *Meta-contexts:*

   (a) $\widetilde{\mathsf{nil}} \simeq_{\mathrm{mc}} \mathsf{nil}$

   (b) $\widetilde{C_1} :: \widetilde{C_2} \simeq_{\mathrm{mc}} C_1 :: C_2$ *iff* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$ *and* $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$

(6) *Configurations:*

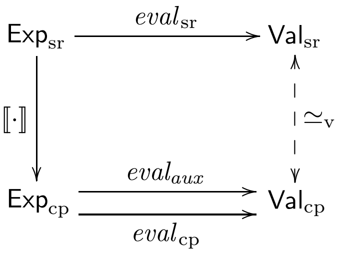
   (a) $\langle \widetilde{e},\, \widetilde{\rho},\, \widetilde{C_1},\, \widetilde{C_2} \rangle_{\widetilde{eval}} \simeq \langle e,\, \rho,\, C_1,\, C_2 \rangle_{eval}$ *iff*
   $\widetilde{e} \simeq_{\mathrm{e}} e$, $\widetilde{\rho} \simeq_{\mathrm{env}} \rho$, $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$, *and* $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$

   (b) $\langle \widetilde{C_1},\, \widetilde{v},\, \widetilde{C_2} \rangle_{\widetilde{cont_1}} \simeq \langle C_1,\, v,\, C_2 \rangle_{cont_1}$ *iff*
   $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$, $\widetilde{v} \simeq_{\mathrm{v}} v$, *and* $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$

   (c) $\langle \widetilde{C_2},\, \widetilde{v} \rangle_{\widetilde{cont_2}} \simeq \langle C_2,\, v \rangle_{cont_2}$ *iff*
   $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$ *and* $\widetilde{v} \simeq_{\mathrm{v}} v$

## 3.3 The formal proof

As a stepping stone, we show that running the abstract machine for `shift` and `reset` on a program $e$ and running the auxiliary abstract machine for `control` and `prompt` on a program $[\![e]\!]$ yield results that are equivalent in the sense of the above relations.




Moreover, we show that the abstract machines operate in lock-step with respect to the relations. To this end we need to prove the following lemmas.

**Lemma 2.** *For all configurations $\widetilde{\delta}$, $\delta$, $\widetilde{\delta}'$ and $\delta'$, if $\widetilde{\delta} \simeq \delta$ then*

$$\widetilde{\delta} \Rightarrow_{sr} \widetilde{\delta}' \text{ if and only if } \delta \Rightarrow_{aux} \delta' \text{ and } \widetilde{\delta}' \simeq \delta'.$$

*Proof.* By case inspection of $\widetilde{\delta} \simeq \delta$. All cases follow directly from the definition of the relation $\simeq$ and the definitions of the abstract machines. We present two crucial cases:

Case: $\widetilde{\delta} = \langle k,\, \widetilde{\rho},\, \widetilde{C_1},\, \widetilde{C_2}\rangle_{\widetilde{eval}}$ and $\delta = \langle \lambda x.\#(k\,x),\, \rho,\, C_1,\, C_2\rangle_{eval}$.
From the definition of the abstract machine for `shift` and `reset`, $\widetilde{\delta} \Rightarrow_{sr} \widetilde{\delta}'$, where $\widetilde{\delta}' = \langle \widetilde{C_1},\, \widetilde{\rho}(k),\, \widetilde{C_2}\rangle_{\widetilde{cont_1}}$.
From the definition of the auxiliary abstract machine for `control` and `prompt`, $\delta \Rightarrow_{cp} \delta'$, where $\delta' = \langle C_1,\, [x,\, \#(k\,x),\, \rho],\, C_2\rangle_{cont_1}$.
By assumption, $\widetilde{\rho}(k) \simeq_{\mathrm{v}} \rho(k)$, $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$ and $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$. Hence, $\widetilde{\delta}' \simeq \delta'$.

Case: $\widetilde{\delta} = \langle \widetilde{\mathsf{FUN}}\,(\widetilde{C_1}',\, \widetilde{C_1}),\, \widetilde{v},\, \widetilde{C_2}\rangle_{\widetilde{eval}}$ and $\delta = \langle \mathsf{FUN}\,([x,\, \#(k\,x),\, \rho],\, C_1),\, v,\, C_2\rangle_{eval}$.
From the definition of the abstract machine for `shift` and `reset`, $\widetilde{\delta} \Rightarrow_{sr} \widetilde{\delta}'$, where $\widetilde{\delta}' = \langle \widetilde{C_1}',\, \widetilde{v},\, \widetilde{C_1} :: \widetilde{C_2}\rangle_{\widetilde{cont_1}}$.
From the definition of the auxiliary abstract machine for `control` and `prompt`, $\delta \Rightarrow_{cp} \delta'$, where $\delta' = \langle C_1',\, v,\, C_1 :: C_2\rangle_{cont_1}$, and $C_1' = \rho(k)$.
By assumption, $\widetilde{C_1}' \simeq_{\mathrm{v}} C_1'$, $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$ and $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$. Hence, $\widetilde{\delta}' \simeq \delta'$.

□

**Lemma 3.** *For all configurations $\widetilde{\delta}$, $\delta$, $\widetilde{\delta}'$ and $\delta'$, and for any $n \geq 1$, if $\widetilde{\delta} \simeq \delta$ then*

$$\widetilde{\delta} \Rightarrow_{sr}^{n} \widetilde{\delta}' \text{ if and only if } \delta \Rightarrow_{aux}^{n} \delta' \text{ and } \widetilde{\delta}' \simeq \delta'.$$

*Proof.* By induction on $n$, using Lemma 2. □

We are now in position to prove the formal statement of the equivalence between the two abstract machines:

**Proposition 2.** *For any program $e$, and for any values $\widetilde{v}$ and $v$, $eval_{\mathrm{sr}}(e) = \widetilde{v}$ if and only if $eval_{aux}(\llbracket e \rrbracket) = v$ and $\widetilde{v} \simeq_{\mathrm{v}} v$.*

*Proof.* The initial configurations $\langle e, \widetilde{\rho_{mt}}, \widetilde{\mathsf{END}}, \widetilde{\mathsf{nil}} \rangle_{\widetilde{eval}}$ and $\langle \llbracket e \rrbracket, \rho_{mt}, \mathsf{END}, \mathsf{nil} \rangle_{eval}$ are in the relation $\simeq$ and thus by Lemma 3 both abstract machines reach their final configurations $\langle \widetilde{\mathsf{nil}}, \widetilde{v} \rangle_{\widetilde{cont_2}}$ and $\langle \mathsf{nil}, v \rangle_{cont_2}$ after the same number of transitions and with $\widetilde{v} \simeq_{\mathrm{v}} v$, or both diverge. □

**Theorem 1.** *For any program $e$, and for any values $\widetilde{v}$ and $v$, $eval_{\mathrm{sr}}(e) = \widetilde{v}$ if and only if $eval_{\mathrm{cp}}(\llbracket e \rrbracket) = v$ and $\widetilde{v} \simeq_{\mathrm{v}} v$.*

*Proof.* Follows directly from Proposition 1 and Proposition 2. □

**Corollary 1 (Folklore).** *For any program $e$, and for any integer $n$, $eval_{\mathrm{sr}}(e) = n$ if and only if $eval_{\mathrm{cp}}(\llbracket e \rrbracket) = n$.*

Extending the source language with more syntactic constructs (other ground values and primitive operations, conditional expressions, recursive definitions, etc.) is straightforward. It is equally simple to extend the proofs.

# 4 Conclusion

We have formalized and proved that the dynamic delimited-control operators `control` and `prompt` can simulate the static delimited-control operators `shift` and `reset` by delimiting the context of the resumption of captured continuations. Shan has recently presented a converse simulation [12]. This converse simulation is considerably more involved than the present one, and it has not been formalized and proved yet.

**Acknowledgments:** We are grateful to Mads Sig Ager, Małgorzata Biernacka, Julia Lawall, Kevin Millikin, and Kristian Støvring for their comments. This work is supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

# References

[1] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A type-theoretic foundation of continuations and prompts. In Kathleen Fisher, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, pages 40–53, Snowbird, Utah, September 2004. ACM Press.

[2] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. Technical Report BRICS RS-04-29, DAIMI, Department of Computer Science, University

of Aarhus, Aarhus, Denmark, December 2004. A preliminary version was presented at the the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).

[3] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Technical Report BRICS RS-05-5, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, February 2005.

[4] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[5] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for subcontinuations. Available at `http://www.cs.indiana.edu/~sabry/research.html`, February 2005.

[6] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[7] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988. ACM Press.

[8] Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, pages 271–282, Pittsburgh, Pennsylvania, September 2002. ACM Press.

[9] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised$^5$ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[10] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and John Wiley, New York, 1976.

[11] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[12] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Snowbird, Utah, September 2004.

## Recent BRICS Report Series Publications

**RS-05-10** **Dariusz Biernacki and Olivier Danvy.** ***A Simple Proof of a Folklore Theorem about Delimited Control.*** **March 2005. ii+11 pp.**

**RS-05-9** **Gudmund Skovbjerg Frandsen and Peter Bro Miltersen.** ***Reviewing Bounds on the Circuit Size of the Hardest Functions.*** **March 2005. 6 pp. To appear in *Information Processing Letters*.**

**RS-05-8** **Peter D. Mosses.** ***Exploiting Labels in Structural Operational Semantics.*** **February 2005. 15 pp. Appears in *Fundamenta Informaticae*, 60:17–31, 2004.**

**RS-05-7** **Peter D. Mosses.** ***Modular Structural Operational Semantics.*** **February 2005. 46 pp. Appears in *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004.**

**RS-05-6** **Karl Krukow and Andrew Twigg.** ***Distributed Approximation of Fixed-Points in Trust Structures.*** **February 2005. 41 pp.**

**RS-05-5** **A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations.** ***Dariusz Biernacki and Olivier Danvy and Kevin Millikin.*** **February 2005.**

**RS-05-4** **Andrzej Filinski and Henning Korsholm Rohde.** ***Denotational Aspects of Untyped Normalization by Evaluation.*** **February 2005.**

**RS-05-3** **Olivier Danvy and Mayer Goldberg.** ***There and Back Again.*** **January 2005. iii+16 pp. Extended version of an article to appear in *Fundamenta Informatica*. This version supersedes BRICS RS-02-12.**

**RS-05-2** **Dariusz Biernacki and Olivier Danvy.** ***On the Dynamic Extent of Delimited Continuations.*** **January 2005. ii+30 pp.**

**RS-05-1** **Mayer Goldberg.** ***On the Recursive Enumerability of Fixed-Point Combinators.*** **January 2005. 7 pp. Superseedes BRICS report RS-04-25.**

**RS-04-41** **Olivier Danvy.** ***Sur un Exemple de Patrick Greussay.*** **December 2004. 14 pp.**

**RS-04-40** **Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde.** ***Fast Partial Evaluation of Pattern Matching in Strings.*** **December 2004. 22 pp. To appear in TOPLAS. Supersedes BRICS report RS-03-20.**