BRICS RS-05-16 Biernacki et al.: A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations

Basic Research in Computer Science

A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations

Dariusz Biernacki
Olivier Danvy
Kevin Millikin

BRICS Report Series **RS-05-16**

ISSN 0909-0878 **May 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/05/16/`

A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations

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

May 2005

Abstract

We present a new abstract machine that accounts for dynamic delimited continuations. We prove the correctness of this new abstract machine with respect to a pre-existing, definitional abstract machine. Unlike this definitional abstract machine, the new abstract machine is in defunctionalized form, which makes it possible to state the corresponding higher-order evaluator. This evaluator is in continuation+state passing style and threads a trail of delimited continuations and a meta-continuation. Since this style accounts for dynamic delimited continuations, we refer to it as 'dynamic continuation-passing style.'

We show that the new machine operates more efficiently than the definitional one and that the notion of computation induced by the corresponding evaluator takes the form of a monad. We also present new examples and a new simulation of dynamic delimited continuations in terms of static ones.

*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,kmillikin`}`@brics.dk`

Contents

List of Figures

1 Introduction

The control operator `call/cc` [9, 26, 32, 39], by now, is an accepted component in the landscape of eager functional programming, where it provides the expressive power of CPS (continuation-passing style) in direct-style programs. An integral part of its success is its surrounding array of computational artifacts: simple motivating examples as well as more complex ones, a functional encoding in the form of a continuation-passing evaluator, the corresponding continuation-passing style and CPS transformation, their first-order counterparts (e.g., the corresponding abstract machine), and the continuation monad.

The delimited-control operators `control` (alias $\mathcal{F}$) and `prompt` (alias #) [20, 23, 41] were designed to go 'beyond continuations' [22]. This vision was investigated in the early 1990's [25,28,29,36,38,42] and today it is receiving renewed attention: Shan and Kiselyov are studying its simulation properties [33, 40], and Dybvig, Peyton Jones, and Sabry are proposing a general framework where multiple control delimiters can coexist [19], on the basis of Hieb, Dybvig, and Anderson's earlier work on 'subcontinuations' [29].

We observe, though, that none of these recent works on `control` and `prompt` uses the entire array of artifacts that organically surrounds `call/cc`. Our goal here is to fill this vacuum.

This work: We present a new abstract machine that accounts for dynamic delimited continuations and that is in defunctionalized form [18,39], and we prove its equivalence with a definitional abstract machine that is not in defunctionalized form. We also present the corresponding higher-order evaluator from which one can obtain the corresponding new CPS transformer. The resulting 'dynamic continuation-passing style' (dynamic CPS) threads a list of trailing delimited continuations, i.e., it is a continuation+state-passing style. This style is equivalent to, but simpler than the one recently proposed by Shan [40], and structurally related to the one recently proposed by Dybvig, Peyton Jones, and Sabry [19]. We also show that it corresponds to a computational monad, and we present some new examples.

Overview: We first present the definitional machine for dynamic delimited continuations in Section 2. We then present the new machine in Section 3, we establish their equivalence in Section 4, and we compare their efficiency in Section 5. The new machine is in defunctionalized form and we present the corresponding higher-order evaluator in Section 6. This evaluator is expressed in a dynamic continuation-passing style and we present the corresponding dynamic CPS transformer in Section 7 and the corresponding direct-style evaluator in Section 8. We illustrate dynamic continuation-passing style in Section 9 and in Section 10, we show that it can be characterized with a computational monad. In Section 11, we present a new simulation of `control` and `prompt` based on dynamic CPS. Finally, we address related work and conclude.

Prerequisites and notation: We assume some basic familiarity with operational semantics, abstract machines, eager functional programming in (Standard) ML, defunctionalization, and continuations.

2 The definitional abstract machine

In our earlier work [4], we obtained an abstract machine for the static delimited-control operators `shift` and `reset` by defunctionalizing a definitional evaluator that had two layered continuations [14, 15]. In this abstract machine, the first continuation takes the form of an evaluation context and the second takes the form of a stack of evaluation contexts. By construction, this abstract machine is an extension of Felleisen et al.'s CEK machine [21], which has one evaluation context and is itself a defunctionalized evaluator with one continuation [1, 2, 12, 39].

The abstract machine for static delimited continuations implements the application of a delimited continuation (represented as a captured context) by pushing the current context onto the stack of contexts and installing the captured context as the new current context [4]. In contrast, the abstract machine for dynamic delimited continuations implements the application of a delimited continuation (also represented as a captured context) by concatenating the captured context to the current context [23]. As a result, static and dynamic delimited continuations differ because a subsequent control operation will capture either the remainder of the reinstated context (in the static case) or the remainder of the reinstated context together with the then-current context (in the dynamic case). An abstract machine implementing dynamic delimited continuations therefore requires defining an operation to concatenate contexts.

Figure 1 displays the definitional abstract machine for dynamic delimited continuations, including the operation to concatenate contexts. It only differs from our earlier abstract machine for static delimited continuations [4, Figure 7 and Section 4.5] in the way captured delimited continuations are applied, by concatenating their representation with the representation of the current continuation (the shaded transition in Figure 1).[1]

Contexts form a monoid:

Proposition 1. *The operation $\star$ defined in Figure 1 satisfies the following properties:*

(1) $C_1 \star \mathsf{END} = C_1 = \mathsf{END} \star C_1$,

(2) $(C_1 \star C_1') \star C_1'' = C_1 \star (C_1' \star C_1'')$.

Proof. By induction on the structure of C_1. □

In the definitional machine, the constructors of contexts are not solely consumed in the *cont₁* transitions, but also by $\star$. Therefore, the definitional abstract machine is not in the range of defunctionalization [18, 39]: it does not correspond to a higher-order evaluator. In the next section, we present a new abstract machine that implements dynamic delimited continuations and is in the range of defunctionalization.

3 The new abstract machine

The definitional machine is not in the range of defunctionalization because of the concatenation of contexts. We therefore introduce a new component in the machine to avoid this concatenation. This new component, the *trail of contexts*, holds the then-current contexts that would have been concatenated to the captured context in the definitional machine. These then-current contexts are then reinstated in turn when the captured context completes. Together, the current context and the trail of contexts represent the current dynamic context. The final component of the machine holds a stack of dynamic contexts (represented as a list: nil denotes the empty list, the infix operator :: denotes list construction, and the infix operator @ denotes list concatenation, as in ML).

[1] In contrast, static delimited continuations are applied as follows:

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

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

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

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

e	$\Rightarrow_{def}$	$\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{eval}$
$\langle x,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle C_1,\, \rho(x),\, C_2\rangle_{cont_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle C_1,\, [x,\, e,\, \rho],\, C_2\rangle_{cont_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1,\rho),\ C_1),\, C_2\rangle_{eval}$
$\langle \#e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle e,\, \rho,\, \mathsf{END},\, C_1 :: C_2\rangle_{eval}$
$\langle \mathcal{F}x.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow_{def}$	$\langle e,\, \rho\{x \mapsto C_1\},\, \mathsf{END},\, C_2\rangle_{eval}$
$\langle \mathsf{END},\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow_{def}$	$\langle C_2,\, v\rangle_{cont_2}$
$\langle \mathsf{ARG}\,((e,\rho),\ C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow_{def}$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\ C_1),\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\ C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow_{def}$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow_{def}$	$\langle C_1' \star C_1,\, v,\, C_2\rangle_{cont_1}$
$\langle C_1 :: C_2,\, v\rangle_{cont_2}$	$\Rightarrow_{def}$	$\langle C_1,\, v,\, C_2\rangle_{cont_1}$
$\langle \mathsf{nil},\, v\rangle_{cont_2}$	$\Rightarrow_{def}$	v

Figure 1: The definitional call-by-value abstract machine
for the λ-calculus extended with $\mathcal{F}$ and #

Figure 2 displays the new abstract machine for dynamic delimited continuations. It only differs from the definitional abstract machine in the way dynamic contexts are represented (a context and a trail of contexts (represented as a list) instead of one concatenated context). In Section 4, we establish the equivalence of the two machines.

In the new machine, the constructors of contexts are solely consumed in the *cont₁* transitions. Therefore the new machine, unlike the definitional machine, is in the range of defunctionalization: it can be refunctionalized into a higher-order evaluator, which we present in Section 6.

- Terms: $e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}x.e$
- Values (closures and captured continuations): $v ::= [x,\, e,\, \rho] \mid [C_1,\, T_1]$
- Environments: $\rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((e, \rho),\, C_1) \mid \mathsf{FUN}\,(v,\, C_1)$
- Trail of contexts: $T_1 ::= \mathsf{nil} \mid C_1 :: T_1$
- Meta-contexts: $C_2 ::= \mathsf{nil} \mid (C_1, T_1) :: C_2$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow_{new}$	$\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil},\, \mathsf{nil}\rangle_{eval}$
$\langle x,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle C_1,\, \rho(x),\, T_1,\, C_2\rangle_{cont_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle C_1,\, [x,\, e,\, \rho],\, T_1,\, C_2\rangle_{cont_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle e_0,\, \rho,\, \mathsf{ARG}\,((e_1, \rho),\, C_1),\, T_1,\, C_2\rangle_{eval}$
$\langle \#e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle e,\, \rho,\, \mathsf{END},\, \mathsf{nil},\, (C_1, T_1) :: C_2\rangle_{eval}$
$\langle \mathcal{F}x.e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow_{new}$	$\langle e,\, \rho\{x \mapsto [C_1,\, T_1]\},\, \mathsf{END},\, \mathsf{nil},\, C_2\rangle_{eval}$
$\langle \mathsf{END},\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow_{new}$	$\langle T_1,\, v,\, C_2\rangle_{trail_1}$
$\langle \mathsf{ARG}\,((e, \rho),\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow_{new}$	$\langle e,\, \rho,\, \mathsf{FUN}\,(v,\, C_1),\, T_1,\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,([x,\, e,\, \rho],\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow_{new}$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, T_1,\, C_2\rangle_{eval}$
$\langle \mathsf{FUN}\,([C_1',\, T_1'],\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow_{new}$	$\langle C_1',\, v,\, T_1' \mathbin{@} (C_1 :: T_1),\, C_2\rangle_{cont_1}$
$\langle \mathsf{nil},\, v,\, C_2\rangle_{trail_1}$	$\Rightarrow_{new}$	$\langle C_2,\, v\rangle_{cont_2}$
$\langle C_1 :: T_1,\, v,\, C_2\rangle_{trail_1}$	$\Rightarrow_{new}$	$\langle C_1,\, v,\, T_1,\, C_2\rangle_{cont_1}$
$\langle (C_1, T_1) :: C_2,\, v\rangle_{cont_2}$	$\Rightarrow_{new}$	$\langle C_1,\, v,\, T_1,\, C_2\rangle_{cont_1}$
$\langle \mathsf{nil},\, v\rangle_{cont_2}$	$\Rightarrow_{new}$	v

Figure 2: A new call-by-value abstract machine
for the λ-calculus extended with $\mathcal{F}$ and $\#$

N.B.: The trail concatenation, in Figure 2, could be avoided by adding a new component to the machine—a meta-trail of pairs of contexts and trails, managed last-in, first-out—and the corresponding new transitions. A captured continuation would then be a triple of context, trail, and meta-trail, and applying it would require this meta-trail to be concatenated to the current trail. In turn, this concatenation could be avoided by adding a meta-meta-trail, etc. Because each of the meta^n-trails (for $n \geq 1$) but the last one has one point of consumption, they all are in defunctionalized form except the last one. Adding meta^n-trails amounts to trading space for time.

4 Equivalence of the definitional machine and of the new machine

We relate the configurations and transitions of the definitional abstract machine to those of the new abstract machine. As a diacritical convention [34], we annotate the components, configurations, and transitions of the definitional machine with a tilde ($\widetilde{\cdot}$). In order to relate a dynamic context of the new machine (a context and a trail of contexts) to a context of the definitional machine, we convert it into a context of the new machine:

Definition 1. *We define an operation $\widehat{\star}$, concatenating a new context and a trail of new contexts, by induction on its second argument:*

$$C_1 \mathbin{\widehat{\star}} \mathsf{nil} \stackrel{\text{def}}{=} C_1$$
$$C_1 \mathbin{\widehat{\star}} (C_1' :: T_1) \stackrel{\text{def}}{=} C_1 \star (C_1' \mathbin{\widehat{\star}} T_1)$$

Proposition 2. $C_1 \mathbin{\widehat{\star}} (C_1' :: T_1) = (C_1 \star C_1') \mathbin{\widehat{\star}} T_1$,

Proof. Follows from Definition 1 and from the associativity of $\star$ (Proposition 1(2)). □

Proposition 3. $(C_1 \mathbin{\widehat{\star}} T_1) \mathbin{\widehat{\star}} T_1' = C_1 \mathbin{\widehat{\star}} (T_1 \mathbin{@} T_1')$.

Proof. By induction on the structure of T_1. □

Definition 2. *We relate the definitional abstract machine and the new abstract machine with the following family of relations $\simeq$:*

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

(2) Values:

- *(a)* $[\widetilde{x}, \widetilde{e}, \widetilde{\rho}] \simeq_{\mathrm{v}} [x, e, \rho]$ *iff* $\widetilde{x} = x$, $\widetilde{e} \simeq_{\mathrm{e}} e$ *and* $\widetilde{\rho} \simeq_{\mathrm{env}} \rho$
- *(b)* $\widetilde{C_1} \simeq_{\mathrm{v}} [C_1, T_1]$ *iff* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \mathbin{\widehat{\star}} T_1$

(3) Environments:

- *(a)* $\widetilde{\rho_{mt}} \simeq_{\mathrm{env}} \rho_{mt}$
- *(b)* $\widetilde{\rho}\{x \mapsto \widetilde{v}\} \simeq_{\mathrm{env}} \rho\{x \mapsto v\}$ *iff* $\widetilde{v} \simeq_{\mathrm{v}} v$ *and* $\widetilde{\rho} \setminus \{x\} \simeq_{\mathrm{env}} \rho \setminus \{x\}$,
  where $\rho \setminus \{x\}$ *denotes the restriction of* ρ *to its domain excluding* x

(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* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$

(5) Meta-contexts:

- *(a)* $\widetilde{\mathsf{nil}} \simeq_{\mathrm{mc}} \mathsf{nil}$
- *(b)* $\widetilde{C_1} :: \widetilde{C_2} \simeq_{\mathrm{mc}} (C_1, T_1) :: C_2$ *iff* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \mathbin{\widehat{\star}} T_1$ *and* $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$

(6) Configurations:

(a) $\langle\widetilde{e},\, \widetilde{\rho},\, \widetilde{C_1},\, \widetilde{C_2}\rangle_{\widetilde{eval}} \simeq \langle e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$ *iff*
$\widetilde{e} \simeq_{\mathrm{e}} e$*,* $\widetilde{\rho} \simeq_{\mathrm{env}} \rho$*,* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \,\widehat{\star}\, T_1$*, and* $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$

(b) $\langle\widetilde{C_1},\, \widetilde{v},\, \widetilde{C_2}\rangle_{\widetilde{cont_1}} \simeq \langle C_1,\, v,\, T_1,\, C_2\rangle_{cont_1}$ *iff*
$\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \,\widehat{\star}\, T_1$*,* $\widetilde{v} \simeq_{\mathrm{v}} v$*, 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* $\widetilde{v} \simeq_{\mathrm{v}} v$

By writing $\delta \Rightarrow^{*} \delta'$, $\delta \Rightarrow^{+} \delta'$ and $\delta \Rightarrow^{1} \delta'$, we mean that there is respectively zero or more, one or more, and at most one transition leading from the configuration δ to the configuration δ'.

Definition 3. *The partial evaluation functions* $eval_{def}$ *and* $eval_{new}$ *mapping terms to values are defined as follows:*

(1) $eval_{def}\,(e) = v$ *if and only if* $\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil}\rangle_{eval} \Rightarrow^{+}_{def} \langle \mathsf{nil},\, v\rangle_{cont_2}$*,*

(2) $eval_{new}\,(e) = v$ *if and only if* $\langle e,\, \rho_{mt},\, \mathsf{END},\, \mathsf{nil},\, \mathsf{nil}\rangle_{eval} \Rightarrow^{+}_{new} \langle \mathsf{nil},\, v\rangle_{cont_2}$*.*

We want to prove that $eval_{def}$ and $eval_{new}$ are defined on the same programs (i.e., closed terms), and that for any given program, they yield equivalent values.

Theorem 1 (Equivalence). *For any program* e*,* $eval_{def}\,(e) = \widetilde{v}$ *if and only if* $eval_{new}\,(e) = v$ *and* $\widetilde{v} \simeq_{\mathrm{v}} v$*.*

Proving Theorem 1 requires proving the following lemmas.

Lemma 1. *If* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1$ *and* $\widetilde{C_1'} \simeq_{\mathrm{c}} C_1'$ *then* $\widetilde{C_1} \,\widetilde{\star}\, \widetilde{C_1'} \simeq_{\mathrm{c}} C_1 \star C_1'$*.*

Proof. By induction on the structure of $\widetilde{C_1}$. □

The following lemma addresses the configurations of the new abstract machine that break the one-to-one correspondence with the definitional abstract machine.

Lemma 2. *Let* $\delta = \langle \mathsf{END},\, v,\, T_1,\, C_2\rangle_{cont_1}$*.*

(1) If $T_1 = \underbrace{\mathsf{END} :: \ldots :: \mathsf{END}}_{n} :: \mathsf{nil}$*, where* $n \geq 0$*, then* $\delta \Rightarrow^{+}_{new} \langle C_2,\, v\rangle_{cont_2}$*.*

(2) If $T_1 = \underbrace{\mathsf{END} :: \ldots :: \mathsf{END}}_{n} :: C_1 :: T_1'$*, where* $n \geq 0$ *and* $C_1 \neq \mathsf{END}$*, then* $\delta \Rightarrow^{+}_{new} \langle C_1,\, v,\, T_1',\, C_2\rangle_{cont_1}$*.*

Proof. By induction on n. □

The following key lemma relates single transitions of the two abstract machines.

Lemma 3. *If* $\widetilde{\delta} \simeq \delta$ *then*

(1) if $\widetilde{\delta} \Rightarrow_{def} \widetilde{\delta}'$ *then there exists a configuration* δ' *such that* $\delta \Rightarrow^{+}_{new} \delta'$ *and* $\widetilde{\delta}' \simeq \delta'$*;*

(2) if $\delta \Rightarrow_{new} \delta'$ *then there exist configurations* $\widetilde{\delta}'$ *and* δ'' *such that* $\widetilde{\delta} \Rightarrow^{1}_{def} \widetilde{\delta}'$*,* $\delta' \Rightarrow^{*}_{new} \delta''$ *and* $\widetilde{\delta}' \simeq \delta''$*.*

Proof. By case analysis of $\widetilde{\delta} \simeq \delta$. Most of the cases follow directly from the definition of the relation $\simeq$. We show the proof of one such case:

Case: $\widetilde{\delta} = \langle \widetilde{x},\, \widetilde{\rho},\, \widetilde{C_1},\, \widetilde{C_2}\rangle_{\widetilde{eval}}$ and $\delta = \langle x,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$.
From the definition of the definitional abstract machine, $\widetilde{\delta} \Rightarrow_{def} \widetilde{\delta}'$, where
$\widetilde{\delta}' = \langle \widetilde{C_1},\, \widetilde{\rho}(\widetilde{x}),\, \widetilde{C_2}\rangle_{\widetilde{cont_1}}$.
From the definition of the new abstract machine, $\delta \Rightarrow_{new} \delta'$, where
$\delta' = \langle C_1,\, \rho(x),\, T_1,\, C_2\rangle_{cont_1}$.
By assumption, $\widetilde{\rho}(\widetilde{x}) \simeq_{\mathrm{v}} \rho(x)$, $\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \,\widehat{\star}\, T_1$ and $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$. Hence, $\widetilde{\delta}' \simeq \delta'$ and both directions of Lemma 3 are proved in this case.

There are only three interesting cases. One of them arises when a captured continuation is applied, and the remaining two explain why the two abstract machines do not operate in lockstep:

Case: $\widetilde{\delta} = \langle \widetilde{\mathsf{FUN}\,(\widetilde{C_1'},\, \widetilde{C_1})},\, \widetilde{v},\, \widetilde{C_2}\rangle_{\widetilde{cont_1}}$ and $\delta = \langle \mathsf{FUN}\,([C_1',\, T_1'],\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$
From the definition of the definitional abstract machine, $\widetilde{\delta} \Rightarrow_{def} \widetilde{\delta}'$, where
$\widetilde{\delta}' = \langle \widetilde{C_1'} \,\widetilde{\star}\, \widetilde{C_1},\, \widetilde{v},\, \widetilde{C_2}\rangle_{\widetilde{cont_1}}$.
From the definition of the new abstract machine, $\delta \Rightarrow_{new} \delta'$, where
$\delta' = \langle C_1',\, v,\, T_1' \,@\, (C_1 :: T_1),\, C_2\rangle_{cont_1}$.
By assumption, $\widetilde{C_1'} \simeq_{\mathrm{c}} C_1' \,\widehat{\star}\, T_1'$ and $\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \,\widehat{\star}\, T_1$.
By Lemma 1, we have $\widetilde{C_1'} \,\widetilde{\star}\, \widetilde{C_1} \simeq_{\mathrm{c}} (C_1' \,\widehat{\star}\, T_1') \star (C_1 \,\widehat{\star}\, T_1)$.
By the definition of $\widehat{\star}$, $(C_1' \,\widehat{\star}\, T_1') \star (C_1 \,\widehat{\star}\, T_1) = (C_1' \,\widehat{\star}\, T_1') \,\widehat{\star}\, (C_1 :: T_1)$.
By Proposition 3, $(C_1' \,\widehat{\star}\, T_1') \,\widehat{\star}\, (C_1 :: T_1) = C_1' \,\widehat{\star}\, (T_1' \,@\, (C_1 :: T_1))$.
Since $\widetilde{v} \simeq_{\mathrm{v}} v$ and $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$, we infer that $\widetilde{\delta}' \simeq \delta'$ and both directions of Lemma 3 are proved in this case.

Case: $\widetilde{\delta} = \langle \widetilde{\mathsf{END}},\, \widetilde{v},\, \widetilde{C_2}\rangle_{\widetilde{cont_1}}$ and $\delta = \langle \mathsf{END},\, v,\, T_1,\, C_2\rangle_{cont_1}$
From the definition of the definitional abstract machine, $\widetilde{\delta} \Rightarrow_{def} \widetilde{\delta}'$, where $\widetilde{\delta}' = \langle \widetilde{C_2},\, \widetilde{v}\rangle_{\widetilde{cont_2}}$.
By the definition of $\simeq$, $\widetilde{v} \simeq_{\mathrm{v}} v$, $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$, and $\widetilde{\mathsf{END}} \simeq_{\mathrm{c}} \mathsf{END} \,\widehat{\star}\, T_1$. Hence, it follows from the definition of $\simeq_{\mathrm{c}}$ that $\mathsf{END} \,\widehat{\star}\, T_1 = \mathsf{END}$, which is possible only when $T_1 = \underbrace{\mathsf{END} :: \ldots :: \mathsf{END}}_{n} :: \mathsf{nil}$ for some $n \geq 0$.
Then by Lemma 2(1), $\delta \Rightarrow^{+}_{new} \delta'$, where $\delta' = \langle C_2,\, v\rangle_{cont_2}$ and $\widetilde{\delta}' \simeq \delta'$, and both directions of the lemma are proved in this case.

Case: $\widetilde{\delta} = \langle \widetilde{C_1},\, \widetilde{v},\, \widetilde{C_2}\rangle_{\widetilde{cont_1}}$ and $\delta = \langle \mathsf{END},\, v,\, T_1,\, C_2\rangle_{cont_1}$, where $\widetilde{C_1} \neq \widetilde{\mathsf{END}}$.
By the definition of $\simeq$, $\widetilde{v} \simeq_{\mathrm{v}} v$, $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$, and $\widetilde{C_1} \simeq_{\mathrm{c}} \mathsf{END} \,\widehat{\star}\, T_1$. Hence, it follows from the definition of $\simeq_{\mathrm{c}}$ that $\mathsf{END} \,\widehat{\star}\, T_1 \neq \mathsf{END}$, which is possible only when $T_1 = \underbrace{\mathsf{END} :: \ldots :: \mathsf{END}}_{n} :: C_1 :: T_1'$ for some $n \geq 0$ and $C_1 \neq \mathsf{END}$. Then by Lemma 2(2), $\delta \Rightarrow^{+}_{new} \delta'$, where $\delta' = \langle C_1,\, v,\, T_1',\, C_2\rangle_{cont_1}$, $C_1 \neq \mathsf{END}$, and since $\mathsf{END} \,\widehat{\star}\, T_1 = C_1 \,\widehat{\star}\, T_1'$, we have $\widetilde{\delta} \simeq \delta'$. Therefore, we have proved part (2) of Lemma 3 and reduced the proof of part (1) to one of the trivial cases (not shown in the proof), where $\widetilde{\delta} \simeq \delta'$.

□

Given the relation between single-step transitions of the two abstract machines, it is straightforward to generalize it to the relation between their multi-step transitions.

Lemma 4. *If $\widetilde{\delta} \simeq \delta$ then*

(1) *if $\widetilde{\delta} \Rightarrow^{+}_{def} \widetilde{\delta}'$ then there exists a configuration δ' such that $\delta \Rightarrow^{+}_{new} \delta'$ and $\widetilde{\delta}' \simeq \delta'$;*

(2) *if $\delta \Rightarrow^{+}_{new} \delta'$ then there exist configurations $\widetilde{\delta}'$ and δ'' such that $\widetilde{\delta} \Rightarrow^{*}_{def} \widetilde{\delta}'$, $\delta' \Rightarrow^{*}_{new} \delta''$ and $\widetilde{\delta}' \simeq \delta''$.*

Proof. Both directions follow from Lemma 3 by induction on the number of transitions. □

We are now in position to prove the equivalence theorem.

Proof of Theorem 1. The initial configuration of the definitional abstract machine, i.e., $\langle e, \widetilde{\rho_{mt}}, \widetilde{\mathsf{END}}, \widetilde{\mathsf{nil}}\rangle_{\widetilde{eval}}$, and that of the new abstract machine, i.e., $\langle e, \rho_{mt}, \mathsf{END}, \mathsf{nil}, \mathsf{nil}\rangle_{eval}$, are in the relation $\simeq$. Therefore, if the definitional abstract machine reaches the final configuration $\langle\widetilde{\mathsf{nil}}, \widetilde{v}\rangle_{\widetilde{cont_2}}$, then by Lemma 4(1), there is a configuration δ' such that $\delta \Rightarrow^{+}_{new} \delta'$ and $\widetilde{\delta}' \simeq \delta'$. By the definition of $\simeq$, δ' must be $\langle \mathsf{nil}, v\rangle_{cont_2}$, with $\widetilde{v} \simeq_{\mathrm{v}} v$. The proof of the converse direction follows similar steps.

□

5 Efficiency issues

The new abstract machine implements the dynamic delimited control operators $\mathcal{F}$ and $\#$ more efficiently than the definitional abstract machine. The efficiency gain comes from allowing continuations to be implemented as lists of stack segments—which is generally agreed to be the most efficient implementation for first-class continuations [10, 11, 30]—without imposing a choice of representation on the stack segments.

In particular, when the definitional abstract machine applies a captured context C'_1 in a current context C_1, the new context is $C'_1 \star C_1$, and constructing it requires work proportional to the length of C'_1. In contrast, when the new abstract machine applies a captured context $[C'_1, T'_1]$ in a current context C_1 with a current trail of contexts T_1, the new trail is $T'_1 @ (C_1 :: T_1)$, and constructing it requires work proportional to the number of contexts (i.e., stack segments) in T'_1, independently of the length of each of these contexts. In the worst case, each context in the trail has length one and the new abstract machine does the same amount of work as the definitional machine. In all other cases it does less.

The following implementation of a list copy function (written in the syntax of ML) illustrates the situation:

```
fun list_copy1 xs
    = let fun visit nil
              = control (fn k => k nil)
            | visit (x :: xs)
              = x :: (visit xs)
      in prompt (fn () => visit xs)
      end
```

The initial call to `visit` is delimited by `prompt` (alias #), and in the base case, the (delimited) continuation is captured with `control` (alias $\mathcal{F}$). This delimited continuation is represented by a context whose size is proportional to the length of the list. In the definitional abstract machine, the entire context must be traversed and copied when invoked (i.e., immediately). In the new machine, only the (empty) trail of contexts is traversed and copied. Therefore, the definitional abstract machine does work proportional to the length of the input list, whereas the new abstract machine does the same work in constant time.

A small variation on the function above causes the definitional machine to perform an amount of work which is quadratic in the length of the input list, by copying contexts whose size is proportional to the length of the list on *every* recursive call:

```
fun list_copy2 xs
    = let fun visit nil
              = control (fn k => k nil)
            | visit (x :: xs)
              = x :: (control (fn k => k (visit xs)))
      in prompt (fn () => visit xs)
      end
```

In contrast to this quadratic behavior, the new abstract machine performs an amount of work that is linear in the length of the input list since it performs a constant amount of work at each application of a continuation (i.e., once per recursive call).

Implementing the composition of delimited continuations by concatenating their representations incurs an overhead proportional to the size of one of the delimited continuations, and is therefore subject to pathological situations such as the one illustrated in this section.

6 The evaluator corresponding to the new abstract machine

The *raison d'être* of the new abstract machine is that it is in defunctionalized form. Refunctionalizing the contexts and meta-contexts of the new abstract machine yields the higher-order evaluator of Figure 3. This evaluator is expressed in a continuation+state-passing style where the state consists of a trail of continuations and a meta-continuation, and defunctionalizing it gives the abstract machine of Figure 2. Since this continuation+state-passing style came into being to account for dynamic delimited continuations, we refer to it as a 'dynamic continuation-passing style' (dynamic CPS).

7 The CPS transformer corresponding to the new evaluator

The dynamic CPS transformer corresponding to the evaluator of Figure 3 can be immediately obtained as the associated syntax-directed encoding into the term model of the meta-language (using fresh variables):

$$
\begin{aligned}
[\![x]\!] &= \lambda(k_1, t_1, k_2).k_1\,(x, t_1, k_2)\\
[\![\lambda x.e]\!] &= \lambda(k_1, t_1, k_2).k_1\,(\lambda(x, k_1, t_1, k_2).\,[\![e]\!]\,(k_1, t_1, k_2), t_1, k_2)\\
[\![e_0\, e_1]\!] &= \lambda(k_1, t_1, k_2).[\![e_0]\!]\,(\lambda(v_0, t_1, k_2).\,[\![e_1]\!]\,(\lambda(v_1, t_1, k_2).\,v_0\,(v_1, k_1, t_1, k_2), t_1, k_2), t_1, k_2)\\
[\![\#e]\!] &= \lambda(k_1, t_1, k_2).[\![e]\!]\,(\theta_1, \mathsf{nil}, \lambda v.\,k_1\,(v, t_1, k_2))\\
[\![\mathcal{F}x.e]\!] &= \lambda(k_1, t_1, k_2).\mathit{let}\ x = \lambda(v, k_1', t_1', k_2).\,k_1\,(v, t_1 @ (k_1' :: t_1'), k_2)\\
&\qquad\qquad\qquad\quad \mathit{in}\ [\![e]\!]\,(\theta_1, \mathsf{nil}, k_2)
\end{aligned}
$$

It is straightforward to write a one-pass version of the dynamic CPS transformer [15], and we have implemented it. For example, transforming `list_copy1` (in Section 5) yields the following program, which we write in the syntax of ML:

```
datatype 'a cont1 = CONT1 of 'a * 'a trail1 * 'a cont2 -> 'a
withtype 'a trail1 = 'a cont1 list
     and 'a cont2 = 'a -> 'a
```

- Terms: $\mathsf{Exp} \ni e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}x.e$
- Answers, meta-continuations, continuations, values, and trails of continuations:

$$
\begin{array}{rcl}
\mathsf{Ans} & = & \mathsf{Val} \\
\theta_2, k_2 \in \mathsf{Cont}_2 & = & \mathsf{Val} \to \mathsf{Ans} \\
\theta_1, k_1 \in \mathsf{Cont}_1 & = & \mathsf{Val} \times \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans} \\
v \in \mathsf{Val} & = & \mathsf{Val} \times \mathsf{Cont}_1 \times \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans} \\
t_1 \in \mathsf{Trail}_1 & = & \mathsf{List}(\mathsf{Cont}_1)
\end{array}
$$

- Initial meta-continuation: $\theta_2 = \lambda v.\, v$
- Initial continuation: $\theta_1 = \lambda(v, t_1, k_2).\ \mathit{case}\ t_1$

$$
\begin{array}{l}
\mathit{of}\ \mathsf{nil} \Rightarrow k_2\, v \\
\mid k_1 :: t_1' \Rightarrow k_1\,(v, t_1', k_2)
\end{array}
$$

- Environments: $\mathsf{Env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation function: $\mathsf{eval} : \mathsf{Exp} \times \mathsf{Env} \times \mathsf{Cont}_1 \times \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Ans}$

$$
\begin{array}{rcl}
\mathsf{eval}_{\mathsf{dcps}}\,(x, \rho, k_1, t_1, k_2) & = & k_1\,(\rho(x), t_1, k_2) \\
\mathsf{eval}_{\mathsf{dcps}}\,(\lambda x.e, \rho, k_1, t_1, k_2) & = & k_1\,(\lambda(v, k_1, t_1, k_2).\, \mathsf{eval}_{\mathsf{dcps}}\,(e, \rho\{x \mapsto v\}, k_1, t_1, k_2), t_1, k_2) \\
\mathsf{eval}_{\mathsf{dcps}}\,(e_0\, e_1, \rho, k_1, t_1, k_2) & = & \mathsf{eval}_{\mathsf{dcps}}\,(e_0, \rho, \lambda(v_0, t_1, k_2).\, \mathsf{eval}_{\mathsf{dcps}}\,(e_1, \rho, \lambda(v_1, t_1, k_2).\, v_0\,(v_1, k_1, t_1, k_2), t_1, k_2), t_1, k_2) \\
\mathsf{eval}_{\mathsf{dcps}}\,(\#e, \rho, k_1, t_1, k_2) & = & \mathsf{eval}_{\mathsf{dcps}}\,(e, \rho, \theta_1, \mathsf{nil}, \lambda v.\, k_1\,(v, t_1, k_2)) \\
\mathsf{eval}_{\mathsf{dcps}}\,(\mathcal{F}x.e, \rho, k_1, t_1, k_2) & = & \mathsf{eval}_{\mathsf{dcps}}\,(e, \rho\{x \mapsto \lambda(v, k_1', t_1', k_2).\, k_1\,(v, t_1 \mathbin{@} (k_1' :: t_1'), k_2)\}, \theta_1, \mathsf{nil}, k_2)
\end{array}
$$

- Main function: $\mathsf{evaluate} : \mathsf{Exp} \to \mathsf{Val}$

$$
\mathsf{evaluate}_{\mathsf{dcps}}\,(e) = \mathsf{eval}_{\mathsf{dcps}}\,(e, \rho_{mt}, \theta_1, \mathsf{nil}, \theta_2)
$$

Figure 3: A call-by-value evaluator for the λ-calculus extended with $\mathcal{F}$ and $\#$

```
(*  theta1 : 'a * 'a trail1 * 'a cont2 -> 'a  *)
fun theta1 (v, nil, k2)
    = k2 v
  | theta1 (v, (CONT1 k1) :: t1, k2)
    = k1 (v, t1, k2)

(*  theta2 : 'a -> 'a  *)
fun theta2 v
    = v

(*  list_copy1_dcps : 'b list -> 'b list  *)
fun list_copy1_dcps xs
    = let (*  visit : 'b list * 'b list trail1 * 'b list cont2 -> 'b list  *)
          fun visit (nil, k1, t1, k2)
              = let fun k (v, k1', t1', k2)
                            = k1 (v, t1 @ ((CONT1 k1') :: t1'), k2)
                in k (nil, theta1, nil, k2)
                end
            | visit (x :: xs, k1, t1, k2)
              = visit (xs, fn (r, t1', k2') => k1 (x :: r, t1', k2'), t1, k2)
      in visit (xs, theta1, nil, theta2)
      end
```

or again, unfolding the inner let expression:

```
fun list_copy1_dcps_simplified xs
    = let fun visit (nil, k1, t1, k2)
              = k1 (nil, t1 @ ((CONT1 theta1) :: nil), k2)
            | visit (x :: xs, k1, t1, k2)
              = visit (xs, fn (r, t1', k2') => k1 (x :: r, t1', k2'), t1, k2)
      in visit (xs, theta1, nil, theta2)
      end
```

In our experience, out-of-the-box dynamic CPS programs are rarely enlightening the way normal CPS programs tend to be (at least after some practice). However, again in our experience, a combination of simplifications (e.g., inlining `k2` and `k_init` in the example just above) and defunctionalization often clarifies the intent and the behavior of the original direct-style program. We illustrate this point in Section 9.

8 The direct-style evaluator corresponding to the new evaluator

Figure 4 shows a direct-style evaluator for the λ-calculus extended with $\mathcal{F}$ and # written in a meta-language enriched with $\mathcal{F}$ and #. Transforming this direct-style evaluator into dynamic continuation-passing style, using the one-pass version of the dynamic CPS transformer of Section 7, yields the evaluator of Figure 3. This experiment is an adaptation of Danvy and Filinski's experiment [14], where a direct-style evaluator for the λ-calculus extended with shift and reset written in a meta-language extended with shift and reset was CPS-transformed into the definitional interpreter for the λ-calculus extended with shift and reset. (In the same spirit, Danvy and Lawall have transformed into direct style a continuation-passing evaluator for the λ-calculus extended with callcc, obtaining a direct-style evaluator interpreting callcc with callcc [16].)

- Terms: $\mathsf{Exp} \ni e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}x.e$
- Values: $v \in \mathsf{Val} = \mathsf{Val} \to \mathsf{Val}$
- Environments: $\mathsf{Env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation function: $\mathsf{eval} : \mathsf{Exp} \times \mathsf{Env} \to \mathsf{Ans}$

$$\begin{aligned}
\mathsf{eval}_{\mathsf{ds}}\,(x, \rho) &= \rho(x) \\
\mathsf{eval}_{\mathsf{ds}}\,(\lambda x.e, \rho) &= \lambda v.\, \mathsf{eval}_{\mathsf{ds}}\,(e, \rho\{x \mapsto v\}) \\
\mathsf{eval}_{\mathsf{ds}}\,(e_0\, e_1, \rho) &= \mathsf{eval}_{\mathsf{ds}}\,(e_0, \rho)\,(\mathsf{eval}_{\mathsf{ds}}\,(e_1, \rho)) \\
\mathsf{eval}_{\mathsf{ds}}\,(\#e, \rho) &= \#(\mathsf{eval}_{\mathsf{ds}}\,(e, \rho)) \\
\mathsf{eval}_{\mathsf{ds}}\,(\mathcal{F}x.e, \rho) &= \mathcal{F}v.\mathsf{eval}_{\mathsf{ds}}\,(e, \rho\{x \mapsto v\})
\end{aligned}$$

- Main function: $\mathsf{evaluate} : \mathsf{Exp} \to \mathsf{Val}$

$$\mathsf{evaluate}_{\mathsf{ds}}\,(e) = \mathsf{eval}_{\mathsf{ds}}\,(e, \rho_{mt})$$

Figure 4: A direct-style evaluator for the λ-calculus extended with $\mathcal{F}$ and #

9 Static and dynamic continuation-passing style

Biernacka, Biernacki, and Danvy have recently presented the following simple example to contrast the effects of `shift` and of `control` [4, Section 4.5]. We write it below in ML, using Filinski's implementation of `shift` and `reset` [24], and using the implementation of `control` and `prompt` presented in Section 11. In both cases, the type of the intermediate answers is `int list`:

```
(*  foo : int list -> int list  *)
fun foo xs
    = let fun visit nil
              = nil
            | visit (x :: xs)
              = visit (shift (fn k => x :: (k xs)))
      in reset (fn () => visit xs)
      end

(*  bar : int list -> int list  *)
fun bar xs
    = let fun visit nil
              = nil
            | visit (x :: xs)
              = visit (control (fn k => x :: (k xs)))
      in prompt (fn () => visit xs)
      end
```

The two functions traverse their input list recursively, and construct an output list. They only differ in that to abstract the recursive call to `visit` into a delimited continuation, `foo` uses `shift` and `reset` whereas `bar` uses `control` and `prompt`. This seemingly minor difference has a major effect since it makes `foo` behave as a *list-copying* function and `bar` as a *list-reversing* function.

To illustrate this difference of behavior, Biernacka, Biernacki, and Danvy have used contexts and meta-contexts [4, Section 4.5], and Biernacki and Danvy have used an intuitive source representation of the successive contexts [5, Section 2.3]. In this section, we use static and dynamic continuation-passing style to illustrate the difference of behavior.

9.1 Static continuation-passing style

Applying the canonical CPS transformation for `shift` and `reset` [14] to the definition of `foo` yields the following purely functional program:

```
fun foo_scps xs
    = let fun visit (nil, k1, k2)
              = k1 (nil, k2)
            | visit (x :: xs, k1, k2)
              = let fun k (v, k1', k2')
                        = visit (v, k1, fn v => k1' (v, k2'))
                in k (xs, fn (v, k2) => k2 (x :: v), k2)
                end
      in visit (xs, fn (v, k2) => k2 v, fn v => v)
      end
```

Inlining `k`, `k1'`, and `k1` yields the following simpler program:

```
fun foo_scps_simplified xs
    = let fun visit (nil, k2)
              = k2 nil
            | visit (x :: xs, k2)
              = visit (xs, fn v => k2 (x :: v))
      in visit (xs, fn v => v)
      end
```

Defunctionalizing `k2` yields the following first-order program:

```
fun foo_scps_defunct xs
    = let fun visit (nil, k2)
              = continue (k2, nil)
            | visit (x :: xs, k2)
              = visit (xs, x :: k2)
          and continue (nil, v)
              = v
            | continue (x :: k2, v)
              = continue (k2, x :: v)
      in visit (xs, nil)
      end
```

These equivalent views make it clearer that the program copies its input list by first reversing it using the meta-continuation as an accumulator, and then by reversing the accumulator.

9.2 Dynamic continuation-passing style

Applying the dynamic CPS transformation for `control` and `prompt` (Section 7) to the definition of `bar` yields the following purely functional program:

```
datatype 'a cont1 = CONT1 of 'a * 'a trail1 * 'a cont2 -> 'a
withtype 'a trail1 = 'a cont1 list
     and 'a cont2 = 'a -> 'a

fun theta1 (v, nil, k2)
    = k2 v
  | theta1 (v, (CONT1 k1) :: t1, k2)
    = k1 (v, t1, k2)

fun theta2 v
    = v

fun bar_dcps xs
    = let fun visit (nil, k1, t1, k2)
              = k1 (nil, t1, k2)
            | visit (x :: xs, k1, t1, k2)
              = let fun k (v, k1', t1', k2)
                        = visit (v, k1, t1 @ (k1' :: t1'), k2)
                in k (xs, CONT1 (fn (v, t1, k2) => theta1 (x :: v, t1, k2)),
                      nil, k2)
                end
      in visit (xs, theta1, nil, theta2)
      end
```

Inlining `k`, `k1'`, `k1`, and `k2`, defunctionalizing the continuation into the ML `option` type, and using an auxiliary function `continue_aux` to interpret the trail, yields the following first-order program:

```
fun bar_dcps_defunct xs
    = let fun visit (nil, t1)
              = continue (NONE, nil, t1)
            | visit (x :: xs, t1)
              = visit (xs, t1 @ ((SOME x) :: nil))
          and continue (NONE, v, t1)
              = continue_aux (t1, v)
            | continue (SOME x, v, t1)
              = continue (NONE, x :: v, t1)
          and continue_aux (nil, v)
              = v
            | continue_aux (k1 :: t1, v)
              = continue (k1, v, t1)
      in visit (xs, nil)
      end
```

Further simplifications lead one to the following program:

```
fun bar_dcps_defunct_simplified xs
    = let fun visit (nil, t1)
              = continue_aux (t1, nil)
            | visit (x :: xs, t1)
              = visit (xs, t1 @ (x :: nil))
          and continue_aux (nil, v)
              = v
            | continue_aux (k1 :: t1, v)
              = continue_aux (t1, k1 :: v)
      in visit (xs, nil)
      end
```

These equivalent views make it clearer that the program reverses its input list by first copying it to the trail through a series of concatenations, and then by reversing the trail.

9.3 A generalization

Let us briefly generalize the programming pattern above from lists to binary trees:

```
datatype tree = EMPTY
              | NODE of tree * int * tree
```

In the following two definitions, the type of the intermediate answers is `int list`:

- Here, the two recursive calls to `visit` are abstracted into a static delimited continuation using `shift` and `reset`:

  ```
  fun traverse_sr t
      = let fun visit (EMPTY, a)
                  = a
              | visit (NODE (t1, i, t2), a)
                  = visit (t1, visit (t2, shift (fn k => i :: (k a))))
        in reset (fn () => visit (t, nil))
        end
  ```

- Here, the two recursive calls to `visit` are abstracted into a dynamic delimited continuation using `control` and `prompt`:

  ```
  fun traverse_cp t
      = let fun visit (EMPTY, a)
                  = a
              | visit (NODE (t1, i, t2), a)
                  = visit (t1, visit (t2, control (fn k => i :: (k a))))
        in prompt (fn () => visit (t, nil))
        end
  ```

The static delimited continuations yield a *preorder* and *right-to-left* traversal, whereas the dynamic delimited continuation yield a *postorder* and *left-to-right* traversal. The resulting two lists are reverse of each other.

Again, CPS transformation and defunctionalization yield first-order programs whose behavior is more patent.

9.4 Further examples

We now turn to the lazy depth-first and breadth-first traversals recently presented by Biernacki, Danvy, and Shan [6]. To support laziness, they used the following signature of generators:

```
signature GENERATOR
= sig
    type 'a computation
    datatype sequence = END
                      | NEXT of int * sequence computation

    val make_sequence : tree -> sequence
    val compute : sequence computation -> sequence
  end
```

The following generator is parameterized by a scheduler that is given four thunks to be applied in turn:

```
structure Lazy_generator : GENERATOR
= struct
    datatype sequence = END
                      | NEXT of int * sequence computation
    withtype 'a computation = unit -> 'a

    structure CP = Control_and_Prompt (type intermediate_answer = sequence)

    fun visit EMPTY
        = ()
      | visit (NODE (t1, i, t2))
        = CP.control (fn k => (schedule
                                (fn () => visit t1,
                                 fn () => CP.control (fn k' => NEXT (i, k')),
                                 fn () => visit t2,
                                 k);
                               END))

    fun make_sequence t
        = CP.prompt (fn () => let val () = visit t
                              in END
                              end)

    fun compute k
        = CP.prompt (fn () => k ())
  end
```

The relative scheduling of the first and third thunks determines whether the traversal of the input tree is from left to right or from right to left. The relative scheduling of the second thunk with respect to the first and the third determines whether the traversal is preorder, inorder, or postorder. The relative scheduling of the fourth thunk determines whether the traversal is depth-first, breadth-first, or a mix of both.

In each case, dynamic CPS transformation and defunctionalization yield first-order programs whose behavior is patent in that the depth-first traversal uses a stack, the breadth-first traversal uses a queue, and the mixed traversal uses a queue to hold the right (respectively the left) subtrees while visiting the left (respectively the right) ones.

10 A monad for dynamic continuation-passing style

The evaluator of Figure 3 is compositional, and has the following type:

$$\mathsf{Exp} \times \mathsf{Env} \times \mathsf{Cont}_1 \times \mathsf{Trail}_1 \times \mathsf{Cont}_2 \rightarrow \mathsf{Ans}$$

Let us curry it to exhibit its notion of computation [35]:

$$\mathsf{Exp} \times \mathsf{Env} \rightarrow \mathsf{Cont}_1 \rightarrow \mathsf{Trail}_1 \times \mathsf{Cont}_2 \rightarrow \mathsf{Ans}$$

Proposition 4. *The type constructor*

$$D(\mathsf{Val}) \;=\; \mathsf{Cont}_1 \rightarrow \mathsf{Trail}_1 \times \mathsf{Cont}_2 \rightarrow \mathsf{Ans}$$

$$\begin{array}{rrcl} \textit{where} & \mathsf{Ans} & = & \mathsf{Val} \\ & \mathsf{Cont}_2 & = & \mathsf{Val} \rightarrow \mathsf{Ans} \\ & \mathsf{Cont}_1 & = & \mathsf{Val} \rightarrow \mathsf{Trail}_1 \times \mathsf{Cont}_2 \rightarrow \mathsf{Ans} \\ & \mathsf{Val} & = & \mathsf{Val} \rightarrow \mathsf{Cont}_1 \rightarrow \mathsf{Trail}_1 \times \mathsf{Cont}_2 \rightarrow \mathsf{Ans} \\ & \mathsf{Trail}_1 & = & \mathsf{List}(\mathsf{Cont}_1) \end{array}$$

together with the functions

$$\begin{array}{rcl} \mathsf{unit} & : & \mathsf{Val} \rightarrow D(\mathsf{Val}) \\ \mathsf{unit}\,(v) & = & \lambda k_1 .\, \lambda(t_1, k_2).\, k_1\, v\, (t_1, k_2) \\ \\ \mathsf{bind} & : & D(\mathsf{Val}) \times (\mathsf{Val} \rightarrow D(\mathsf{Val})) \rightarrow D(\mathsf{Val}) \\ \mathsf{bind}\,(c, f) & = & \lambda k_1 .\, \lambda(t_1, k_2).\, c\, (\lambda v.\, \lambda(t_1', k_2').\, f\, v\, k_1\, (t_1', k_2'))\, (t_1, k_2) \end{array}$$

form a continuation+state monad, where the state pairs the trail of continuations and the meta-continuation. (The state could be η-reduced in the definitions of unit *and* bind*, yielding the definition of the continuation monad.)*

Proof. A simple equational verification of the three monad laws [35]. □

As in Wadler's study of monads and static delimited continuations [44], the type of bind, instead of the usual $D(\alpha) \times (\alpha \rightarrow D(\beta)) \rightarrow D(\beta)$, has $\alpha = \beta = \mathsf{Val}$, making the triple ($D$, unit, bind) more specific than a monad. As in Wadler's work, this extra specificity is coincidental here since we consider only one type, Val.

Having identified the monad for dynamic continuation-passing style, we are now in position to define `control` and `prompt` as operations in this monad:

Definition 4. *We define the monad operations* control*,* prompt *and* compute *as follows:*

$$\begin{array}{rcl} \mathsf{prompt} & : & D(\mathsf{Val}) \rightarrow D(\mathsf{Val}) \\ \mathsf{prompt}\,(c) & = & \lambda k_1 .\, \lambda(t_1, k_2).\, c\, \theta_1\, (\mathsf{nil}, \lambda v.\, k_1\, v\, (t_1, k_2)) \\ \\ \mathsf{control} & : & ((\mathsf{Val} \rightarrow D(\mathsf{Val})) \rightarrow D(\mathsf{Val})) \rightarrow D(\mathsf{Val}) \\ \mathsf{control}\,(e) & = & \lambda k_1 .\, \lambda(t_1, k_2).\, e\, k\, \theta_1\, (\mathsf{nil}, k_2) \\ & & \textit{where } k = \lambda v.\, \lambda k_1' .\, \lambda(t_1', k_2').\, k_1\, v\, (t_1 \mathbin{@} (k_1' :: t_1'), k_2') \\ \\ \mathsf{compute} & : & D(\mathsf{Val}) \rightarrow \mathsf{Val} \\ \mathsf{compute}\,(c) & = & c\, \theta_1\, (\mathsf{nil}, \theta_2) \end{array}$$

We can now extend the usual call-by-value monadic evaluator for the λ-calculus to $\mathcal{F}$ and # (see Figure 5). Inlining the abstraction layer provided by the monad yields the evaluator of Figure 3. Dynamic continuation-passing style therefore fits the functional correspondence between evaluators and abstract machines advocated by the first two authors [1,2,13]. Furthermore, and as has been observed before for other CPS transformations and for the continuation monad [27,43], the dynamic CPS transformation itself can be factored through Moggi's monadic metalanguage and the monad above.

11 A new implementation of `control` and `prompt`

As pointed out in Section 10, if one curries the evaluator of Figure 3 and η-reduces the parameters t_1 and k_2 in the first three clauses interpreting the λ-calculus, one can observe that dynamic CPS conservatively extends CPS. Therefore, since the continuations k_1 live in the continuation monad, it is straightforward to express `control` and `prompt` in terms of `shift` and `reset`, essentially, by

- Terms: $\mathsf{Exp} \ni e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}x.e$
- Values: $v \in \mathsf{Val} = \mathsf{Val} \rightarrow D(\mathsf{Val})$
- Environments: $\mathsf{Env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Evaluation function: $\mathsf{eval} : \mathsf{Exp} \times \mathsf{Env} \rightarrow D(\mathsf{Val})$

$$\begin{aligned}
\mathsf{eval}_{\mathsf{mon}}\,(x, \rho) &= \mathsf{unit}\,(\rho(x)) \\
\mathsf{eval}_{\mathsf{mon}}\,(\lambda x.e, \rho) &= \mathsf{unit}\,(\lambda v.\, \mathsf{eval}_{\mathsf{mon}}\,(e, \rho\{x \mapsto v\})) \\
\mathsf{eval}_{\mathsf{mon}}\,(e_0\, e_1, \rho) &= \mathsf{bind}\,(\mathsf{eval}_{\mathsf{mon}}\,(e_0, \rho), \lambda v_0.\, \mathsf{bind}\,(\mathsf{eval}_{\mathsf{mon}}\,(e_1, \rho), \lambda v_1.\, v_0\, v_1)) \\
\mathsf{eval}_{\mathsf{mon}}\,(\#e, \rho) &= \mathsf{prompt}\,(\mathsf{eval}_{\mathsf{mon}}\,(e, \rho)) \\
\mathsf{eval}_{\mathsf{mon}}\,(\mathcal{F}x.e, \rho) &= \mathsf{control}\,(\lambda v.\, \mathsf{eval}_{\mathsf{mon}}\,(e, \rho\{x \mapsto v\}))
\end{aligned}$$

- Main function: $\mathsf{evaluate} : \mathsf{Exp} \rightarrow \mathsf{Val}$

$$\mathsf{evaluate}_{\mathsf{mon}}\,(e) = \mathsf{compute}\,(\mathsf{eval}_{\mathsf{mon}}\,(e, \rho_{mt}))$$

Figure 5: A monadic evaluator for the λ-calculus extended with $\mathcal{F}$ and #

(1) transforming the evaluator of Figure 3 into direct style with respect to k_2 (the result is in continuation-composing style [14]), and

(2) transforming the resulting evaluator into direct style with respect to k_1 (as opposed to Section 8, where in order to obtain the direct-style evaluator written with `control` and `prompt`, we transformed the resulting evaluator into direct style with respect to both k_1 and t_1).

Building on this observation, we present below an implementation of `control` and `prompt` in Standard ML of New Jersey, based on Filinski's implementation of `shift` and `reset` [24]. The implementation takes the form of a functor mapping a type of intermediate answers to an instance of `control` and `prompt` at that type:

```
signature CONTROL_AND_PROMPT
= sig
    type intermediate_answer
    val control : (('a -> intermediate_answer) -> intermediate_answer) -> 'a
    val prompt : (unit -> intermediate_answer) -> intermediate_answer
  end

functor Control_and_Prompt (type intermediate_answer) : CONTROL_AND_PROMPT
= struct
    datatype ('a, 'b) cont1 = CONT1 of 'a -> 'b trail1 -> 'b
    withtype 'b trail1 = ('b, 'b) cont1 list

    structure SR
    = Shift_and_Reset
       (type intermediate_answer
             = intermediate_answer trail1 -> intermediate_answer)

    type intermediate_answer = intermediate_answer
```

```
  fun theta1 v nil
      = v
    | theta1 v ((CONT1 k) :: t)
      = k v t

  fun prompt thunk
      = SR.reset (fn () => theta1 (thunk ())) nil

  exception MISSING_PROMPT

  fun control function
      = SR.shift
            (fn k1 =>
              fn t1 =>
               let val f = fn v =>
                            SR.shift
                                (fn k1' =>
                                  fn t1' =>
                                   k1 v (t1 @ (CONT1 k1' :: t1')))
               in SR.reset (fn () => theta1 (function f)) nil
               end) handle MISSING_RESET => raise MISSING_PROMPT
end
```

In the definition of `prompt`, the expression delayed in `thunk` is computed with the initial continuation `theta1` and with an empty trail of continuations.

In the definition of `control`, the continuations `k1` and `k1'` are captured with `shift`, the function `f` is constructed according to its definition in the evaluator, and the application `function f` is computed with the initial continuation `theta1` and with an empty trail of continuations.

Hence, the standard continuation semantics of the definitions above coincides with the dynamic continuation semantics of `prompt` and `control` given by the evaluator. A more formal justification, however, is out of scope here.

12 Related work

The concept of meta-continuation and its representation as a function originates in Wand and Friedman's formalization of reflective towers [46], and its representation as a list in Danvy and Malmkjær's followup study [17]. Danvy and Filinski then realized that a meta-continuation naturally arises by "one more" CPS transformation, giving rise to success and failure continuations [14], and later Danvy and Nielsen observed that the list representation naturally arises by defunctionalization [18]. Just as repeated CPS transformations give rise to a static CPS hierarchy [4, 14, 31, 37], repeated dynamic CPS transformations should give rise to a dynamic CPS hierarchy—a future work.

The original approaches to delimited continuations were split between composing continuations dynamically by concatenating their representations [23] and composing them statically using continuation-passing function composition [14]. Recently, Shan [40] and Dybvig, Peyton Jones, and Sabry [19] each have proposed an account of dynamic delimited continuations using a continuation+state-passing style:

- Shan's development extends Wand et al.'s idea of an algebra of contexts [23] (the state represents the prefix of a meta-continuation and is equipped with algebraic operators *Send* and *Compose* to propagate intermediate results and compose the representation of delimited continuations). Like our dynamic continuation-passing style, Shan's continuation-passing style hinges on the requirement that the answer type of continuations be recursive. Our dynamic continuation-passing style also uses a state, namely a trail of contexts and a meta-continuation. This representation, however, only requires the usual list operations, instead of the dedicated algebraic operations provided by *Send* and *Compose*. Consequently, the abstract machine of Section 3 is simpler than the abstract machine corresponding to Shan's continuation-passing style. (We have constructed this abstract machine.) Shan's transformation can account for two other variations on $\mathcal{F}$. Our continuation-passing style can be adapted to account for these as well, by defunctionalizing the meta-continuation.

- Dybvig, Peyton Jones, and Sabry's continuation+state-passing style threads a state which is a list of continuations annotated with multiple control delimiters. This state is structurally similar to ours in the sense that defunctionalizing and flattening our meta-continuation and appending it to our trail of continuations yields their state without annotations. We find this coincidence of result remarkable considering the difference of motivation and methodology:
  - Dybvig, Peyton Jones, and Sabry sought "a typed monadic framework in which one can define and experiment with arbitrary [delimited] control operators" [19, Section 7], using Hieb, Dybvig, and Anderson's control operators for subcontinuations [29] as a common basis, whereas
  - we wanted an abstract machine for dynamic delimited continuations that is in the range of Reynolds's defunctionalization in order to provide a consistent spectrum of tools for programming with and reasoning about delimited continuations, both in direct style and in continuation-passing style.

Finally, as an alternative to Wand and Friedman's use of a meta-continuation [46], Bawden has used trampolining to investigate reflective towers [3]. Recently, Kiselyov has revisited trampolining to study the expressivity of static and dynamic delimited-continuation operators [33].

13 Conclusion and issues

In our earlier work [6], we argued that dynamic delimited continuations need examples, reasoning tools, and meaning-preserving program transformations, not only new variations, new formalizations, or new implementations. The present work partly fulfills these wishes by providing, in a concerted way, an abstract machine that is in defunctionalized form, the corresponding evaluator, the corresponding continuation-passing style and CPS transformer, a monadic account of this continuation-passing style, a new simulation of a dynamic delimited-control operator in terms of a static one, and several new examples.

Compared to static delimited continuations, and despite recent implementation advances, the topic of dynamic delimited continuations still remains largely unexplored. We believe that the spectrum of compatible computational artifacts presented here—abstract machine, evaluator, computational monad, and dynamic continuation-passing style—puts one in a better position to assess them.

Acknowledgments: We are grateful to Mads Sig Ager, Małgorzata Biernacka, Julia Lawall, Kristian Støvring, and the anonymous reviewers of MFPS XXI for their comments.

This work is supported by the ESPRIT Working Group APPSEM (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, August 2003.

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 2005. To appear. Extended version available as the technical report BRICS RS-04-28.

[3] Alan Bawden. Reification without evaluation. In Cartwright [8], pages 342–351.

[4] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy (revised version). Research Report BRICS RS-05-11, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW 2004).

[5] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. Research Report BRICS RS-05-10, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2005.

[6] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the dynamic extent of delimited continuations. Research Report BRICS RS-05-13, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, April 2005. Extended version of an article to appear in Information Processing Letters.

[7] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[8] Robert (Corky) Cartwright, editor. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988. ACM Press.

[9] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[10] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

[11] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 88–103, Berlin, Germany, March 2000. Springer-Verlag.

[12] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk.

[13] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck and Frank Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin price. Extended version available as the technical report BRICS-RS-03-33.

[14] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [45], pages 151–160.

[15] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[16] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.

[17] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In Cartwright [8], pages 327–341.

[18] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[19] 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.

[20] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.

[21] 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.

[22] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, February 1987.

[23] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Cartwright [8], pages 52–62.

[24] Andrzej Filinski. Representing monads. In Boehm [7], pages 446–457.

[25] Carl Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, La Jolla, California, June 1995. ACM Press.

[26] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[27] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [7], pages 458–471.

[28] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, SIGPLAN Notices, Vol. 25, No. 3, pages 128–136, Seattle, Washington, March 1990. ACM Press.

[29] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[30] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[31] Yukiyoshi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 442–457, Karpacz, Poland, September 2004. Springer.

[32] 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.

[33] Oleg Kiselyov. 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, March 2005.

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

[35] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[36] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 182–197, Madrid, Spain, September 1994. Springer-Verlag.

[37] Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn L. Talcott, editors, *Proceedings*

of the First ACM SIGPLAN Workshop on Continuations (CW 1992), Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.

[38] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, Orlando, Florida, January 1991. ACM Press.

[39] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[40] 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.

[41] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

[42] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [45], pages 161–175.

[43] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[44] Philip Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, January 1994.

[45] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[46] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A nonreflective description of the reflective tower. In William L. Scherlis and John H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 298–307, Cambridge, Massachusetts, August 1986. ACM Press.

Recent BRICS Report Series Publications

RS-05-16 Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. ***A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations*****. May 2005. ii+24 pp.**

RS-05-15 Małgorzata Biernacka and Olivier Danvy. ***A Concrete Framework for Environment Machines*****. May 2005. ii+25 pp.**

RS-05-14 Olivier Danvy and Henning Korsholm Rohde. ***On Obtaining the Boyer-Moore String-Matching Algorithm by Partial Evaluation*****. April 2005. ii+8 pp.**

RS-05-13 Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. ***On the Dynamic Extent of Delimited Continuations*****. April 2005. ii+32 pp. Extended version of an article to appear in** ***Information Processing Letters*****. Subsumes BRICS RS-05-2.**

RS-05-12 Małgorzata Biernacka, Olivier Danvy, and Kristian Støvring. ***Program Extraction from Proofs of Weak Head Normalization*****. April 2005. 19 pp. Extended version of an article to appear in the preliminary proceedings of MFPS XXI, Birmingham, UK, May 2005.**

RS-05-11 Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. ***An Operational Foundation for Delimited Continuations in the CPS Hierarchy*****. March 2005. iii+42 pp. A preliminary version appeared in Thielecke, editor,** ***4th ACM SIGPLAN Workshop on Continuations*****, CW '04 Proceedings, Association for Computing Machinery (ACM) SIGPLAN Technical Reports CSR-04-1, 2004, pages 25–33. This version supersedes BRICS RS-04-29.**

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.**