Basic Research in Computer Science

A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations

(Preliminary Version)

Dariusz Biernacki
Olivier Danvy
Kevin Millikin

BRICS Report Series **RS-05-5**

ISSN 0909-0878 **February 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/05/5/`

A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations*

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

February 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 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 illustrate that the new machine is more efficient than the definitional one, and we show that the notion of computation induced by the corresponding evaluator takes the form of a monad.

*Preliminary version.

†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`

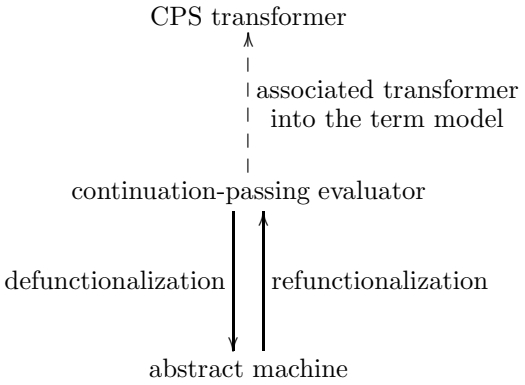
Contents

List of Figures

1 Introduction

Delimited continuations have been a topic of study for 15 years now [9,13], with two main lines of work: Felleisen's operational approach [13, 15] where dynamic delimited continuations are represented as lists of control-stack frames and composed by list concatenation, and Danvy and Filinski's denotational approach [9] where static delimited continuations are represented with continuation-passing functions and composed by continuation-passing function composition. It is well known that static and dynamic delimited continuations differ in behavior, even though they have the same expressive power [21]. Recently, we have pointed out in which sense dynamic delimited continuations are incompatible with continuation-passing style (CPS) [3, Section 4.5], and how they make it possible to program a breadth-first tree traversal in direct style and with no auxiliary parameter [4].

Static delimited continuations are compatible with continuation-passing style because a program using them can be CPS-transformed using a traditional notion of CPS transformation [10, 19, 22]. The abstract machine accounting for static delimited continuations is in defunctionalized form [11,20] and corresponds to a definitional evaluator in CPS [3,20], which itself corresponds to the associated CPS transformer:



In contrast, dynamic delimited continuations are specified with an abstract machine which is not in defunctionalized form [3, Section 4.5], and only recently have they been characterized with a non-standard notion of CPS [21].

This work: We present a new abstract machine that accounts for dynamic delimited continuations and that is in defunctionalized form, and we prove its equivalence with a definitional abstract machine that is not in defunctionalized form. We also present the corresponding new evaluator from which one can obtain the corresponding new CPS transformer. The resulting 'dynamic continuation-passing style' 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 [21], and structurally similar to the one recently proposed by Dybvig, Peyton Jones, and Sabry [12]. We also show that it corresponds to a computational monad.

Prerequisites: We assume a passing familiarity with the notions of continuation, of delimited continuation, and of defunctionalization. In particular, we use Danvy and Nielsen's characterization of a program being in the range of defunctionalization [11]: the first-order representation of functions should have a single point of consumption. So for example, the following ML program traverses a binary tree in depth-first order, using a stack (represented as a list):

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

(*  depth_first_stack_based_enumeration : tree -> int list  *)
fun depth_first_stack_based_enumeration t
    = let (*  visit : tree * tree list -> int list  *)
          fun visit (LEAF i, a)
              = i :: (continue (a, ()))
            | visit (NODE (t1, i, t2), a)
              = i :: (visit (t1, t2 :: a))
          (*  continue : tree list * unit -> int list  *)
          and continue (nil, ())
              = nil
            | continue (t :: a, ())
              = visit (t, a)
      in visit (t, nil)
      end
```

The intermediate list of trees is constructed in the inductive case (with the expression `t2 :: a`) and consumed by `continue`. This program is therefore in defunctionalized form, and the corresponding higher-order program reads as follows:

```
(*  depth_first_higher_order_enumeration : tree -> int list  *)
fun depth_first_higher_order_enumeration t
    = let (*  visit : tree * (unit -> int list) -> int list  *)
          fun visit (LEAF i, a)
              = i :: (a ())
            | visit (NODE (t1, i, t2), a)
              = i :: (visit (t1, fn () => visit (t2, a)))
      in visit (t, fn () => nil)
      end
```

Defunctionalizing the higher-order program yields the stack-based program, and conversely, Church-encoding the list in the stack-based program yields the higher-order program [11].

By folding the definition of `continue` in the induction case of the stack-based definition, we can make it even more clear that the definition uses a stack:

```
            | visit (NODE (t1, i, t2), a)
              = i :: (continue (t1 :: t2 :: a, ()))
```

By replacing the stack with a queue, we obtain a program that traverses the source tree in breadth-first order:

```
  | visit (NODE (t1, i, t2), a)
    = i :: (continue (a @ (t1 :: t2 :: nil), ()))
```

(Nothing else changes in the definition.) This queue-based program is not in defunctionalized form because the intermediate list of trees, which is constructed in the inductive case (with the expression `t1 :: t2 :: nil`), is not solely consumed by `continue`—it may also be consumed by `@` (i.e., the list-concatenation function) in a subsequent recursive call.

Overview: We first present the definitional machine for dynamic delimited continuations in Section 2. We then present the new machine in Section 3 and we establish their equivalence in Section 4. The new machine is in defunctionalized form and we present the corresponding higher-order evaluator in Section 5. This evaluator is expressed in a dynamic continuation-passing style. We address the issue of efficiency in Section 6 and in Section 7, we show that dynamic continuation-passing style can be characterized with a computational monad.

2 The definitional abstract machine

In our earlier work [3], we obtained an abstract machine for static delimited continuations by defunctionalizing a definitional evaluator that had two layered continuations. In this abstract machine, the first continuation takes the form of an evaluation context and the second one takes the form of a stack of evaluation contexts. By construction, this abstract machine is an extension of Felleisen et al.'s CEK machine [14], which has one evaluation context and is itself a defunctionalized evaluator with one continuation [8].

The abstract machine for static delimited continuations implements the application of a delimited continuation (represented as a captured context) by pushing the current context on the stack of contexts and installing the captured context as the new current context. In contrast, applying a dynamic delimited continuation (also represented as a captured context) is implemented by concatenating the captured context to the current context. 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 one to define 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 [3, Figure 7 and Section 4.5] in the way captured delimited continuations are applied, and is otherwise consistent with Felleisen et al.'s definition of delimited-continuation composition by concatenation of their representation [15].

Contexts form a monoid:

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

- Terms: $e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}k.e$
- 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	$\Rightarrow$	$\langle e,\, \rho_{mt},\, \mathsf{end},\, \mathsf{nil}\rangle_{eval}$
$\langle x,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, \rho(x),\, C_2\rangle_{cont_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, [x,\, e,\, \rho],\, C_2\rangle_{cont_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle e_0,\, \rho,\, \mathsf{arg}\,((e_1,\rho),\, C_1),\, C_2\rangle_{eval}$
$\langle \#e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{end},\, C_1 :: C_2\rangle_{eval}$
$\langle \mathcal{F}k.e,\, \rho,\, C_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho\{k \mapsto C_1\},\, \mathsf{end},\, C_2\rangle_{eval}$
$\langle \mathsf{end},\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle C_2,\, v\rangle_{cont_2}$
$\langle \mathsf{arg}\,((e,\rho),\, C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{fun}\,(v,\, C_1),\, C_2\rangle_{eval}$
$\langle \mathsf{fun}\,([x,\, e,\, \rho],\, C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, C_2\rangle_{eval}$
$\langle \mathsf{fun}\,(C_1',\, C_1),\, v,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle C_1' \star C_1,\, v,\, C_2\rangle_{cont_1}$
$\langle C_1 :: C_2,\, v\rangle_{cont_2}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2\rangle_{cont_1}$
$\langle \mathsf{nil},\, v\rangle_{cont_2}$	$\Rightarrow$	v

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

(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 Figure 1, the constructors of contexts are not solely consumed by the $cont_1$ transitions of the abstract machine, but also by $\star$. Therefore, the definitional abstract machine is not in the range of defunctionalization, and 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).

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 definitional machine and of the new machine.

In the new machine, and unlike in the definitional machine, context constructors are only consumed in the $cont_1$ transitions (instead of also in the context-concatenation function). Therefore, the new machine, unlike the definitional machine, is in the range of defunctionalization. It can be refunctionalized to produce a higher-order evaluator, which we present in Section 5.

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 [23], we annotate the components, configurations, and transitions of the definitional machine with a tilde ($\widetilde{\cdot}$). Also, we convert a dynamic context of the new machine (a context and a trail of contexts) into another context of the new machine in order to relate it to a context of the definitional 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:*

$$C_1 \mathbin{\widehat{\star}} \mathsf{nil} \stackrel{\text{def}}{=} C_1$$

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

- Terms: $e ::= x \mid \lambda x.e \mid e_0\, e_1 \mid \#e \mid \mathcal{F}k.e$
- 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	$\Rightarrow$	$\langle e,\, \rho_{mt},\, \mathsf{end},\, \mathsf{nil},\, \mathsf{nil}\rangle_{eval}$
$\langle x,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, \rho(x),\, T_1,\, C_2\rangle_{cont_1}$
$\langle \lambda x.e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, [x,\, e,\, \rho],\, T_1,\, C_2\rangle_{cont_1}$
$\langle e_0\, e_1,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle e_0,\, \rho,\, \mathsf{arg}\,((e_1, \rho),\, C_1),\, T_1,\, C_2\rangle_{eval}$
$\langle \#e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{end},\, \mathsf{nil},\, (C_1, T_1) :: C_2\rangle_{eval}$
$\langle \mathcal{F}k.e,\, \rho,\, C_1,\, T_1,\, C_2\rangle_{eval}$	$\Rightarrow$	$\langle e,\, \rho\{k \mapsto [C_1,\, T_1]\},\, \mathsf{end},\, \mathsf{nil},\, C_2\rangle_{eval}$
$\langle \mathsf{end},\, v,\, \mathsf{nil},\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle C_2,\, v\rangle_{cont_2}$
$\langle \mathsf{end},\, v,\, C_1 :: T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle C_1,\, v,\, T_1,\, C_2\rangle_{cont_1}$
$\langle \mathsf{arg}\,((e, \rho),\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle e,\, \rho,\, \mathsf{fun}\,(v,\, C_1),\, T_1,\, C_2\rangle_{eval}$
$\langle \mathsf{fun}\,([x,\, e,\, \rho],\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle e,\, \rho\{x \mapsto v\},\, C_1,\, T_1,\, C_2\rangle_{eval}$
$\langle \mathsf{fun}\,([C_1',\, T_1'],\, C_1),\, v,\, T_1,\, C_2\rangle_{cont_1}$	$\Rightarrow$	$\langle C_1',\, v,\, T_1' \mathbin{@} (C_1 :: T_1),\, C_2\rangle_{cont_1}$
$\langle (C_1, T_1) :: C_2,\, v\rangle_{cont_2}$	$\Rightarrow$	$\langle C_1,\, v,\, T_1,\, C_2\rangle_{cont_1}$
$\langle \mathsf{nil},\, v\rangle_{cont_2}$	$\Rightarrow$	v

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

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: $\widetilde{\rho} \simeq_{\mathrm{env}} \rho$ *iff* $dom\,(\widetilde{\rho}) = dom\,(\rho)$ *and for all* $x \in dom\,(\widetilde{\rho})$, $\widetilde{\rho}(x) \simeq_{\mathrm{v}} \rho(x)$

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

Definition 3. *The partial functions* $\widetilde{eval}$ *and eval mapping terms to values are defined as follows:*

(1) $\widetilde{eval}\,(e) = \widetilde{v}$ *if and only if the definitional abstract machine, started with the term* e*, stops with the value* $\widetilde{v}$*,*

(2) $eval\,(e) = v$ *if and only if the new abstract machine, started with the term* e*, stops with the value* v*.*

We want to prove that $\widetilde{eval}$ and *eval* 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, $\widetilde{eval}\,(e) = \widetilde{v}$ *if and only if* $eval\,(e) = v$ *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* $\widetilde{C_1} \mathbin{\widetilde{\star}} \widetilde{C_1'} \simeq_{\mathrm{c}} C_1 \star C_1'$.

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

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

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

(1) If $\mathsf{end} \mathbin{\widehat{\star}} T_1 = \mathsf{end}$ *then* $\delta \Rightarrow^* \langle \mathsf{end},\, v,\, \mathsf{nil},\, C_2 \rangle_{cont_1}$.

(2) If $\mathsf{end} \mathbin{\widehat{\star}} T_1 = C_1 \mathbin{\widehat{\star}} T_1' \neq \mathsf{end}$ *then* $\delta \Rightarrow^* \langle C_1,\, v,\, T_1',\, C_2 \rangle_{cont_1}$.

(3) If $T_1 \neq \mathsf{nil}$ *and* $\widetilde{\delta} \simeq \delta$ *and* $\delta \Rightarrow \delta'$ *then* $\widetilde{\delta} \simeq \delta'$.

Proof. By induction on the structure of T_1. □

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

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

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

(2) if $\delta \Rightarrow \delta'$ *then there exists a configuration* $\widetilde{\delta}'$ *such that* $\widetilde{\delta} \Rightarrow^1 \widetilde{\delta}'$ *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, $\widetilde{\delta} \Rightarrow \widetilde{\delta}'$, 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, $\delta \Rightarrow \delta'$, 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 the lemma 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 lock step:

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, $\widetilde{\delta} \Rightarrow \widetilde{\delta}'$, where
$\widetilde{\delta}' = \langle \widetilde{C_1'} \mathbin{\widetilde{\star}} \widetilde{C_1},\, \widetilde{v},\, \widetilde{C_2} \rangle_{\widetilde{cont_1}}$.

From the definition of the new abstract machine, $\delta \Rightarrow \delta'$, where
$\delta' = \langle C_1', v, T_1' \mathbin{@} (C_1 :: T_1), C_2 \rangle_{cont_1}$.
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 the lemma are proved in this case.

Case: $\widetilde{\delta} = \langle \widetilde{\mathsf{end}}, \widetilde{v}, \widetilde{C_2} \rangle_{\widetilde{cont_1}}$ and $\delta = \langle \mathsf{end}, v, T_1, C_2 \rangle_{cont_1}$

(1) From the definition of the definitional abstract machine, $\widetilde{\delta} \Rightarrow \widetilde{\delta'}$, where $\widetilde{\delta'} = \langle \widetilde{C_2}, \widetilde{v} \rangle_{\widetilde{cont_2}}$.
By the definition of $\simeq$, $\widetilde{\mathsf{end}} \simeq_{\mathrm{c}} \mathsf{end} \mathbin{\widehat{\star}} T_1$, so $\mathsf{end} \mathbin{\widehat{\star}} T_1 = \mathsf{end}$ by the definition of $\simeq_{\mathrm{c}}$.
Then by Lemma 2(1), $\delta \Rightarrow^* \langle \mathsf{end}, v, \mathsf{nil}, C_2 \rangle_{cont_1}$.
Hence, $\delta \Rightarrow^+ \delta'$, where $\delta' = \langle C_2, v \rangle_{cont_2}$ and $\widetilde{\delta'} \simeq \delta'$.

(2) If $T_1 = \mathsf{nil}$ then $\delta \Rightarrow \delta'$, where $\delta' = \langle C_2, v \rangle_{cont_2}$, and $\widetilde{\delta} \Rightarrow \widetilde{\delta'}$, where $\widetilde{\delta'} = \langle \widetilde{C_2}, \widetilde{v} \rangle_{\widetilde{cont_2}}$, with $\widetilde{\delta'} \simeq \delta'$.
Otherwise, $T_1 = \mathsf{end} :: T_1'$ and $\delta \Rightarrow \delta'$, where $\delta' = \langle \mathsf{end}, v, T_1', C_2 \rangle_{cont_1}$. Obviously, $\widetilde{\delta'} \simeq \delta'$.

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 $\widetilde{C_1} \neq \mathsf{end}$.

(1) Assume that $\widetilde{\delta} \Rightarrow \widetilde{\delta'}$. By Lemma 2(2), $\delta \Rightarrow^+ \delta'$, where
$\delta' = \langle C_1, v, T_1', C_2 \rangle_{cont_1}$ and $C_1 \neq \mathsf{end}$ and $\widetilde{\delta} \simeq \delta'$. Hence, we have reduced this case to one of the trivial cases (not shown in the proof), where $\widetilde{\delta} \simeq \delta'$ and $\widetilde{\delta} \Rightarrow \widetilde{\delta'}$. Therefore, there exists a configuration δ'' such that $\delta' \Rightarrow \delta''$ and $\widetilde{\delta'} \simeq \delta''$.

(2) Assume that $\delta \Rightarrow \delta'$. By Lemma 2(3), $\widetilde{\delta} \simeq \delta'$.

□

Given the relation between single-step transitions of the two abstract machines, we can generalize it straightforwardly to the relation between their multi-step transitions.

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

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

(2) if $\delta \Rightarrow^+ \delta'$ then there exists a configuration $\widetilde{\delta'}$ such that $\widetilde{\delta} \Rightarrow^* \widetilde{\delta'}$ 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 the initial configuration 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, there is a configuration δ' such that $\delta \Rightarrow^+ \delta'$ 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 the same steps. □

5 The evaluator corresponding to the new abstract machine

The *raison d'être* of the new abstract machine is that it is in defunctionalized form. We present the corresponding higher-order evaluator in 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. 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.'

The corresponding dynamic CPS transformer can be immediately obtained as the associated syntax-directed encoding into the term model of the meta-language. The full version of this article presents it in detail [5].

6 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 [6, 7, 17]—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 the context C_1'. In contrast, when the new abstract machine applies the equivalent context $[C_1', T_1']$ in a current context C_1 with a current trail of contexts T_1, the new trail is $T_1' \mathbin{@} (C_1 :: T_1)$, and constructing it requires work proportional to the number of contexts (i.e., stack segments) in the trail, 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 (expressed in ML syntax) illustrates the situation:

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

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

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

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

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

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

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

```
(*  list_copy : 'a list -> 'a list  *)
fun list_copy 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:

```
(*  list_copy' : 'a list -> 'a list  *)
fun list_copy' 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
```

The delimited continuation captured by `control` is represented by a context whose size is proportional to the length of the list traversed so far (i.e., 0, 1, 2, etc.). 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.

7 A monad for dynamic continuation-passing style

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

$$\mathsf{Exp} \times \mathsf{Env} \times \mathsf{Cont}_1 \times \mathsf{List}(\mathsf{Cont}_1) \times \mathsf{Cont}_2 \to \mathsf{Ans}$$

Let us curry it to exhibit its notion of computation:

$$\mathsf{Exp} \times \mathsf{Env} \to \mathsf{Cont}_1 \to \mathsf{List}(\mathsf{Cont}_1) \times \mathsf{Cont}_2 \to \mathsf{Ans}$$

Proposition 4. *The type constructor*

$$D(\mathsf{Val}) = \mathsf{Cont}_1 \to \mathsf{List}(\mathsf{Cont}_1) \times \mathsf{Cont}_2 \to \mathsf{Ans}$$

where

$$\begin{array}{rcl}
\mathsf{Ans} & = & \mathsf{Val} \\
\mathsf{Cont}_2 & = & \mathsf{Val} \to \mathsf{Ans} \\
\mathsf{Cont}_1 & = & \mathsf{Val} \to \mathsf{List}(\mathsf{Cont}_1) \times \mathsf{Cont}_2 \to \mathsf{Ans} \\
\mathsf{Val} & = & \mathsf{Val} \to \mathsf{Cont}_1 \to \mathsf{List}(\mathsf{Cont}_1) \times \mathsf{Cont}_2 \to \mathsf{Ans}
\end{array}$$

together with the functions

$$\begin{array}{rcl}
\mathsf{unit} & : & \mathsf{Val} \to D(\mathsf{Val}) \\
\mathsf{unit}\,(v) & = & \lambda k_1.\lambda(t_1, k_2).k_1\, v\, (t_1, k_2)
\end{array}$$

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

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 [18]. □

Therefore the evaluator of Figure 3 is a specialized version of the usual call-by-value monadic evaluator with respect to the monad above, given two monad operators, one for delimiting control with # and one for abstracting control with $\mathcal{F}$. (The full version of this article contains more detail [5].) Dynamic continuation-passing style therefore fits the functional correspondence between evaluators and abstract machines advocated by the two first authors [1, 2]. In particular, we are now in position to make dynamic delimited continuations coexist with arbitrary computational effects expressed as monads.

8 Related work

As mentioned in the introduction, the original approaches to delimited continuations were split between composing continuations by concatenating their representations and composing them using continuation-passing function composition. Recently, Shan [21] and Dybvig, Peyton Jones, and Sabry [12] each have proposed an account of dynamic delimited continuations using a continuation+state-passing style.

Shan's development extends Felleisen et al.'s idea of an algebra of contexts [15] (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 prompt-annotated list of continuations. This state is structurally similar to ours in the sense that defunctionalizing and flattening our meta-continuation and appending to it our trail of continuations yields their state without prompt annotations. In particular, enriching our meta-continuation with named prompts precisely yields Dybvig, Peyton Jones, and Sabry's state. 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" [12, Section 7] 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.

9 Conclusion and issues

In our earlier work [4], we argued that dynamic delimited continuations need examples, reasoning tools, and meaning-preserving program transformations, not only new variations, new formalizations, or new implementations. Our present work partly fulfills these wishes by providing an abstract machine that is in defunctionalized form, the corresponding evaluator, the corresponding CPS transformer, and a monadic notion of continuation-passing style that accounts for dynamic delimited continuations.

In the full version of this article [5], we revisit the breadth-first traversal in direct style we presented in our earlier work [4] and that we briefly touched upon in Section 1. This breadth-first traversal uses dynamic delimited continuations; CPS-transforming it and defunctionalizing the resulting continuations yields an iterative program using a queue—a situation which is pleasingly symmetric to the depth-first counterpart of this breadth-first traversal: the depth-first traversal uses static delimited continuations; CPS-transforming it and defunctionalizing the resulting continuations yields an iterative program using a stack.

Compared to static delimited continuations, dynamic delimited continuations still remain 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, and Kristian Støvring for timely 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] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, Aug. 2003.

[2] M. S. Ager, O. Danvy, and J. 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] M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy. Tech. report BRICS RS-04-29, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Dec. 2004.

[4] D. Biernacki and O. Danvy. On the dynamic extent of delimited continuations. Tech. report BRICS RS-05-2, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Jan. 2005.

[5] D. Biernacki, O. Danvy, and K. Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Tech. report BRICS RS-05-5, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Feb. 2005.

[6] W. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

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

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

[9] O. Danvy and A. Filinski. Abstracting control. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

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

[11] O. Danvy and L. R. Nielsen. Defunctionalization at work. In H. Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174,

Firenze, Italy, Sept. 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[12] R. K. Dybvig, S. Peyton-Jones, and A. Sabry. A monadic framework for subcontinuations. Available at `http://www.cs.indiana.edu/~sabry/research.html`, Feb. 2005.

[13] M. Felleisen. The theory and practice of first-class prompts. In J. Ferrante and P. Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, Jan. 1988. ACM Press.

[14] M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V., Amsterdam, 1986.

[15] M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In R. C. Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988. ACM Press.

[16] C. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In S. 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.

[17] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In B. 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.

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

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

[20] J. 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.

[21] C. Shan. Shift to control. In O. Shivers and O. Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Snowbird, Utah, Sept. 2004.

[22] G. L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Tech. report AI-TR-474.

[23] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

Recent BRICS Report Series Publications

RS-05-5 **Dariusz Biernacki, Olivier Danvy, and Kevin Millikin.** ***A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations (Preliminary Version)*****. February 2005. ii+16 pp. Superseded by BRICS RS-05-16.**

RS-05-4 **Andrzej Filinski and Henning Korsholm Rohde.** ***Denotational Aspects of Untyped Normalization by Evaluation*****. February 2005. 51 pp. Extended version of an article to appear in the FOSSACS 2004 special issue of RAIRO,** ***Theoretical Informatics and Applications*****.**

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

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

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

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

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

RS-04-39 **Olivier Danvy and Lasse R. Nielsen.** ***CPS Transformation of Beta-Redexes*****. December 2004. ii+11 pp. Extended version of an article appearing in** ***Information Processing Letters*****, 94(5):217–224, 2005. Also superseedes BRICS report RS-00-35.**

RS-04-38 **Olin Shivers and Mitchell Wand.** ***Bottom-Up β-Substitution: Uplinks and λ-DAGs*****. December 2004. iv+32 pp.**

RS-04-37 **Jørgen Iversen and Peter D. Mosses.** ***Constructive Action Semantics for Core ML*****. December 2004. 68 pp. To appear in a special** ***Language Definitions and Tool Generation*** **issue of the journal** ***IEE Proceedings Software*****.**