JFP **16** (3): 269–280, 2006. © 2005 Cambridge University Press
doi:10.1017/S0956796805005782 First published online 17 November 2005 Printed in the United Kingdom

THEORETICAL PEARL
A simple proof of a folklore theorem about delimited control

DARIUSZ BIERNACKI and OLIVIER DANVY
BRICS, Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark*
(*e-mail:* {dabi,danvy}@brics.dk)

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.

1 Introduction

In the recent upsurge of interest in delimited continuations (Ariola *et al.*, 2004; Dybvig *et al.*, 2005; Gasbichler & Sperber, 2002; Shan, 2004; Kiselyov, 2005) it appears to be taken 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 (Danvy & Filinski, 1990, 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` (Danvy & Filinski, 1990) and inducing a partial evaluation function $eval_{\mathrm{sr}}$, and one for dynamic delimited continuations as provided by the control operators `control` and `prompt` (Felleisen *et al.*, 1988) 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:

$$\begin{tikzcd}
\mathsf{Exp}_{\mathrm{sr}} \arrow[r, "eval_{\mathrm{sr}}"] \arrow[d, "{[\![\cdot]\!]}"'] & \mathsf{Val}_{\mathrm{sr}} \arrow[d, dashed, leftrightarrow, "\simeq_{\mathrm{v}}"] \\
\mathsf{Exp}_{\mathrm{cp}} \arrow[r, "eval_{\mathrm{cp}}"'] & \mathsf{Val}_{\mathrm{cp}}
\end{tikzcd}$$

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

* Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

2 The formalization

Figures 1 and 2 display two abstract machines, one for the λ-calculus extended with `shift` and `reset`, and one for the λ-calculus extended with `control` and `prompt`. These two machines only differ in the application of captured contexts (which represent delimited continuations in the course of executing source programs).

For simplicity, in the source syntax, we distinguish between λ-bound variables (x) and `shift`- or `control`-bound variables (k). We use the same meta-variables (e, n, i, x, k, v, ρ, 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 (Biernacka *et al.*, 2005), we derived a definitional abstract machine for `shift` and `reset` by defunctionalizing the continuation and meta-continuation of Danvy and Filinski's definitional evaluator (Danvy & Filinski, 1990). This definitional abstract machine is displayed in Figure 1; it is a straightforward extension of Felleisen *et al.*'s CEK machine (Felleisen & Friedman, 1986) with a meta-context. The source language is the untyped λ-calculus extended with integers, the successor function, `shift` (noted $\mathscr{S}$), and `reset` (noted $\langle\cdot\rangle$). The machine is an extension of the CEK machine because when given a program that does not use `shift` and `reset`, it operates in lock step with the CEK machine. When delimiting control with `reset`, the machine pushes the current context on the current meta-context, and proceeds with an empty context. When abstracting control with `shift`, the machine captures the current context and proceeds with an empty context. When applying a captured context, the machine pushes the current context on the current meta-context, and proceeds with the captured context.

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

We could define the function $eval_{\text{sr}}$ in terms of the initial and final transition, but they play only an administrative role, i.e., to load an input term to the machine and to unload the computed value from the machine.

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

In our earlier work (Biernacka *et al.*, 2005), we also showed how to modify the abstract machine for `shift` and `reset` to obtain a definitional abstract machine for `control` and `prompt` (Felleisen *et al.*, 1988; Felleisen, 1988). This abstract machine is displayed in Figure 2. The source language is the λ-calculus extended with integers, the successor function, `control` (noted $\mathscr{F}$) and `prompt` (noted $\#$). The machine is an extension of the CEK machine because when given a program that does not use `control` and `prompt`, it operates in lock step with the CEK machine. When delimiting control with `prompt`, the machine pushes the current context on the current meta-context, and proceeds with an empty context. When abstracting control with `control`,

- Terms and identifiers: $e ::= \ulcorner n \urcorner \mid i \mid \lambda x.e \mid e_0\, e_1 \mid succ\ e \mid \langle\!\langle e \rangle\!\rangle \mid \mathcal{S}k.e$
  $i ::= x \mid k$
- 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:

$$
\begin{array}{rcl}
\hline
e & \Rightarrow_{\mathrm{sr}} & \langle e,\ \rho_{mt},\ \mathsf{END},\ \mathsf{nil}\rangle_{eval} \\
\hline
\langle \ulcorner n \urcorner,\ \rho,\ C_1,\ C_2\rangle_{eval} & \Rightarrow_{\mathrm{sr}} & \langle C_1,\ n,\ C_2\rangle_{cont_1} \\
\langle i,\ \rho,\ C_1,\ C_2\rangle_{eval} & \Rightarrow_{\mathrm{sr}} & \langle C_1,\ \rho(i),\ C_2\rangle_{cont_1} \\
\langle \lambda x.e,\ \rho,\ C_1,\ C_2\rangle_{eval} & \Rightarrow_{\mathrm{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_{\mathrm{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_{\mathrm{sr}} & \langle e,\ \rho,\ \mathsf{SUCC}\,(C_1),\ C_2\rangle_{eval} \\
\langle \langle\!\langle e \rangle\!\rangle,\ \rho,\ C_1,\ C_2\rangle_{eval} & \Rightarrow_{\mathrm{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_{\mathrm{sr}} & \langle e,\ \rho\{k \mapsto C_1\},\ \mathsf{END},\ C_2\rangle_{eval} \\
\hline
\langle \mathsf{END},\ v,\ C_2\rangle_{cont_1} & \Rightarrow_{\mathrm{sr}} & \langle C_2,\ v\rangle_{cont_2} \\
\langle \mathsf{ARG}\,((e, \rho),\ C_1),\ v,\ C_2\rangle_{cont_1} & \Rightarrow_{\mathrm{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_{\mathrm{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_{\mathrm{sr}} & \langle C_1',\ v,\ C_1 :: C_2\rangle_{cont_1} \\
\langle \mathsf{SUCC}\,(C_1),\ n,\ C_2\rangle_{cont_1} & \Rightarrow_{\mathrm{sr}} & \langle C_1,\ n+1,\ C_2\rangle_{cont_1} \\
\hline
\langle C_1 :: C_2,\ v\rangle_{cont_2} & \Rightarrow_{\mathrm{sr}} & \langle C_1,\ v,\ C_2\rangle_{cont_1} \\
\hline
\langle \mathsf{nil},\ v\rangle_{cont_2} & \Rightarrow_{\mathrm{sr}} & v \\
\hline
\end{array}
$$

Fig. 1. A definitional abstract machine for `shift` and `reset`.

the machine captures the current context and proceeds with an empty context. When applying a captured context, the machine concatenates the captured context to the current context and proceeds with the resulting context.

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

- Terms and identifiers: $e ::= \ulcorner n \urcorner \mid i \mid \lambda x.e \mid e_0\, e_1 \mid succ\ e \mid \#e \mid \mathscr{F}k.e$
  $i ::= x \mid k$
- 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:

$$\begin{array}{rcl}
\hline
e & \Rightarrow_{\text{cp}} & \langle e,\ \rho_{mt},\ \mathsf{END},\ \mathsf{nil}\rangle_{eval} \\
\hline
\langle \ulcorner n \urcorner,\ \rho,\ C_1,\ C_2\rangle_{eval} & \Rightarrow_{\text{cp}} & \langle C_1,\ n,\ C_2\rangle_{cont_1} \\
\langle i,\ \rho,\ C_1,\ C_2\rangle_{eval} & \Rightarrow_{\text{cp}} & \langle C_1,\ \rho(i),\ C_2\rangle_{cont_1} \\
\langle \lambda x.e,\ \rho,\ C_1,\ C_2\rangle_{eval} & \Rightarrow_{\text{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_{\text{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_{\text{cp}} & \langle e,\ \rho,\ \mathsf{SUCC}\,(C_1),\ C_2\rangle_{eval} \\
\langle \#e,\ \rho,\ C_1,\ C_2\rangle_{eval} & \Rightarrow_{\text{cp}} & \langle e,\ \rho,\ \mathsf{END},\ C_1 :: C_2\rangle_{eval} \\
\langle \mathscr{F}k.e,\ \rho,\ C_1,\ C_2\rangle_{eval} & \Rightarrow_{\text{cp}} & \langle e,\ \rho\{k \mapsto C_1\},\ \mathsf{END},\ C_2\rangle_{eval} \\
\hline
\langle \mathsf{END},\ v,\ C_2\rangle_{cont_1} & \Rightarrow_{\text{cp}} & \langle C_2,\ v\rangle_{cont_2} \\
\langle \mathsf{ARG}\,((e,\rho),\ C_1),\ v,\ C_2\rangle_{cont_1} & \Rightarrow_{\text{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_{\text{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_{\text{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_{\text{cp}} & \langle C_1,\ n+1,\ C_2\rangle_{cont_1} \\
\hline
\langle C_1 :: C_2,\ v\rangle_{cont_2} & \Rightarrow_{\text{cp}} & \langle C_1,\ v,\ C_2\rangle_{cont_1} \\
\hline
\langle \mathsf{nil},\ v\rangle_{cont_2} & \Rightarrow_{\text{cp}} & v \\
\hline
\end{array}$$

Fig. 2. A definitional abstract machine for `control` and `prompt`.

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:

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

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:

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

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

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 (Biernacka *et al.*, 2005; Danvy & Filinski, 1990; Plotkin, 1975) whereas programs using `control` and `prompt` give rise to a more complex notion of continuation-passing style that threads a dynamic state (Biernacki *et al.*, 2005; Dybvig *et al.*, 2005; Shan, 2004). This difference in the semantics of `shift` and `control` also induces distinct computational behaviors, as illustrated in the following example.

Copying vs. reversing a list: 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 (Biernacka *et al.*, 2005, section 4.5); this function is written in the syntax of Scheme (Kelsey *et al.*, 1998):

```
(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 of the call to `visit`. Intuitively, this delimited context reads as follows:

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

- The function reverses 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 of 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))))
```

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* η*-expanded and delimited version* $\lambda x.\#(k\ x)$*. (As a* β_v*-optimization, every application of* k *to a trivial expression* e *(typically a value) 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, $eval_{\mathrm{sr}}(e)$ and $eval_{\mathrm{cp}}([\![e]\!])$ are equivalent (in the sense of Definition 5 below) 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 η-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 δ, 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 $\delta \Rightarrow_{\text{aux}} \delta'$, 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, δ' is the configuration such that $\delta \Rightarrow_{\text{cp}} \delta'$, if it exists.
3. The partial evaluation function $eval_{aux}$ is defined in the usual way: $eval_{aux}\,(e) = v$ if and only if $\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{eval} \Rightarrow^{+}_{\text{aux}} \langle \mathsf{nil},\, v\rangle_{cont_2}$.

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,

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

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

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

□

Proposition 1
For any program e and for any value v, $eval_{\text{cp}}\,(e) = v$ 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 (Milne & Strachey, 1976), we annotate the components of the machine for `shift` and `reset` with a hat.

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: $\widehat{e} \simeq_{\text{e}} e$ iff $[\![\widehat{e}]\!] = e$
2. Values:
   (a) $\widehat{n} \simeq_{\text{v}} n$ iff $\widehat{n} = n$
   (b) $[\widehat{x}, \widehat{e}, \widehat{\rho}] \simeq_{\text{v}} [x, e, \rho]$ iff $\widehat{x} = x$, $\widehat{e} \simeq_{\text{e}} e$ and $\widehat{\rho} \simeq_{\text{env}} \rho$
   (c) $\widehat{C_1} \simeq_{\text{v}} [x, \#(k\ x), \rho]$ iff $\widehat{C_1} \simeq_{\text{c}} \rho(k)$
3. Environments:
   (a) $\widehat{\rho_{mt}} \simeq_{\text{env}} \rho_{mt}$
   (b) $\widehat{\rho}\{x \mapsto \widehat{v}\} \simeq_{\text{env}} \rho\{x \mapsto v\}$ iff $\widehat{v} \simeq_{\text{v}} v$ and $\widehat{\rho} \setminus \{x\} \simeq_{\text{env}} \rho \setminus \{x\}$, where $\rho \setminus \{i\}$ denotes the restriction of ρ to its domain excluding i
   (c) $\widehat{\rho}\{k \mapsto \widehat{C_1}\} \simeq_{\text{env}} \rho\{k \mapsto C_1\}$ iff $\widehat{C_1} \simeq_{\text{c}} C_1$ and $\widehat{\rho} \setminus \{k\} \simeq_{\text{env}} \rho \setminus \{k\}$
4. Contexts:
   (a) $\widehat{\mathsf{END}} \simeq_{\text{c}} \mathsf{END}$
   (b) $\widehat{\mathsf{ARG}}\,((\widehat{e}, \widehat{\rho}),\ \widehat{C_1}) \simeq_{\text{c}} \mathsf{ARG}\,((e, \rho),\ C_1)$ iff $\widehat{e} \simeq_{\text{e}} e$, $\widehat{\rho} \simeq_{\text{env}} \rho$, and $\widehat{C_1} \simeq_{\text{c}} C_1$
   (c) $\widehat{\mathsf{FUN}}\,(\widehat{v},\ \widehat{C_1}) \simeq_{\text{c}} \mathsf{FUN}\,(v,\ C_1)$ iff $\widehat{v} \simeq_{\text{v}} v$ and $\widehat{C_1} \simeq_{\text{c}} C_1$
   (d) $\widehat{\mathsf{SUCC}}\,(\widehat{C_1}) \simeq_{\text{c}} \mathsf{SUCC}\,(C_1)$ iff $\widehat{C_1} \simeq_{\text{c}} C_1$
5. Meta-contexts:
   (a) $\widehat{\mathsf{nil}} \simeq_{\text{mc}} \mathsf{nil}$
   (b) $\widehat{C_1} :: \widehat{C_2} \simeq_{\text{mc}} C_1 :: C_2$ iff $\widehat{C_1} \simeq_{\text{c}} C_1$ and $\widehat{C_2} \simeq_{\text{mc}} C_2$
6. Configurations:
   (a) $\langle \widehat{e}, \widehat{\rho}, \widehat{C_1}, \widehat{C_2} \rangle_{\widehat{eval}} \simeq \langle e, \rho, C_1, C_2 \rangle_{eval}$ iff
   $\widehat{e} \simeq_{\text{e}} e$, $\widehat{\rho} \simeq_{\text{env}} \rho$, $\widehat{C_1} \simeq_{\text{c}} C_1$, and $\widehat{C_2} \simeq_{\text{mc}} C_2$
   (b) $\langle \widehat{C_1}, \widehat{v}, \widehat{C_2} \rangle_{\widehat{cont_1}} \simeq \langle C_1, v, C_2 \rangle_{cont_1}$ iff
   $\widehat{C_1} \simeq_{\text{c}} C_1$, $\widehat{v} \simeq_{\text{v}} v$, and $\widehat{C_2} \simeq_{\text{mc}} C_2$
   (c) $\langle \widehat{C_2}, \widehat{v} \rangle_{\widehat{cont_2}} \simeq \langle C_2, v \rangle_{cont_2}$ iff
   $\widehat{C_2} \simeq_{\text{mc}} C_2$ and $\widehat{v} \simeq_{\text{v}} v$

The relations are intended to capture the equivalence of the abstract machine for `shift` and `reset` and the auxiliary abstract machine for `control` and `prompt` when run on a term $\widehat{e}$ and on its translation $[\![\widehat{e}]\!]$, respectively. Most of the cases are homomorphic on the structure of a component. The critical cases are: 1. – a formalization of the programming folklore formulated in section 2.3, and 2. (*c*) – a formalization of the fact that a `control`-abstracted continuation is applied by concatenating its representation to the current context whereas when a `shift`-abstracted continuation is applied, its representation is kept separate from the current context.

3.3 The formal proof

We first show that indeed, running the abstract machine for `shift` and `reset` on a program $\widehat{e}$ and running the auxiliary abstract machine for `control` and `prompt` on a program $[\![\widehat{e}]\!]$ yield results that are equivalent in the sense of the above relations. Then by Proposition 1, we obtain the equivalence result of the abstract machine for `shift` and `reset` and the definitional abstract machine for `control` and `prompt`, as summarized in the following diagram:

$$\begin{tikzcd}
\mathsf{Exp}_{\mathrm{sr}} \arrow[r, "eval_{\mathrm{sr}}"] \arrow[d, "{[\![\cdot]\!]}"'] & \mathsf{Val}_{\mathrm{sr}} \arrow[d, dashed, leftrightarrow, "\simeq_{\mathrm{v}}"] \\
\mathsf{Exp}_{\mathrm{cp}} \arrow[r, shift left, "eval_{aux}"] \arrow[r, shift right, "eval_{\mathrm{cp}}"'] & \mathsf{Val}_{\mathrm{cp}}
\end{tikzcd}$$

More precisely, we show that the abstract machine for `shift` and `reset` and the auxiliary abstract machine for `control` and `prompt` operate in lock-step with respect to the relations. To this end, we need to prove the following lemmas.

Lemma 2
For all configurations $\widehat{\delta}$, δ, $\widehat{\delta}'$ and δ', if $\widehat{\delta} \simeq \delta$ then

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

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

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

Case: $\widehat{\delta} = \langle \mathsf{FUN}(\widehat{C_1}', \widehat{C_1}), \widehat{v}, \widehat{C_2}\rangle_{\widehat{eval}}$ 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`, $\widehat{\delta} \Rightarrow_{\mathrm{sr}} \widehat{\delta}'$, where $\widehat{\delta}' = \langle \widehat{C_1}', \widehat{v}, \widehat{C_1} :: \widehat{C_2}\rangle_{cont_1}$.
From the definition of the auxiliary abstract machine for `control` and `prompt`,

$\delta \Rightarrow_{\mathrm{aux}} \delta'$, where $\delta' = \langle C_1', v, C_1 :: C_2\rangle_{cont_1}$, and $C_1' = \rho(k)$.
By assumption, $\widehat{C_1}' \simeq_{\mathrm{c}} C_1'$, $\widehat{v} \simeq_{\mathrm{v}} v$, $\widehat{C_1} \simeq_{\mathrm{c}} C_1$ and $\widehat{C_2} \simeq_{\mathrm{mc}} C_2$. Hence, $\widehat{\delta}' \simeq \delta'$.

□

Lemma 3
For all configurations $\widehat{\delta}$, δ, $\widehat{\delta}'$ and δ', and for any $n \geqslant 1$, if $\widehat{\delta} \simeq \delta$ then

$$\widehat{\delta} \Rightarrow^n_{\mathrm{sr}} \widehat{\delta}' \text{ if and only if } \delta \Rightarrow^n_{\mathrm{aux}} \delta' \text{ and } \widehat{\delta}' \simeq \delta'.$$

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 $\widehat{e}$, either both $eval_{\mathrm{sr}}(\widehat{e})$ and $eval_{aux}([\![\widehat{e}]\!])$ are undefined or there exist values $\widehat{v}$ and v such that $eval_{\mathrm{sr}}(\widehat{e}) = \widehat{v}$, $eval_{aux}([\![\widehat{e}]\!]) = v$, and $\widehat{v} \simeq_{\mathrm{v}} v$.

Proof
Since the initial configurations $\langle \widehat{e}, \widehat{\rho_{mt}}, \widehat{\mathsf{END}}, \widehat{\mathsf{nil}}\rangle_{\widehat{eval}}$ and $\langle [\![\widehat{e}]\!], \rho_{mt}, \mathsf{END}, \mathsf{nil}\rangle_{eval}$ are in the relation $\simeq$, then by Lemma 3 both abstract machines reach their final configurations $\langle\widehat{\mathsf{nil}}, \widehat{v}\rangle_{cont_2}$ and $\langle \mathsf{nil}, v\rangle_{cont_2}$ after the same number of transitions and with $\widehat{v} \simeq_{\mathrm{v}} v$, or both diverge. □

Theorem 1
For any program $\widehat{e}$, either both $eval_{\mathrm{sr}}(\widehat{e})$ and $eval_{\mathrm{cp}}([\![\widehat{e}]\!])$ are undefined or there exist values $\widehat{v}$ and v such that $eval_{\mathrm{sr}}(\widehat{e}) = \widehat{v}$, $eval_{\mathrm{cp}}([\![\widehat{e}]\!]) = v$, and $\widehat{v} \simeq_{\mathrm{v}} v$.

Proof
Follows directly from Proposition 1 and Proposition 2. □

Corollary 1 (Folklore)
For any program $\widehat{e}$, and for any integer n, $eval_{\mathrm{sr}}(\widehat{e}) = n$ if and only if $eval_{\mathrm{cp}}([\![\widehat{e}]\!]) = n$.

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 proof.

Our simple proof is based on the original (operational) specification of static and dynamic delimited continuations. An alternative proof could be based, for example, on equational reasoning (Felleisen, 1988; Kameyama & Hasegawa, 2003).

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. Several converse simulations have been presented recently (Shan, 2004; Biernacki *et al.*, 2005; Kiselyov, 2005). These converse simulations are considerably more involved than the present one, and have 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. Special thanks to the anonymous reviewers for an insightful e-mail exchange. This work is partially 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

Ariola, Z. M., Herbelin, H. and Sabry, A. (2004) A type-theoretic foundation of continuations and prompts. In: Fisher, K. (editor), *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, pp. 40–53. ACM Press.

Biernacka, M., Biernacki, D. and Danvy, O. (2005) *An operational foundation for delimited continuations in the CPS hierarchy*. Research Report BRICS RS-05-24. DAIMI, Department of Computer Science, University of Aarhus, Denmark. (To appear in *Logical Methods in Computer Science*. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).)

Biernacki, D., Danvy, O. and Millikin, K. (2005) *A dynamic continuation-passing style for dynamic delimited continuations*. Research Report BRICS RS-05-16. DAIMI, Department of Computer Science, University of Aarhus, Denmark.

Danvy, O. and Filinski, A. (1990) Abstracting control. In: Wand, M. (editor), *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160. Nice, France. ACM Press.

Dybvig, R. K., Peyton-Jones, S. and Sabry, A. (2005) *A monadic framework for subcontinuations*. Technical Report 615. Computer Science Department, Indiana University, Bloomington, Indiana.

Felleisen, M. (1988) The theory and practice of first-class prompts. In: Ferrante, J. and Mager, P. (editors), *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 180–190. San Diego, CA. ACM Press.

Felleisen, M. and Friedman, D. P. (1986) Control operators, the SECD machine, and the λ-calculus. In: Wirsing, M. (editor), *Formal Description of Programming Concepts III*, pp. 193–217. North-Holland.

Felleisen, M., Wand, M., Friedman, D. P. and Duba, B. F. (1988) Abstract continuations: A mathematical semantics for handling full functional jumps. In: Cartwright, R. (Corky) (editor), *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pp. 52–62. Snowbird, UT. ACM Press.

Gasbichler, M. and Sperber, M. (2002) Final shift for call/cc: direct implementation of shift and reset. In: Peyton Jones, S. (editor), *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, pp. 271–282. Pittsburgh, PA. ACM Press. (*SIGPLAN Notices*, **37**(9).)

Kameyama, Y. and Hasegawa, M. (2003) A sound and complete axiomatization of delimited continuations. In: Shivers, O. (editor), *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, pp. 177–188. Uppsala, Sweden. ACM Press.

Kelsey, R., Clinger, W. and Rees, J. (editors) (1998) Revised^5 report on the algorithmic language Scheme. *Higher-order & Symbolic Computation*, **11**(1), 7–105.

Kiselyov, O. (2005) *How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible*. Technical Report 611. Computer Science Department, Indiana University, Bloomington, Indiana.

Milne, R. E. and Strachey, C. (1976) *A Theory of Programming Language Semantics.* Chapman & Hall, London, and John Wiley, New York.

Plotkin, G. D. (1975) Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.* **1**, 125–159.

Shan, C.-c. (2004) Shift to control. In: Shivers, O. and Waddell, O. (editors), *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*. Technical report TR600, Computer Science Department, Indiana University.