A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations

Dariusz Biernacki
Olivier Danvy
Kevin Millikin

BRICS Report Series RS-06-15

ISSN 0909-0878 October 2006

A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations[*]

Dariusz Biernacki[†], Olivier Danvy, and Kevin Millikin
BRICS[‡]
Department of Computer Science
University of Aarhus[§]

October 2006

Abstract

We put a pre-existing definitional abstract machine for dynamic delimited continuations in defunctionalized form, and we present the consequences of this adjustment.

We first prove the correctness of the adjusted abstract machine. Because it is in defunctionalized form, we can refunctionalize it into a higher-order evaluation function. This evaluation function, which is compositional, 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' and we present the corresponding dynamic CPS transformation. We show that the notion of computation induced by dynamic CPS takes the form of a continuation monad with a recursive answer type and we present a new simulation of dynamic delimited continuations in terms of static ones as well as new applications of dynamic delimited continuations.

The significance of the present work is that the computational artifacts surrounding dynamic CPS (i.e., simple motivating examples as well as more complex applications and simulations, a functional encoding in the form of a continuation-passing evaluator, the corresponding CPS transformation, their first-order counterparts, and the continuation monad) are not independent designs: they are mechanical consequences of having put the definitional abstract machine in defunctionalized form.

Technical Report BRICS RS-06-15
`http://www.brics.dk/RS/06/Abs/BRICS-RS-06-Abs/BRICS-RS-06-Abs.html#BRICS-RS-06-15`

---

[*]Revised version of BRICS RS-05-16.

[†]Current affiliation: INRIA Futurs, Orsay, France.

[‡]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` [11, 35, 42, 49], 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 applications, 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 #) [27, 30, 53] were designed to go 'beyond continuations' [29]. This vision was investigated in the early 1990's [34, 37, 38, 46, 48, 54] and today it is receiving renewed attention: Shan and Kiselyov are studying its simulation properties [43, 52], and Dybvig, Peyton Jones, and Sabry are proposing a general framework where multiple control delimiters can coexist [25].

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 do so.

This work: We present an abstract machine that accounts for dynamic delimited continuations and that is in defunctionalized form [23, 49], 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 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 [52]. It is structurally related to the one recently proposed by Dybvig, Peyton Jones, and Sabry [25]. 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. In Section 3, we put the definitional machine into defunctionalized form and we establish the equivalence of the two machines in Section 4. We present the corresponding higher-order evaluator, which is compositional, in Section 5. This evaluator is expressed in a dynamic continuation-passing style and we present the corresponding dynamic CPS transformer in Section 6 and the corresponding direct-style evaluator in Section 7: transforming this direct-style evaluator into dynamic CPS yields the evaluator of Section 5. We illustrate dynamic continuation-passing style in Section 8 and in Section 9, we show that it can be characterized with a computational monad: macro-expanding the definition of this monad into a monadic evaluator and CPS transforming the result yields a curried version of the evaluator of Section 5. In Section 10, we present a new simulation of `control` and `prompt` based on dynamic CPS. We then address related work and conclude. In Appendices A and B we consider the abstract machines corresponding to the control operators `control0` and `prompt0`. Finally, for completeness, we give an ML implementation of `shift` and `reset` in Appendix C.

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 [18, 19]. 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 [28], which has one evaluation context and is itself a defunctionalized evaluator with one continuation [2, 3, 13, 49].

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 [30]. 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 a priori 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] Biernacka and Danvy present the corresponding calculus elsewhere [5, Section 6.2].

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 [16, 23, 49]: 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 adjusted 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:

$$\boxed{\langle \mathsf{FUN}\,(C_1',\, C_1),\, v,\, C_2\rangle_{\mathit{cont_1}} \quad\Rightarrow\quad \langle C_1',\, v,\, C_1 :: C_2\rangle_{\mathit{cont_1}}}$$

- 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:

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

- 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 [16]: it can be refunctionalized into a higher-order evaluator, which we present in Section 5.

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

Figure 2: The adjusted 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 adjusted machine

We relate the configurations and transitions of the definitional abstract machine to those of the adjusted abstract machine. As a diacritical convention [44], 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 adjusted machine (a context and a trail of contexts) to a context of the definitional machine, we convert it into a context of the adjusted machine:

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

$$C_1 \mathbin{\widehat{\star}} \mathsf{nil} \overset{\text{def}}{=} C_1$$
$$C_1 \mathbin{\widehat{\star}} (C'_1 :: T_1) \overset{\text{def}}{=} C_1 \star (C'_1 \mathbin{\widehat{\star}} T_1)$$

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 adjusted abstract machine with the following family of relations $\simeq$:*

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

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* $\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \mathbin{\widehat{\star}} T_1$

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

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

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

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* $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$

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*

$\widetilde{e} \simeq_{\mathrm{e}} e$, $\widetilde{\rho} \simeq_{\mathrm{env}} \rho$, $\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \mathbin{\widehat{\star}} T_1$, *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*

$\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \mathbin{\widehat{\star}} T_1$, $\widetilde{v} \simeq_{\mathrm{v}} v$, *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 $\delta \Rightarrow^* \delta'$ and $\delta \Rightarrow^+ \delta'$, we mean that there is respectively zero or more and one or more transitions leading from the configuration δ to the configuration δ'.

Definition 3. *The partial evaluation functions* $\mathit{eval}_{\mathit{def}}$ *and* $\mathit{eval}_{\mathit{adj}}$ *mapping terms to values are defined as follows:*

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

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

We want to prove that $\mathit{eval}_{\mathit{def}}$ and $\mathit{eval}_{\mathit{adj}}$ 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 programs* $\widetilde{e}$ *and* e *such that* $\widetilde{e} \simeq_{\mathrm{e}} e$ *(i.e.,* $\widetilde{e} = e$*),* $\mathit{eval}_{\mathit{def}}\,(\widetilde{e}) = \widetilde{v}$ *for some value* $\widetilde{v}$ *if and only if* $\mathit{eval}_{\mathit{adj}}\,(e) = v$ *for some value* v *such that* $\widetilde{v} \simeq_{\mathrm{v}} v$*.*

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* $\widetilde{C_1} \mathbin{\widetilde{\star}} \widetilde{C_1'} \simeq_{\mathrm{c}} C_1 \star C_1'$*.*

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

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

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

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

(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 $\delta \Rightarrow^+_{\mathit{adj}} \langle C_1,\, v,\, T_1',\, C_2\rangle_{cont_1}$*.*

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 $\widetilde{\delta} \Rightarrow_{\mathit{def}} \widetilde{\delta}'$ *then there exists a configuration* δ' *such that* $\delta \Rightarrow^+_{\mathit{adj}} \delta'$ *and* $\widetilde{\delta}' \simeq \delta'$*;*

(2) if $\delta \Rightarrow_{\mathit{adj}} \delta'$ *then there exist configurations* $\widetilde{\delta}'$ *and* δ'' *such that* $\widetilde{\delta} \Rightarrow_{\mathit{def}} \widetilde{\delta}'$*,* $\delta' \Rightarrow^*_{\mathit{adj}} \delta''$ *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, $\widetilde{\delta} \Rightarrow_{\mathit{def}} \widetilde{\delta}'$, where $\widetilde{\delta}' = \langle \widetilde{C_1},\, \widetilde{\rho}(\widetilde{x}),\, \widetilde{C_2}\rangle_{\widetilde{cont_1}}$.
From the definition of the adjusted abstract machine, $\delta \Rightarrow_{\mathit{adj}} \delta'$, where $\delta' = \langle C_1,\, \rho(x),\, T_1,\, C_2\rangle_{cont_1}$.
By assumption, $\widetilde{\rho}(\widetilde{x}) \simeq_{\mathrm{v}} \rho(x)$, $\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \mathbin{\widehat{\star}} T_1$ 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: $\widetilde{\delta} = \langle \widetilde{\mathsf{FUN}\,(\widetilde{C'_1},\ \widetilde{C_1})},\ \widetilde{v},\ \widetilde{C_2}\rangle_{\widetilde{\mathit{cont_1}}}$ and $\delta = \langle \mathsf{FUN}\,([C'_1,\ T'_1],\ C_1),\ v,\ T_1,\ C_2\rangle_{\mathit{cont_1}}$
From the definition of the definitional abstract machine, $\widetilde{\delta} \Rightarrow_{\mathit{def}}\ \widetilde{\delta}'$, where
$\widetilde{\delta}' = \langle \widetilde{C'_1} \mathbin{\widetilde{\star}} \widetilde{C_1},\ \widetilde{v},\ \widetilde{C_2}\rangle_{\widetilde{\mathit{cont_1}}}$.
From the definition of the adjusted abstract machine, $\delta \Rightarrow_{\mathit{adj}}\ \delta'$, where
$\delta' = \langle C'_1,\ v,\ T'_1 \mathbin{@} (C_1 :: T_1),\ C_2\rangle_{\mathit{cont_1}}$.
By assumption, $\widetilde{C'_1} \simeq_{\mathrm{c}} C'_1 \mathbin{\widehat{\star}} T'_1$ and $\widetilde{C_1} \simeq_{\mathrm{c}} C_1 \mathbin{\widehat{\star}} T_1$.
By Lemma 1, we have $\widetilde{C'_1} \mathbin{\widetilde{\star}} \widetilde{C_1} \simeq_{\mathrm{c}} (C'_1 \mathbin{\widehat{\star}} T'_1) \star (C_1 \mathbin{\widehat{\star}} T_1)$.
By the definition of $\widehat{\star}$, $(C'_1 \mathbin{\widehat{\star}} T'_1) \star (C_1 \mathbin{\widehat{\star}} T_1) = (C'_1 \mathbin{\widehat{\star}} T'_1) \mathbin{\widehat{\star}} (C_1 :: T_1)$.
By Proposition 3, $(C'_1 \mathbin{\widehat{\star}} T'_1) \mathbin{\widehat{\star}} (C_1 :: T_1) = C'_1 \mathbin{\widehat{\star}} (T'_1 \mathbin{@} (C_1 :: T_1))$.
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: $\widetilde{\delta} = \langle \widetilde{\mathsf{END}},\ \widetilde{v},\ \widetilde{C_2}\rangle_{\widetilde{\mathit{cont_1}}}$ and $\delta = \langle \mathsf{END},\ v,\ T_1,\ C_2\rangle_{\mathit{cont_1}}$
From the definition of the definitional abstract machine, $\widetilde{\delta} \Rightarrow_{\mathit{def}}\ \widetilde{\delta}'$, where $\widetilde{\delta}' = \langle \widetilde{C_2},\ \widetilde{v}\rangle_{\widetilde{\mathit{cont_2}}}$.
By the definition of $\simeq$, $\widetilde{v} \simeq_{\mathrm{v}} v$, $\widetilde{C_2} \simeq_{\mathrm{mc}} C_2$, and $\widetilde{\mathsf{END}} \simeq_{\mathrm{c}} \mathsf{END} \mathbin{\widehat{\star}} T_1$.
Hence, it follows from the definition of $\simeq_{\mathrm{c}}$ that $\mathsf{END} \mathbin{\widehat{\star}} T_1 = \mathsf{END}$, 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), $\delta \Rightarrow^{+}_{\mathit{adj}}\ \delta'$, where $\delta' = \langle C_2,\ v\rangle_{\mathit{cont_2}}$ and $\widetilde{\delta}' \simeq \delta'$, and both directions of the lemma are proved in this case.

Case: $\widetilde{\delta} = \langle \widetilde{C_1},\ \widetilde{v},\ \widetilde{C_2}\rangle_{\widetilde{\mathit{cont_1}}}$ and $\delta = \langle \mathsf{END},\ v,\ T_1,\ C_2\rangle_{\mathit{cont_1}}$, 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 $\widetilde{C_1} \simeq_{\mathrm{c}} \mathsf{END} \mathbin{\widehat{\star}} T_1$.
Hence, it follows from the definition of $\simeq_{\mathrm{c}}$ that $\mathsf{END} \mathbin{\widehat{\star}} T_1 \neq \mathsf{END}$, 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), $\delta \Rightarrow^{+}_{\mathit{adj}}\ \delta'$, where $\delta' = \langle C_1,\ v,\ T'_1,\ C_2\rangle_{\mathit{cont_1}}$, $C_1 \neq \mathsf{END}$, and since $\mathsf{END} \mathbin{\widehat{\star}} T_1 = C_1 \mathbin{\widehat{\star}} T'_1$, we have $\widetilde{\delta} \simeq \delta'$. By one of the trivial cases for $\widetilde{\delta} \simeq \delta'$ (not shown in the proof), both directions of the lemma are proved in this case.

$\square$

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 $\widetilde{\delta} \Rightarrow^{+}_{\mathit{def}}\ \widetilde{\delta}'$ then there exists a configuration δ' such that $\delta \Rightarrow^{+}_{\mathit{adj}}\ \delta'$ and $\widetilde{\delta}' \simeq \delta'$;

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

Proof. Both directions follow from Lemma 3 by induction on the number of transitions. $\square$

We are now in position to prove the equivalence theorem.

Proof of Theorem 1. The initial configuration of the definitional abstract machine, i.e., $\langle \widetilde{e}, \widetilde{\rho_{mt}}, \widetilde{\mathsf{END}}, \widetilde{\mathsf{nil}} \rangle_{\widetilde{eval}}$, and the initial configuration of the adjusted abstract machine, i.e., $\langle e, \rho_{mt}, \mathsf{END}, \mathsf{nil}, \mathsf{nil} \rangle_{eval}$, are in the relation $\simeq$ when $\widetilde{e} = e$. 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 $\delta \Rightarrow^{+}_{adj} \delta'$ 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.

$\square$

5 The evaluator corresponding to the adjusted machine

The *raison d'être* of the adjusted abstract machine is that it is in defunctionalized form. Refunctionalizing its contexts and meta-contexts yields the higher-order evaluator of Figure 3. This evaluator is compositional, i.e., the recursive calls on each right-hand side are over a proper sub-term of the corresponding left-hand side. The evaluator is expressed in a continuation+state-passing style where the state consists of a trail of continuations and a meta-continuation. 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).

6 The CPS transformer corresponding to the evaluator in dynamic CPS

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):

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

A transformed term is evaluated by supplying an initial continuation, trail, and meta-continuation as follows: $[\![e]\!]\,(\theta_1, \mathsf{nil}, \theta_2)$, where θ_1 and θ_2 are defined in Figure 3. As usual, this initialization is equivalent to delimiting control in the translated term.

It is straightforward to write a one-pass version of the dynamic CPS transformer [19].

An example: In our earlier work [8, Section 2.5], we presented a simple example where using `control` and `prompt` led to one result and using `shift` and `reset` led to another. We displayed the CPS counterpart of the latter and stated that no such simple functional encoding existed for the former. Dynamic CPS, however, provides such a functional encoding. The example reads as follows:

```
fun test ()
    = prompt (fn () => control (fn k => 10 + (k 100)) + control (fn k' => 1))
```

Applying `test` to `()` yields `1` since the second occurrence of `control` wipes out the entire delimited evaluation context. Its `shift` and `reset` counterpart yields `11` since the second occurrence of `control` only wipes out the evaluation context up to the application of `k` to `100`.

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

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

- Environments: $\mathsf{Env} \ni \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\}$
- Initial continuation: $\theta_1 = \lambda(v, t_1, k_2).\ \mathit{case}\ t_1$
  $\quad \mathit{of}\ \mathsf{nil} \Rightarrow k_2\, v$
  $\quad \mid k_1 :: t_1' \Rightarrow k_1\, (v, t_1', k_2)$
- Initial meta-continuation: $\theta_2 = \lambda v.\, v$
- Evaluation function: $\mathsf{eval} : \mathsf{Exp} \times \mathsf{Env} \times \mathsf{Cont}_1 \times \mathsf{Trail}_1 \times \mathsf{Cont}_2 \to \mathsf{Val}$

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

- 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 $\#$

The dynamic CPS transformation yields the following program, which we write in the syntax of ML:

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

(*  theta1 : 'a * ('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 cont2  *)
fun theta2 v
    = v

fun test_dcps () (k1, t1, k2)
    = let fun k (v, k1', t1', k2)
              = let fun k' (v', k1'', t1'', k2)
                          = k1 (v + v', t1' @ (k1'' :: t1''), k2)
                in theta1 (1, nil, k2)
                end
      in k (100, CONT1 (fn (v, t1, k2) => theta1 (10 + v, t1, k2)), nil, k2)
      end
```

or again, unfolding the two let expressions and inlining `theta1`:

```
fun test_dcps () (k1, t1, k2)
    = k2 1
```

The initial call is `test_dcps () (theta1, nil, theta2)`. In our experience, out-of-the-box dynamic CPS programs are rarely enlightening the way normal CPS programs (at least after some practice) tend to be. However, again in our experience, a combination of simplifications (e.g., inlining `theta1` 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 8.

7 The direct-style evaluator corresponding to the evaluator in dynamic CPS

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 #.

The following coherence property holds. Transforming this direct-style evaluator into dynamic continuation-passing style, using the one-pass version of the dynamic CPS transformer of Section 6, yields the evaluator of Figure 3. Earlier on [18, Section 2], Danvy and Filinski have shown that a similar coherence property holds for static delimited continuations: CPS-transforming a direct-style evaluator for the λ-calculus extended with shift and reset written in a meta-language extended with shift and reset yields 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 call/cc, obtaining a traditional direct-style evaluator interpreting call/cc with call/cc [20, Section 1.2.1].)

- 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 $\#$

8 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 4.6]. We write it below in ML, using Filinski's implementation of `shift` and `reset` [32] shown in Appendix C, and using the implementation of `control` and `prompt` presented in Section 10. 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 4.6], and Biernacki and Danvy have used an intuitive source representation of the successive contexts [7, Section 2.3]: given the list `[1,2,3]`, the captured delimited continuation in `foo` is always `fn v => visit v`, whereas for `bar`, it is successively `fn v => visit v`, `fn v => 1 :: (visit v)`, `fn v => 2 :: 1 :: (visit v)`, and `fn v => 3 :: 2 :: 1 :: (visit v)`, making it clear that `foo` copies its argument whereas `bar` reverses it. In this section, we use static and dynamic continuation-passing style to illustrate the difference of behavior.

8.1 Static continuation-passing style

Applying the canonical CPS transformation for `shift` and `reset` [18] 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` and `k1'` and lambda-dropping `k1` [24] and then inlining it 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` into a list 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 successive equivalent views make it increasingly clearer that the program copies its input list by first reversing it using the meta-continuation as an accumulator (in `visit`), and then by reversing the accumulator (in `continue`).

8.2 Dynamic continuation-passing style

Applying the dynamic CPS transformation for `control` and `prompt` (Section 6) to the definition of `bar` yields the following purely functional program:

```
datatype ('a, 'b) cont1 = CONT1 of 'a * 'b trail1 * 'b cont2 -> 'b
withtype 'a trail1 = ('a, '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`, lambda-dropping `k1` and `k2` and then inlining them, 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 (essentially inlining the calls to `continue`) 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 successive equivalent views make it increasingly clearer that the program reverses its input list by first copying it to the trail through a series of concatenations (with `visit`), and then by reversing the trail (with `continue_aux`).

8.3 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.

8.4 Further examples

We now turn to the lazy depth-first and breadth-first traversals recently presented by Biernacki, Danvy, and Shan [8]. 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 commands (i.e., unit-yieldings thunks) to be applied in turn. The functor `make_Control_and_Prompt` is defined in Section 10.

```
signature SCHEDULER
= sig
    type command = unit -> unit
    val schedule : command * command * command * command -> unit
  end

functor make_Lazy_Generator (S : SCHEDULER) : GENERATOR
= struct
    datatype sequence = END
                      | NEXT of int * sequence computation
    withtype 'a computation = unit -> 'a

    structure CP = make_Control_and_Prompt (type answer = sequence)

    (*  visit : tree -> unit *)
    fun visit EMPTY
        = ()
      | visit (NODE (t1, i, t2))
        = CP.control
            (fn k =>
               let val () = S.schedule
                              (fn () => visit t1,
                               fn () => CP.control (fn k' => NEXT (i, k')),
                               fn () => visit t2,
                               fn () => let val _ = k () in () end)
               in END
               end)

    (*  make_sequence : tree -> sequence  *)
    fun make_sequence t
        = CP.prompt (fn () => let val () = visit t
                              in END
                              end)

    (*  compute : sequence computation -> sequence  *)
    fun compute k
        = CP.prompt (fn () => k ())
  end
```

The relative scheduling of the first and third commands determines whether the traversal of the input tree is from left to right or from right to left. The relative scheduling of the second command with respect to the first and the third determines whether the traversal is preorder, inorder, or postorder. The relative scheduling of the fourth command 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.

9 A monad for dynamic continuation-passing style

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

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

Let us curry and map the evaluator to direct style with respect to the meta-continuation [12]. The type signature of the resulting evaluator $\mathsf{eval}'_{\mathsf{dcps}}$ is as follows:

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

where

$$\begin{aligned} \mathsf{Cont}_1 &= \mathsf{Val} \rightarrow \mathsf{Trail}_1 \rightarrow \mathsf{Val} \\ \mathsf{Val} &= \mathsf{Val} \rightarrow \mathsf{Cont}_1 \rightarrow \mathsf{Trail}_1 \rightarrow \mathsf{Val} \\ \mathsf{Trail}_1 &= \mathsf{List}(\mathsf{Cont}_1) \end{aligned}$$

In all clauses of the evaluator but the one defining `prompt`, the direct-style transformation consists of eliminating the meta-continuation, whereas the new clause defining `prompt` is transformed into continuation-composing style [18]:

$$\mathsf{eval}'_{\mathsf{dcps}}\,(\#e, \rho)\, k_1\, t_1 = k_1\,(\mathsf{eval}'_{\mathsf{dcps}}\,(e, \rho)\,\theta\,\mathsf{nil})\,t_1$$

The initial continuation also is transformed:

$$\begin{aligned} \theta_1 = \lambda v.\, \lambda t_1.\, &\mathit{case}\ t_1 \\ &\mathit{of}\ \mathsf{nil} \Rightarrow v \\ &\mid k'_1 :: t'_1 \Rightarrow k'_1\, v\, t'_1 \end{aligned}$$

The new evaluator operates on values only of one type Val, so in order to exhibit the general notion of computation dynamic CPS induces, we abstract both the argument type (α) and the final answer type (o) of continuations and we introduce the following type constructor [45,55]:

$$D(\alpha) \quad = \quad \mathsf{Cont}_1(\alpha) \rightarrow \mathsf{Trail}_1 \rightarrow o$$

where

$$\begin{aligned} \mathsf{Cont}_1(\alpha) &= \alpha \rightarrow \mathsf{Trail}_1 \rightarrow o \\ \mathsf{Trail}_1 &= \mathsf{List}(\mathsf{Cont}_1(o)) \end{aligned}$$

We observe that for a fixed type o, D can be expressed as follows:

$$D(\alpha) \quad = \quad (\alpha \rightarrow \mathsf{Ans}) \rightarrow \mathsf{Ans}$$

where

$$\mathsf{Ans} \quad = \quad \mathsf{List}(o \rightarrow \mathsf{Ans}) \rightarrow o$$

Therefore, the notion of computation induced by dynamic CPS takes the form of the continuation monad with the recursive answer type Ans (which confirms Shan's point that a static simulation of dynamic continuations requires a recursive answer type [52]), the type constructor D, and the usual continuation monad operations unit and bind:

$$\begin{aligned} \mathsf{unit} &: \alpha \rightarrow D(\alpha) \\ \mathsf{unit}\, v &= \lambda k.\, k\, v \\[1ex] \mathsf{bind} &: D(\alpha) \rightarrow (\alpha \rightarrow D(\beta)) \rightarrow D(\beta) \\ \mathsf{bind}\, c\, f &= \lambda k.\, c\,(\lambda v.\, f\, v\, k) \end{aligned}$$

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

- 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: $\mathsf{eval} : \mathsf{Exp} \times \mathsf{Env} \rightarrow D(\mathsf{Val})$

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

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

$$\mathsf{evaluate_{mon}}\,(e) = \mathsf{run}\,(\mathsf{eval_{mon}}\,(e, \rho_{mt}))$$

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

Definition 4. *We define the monad operations* $\mathsf{control}$, prompt *and* run *as follows:*

$$\begin{array}{rcl}
\mathsf{control} & : & ((\alpha \rightarrow D(o)) \rightarrow D(o)) \rightarrow D(\alpha) \\
\mathsf{control}\, f & = & \lambda k_1.\, \lambda t_1.\, \mathit{let}\ x = \lambda v.\, \lambda k_1'.\, \lambda t_1'.\, k_1\, v\, (t_1 \mathbin{@} (k_1' :: t_1')) \\
 & & \qquad\qquad \mathit{in}\ f\, x\, \theta_1\, \mathsf{nil} \\
\mathsf{prompt} & : & D(o) \rightarrow D(o) \\
\mathsf{prompt}\, c & = & \lambda k_1.\, \lambda t_1.\, k_1\, (c\, \theta_1\, \mathsf{nil})\, t_1 \\
\mathsf{run} & : & D(o) \rightarrow o \\
\mathsf{run}\, c & = & c\, \theta_1\, \mathsf{nil}
\end{array}$$

We can now extend the usual call-by-value monadic evaluator for the λ-calculus to $\mathcal{F}$ and # by taking $\alpha = o = \mathsf{Val}$ (see Figure 5). Inlining the abstraction layer provided by the monad yields the evaluator $\mathsf{eval}'_{\mathsf{dcps}}$, and uncurrying and CPS-transforming the evaluator $\mathsf{eval}'_{\mathsf{dcps}}$ 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 [1–3, 6, 14, 15]. Furthermore, and as has been observed before for other CPS transformations and for the continuation monad [36, 55], the dynamic CPS transformation itself can be factored through Moggi's monadic metalanguage and the monad above.

The monad presented in this section determines a simple type system that accounts for dynamic delimited-control operators, where, for a fixed type o, `control` and `prompt` have the following type signatures:

$$\begin{array}{rcl}
\texttt{control} & : & ((\alpha \rightarrow o) \rightarrow o) \rightarrow \alpha \\
\texttt{prompt} & : & o \rightarrow o
\end{array}$$

This type system is expressive enough to type most of the existing programming examples and makes it possible to implement `control` and `prompt` in ML (see Section 10). It requires, however, that the answer type of each delimited continuation be the same as the final answer type of the entire program. In order to obtain more liberal and expressive type systems for dynamic continuations, we could follow Wadler [56] and parameterize the monad by intermediate answer types. Allowing for one additional type parameter would yield a type-and-effect system à la Murthy [47], whereas allowing for two additional type parameters would yield a type-and-effect system à la Danvy and Filinski [17].

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

The operations $\mathsf{control}$ and prompt in the continuation monad can be implemented in direct style in terms of shift and reset. In direct style, we use the monadic reflection operators reify and reflect to convert between implicit and explicit representations of computations [32]. They have types:

$$\begin{aligned}\mathsf{reify} &: (\mathsf{unit} \to \alpha) \to D(\alpha)\\ \mathsf{reflect} &: D(\alpha) \to \alpha\end{aligned}$$

The reify operator takes a computation (represented as a thunk in call by value) that may have implicit effects and coerces it into an effect-free value that represents those effects. The $\mathsf{reflect}$ operator takes an effect-free value and coerces it into a computation, performing the effects represented by the value.

We insert occurrences of reify and $\mathsf{reflect}$ in the definitions of $\mathsf{control}$ and prompt from Section 9 guided by the types: we reify a computation that possibly has control effects in order to explicitly apply it to a continuation, and we reflect pure functional values that expect a continuation. Since our implementation language is a call-by-value functional language, we require the argument to prompt (a computation which may have control effects) to be a thunk:

$$\begin{aligned}\mathsf{control} &: ((\alpha \to o) \to o) \to \alpha\\ \mathsf{control}\, f &= \mathsf{reflect}\, \lambda k_1 .\, \lambda t_1 .\, \mathit{let}\ x = \lambda v.\, \mathsf{reflect}\, \lambda k_1' .\, \lambda t_1' .\, k_1\, v\, (t_1 \mathbin{@} (k_1' :: t_1'))\\ &\qquad\qquad\qquad\qquad \mathit{in}\ \mathsf{reify}\, (\lambda().\, f\, x)\, \theta_1\, \mathsf{nil}\end{aligned}$$

$$\begin{aligned}\mathsf{prompt} &: (\mathsf{unit} \to o) \to o\\ \mathsf{prompt}\, c &= \mathsf{reflect}\, \lambda k_1 .\, \lambda t_1 .\, k_1\, (\mathsf{reify}\, c\, \theta_1\, \mathsf{nil})\, t_1\end{aligned}$$

Since $\mathsf{control}$ and prompt are operations in the continuation monad, we use the definitions of reify and $\mathsf{reflect}$ for the continuation monad [33]:[2]

$$\begin{aligned}\mathsf{reify}\, c &= \lambda k.\, \mathsf{reset}\, \lambda().\, k\, (c\, ())\\ \mathsf{reflect}\, f &= \mathsf{shift}\, f\end{aligned}$$

We obtain definitions of $\mathsf{control}$ and prompt in terms of shift and reset by inlining the occurrences of reify and $\mathsf{reflect}$ and simplifying:

$$\begin{aligned}\mathsf{control}\, f =&\quad \{\mathit{definition\ of}\ \mathsf{control}\}\\ &\mathsf{reflect}\, \lambda k_1 .\, \lambda t_1 .\, \mathit{let}\ x = \lambda v.\, \mathsf{reflect}\, \lambda k_1' .\, \lambda t_1' .\, k_1\, v\, (t_1 \mathbin{@} (k_1' :: t_1'))\\ &\qquad\qquad\qquad \mathit{in}\ \mathsf{reify}\, (\lambda().\, f\, x)\, \theta_1\, \mathsf{nil}\\ =&\quad \{\mathit{definition\ of}\ \mathsf{reify}\ \mathit{and}\ \mathsf{reflect}\}\\ &\mathsf{shift}\, \lambda k_1 .\, \lambda t_1 .\, \mathit{let}\ x = \lambda v.\, \mathsf{shift}\, \lambda k_1' .\, \lambda t_1' .\, k_1\, v\, (t_1 \mathbin{@} (k_1' :: t_1'))\\ &\qquad\qquad \mathit{in}\ (\lambda k.\, \mathsf{reset}\, \lambda().\, k\, ((\lambda().\, f\, x)\, ()))\, \theta_1\, \mathsf{nil}\\ =&\quad \{\mathit{two}\ \beta_v\text{-}\mathit{reductions}\}\\ &\mathsf{shift}\, \lambda k_1 .\, \lambda t_1 .\, \mathit{let}\ x = \lambda v.\, \mathsf{shift}\, \lambda k_1' .\, \lambda t_1' .\, k_1\, v\, (t_1 \mathbin{@} (k_1' :: t_1'))\\ &\qquad\qquad \mathit{in}\ \mathsf{reset}\, (\lambda().\, \theta_1\, (f\, x))\, \mathsf{nil}\end{aligned}$$

$$\begin{aligned}\mathsf{prompt}\, c =&\quad \{\mathit{definition\ of}\ \mathsf{prompt}\}\\ &\mathsf{reflect}\, \lambda k_1 .\, \lambda t_1 .\, k_1\, (\mathsf{reify}\, c\, \theta_1\, \mathsf{nil})\, t_1\\ =&\quad \{\mathit{definition\ of}\ \mathsf{reify}\ \mathit{and}\ \mathsf{reflect}\}\\ &\mathsf{shift}\, \lambda k_1 .\, \lambda t_1 .\, k_1\, ((\lambda k.\, \mathsf{reset}\, \lambda().\, k\, (c\, ()))\, \theta_1\, \mathsf{nil})\, t_1\end{aligned}$$

[2]The definitions given here for reify and $\mathsf{reflect}$ are simplifications of the ones given in Filinski's dissertation [33, page 82]. They are obtained by erasing the level tags and then simplifying according to standard call-by-value reasoning.

$= \quad \{\beta_v\text{-}reduction\}$
$\quad \mathsf{shift}\,\lambda k_1.\,\lambda t_1.\,k_1\,(\mathsf{reset}\,(\lambda().\,\theta_1\,(c\,()))\,\mathsf{nil})\,t_1$
$= \quad \{\eta\text{-}reduction\}$
$\quad \mathsf{shift}\,\lambda k_1.\,k_1\,(\mathsf{reset}\,(\lambda().\,\theta_1\,(c\,()))\,\mathsf{nil})$
$= \quad \{\mathcal{S}\text{-}elim\text{: } \mathcal{S}k.k\,M = M,\ if\ k \notin FV(M)\}$
$\quad \mathsf{reset}\,(\lambda().\,\theta_1\,(c\,()))\,\mathsf{nil}$

where the final step is justified by Kameyama and Hasegawa's $\mathcal{S}$-*elim* axiom [41].

Translating these definitions into an implementation in Standard ML is straightforward:

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

functor make_Control_and_Prompt (type answer) : CONTROL_AND_PROMPT
= struct
    type answer = answer                                (* final answer type *)

    datatype ans = ANS of trail1 -> answer             (* answer type of the *)
    withtype 'a cont1 = 'a -> ans                      (* continuation monad *)
         and trail1 = answer cont1 list

    exception MISSING_PROMPT

    structure SR = make_Shift_and_Reset (type answer = ans)

    fun continue (ANS f) t1           (*  continue : ans -> trail1 -> answer *)
        = f t1

    fun theta1 v                                       (*  theta1 : 'a cont1 *)
        = ANS (fn nil        => v
                | (k1 :: t1) => continue (k1 v) t1)

    fun control f              (* control : (('a -> answer) -> answer) -> 'a *)
        = SR.shift
            (fn k1 => ANS (fn t1 =>
               let val x = fn v =>
                             SR.shift
                               (fn k1' => ANS (fn t1' =>
                                  continue (k1 v) (t1 @ (k1' :: t1'))))
               in continue (SR.reset (fn () => theta1 (f x))) nil
               end)) handle MISSING_RESET => raise MISSING_PROMPT

    fun prompt c                      (* prompt : (unit -> answer) -> answer *)
        = continue (SR.reset (fn () => theta1 (c ()))) nil
  end
```

This implementation is a direct consequence of the abstract-machine semantics of control and prompt. It is formally connected to the operations in the continuation monad by monadic reflection and the monad is formally connected to the abstract machine by defunctionalization.

As usual with implementations of delimited control operators, a program using `control` and `prompt` needs a toplevel control delimiter: not only the capture of the current delimited continuation (like for `shift` and `reset`) but also the application of a captured continuation (unlike for `shift` and `reset`) must occur within a delimited context.

11 Related work

The concept of meta-continuation and its representation as a function originate in Wand and Friedman's formalization of reflective towers [58], and its representation as a list in Danvy and Malmkjær's followup study [21]. Danvy and Filinski then realized that a meta-continuation naturally arises by "one more" CPS transformation, giving rise to success and failure continuations [18], and later Danvy and Nielsen observed that the list representation naturally arises by defunctionalization [23].

As for delimited continuations, the representation of the meta-continuation as a list has a long history. Johnson and Duggan use it in their early work on composable continuations [39]. Danvy and Filinski use it to specify (what is now known as) `shift0` [17, Appendix C]. Moreau and Queinnec later used it to specify their control operators `call/pc` and `marker` [46]. Dybvig, Peyton Jones, and Sabry have recently used it in their monadic framework for delimited continuations [25].

The original approaches to delimited continuations were split between composing continuations dynamically by concatenating their representations [30] and composing them statically using continuation-passing function composition [18]. Recently, Shan [52], Dybvig, Peyton Jones, and Sabry [25], and Kiselyov [43] have each proposed accounts of dynamic delimited continuations:

- Shan gives a continuation semantics for dynamic delimited continuations. His continuation semantics threads a state equivalent to our trail. This state is a functional representation of a binary tree with continuations at the leaves. He extends Felleisen et al.'s idea of an algebra of contexts [30] and uses the algebraic operators *Send* and *Compose* rather than standard list operators to propagate intermediate results and compose delimited continuations.

  The abstract machine corresponding to Shan's continuation semantics is similar to our adjusted abstract machine. Like ours it uses an extra component to delay the concatenation of contexts. Shan's approach corresponds to viewing the composition operator ($\star$) as a context constructor rather than a meta-level operation, thus obtaining a machine in defunctionalized form. He justifies the correctness of his continuation semantics by (1) using defunctionalization to obtain the corresponding abstract machine, and (2) relating it to the definitional machine for control and prompt given here in Section 2.

  Shan informally connects his continuation semantics to his direct-style implementation in Scheme via a pair of transformations. He transforms an abstraction over a continuation into a use of `shift` and transforms an application to a continuation into an application of a continuation guarded by a `reset`. Here we show that these transformations are formally justified by the monadic reflection operators reflect and reify for the continuation monad.

  Shan considers two other dynamic delimited continuation operators. He gives them continuation semantics and implementations that correspond to two other abstract machines. Our adjusted abstract machine and the corresponding dynamic continuation-passing style can be adapted to account for either of these variations as well, by leaving the meta-continuation defunctionalized (see Appendices A and B).

- Dybvig, Peyton Jones, and Sabry provide a general framework for delimited continuations. They give a continuation semantics that threads a state that is a list of continuations annotated with multiple control delimiters. This state is related to our trail and meta-continuation. Defunctionalizing our meta-continuation, inserting explicit delimiters between its segments, flattening it, and appending it to our trail of continuations yields their state specialized to a single delimiter. Their framework, however, was designed independently of defunctionalization.

  They exhibit an abstract machine that corresponds to their continuation semantics. Our adjusted abstract machine is related to their machine specialized to a single delimiter and restricted to `control` and `prompt`. 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 control operators that manipulate delimited continuations" [25, Section 8], based on Moreau and Queinnec's representation of the meta-continuation as a list of control-delimiter tags and of continuations, and using Gunter, Rémy, and Riecke's control operators `set` and `cupto` [34] as a common basis, whereas

  – we wanted an abstract machine for `control` and `prompt` 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.

  Dybvig et al. give a direct-style Scheme implementation of their framework in terms of `call/cc` and state, but do not formally justify the correctness of this implementation.

  Their framework is more general than the present work: it can account for all the variations on control operators considered by Shan, as well as variations with multiple delimiters. In contrast, we have focused on specifying `control` and `prompt` and on illustrating them.

- Kiselyov gives an encoding of dynamic delimited continuations in terms of `shift` and `reset` and recursion. His approach is qualitatively different from ours, Shan's, and Dybvig et al.'s. It does not thread an extra state parameter, but rather tags values sent to the meta-continuation to indicate control effects or their absence. His transformation wraps code around delimiters and the bodies of continuation functions in order to handle control effects.

  The abstract machine corresponding to Kiselyov's encoding does not have an extra component to delay the concatenation of contexts. Instead, it uses the meta-continuation for temporary storage of continuations to delay their concatenation to the current continuation.

  Kiselyov proves his encoding correct by showing that its behavior under reduction agrees with the reduction semantics of `control` and `prompt`.

  His approach allows variations on delimited continuations by choosing how to handle delimiters and continuation application.

As for adjusting an abstract machine to put it in defunctionalized form, there are precedents. For example, as pointed out by the two last authors [22], Felleisen's version of the SECD machine with the J operator [26] differs from its predecessors in that it is in defunctionalized form. (In effect, it uses a control delimiter.)

Finally, just as repeated CPS transformations give rise to a static CPS hierarchy [4,18,40, 47], repeated dynamic CPS transformations give rise to a dynamic CPS hierarchy—a future work.

12 Conclusion and issues

In our earlier work [8], 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 fulfills these wishes for `control` and `prompt` 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 dynamic delimited-control operators in terms of static ones, 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 for their comments. This work is supported by the Danish Natural Science Research Council, Grant no. 21-03-0545 and by the ESPRIT Working Group APPSEM (`http://www.appsem.org`).

Appendices

A `control0`

Shan [52] considers a variation of `control`, `control0` ($^{-}\mathcal{F}^{-}$ in the parlance of Dybvig, Peyton Jones, and Sabry [25]). Informally, `control0` is like `control` except that capturing a delimited continuation removes a delimiter (and therefore programs using `control0` may need more than one toplevel control delimiter or alternatively a "master," undiscardable control delimiter). Our adjusted abstract machine can be modified to account for this variation.

The adjusted machine is modified to implement the semantics of `control0` by replacing the clause for capturing a delimited continuation as follows:

$$\boxed{\langle {}^{-}\mathcal{F}^{-}x.e,\, \rho,\, C_1,\, T_1,\, (C_1', T_1') :: C_2\rangle_{eval} \quad \Rightarrow_{adj} \quad \langle e,\, \rho\{x \mapsto [C_1,\, T_1]\},\, C_1',\, T_1',\, C_2\rangle_{eval}}$$

The machine is still in defunctionalized form with respect to its contexts. It can thus be refunctionalized, which yields a compositional continuation-passing evaluator. The rest of the development (CPS transformation, direct-style evaluator, monad, and implementation) follows Sections 6, 7, 9, and 10 mutatis mutandis.[3]

B `shift0`

The fourth control operator considered by Shan is `shift0` ($^{-}\mathcal{F}^{+}$ in the parlance of Dybvig, Peyton Jones, and Sabry [25]). Informally, `shift0` is like `shift` except that capturing a delimited continuation removes a delimiter (and therefore programs using `shift0` may need more than one toplevel control delimiter). Our adjusted abstract machine can be modified to account for this variation as well.

[3]The meta-contexts remain defunctionalized, and therefore cannot be eliminated from the evaluator (and thus the monad and implementation) via direct-style transformation.

The adjusted machine is modified to implement the semantics of `shift0` by replacing the clause for applying a captured continuation as follows, in addition to the modification of Appendix A:

$$\boxed{\begin{array}{rcl} \langle {}^{-}\mathcal{F}^{+}x.e,\, \rho,\, C_1,\, T_1,\, (C_1', T_1') :: C_2\rangle_{eval} & \Rightarrow_{adj} & \langle e,\, \rho\{x \mapsto [C_1,\, T_1]\},\, C_1',\, T_1',\, C_2\rangle_{eval} \\ \langle \mathsf{FUN}\,([C_1',\, T_1'],\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1} & \Rightarrow_{adj} & \langle C_1',\, v,\, T_1',\, (C_1, T_1) :: C_2\rangle_{cont_1} \end{array}}$$

The machine is still in defunctionalized form with respect to its contexts. It can thus be refunctionalized, which yields a compositional continuation-passing evaluator. Just as in Appendix A, the rest of the development follows Sections 6, 7, 9, and 10.

C An implementation of `shift` and `reset`

We use Filinski's implementation of `shift` and `reset` in SML [32], renaming some identifiers for uniformity:

```
signature ESCAPE
= sig
    type void
    val coerce : void -> 'a
    val escape : (('a -> void) -> 'a) -> 'a
  end

structure Escape : ESCAPE
= struct
    open SMLofNJ.Cont
    datatype void = VOID of void
    fun coerce (VOID v) = coerce v
    fun escape f = callcc (fn k => f (fn x => throw k x))
  end

signature SHIFT_AND_RESET
= sig
    type answer
    val shift : (('a -> answer) -> answer) -> 'a
    val reset : (unit -> answer) -> answer
  end

functor make_Shift_and_Reset (type answer) : SHIFT_AND_RESET
= struct
    open Escape
    type answer = answer
    exception MISSING_RESET
    val mk : (answer -> void) ref = ref (fn _ => raise MISSING_RESET)
    fun abort x = coerce (!mk x)
    fun reset t
        = escape (fn k => let val m = !mk
                          in mk := (fn r => (mk := m; k r));
                             abort (t ())
                          end)
    fun shift h
        = escape (fn k => abort (h (fn v => reset (fn () => coerce (k v)))))
  end
```

References

[1] Mads Sig Ager. *Partial Evaluation of String Matchers & Constructions of Abstract Machines*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[2] 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, Uppsala, Sweden, August 2003. ACM Press.

[3] 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*, 342(1):149–172, 2005. Extended version available as the technical report BRICS RS-04-28.

[4] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).

[5] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. Technical Report BRICS RS-05-38, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2005. Accepted for publication in Theoretical Computer Science (March 2006).

[6] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, December 2005.

[7] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. *Journal of Functional Programming*, 16(3):269–280, 2006.

[8] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60:274–297, 2006.

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

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

[11] 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.

[12] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).

[13] 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 (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk.

[14] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, 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 prize. Extended version available as the technical report BRICS RS-03-33.

[15] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2006.

[16] Olivier Danvy. Refunctionalization at work. In *Preliminary proceedings of the 8th International Conference on Mathematics of Program Construction (MPC '06)*, Kuressaare, Estonia, July 2006. Invited talk.

[17] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[18] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [57], pages 151–160.

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

[20] 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.

[21] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In Cartwright [10], pages 327–341.

[22] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's J operator. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05*, number 4015 in Lecture Notes in Computer Science, pages 55–73, Dublin, Ireland, September 2005. Springer-Verlag. Extended version available as the technical report BRICS RS-06-4 (February 2006).

[23] 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.

[24] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000.

[25] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for subcontinuations. To appear in the Journal of Functional Programming. Available at `<http://www.cs.indiana.edu/~sabry/research.html>`, May 2006.

[26] Matthias Felleisen. Reflections on Landin's J operator: a partly historical note. *Computer Languages*, 12(3/4):197–207, 1987.

[27] Matthias Felleisen. The theory and practice of first-class prompts. In Ferrante and Mager [31], pages 180–190.

[28] 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.

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

[30] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Cartwright [10], pages 52–62.

[31] Jeanne Ferrante and Peter Mager, editors. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988. ACM Press.

[32] Andrzej Filinski. Representing monads. In Boehm [9], pages 446–457.

[33] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996. Technical Report CMU-CS-96-119.

[34] 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.

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

[36] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [9], pages 458–471.

[37] 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.

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

[39] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In Ferrante and Mager [31], pages 158–168.

[40] 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.

[41] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.

[42] 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.

[43] 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.

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

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

[46] 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.

[47] 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'92)*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.

[48] 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.

[49] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [50].

[50] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[51] 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*, Technical report TR600, Computer Science Department, Indiana University, Snowbird, Utah, September 2004.

[52] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 2007. Journal version of [51]. To appear.

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

[54] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [57], pages 161–175.

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

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

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

[58] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–38, May 1988. A preliminary version was presented at the 1986 ACM Conference on Lisp and Functional Programming (LFP 1986).