BRICS

Basic Research in Computer Science

BRICS RS-99-51 O. Danvy: Formalizing Implementation Strategies for First-Class Continuations

Formalizing Implementation Strategies for First-Class Continuations

Olivier Danvy

BRICS Report Series **RS-99-51**

ISSN 0909-0878 **December 1999**

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/99/51/`

Formalizing Implementation Strategies for First-Class Continuations *

Olivier Danvy

BRICS **
Department of Computer Science, University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
E-mail: danvy@brics.dk
Home page: http://www.brics.dk/~danvy

Abstract. We present the first formalization of implementation strategies for first-class continuations. The formalization hinges on abstract machines for continuation-passing style (CPS) programs with a special treatment for the current continuation, accounting for the essence of first-class continuations. These abstract machines are proven equivalent to a standard, substitution-based abstract machine. The proof techniques work uniformly for various representations of continuations. As a byproduct, we also present a formal proof of the two folklore theorems that one continuation identifier is enough for second-class continuations and that second-class continuations are stackable.

A large body of work exists on implementing continuations, but it is predominantly empirical and implementation-oriented. In contrast, our formalization abstracts the essence of first-class continuations and provides a uniform setting for specifying and formalizing their representation.

1 Introduction

Be it for coroutines, threads, mobile code, interactive computer games, or computer sessions, one often needs to suspend and to resume a computation. Suspending a computation amounts to saving away its state, and resuming a suspended computation amounts to restoring the saved state. Such saved copies may be ephemeral and restored at most once (e.g., coroutines, threads, and computer sessions that were 'saved to disk'), or they may need to be restored repeatedly (e.g., in a computer game). This functionality is reminiscent of *continuations*, which represent the rest of a computation [22].

In this article, we consider how to implement first-class continuations. A wealth of empirical techniques exist to take a snapshot of control during the execution of a program (call/cc) and to restore this snapshot (throw): SML/NJ, for example, allocates continuations entirely in the heap, reducing call/cc and throw to a matter of swapping pointers [1]; T and Scheme 48 allocate continuations on a stack, copying this stack in the heap and back to account for

---

* Extended version available as the technical report BRICS RS-99-51.

** Basic Research in Computer Science (http://www.brics.dk), Centre of the Danish National Research Foundation.

call/cc and throw [16, 17];[1] and PC Scheme, Chez Scheme, and Larceny allocate continuations on a segmented stack [2, 4, 15]. Clinger, Hartheimer, and Ost's recent article [4] provides a comprehensive overview of implementation strategies for first-class continuations and of their issues: ideally, first-class continuations should exert zero overhead for programs that do not use them.

Our goal and non-goal: We formalize implementation strategies for first-class continuations. We do not formalize first-class continuations per se (cf., e.g., Felleisen's PhD thesis [12] or Duba, Harper, and MacQueen's formal account of call/cc in ML [10]).

Our work: We consider abstract machines for continuation-passing style (CPS) programs, focusing on the implementation of continuations. As a stepping stone, we formalize the folklore theorem that one register is enough to implement second-class continuations. We then formalize the three implementation techniques for first-class continuations mentioned above: heap, stack, and segmented stack. The formalization and its proof techniques (structural induction on terms and on derivation trees) are uniform: besides clarifying what it means to implement continuations, be they second-class or first-class, our work provides a platform to state and prove the correctness of each implementation. Also, this platform is not restricted to CPS programs: through Flanagan et al.'s results [13], it is applicable to direct-style programs if one represents control with a stack of evaluation contexts instead of a stack of functions.

1.1 Related work

The four works most closely related to ours are Clinger, Hartheimer, and Ost's overview of implementation strategies for first-class continuations [4]; Flanagan, Sabry, Duba, and Felleisen's account of compiling with continuations and more specifically, their two first abstract machines [13]; Danvy and Lawall's syntactic characterization of second-class and first-class continuations in CPS programs [8]; and Danvy, Dzafic, and Pfenning's work on the occurrence of continuation parameters in CPS programs [6, 9, 11].

1.2 Overview

Section 2 presents our source language: the λ-calculus in direct style and in CPS, the CPS transformation, and an abstract machine for CPS programs that will be our reference point here. This standard machine treats continuation identifiers on par with all the other identifiers. The rest of this article focuses on continuation identifiers and how to represent their bindings – i.e., on the essence of how to implement continuations.

[1] This strategy is usually attributed to Drew McDermott in the late 70's [19], but apparently it was already considered in the early '70s at Queen Mary and Westfield College to implement PAL (John C. Reynolds, personal communication, Aarhus, Denmark, fall 1999).

Section 3 addresses second-class continuations. In a CPS program with second-class continuations, continuation identifiers are not only linear (in the sense of Linear Logic), but they also denote a stackable resource, and indeed it is folklore that second-class continuations can be implemented LIFO on a "control stack". We formalize this folklore by characterizing second-class continuations syntactically in a CPS program and by presenting an abstract machine where the bindings of continuation identifiers are represented with a stack. We show this stack machine to be equivalent to the standard one.

Section 4 addresses first-class continuations. In a CPS program with first-class continuations, continuation identifiers do not denote a stackable resource in general. First-class continuations, however, are relatively rare, and thus over the years, "zero-overhead" implementations have been sought [4]: implementations that do support first-class continuations but only tax programs that use them. We consider the traditional strategy of stack-allocating all continuations by default, as if they were all second-class, and of copying this stack in case of first-class continuations. We formalize this empirical strategy with a new abstract machine, which we show to be equivalent to the standard one.

Section 5 outlines how to formalize alternative implementation strategies, such as segmenting the stack and recycling unshared continuations.

2 CPS programs

We consider closed programs: direct-style (DS) λ-terms with literals. The BNF of DS programs is displayed in Figure 1. Assuming a call-by-value evaluation strategy, the BNF of CPS programs is displayed in Figure 2. CPS programs are prototypically obtained by CPS-transforming DS programs, as defined in Figure 3 [7, 20, 21].

Figure 4 displays our starting point: a standard abstract machine implementing β-reduction for CPS programs. This machine is a simplified version of another machine studied jointly with Belmina Dzafic and Frank Pfenning [6, 9, 11]. We use two judgments, indexed by the syntactic categories of CPS terms. The judgment

$$\vdash^{\text{CProg}}_{\text{std}} p \hookrightarrow a$$

is satisfied whenever a CPS program p evaluates to an answer a. The auxiliary judgment

$$\vdash^{\text{CExp}}_{\text{std}} e \hookrightarrow a$$

is satisfied whenever a CPS expression e evaluates to an answer a. The machine starts and stops with the initial continuation k_{init}, which is a distinguished fresh continuation identifier. Answers can be either the trivial expressions ℓ or $\lambda x.\lambda k.e$, or the error token.

For expository simplicity, our standard machine uses substitutions to implement variable bindings. Alternatively and equivalently, it could use an environment and represent functional values as closures [18]. And indeed Flanagan et al. present a similar standard abstract machine which uses an environment [13, Figure 4].

$p \in \mathrm{DProg}$	— DS programs	$p ::= e$
$e \in \mathrm{DExp}$	— DS expressions	$e ::= e_0\, e_1 \mid t$
$t \in \mathrm{DTriv}$	— DS trivial expressions	$t ::= \ell \mid x \mid \lambda x.e$
$\ell \in \mathrm{Lit}$	— literals	
$x \in \mathrm{Ide}$	— identifiers	

Fig. 1. BNF of DS programs

$p \in \mathrm{CProg}$	— CPS programs	$p ::= \lambda k.e$
$e \in \mathrm{CExp}$	— CPS (serious) expressions	$e ::= t_0\, t_1\, c \mid c\, t$
$t \in \mathrm{CTriv}$	— CPS trivial expressions	$t ::= \ell \mid x \mid v \mid \lambda x.\lambda k.e$
$c \in \mathrm{Cont}$	— continuations	$c ::= \lambda v.e \mid k$
$\ell \in \mathrm{Lit}$	— literals	
$x \in \mathrm{Ide}$	— source identifiers	
$k \in \mathrm{IdeC}$	— fresh continuation identifiers	
$v \in \mathrm{IdeV}$	— fresh parameters of continuations	
$a \in \mathrm{Answer}$	— CPS answers	$a ::= \ell \mid \lambda x.\lambda k.e \mid \mathrm{error}$

Fig. 2. BNF of CPS programs

$$
\begin{aligned}
[\![e]\!]^{\mathrm{DProg}}_{\mathrm{cps}} &= \lambda k.[\![e]\!]^{\mathrm{DExp}}_{\mathrm{cps}}\, k && \text{– where } k \text{ is fresh}\\[1ex]
[\![e_0\, e_1]\!]^{\mathrm{DExp}}_{\mathrm{cps}}\, c &= [\![e_0]\!]^{\mathrm{DExp}}_{\mathrm{cps}}\, \lambda v_0.[\![e_1]\!]^{\mathrm{DExp}}_{\mathrm{cps}}\, \lambda v_1.v_0\, v_1\, c && \text{– where } v_0 \text{ and } v_1 \text{ are fresh}\\
[\![t]\!]^{\mathrm{DExp}}_{\mathrm{cps}}\, c &= c\, [\![t]\!]^{\mathrm{DTriv}}_{\mathrm{cps}}\\[1ex]
[\![\ell]\!]^{\mathrm{DTriv}}_{\mathrm{cps}} &= \ell\\
[\![x]\!]^{\mathrm{DTriv}}_{\mathrm{cps}} &= x\\
[\![\lambda x.e]\!]^{\mathrm{DTriv}}_{\mathrm{cps}} &= \lambda x.\lambda k.[\![e]\!]^{\mathrm{DExp}}_{\mathrm{cps}}\, k && \text{– where } k \text{ is fresh}
\end{aligned}
$$

Fig. 3. The left-to-right, call-by-value CPS transformation

$$
\frac{\vdash^{\mathrm{CExp}}_{\mathrm{std}} e[k_{\mathrm{init}}/k] \hookrightarrow a}{\vdash^{\mathrm{CProg}}_{\mathrm{std}} \lambda k.e \hookrightarrow a}
$$

$$
\frac{}{\vdash^{\mathrm{CExp}}_{\mathrm{std}} \ell\, t\, c \hookrightarrow \mathrm{error}}
\qquad
\frac{\vdash^{\mathrm{CExp}}_{\mathrm{std}} e[t/x,\, c/k] \hookrightarrow a}{\vdash^{\mathrm{CExp}}_{\mathrm{std}} (\lambda x.\lambda k.e)\, t\, c \hookrightarrow a}
$$

$$
\frac{\vdash^{\mathrm{CExp}}_{\mathrm{std}} e[t/v] \hookrightarrow a}{\vdash^{\mathrm{CExp}}_{\mathrm{std}} (\lambda v.e)\, t \hookrightarrow a}
\qquad
\frac{}{\vdash^{\mathrm{CExp}}_{\mathrm{std}} k_{\mathrm{init}}\, t \hookrightarrow t}
$$

Fig. 4. Standard machine for CPS programs

3 A stack machine for CPS programs with second-class continuations

As a stepping stone, this section formalizes the folklore theorem that in the absence of first-class continuations, one continuation identifier is enough, i.e., in Figure 2, IdeC can be defined as a singleton set. To this end, we prove that in the output of the CPS transformation, only one continuation identifier is indeed enough. We also prove that this property is closed under arbitrary β-reduction. We then rephrase the BNF of CPS programs with IdeC as a singleton set (Section 3.1). In the new BNF, only CPS programs with second-class continuations can be expressed. We present a stack machine for these CPS programs and we prove it equivalent to the standard machine of Figure 4 (Section 3.2). Flanagan et al. present a similar abstract machine [13, Figure 5], but without relating it formally to their standard abstract machine.

3.1 One continuation identifier is enough

Each expression in a DS program occurs in one evaluation context. Correspondingly, each expression in a CPS program has one continuation. We formalize this observation in terms of continuation identifiers with the judgment defined in Figure 5, where $\mathrm{FC}(t)$ yields the set of continuation identifiers occurring free in t.

$$\frac{k \notin \mathrm{FC}(t_0) \qquad k \notin \mathrm{FC}(t_1) \qquad k \models^{\mathrm{Cont}}_{\mathrm{2cc}} c}{k \models^{\mathrm{CExp}}_{\mathrm{2cc}} t_0\, t_1\, c} \qquad \frac{k \models^{\mathrm{Cont}}_{\mathrm{2cc}} c \qquad k \notin \mathrm{FC}(t)}{k \models^{\mathrm{CExp}}_{\mathrm{2cc}} c\, t}$$

$$\frac{k \models^{\mathrm{CExp}}_{\mathrm{2cc}} e}{k \models^{\mathrm{Cont}}_{\mathrm{2cc}} \lambda v.e} \qquad \frac{}{k \models^{\mathrm{Cont}}_{\mathrm{2cc}} k}$$

Fig. 5. Characterization of a second-class continuation abstraction $\lambda k.e$

Definition 1 (Second-class position, second-class continuations). *In a continuation abstraction $\lambda k.e$, we say that k occurs in* second-class position *and denotes a* second-class continuation *whenever the judgment $k \models^{\mathrm{CExp}}_{2cc} e$ is satisfied.*

Below, we prove that actually, in the output of the CPS transformation, *all* continuation identifiers denote second-class continuations. In Figure 6, we thus generalize our judgment to a whole CPS program.

Definition 2 (2Cont-validity). *We say that a CPS program p is* 2Cont-valid *whenever the judgment $\models^{\mathrm{CProg}}_{2cc*} p$ is satisfied. Informally, $\models^{\mathrm{CProg}}_{2cc*} p$ holds if and only if all continuation abstractions $\lambda k.e$ occurring in p satisfy $k \models^{\mathrm{CExp}}_{2cc} e$.*

Lemma 1 (The CPS transformation yields 2Cont-valid programs). *For any $p \in \mathrm{DProg}$, $\models^{\mathrm{CProg}}_{2cc*} [\![p]\!]^{\mathrm{DProg}}_{\mathrm{cps}}$.*

Proof. A straightforward induction over DS programs. □

$$\frac{k \models^{\mathrm{CExp}}_{2cc^*} e}{\models^{\mathrm{CProg}}_{2cc^*} \lambda k.e}$$

$$\frac{\models^{\mathrm{CTriv}}_{2cc^*} t_0 \qquad \models^{\mathrm{CTriv}}_{2cc^*} t_1 \qquad k \models^{\mathrm{Cont}}_{2cc^*} c}{k \models^{\mathrm{CExp}}_{2cc^*} t_0\, t_1\, c} \qquad \frac{k \models^{\mathrm{Cont}}_{2cc^*} c \qquad \models^{\mathrm{CTriv}}_{2cc^*} t}{k \models^{\mathrm{CExp}}_{2cc^*} c\, t}$$

$$\frac{}{\models^{\mathrm{CTriv}}_{2cc^*} \ell} \qquad \frac{}{\models^{\mathrm{CTriv}}_{2cc^*} x} \qquad \frac{}{\models^{\mathrm{CTriv}}_{2cc^*} v} \qquad \frac{k \models^{\mathrm{CExp}}_{2cc^*} e}{\models^{\mathrm{CTriv}}_{2cc^*} \lambda x.\lambda k.e}$$

$$\frac{k \models^{\mathrm{CExp}}_{2cc^*} e}{k \models^{\mathrm{Cont}}_{2cc^*} \lambda v.e} \qquad \frac{}{k \models^{\mathrm{Cont}}_{2cc^*} k}$$

Fig. 6. Characterization of a CPS program with second-class continuations

Furthermore, 2Cont-validity is closed under β-reduction, which means that it is preserved by regular evaluation as well as by the arbitrary simplifications of a CPS compiler [21]. The corresponding formal statement and its proof are straightforward and omitted here: we rely on them in the proof of Theorem 1.

Therefore each use of each continuation identifier k is uniquely determined, capturing the fact that in the BNF of 2Cont-valid CPS programs, one continuation identifier is enough. To emphasize this fact, let us specialize the BNF of Figure 2 by defining IdeC as the singleton set $\{\star\}$, yielding the BNF of 2CPS programs displayed in Figure 7.

$p \in$ 2CProg	— 2CPS programs	$p ::= \lambda\star.e$
$e \in$ 2CExp	— 2CPS (serious) expressions	$e ::= t_0\, t_1\, c \mid c\, t$
$t \in$ 2CTriv	— 2CPS trivial expressions	$t ::= \ell \mid x \mid v \mid \lambda x.\lambda\star.e$
$c \in$ 2Cont	— continuations	$c ::= \lambda v.e \mid \star$
$\ell \in$ Lit	— literals	
$x \in$ Ide	— source identifiers	
$\star \in$ Token	— single continuation identifier	
$v \in$ IdeV	— fresh parameters of continuations	
$a \in$ 2Answer	— 2CPS answers	$a ::= \ell \mid \lambda x.\lambda\star.e \mid \mathrm{error}$

Fig. 7. BNF of 2CPS programs

Let $[\![\cdot]\!]^{\mathrm{CProg}}_{\mathrm{strip}}$ denote the straightforward homomorphic mapping from a 2Cont-valid CPS program to a 2CPS program and $[\![\cdot]\!]^{\mathrm{2CProg}}_{\mathrm{name}}$ denote its inverse, such that $\forall p \in \mathrm{CProg}$, $[\![[\![p]\!]^{\mathrm{CProg}}_{\mathrm{strip}}]\!]^{\mathrm{2CProg}}_{\mathrm{name}} \equiv_\alpha p$ whenever the judgment $\models^{\mathrm{CProg}}_{2cc^*} p$ is satisfied, and $\forall p' \in \mathrm{2CProg}$, $[\![[\![p]\!]^{\mathrm{2CProg}}_{\mathrm{name}}]\!]^{\mathrm{CProg}}_{\mathrm{strip}} = p'$. These two translations are generalized in Section 4 and thus we omit their definition here.

3.2 A stack machine for 2CPS programs

Figure 8 displays a stack-based abstract machine for 2CPS programs. We obtained it from the standard machine of Section 2, page 4, by implementing the bindings of continuation identifiers with a global "control stack" φ.

$$\varphi \in \text{2CStack} \text{ — control stacks} \qquad\qquad \varphi ::= \bullet \mid \varphi,\, \lambda v.e$$

The machine starts and stops with an empty control stack $\bullet$. When a function is applied, its continuation is pushed on φ. When a continuation is needed, it is popped from φ. If φ is empty, the intermediate result sent to the continuation is the final answer. We distinguish tail calls (i.e., function calls where the continuation is $\star$) by not pushing anything on φ, thereby achieving proper tail recursion.

$$\frac{\bullet \vdash^{\text{2CExp}}_{\text{2cc}} e \hookrightarrow a}{\vdash^{\text{2CProg}}_{\text{2cc}} \lambda\star.e \hookrightarrow a} \qquad \frac{}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} \ell\, t\, c \hookrightarrow \text{error}}$$

$$\frac{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} (\lambda x.\lambda\star.e)\, t\, \star \hookrightarrow a} \qquad \frac{\varphi,\, \lambda v.e' \vdash^{\text{2CExp}}_{\text{2cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} (\lambda x.\lambda\star.e)\, t\, \lambda v.e' \hookrightarrow a}$$

$$\frac{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/v] \hookrightarrow a}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} (\lambda v.e)\, t \hookrightarrow a} \qquad \frac{}{\bullet \vdash^{\text{2CExp}}_{\text{2cc}} \star\, t \hookrightarrow t} \qquad \frac{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/v] \hookrightarrow a}{\varphi,\, \lambda v.e \vdash^{\text{2CExp}}_{\text{2cc}} \star\, t \hookrightarrow a}$$

Fig. 8. Stack machine for 2CPS programs

N.B. The machine does not substitute continuations for continuation identifiers, and therefore one might be surprised by the rule handling the redex $(\lambda v.e)\, t$. Such redexes, however, can occur in the source program.

Formally, the judgment

$$\vdash^{\text{2CProg}}_{\text{2cc}} p \hookrightarrow a$$

is satisfied whenever a CPS program $p \in \text{2CProg}$ evaluates to an answer $a \in \text{2Answer}$. The auxiliary judgment

$$\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e \hookrightarrow a$$

is satisfied whenever an expression $e \in \text{2CExp}$ evaluates to an answer a, given a control stack $\varphi \in \text{2CStack}$.

We prove the equivalence between the stack machine and the standard machine by showing that the computations for each abstract machine (represented by derivations) are in bijective correspondence. To this end, we define a "control-stack substitution" over the state of the stack machine (i.e., expression under evaluation and current control stack) to obtain the state of the standard machine (i.e., expression under evaluation). We define control-stack substitution inductively over 2CPS expressions and continuations.

Definition 3 (Control-stack substitution for 2CPS programs). *Given a stack φ of* 2Cont *continuations, the stack substitution of any $e \in \text{2CExp}$ (resp. $c \in \text{2Cont}$), noted $e\{\varphi\}_2$ (resp. $c\{\varphi\}_2$), yields a* CExp *expression (resp. a* Cont *continuation) and is defined as follows.*

$$(t_0\, t_1\, c)\{\varphi\}_2 = [\![t_0]\!]^{\text{2CTriv}}_{\text{name}}\, [\![t_1]\!]^{\text{2CTriv}}_{\text{name}}\, (c\{\varphi\}_2) \qquad (\lambda v.e)\{\varphi\}_2 = \lambda v.(e\{\varphi\}_2)$$

$$\star\{\bullet\}_2 = k_{init}$$

$$(c\, t)\{\varphi\}_2 = (c\{\varphi\}_2)\, [\![t]\!]^{\text{2CTriv}}_{\text{name}} \qquad \star\{\varphi,\, \lambda v.e\}_2 = \lambda v.(e\{\varphi\}_2)$$

Stack substitution is our key tool for mapping a state of the stack machine into a state of the standard machine. It yields CExp expressions and Cont continuations that have one free continuation identifier: k_{init}.

Lemma 2 (2Cont-validity of stack-substituted expressions and continuations).

1. *For any* $e \in$ 2CExp *and for any stack of* 2Cont *continuations* φ*, the judgment* $k_{init} \models^{\text{CExp}}_{2cc*} e\{\varphi\}_2$ *is satisfied.*
2. *For any* $c \in$ 2Cont *and for any stack of* 2Cont *continuations* φ*, the judgment* $k_{init} \models^{\text{Cont}}_{2cc*} c\{\varphi\}_2$ *is satisfied.*

Proof. By mutual induction on the structure of e and c. □

Lemma 3 (Control-stack substitution for 2CPS programs).

1. *For any* $e' \in$ CExp *satisfying* $k \models^{\text{CExp}}_{2cc*} e'$ *for some* k *and for any stack of* 2Cont *continuations* φ, $[\![e']\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2 = e'[\star\{\varphi\}_2/k]$.
2. *For any* $e \in$ 2CExp*, for any* $t' \in$ CTriv *satisfying* $\models^{\text{CTriv}}_{2cc*} t'$*, for any identifier* i *in* Ide *or in* IdeV*, and for any stack of* 2Cont *continuations* φ, $e[[\![t']\!]^{\text{CTriv}}_{\text{strip}}/i]\{\varphi\}_2 = e\{\varphi\}_2[t'/i]$.

Theorem 1 (Simulation). *The stack machine of Figure 8 and the standard machine are equivalent:*

1. *For any 2Cont-valid CPS program p,*
   $\vdash^{\text{CProg}}_{std} p \hookrightarrow a$ *if and only if* $\vdash^{\text{2CProg}}_{2cc} [\![p]\!]^{\text{CProg}}_{\text{strip}} \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}$.
2. *For any CPS expression e satisfying* $k \models^{\text{CExp}}_{2cc*} e$ *for some k and for any stack of* 2Cont *continuations* φ*,*
   $\vdash^{\text{CExp}}_{std} [\![e]\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2 \hookrightarrow a$ *if and only if* $\varphi \vdash^{\text{2CExp}}_{2cc} [\![e]\!]^{\text{CExp}}_{\text{strip}} \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}$.

Proof. The theorem follows in each direction by an induction over the structure of the derivations, using Lemma 3. Let us show the case of tail calls in one direction.

$$\text{Case } \mathcal{E} = \frac{\overset{\mathcal{E}_1}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/x] \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}}}{\varphi \vdash^{\text{2CExp}}_{\text{2cc}} (\lambda x.\lambda\star.e)\, t\,\star \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}},$$

where $\mathcal{E}_1$ names the derivation ending in $\varphi \vdash^{\text{2CExp}}_{\text{2cc}} e[t/x] \hookrightarrow [\![a]\!]^{\text{Answer}}_{\text{strip}}$.

By applying the induction hypothesis to $\mathcal{E}_1$, we obtain a derivation

$$\overset{\mathcal{E}_1'}{\vdash^{\text{CExp}}_{\text{std}} e[t/x]\{\varphi\}_2 \hookrightarrow a}$$

Since $e[t/x]$ is a 2CPS expression, there exists a CPS expression e' satisfying $k \models^{\text{CExp}}_{2\text{cc}^*} e'$ for some k and there exists a CPS trivial expression t' satisfying $\models^{\text{CTriv}}_{2\text{cc}^*} t'$ such that $e = [\![e']\!]^{\text{CExp}}_{\text{strip}}$ and $t = [\![t']\!]^{\text{CTriv}}_{\text{strip}}$.

By Lemma 3,

$$\begin{aligned}
[\![e']\!]^{\text{CExp}}_{\text{strip}}[[\![t']\!]^{\text{CExp}}_{\text{strip}}/x]\{\varphi\}_2 &= [\![e']\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2[t'/x] \\
&= e'[\star\{\varphi\}_2/k][t'/x] \\
&= e'[t'/x, \star\{\varphi\}_2/k] && \text{– because } t' \text{ has no free } k \\
& && \text{and } \varphi \text{ has no free } x.
\end{aligned}$$

By inference,

$$\frac{\vdash^{\text{CExp}}_{\text{std}} e'[t'/x, \star\{\varphi\}_2/k] \hookrightarrow a}{\vdash^{\text{CExp}}_{\text{std}} (\lambda x.\lambda k.e')\, t'\, (\star\{\varphi\}_2) \hookrightarrow a}$$

Now by definition of stack substitution,

$$(\lambda x.\lambda k.e')\, t'\, (\star\{\varphi\}_2) = [\![(\lambda x.\lambda k.e)\, t\, k']\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2, \qquad \text{– for some } k'.$$

In other words, there exists a derivation

$$\frac{\begin{array}{c}\mathcal{E}_1' \\ \vdash^{\text{CExp}}_{\text{std}} [\![e[t/x]]\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2 \hookrightarrow a\end{array}}{\vdash^{\text{CExp}}_{\text{std}} [\![(\lambda x.\lambda k.e)\, t\, k']\!]^{\text{CExp}}_{\text{strip}}\{\varphi\}_2 \hookrightarrow a}$$

which is what we wanted to show. □

3.3 Summary and conclusion

As a stepping stone towards Section 4, we have formalized and proven two folklore theorems: (1) for CPS programs with second-class continuations, one identifier is enough; and (2) the bindings of continuation identifiers can be implemented with a stack for CPS programs with second-class continuations. To this end, we have considered a simplified abstract machine and taken the same conceptual steps as in our earlier joint work with Dzafic and Pfenning [6, 9, 11]. This earlier work is formalized in Elf, whereas the present work is not (yet). The rest of this article reports an independent foray. In the next section, we adapt the stack machine to CPS programs with first-class continuations, thereby formalizing an empirical implementation strategy for first-class continuations.

4 A stack machine for CPS programs with first-class continuations

First-class continuations occur because of call/cc. The call-by-value CPS transformation of call/cc reads as follows.

$$[\![\text{call/cc}\, e]\!]^{\text{DExp}}_{\text{cps}} c = [\![e]\!]^{\text{DExp}}_{\text{cps}} \lambda f.f\, (\lambda x.\lambda k.c\, x)\, c \quad \text{– where } f,\ x, \text{ and } k \text{ are fresh.}$$

On the right-hand-side of this definitional equation, c occurs twice: once as a regular, second-class continuation, and once more, in $\lambda x.\lambda k.c\, x$. In that term, k is declared but not used – c is used instead and denotes a first-class continuation.

Such CPS programs do not satisfy the judgments of Figures 5 and 6. And indeed, Danvy and Lawall observed that in a CPS program, first-class continuations can be detected through continuation identifiers occurring "out of turn", so to speak [8].

Because it makes no assumptions on the binding discipline of continuation identifiers, the standard machine of Section 2, page 4, properly handles CPS programs with first-class continuations. First-class continuations, however, disqualify the stack machine of Section 3, page 7.

The goal of this section is to develop a stack machine for CPS programs with first-class continuations. To this end, we formalize what it means for a continuation identifier to occur in first-class position. We also prove that arbitrary β-reduction never promotes a continuation identifier occurring in second-class position into one occurring in first-class position. We then rephrase the BNF of CPS programs to single out continuation identifiers occurring in first-class position and their declaration. And similarly to Section 3, we tag with "$\star$" all the declarations of continuation identifiers occurring in second-class position or not occurring at all, and all second-class positions of continuation identifiers (Section 4.1). We then present a stack machine for these 1CPS programs that copies the stack when first-class continuation abstractions are invoked. We prove it equivalent to the standard machine of Figure 4 (Section 4.2).

4.1 One continuation identifier is not enough

Following Danvy and Lawall [8], we now say that a continuation identifier occurs in first-class position whenever it occurs elsewhere than in second-class position, which is syntactically easy to detect. We formalize first-class occurrences with the judgment displayed in Figure 9.

$$\frac{k \in \mathrm{FC}(t_0)}{k \models^{\mathrm{CExp}}_{1\mathrm{cc}} t_0\, t_1\, c} \qquad \frac{k \in \mathrm{FC}(t_1)}{k \models^{\mathrm{CExp}}_{1\mathrm{cc}} t_0\, t_1\, c} \qquad \frac{k \models^{\mathrm{Cont}}_{1\mathrm{cc}} c}{k \models^{\mathrm{CExp}}_{1\mathrm{cc}} t_0\, t_1\, c}$$

$$\frac{k \models^{\mathrm{Cont}}_{1\mathrm{cc}} c}{k \models^{\mathrm{CExp}}_{1\mathrm{cc}} c\, t} \qquad \frac{k \in \mathrm{FC}(t)}{k \models^{\mathrm{CExp}}_{1\mathrm{cc}} c\, t}$$

$$\frac{k \models^{\mathrm{CExp}}_{1\mathrm{cc}} e}{k \models^{\mathrm{Cont}}_{1\mathrm{cc}} \lambda v.e}$$

Fig. 9. Characterization of a first-class continuation abstraction $\lambda k.e$

Definition 4 (First-class position, first-class continuations). *In a continuation abstraction $\lambda k.e$, we say that k occurs in* first-class position *and denotes a* first-class continuation *whenever the judgment $k \models^{\mathrm{CExp}}_{1cc} e$ is satisfied.*

N.B. For any continuation abstraction $\lambda k.e$, at most one of $k \models^{\mathrm{CExp}}_{1\mathrm{cc}} e$ and $k \models^{\mathrm{CExp}}_{2\mathrm{cc}} e$ is satisfied.

In Section 3, we stated that 2Cont-validity is closed under β-reduction. Similarly here, β-reduction may demote a first-class continuation identifier into a second-class one, but it can never promote a second-class continuation identifier into a first-class one. The corresponding formal statement and its proof are straightforward and omitted here: we rely on them in the proof of Theorem 2.

For example, in

$$\lambda k.(\lambda x.\lambda k'.k\, x)\, \ell\, k$$

k occurs in first-class position. However, β-reducing this term yields

$$\lambda k.k\, \ell$$

where k occurs in second-class position.

In Section 3, we capitalized on the fact that each second-class position was uniquely determined. Here, we still capitalize on this fact by only singling out continuation identifiers in first-class position.[2]

Introduction: For all continuation abstractions $\lambda k.e$ satisfying $k \models^{\mathrm{CExp}}_{\mathrm{1cc}} e$, we tag the declaration of k with λ^1 and we keep the name k. Otherwise, we replace it with $\star$.

Elimination: When a continuation identifier occurs, if it is the latest one declared, we replace it with $\star$; otherwise, we keep its name.

The resulting BNF for 1CPS programs is displayed in Figure 10. The back and forth translation functions are displayed in Figures 11 and 12. They generalize their counterpart in Section 3.

Lemma 4 (Inverseness of stripping and naming).
$\forall p \in \mathrm{CProg}, [\![[\![p]\!]^{\mathrm{CProg}}_{\mathrm{strip}}]\!]^{\mathrm{1CProg}}_{\mathrm{name}} \equiv_\alpha p$ *and* $\forall p' \in \mathrm{1CProg}, [\![[\![p']\!]^{\mathrm{1CProg}}_{\mathrm{name}}]\!]^{\mathrm{CProg}}_{\mathrm{strip}} = p'$.

4.2 A stack machine for CPS programs with first-class continuations

We handle first-class continuations by extending the formalization of Section 3 with a new syntactic form:

$$c \in \mathrm{1Cont} \quad \text{— continuations} \qquad\qquad c ::= \lambda v.e \mid \star \mid k \mid \mathrm{swap}\,\varphi$$

The new form $\mathrm{swap}\,\varphi$ makes it possible to represent a copy of the control stack φ. It requires us to extend control-stack substitution as follows.

Definition 5 (Control-stack substitution for 1CPS programs). *Given a stack φ of* 1Cont *continuations, The stack substitution of any $e \in$* 1CExp *(resp. $c \in$* 1Cont*), noted $e\{\varphi\}_1$ (resp. $c\{\varphi\}_1$), yields a* CExp *expression (resp. a* Cont *continuation) and is defined as follows.*

$$
\begin{aligned}
(t_0\, t_1\, c)\{\varphi\}_1 &= ([\![t_0]\!]^{\mathrm{1CTriv}}_{\mathrm{name}}\, [\![t_1]\!]^{\mathrm{1CTriv}}_{\mathrm{name}})\, (c\{\varphi\}_1)\\
(c\, t)\{\varphi\}_1 &= (c\{\varphi\}_1)\, [\![t]\!]^{\mathrm{1CTriv}}_{\mathrm{name}}
\end{aligned}
\qquad
\begin{aligned}
(\lambda v.e)\{\varphi\}_1 &= \lambda v.(e\{\varphi\}_1)\\
\star\{\bullet\}_1 &= k_{init}\\
\star\{\varphi,\, \lambda v.e\}_1 &= \lambda v.(e\{\varphi\}_1)\\
k\{\varphi\}_1 &= k\\
(\mathrm{swap}\,\varphi')\{\varphi\}_1 &= \star\{\varphi'\}_1
\end{aligned}
$$

[2] Andrzej Filinski suggested this concise notation (personal communication, Aarhus, Denmark, summer 1999).

$p \in$ 1CProg	— 1CPS programs	$p ::= \lambda\star.e \mid \lambda^1 k.e$
$e \in$ 1CExp	— 1CPS (serious) expressions	$e ::= t_0\, t_1\, c \mid c\, t$
$t \in$ 1CTriv	— 1CPS trivial expressions	$t ::= \ell \mid x \mid v \mid \lambda x.\lambda\star.e \mid \lambda x.\lambda^1 k.e$
$c \in$ 1Cont	— continuations	$c ::= \lambda v.e \mid \star \mid k$
$\ell \in$ Lit	— literals	
$x \in$ Ide	— source identifiers	
$k \in$ IdeC	— fresh continuation identifiers	
$\star \in$ Token	— single continuation identifier	
$v \in$ IdeV	— fresh parameters of continuations	
$a \in$ 1Answer	— 1CPS answers	$a ::= \ell \mid \lambda x.\lambda\star.e \mid \lambda x.\lambda^1 k.e \mid \text{error}$

Fig. 10. BNF of 1CPS programs

$$[\![\lambda k.e]\!]^{\text{CProg}}_{\text{strip}} = \begin{cases} \lambda^1 k.[\![e]\!]^{\text{CExp}}_{\text{strip}}\, k & \text{if } k \models^{\text{CExp}}_{\text{1cc}} e \\ \lambda\star.[\![e]\!]^{\text{CExp}}_{\text{strip}}\, k & \text{otherwise} \end{cases}$$

$$\begin{aligned} [\![t_0\, t_1\, c]\!]^{\text{CExp}}_{\text{strip}}\, k &= [\![t_0]\!]^{\text{CTriv}}_{\text{strip}}\, [\![t_1]\!]^{\text{CTriv}}_{\text{strip}}\, ([\![c]\!]^{\text{Cont}}_{\text{strip}}\, k) \\ [\![c\, t]\!]^{\text{CExp}}_{\text{strip}}\, k &= ([\![c]\!]^{\text{Cont}}_{\text{strip}}\, k)\, [\![t]\!]^{\text{CTriv}}_{\text{strip}} \end{aligned}$$

$$\begin{aligned} [\![\ell]\!]^{\text{CTriv}}_{\text{strip}} &= \ell \\ [\![x]\!]^{\text{CTriv}}_{\text{strip}} &= x \\ [\![v]\!]^{\text{CTriv}}_{\text{strip}} &= v \end{aligned}$$

$$[\![\lambda x.\lambda k.e]\!]^{\text{CTriv}}_{\text{strip}} = \begin{cases} \lambda x.\lambda^1 k.[\![e]\!]^{\text{CExp}}_{\text{strip}}\, k & \text{if } k \models^{\text{CExp}}_{\text{1cc}} e \\ \lambda x.\lambda\star.[\![e]\!]^{\text{CExp}}_{\text{strip}}\, k & \text{otherwise} \end{cases}$$

$$[\![\lambda v.e]\!]^{\text{Cont}}_{\text{strip}}\, k = \lambda v.[\![e]\!]^{\text{CExp}}_{\text{strip}}\, k$$

$$[\![k']\!]^{\text{Cont}}_{\text{strip}}\, k = \begin{cases} \star & \text{if } k = k' \\ k' & \text{otherwise} \end{cases}$$

Fig. 11. Translation from CPS to 1CPS – stripping continuation identifiers

$$\begin{aligned} [\![\lambda\star.e]\!]^{\text{1CProg}}_{\text{name}} &= \lambda k.[\![e]\!]^{\text{1CExp}}_{\text{name}}\, k & & \text{– where } k \text{ is fresh} \\ [\![\lambda^1 k.e]\!]^{\text{1CProg}}_{\text{name}} &= \lambda k.[\![e]\!]^{\text{1CExp}}_{\text{name}}\, k \end{aligned}$$

$$\begin{aligned} [\![t_0\, t_1\, c]\!]^{\text{1CExp}}_{\text{name}}\, k &= [\![t_0]\!]^{\text{1CTriv}}_{\text{name}}\, [\![t_1]\!]^{\text{1CTriv}}_{\text{name}}\, ([\![c]\!]^{\text{1Cont}}_{\text{name}}\, k) \\ [\![c\, t]\!]^{\text{1CExp}}_{\text{name}}\, k &= ([\![c]\!]^{\text{1Cont}}_{\text{name}}\, k)\, [\![t]\!]^{\text{1CTriv}}_{\text{name}} \end{aligned}$$

$$\begin{aligned} [\![\ell]\!]^{\text{1CTriv}}_{\text{name}} &= \ell \\ [\![x]\!]^{\text{1CTriv}}_{\text{name}} &= x \\ [\![v]\!]^{\text{1CTriv}}_{\text{name}} &= v \end{aligned}$$

$$\begin{aligned} [\![\lambda x.\lambda\star.e]\!]^{\text{1CTriv}}_{\text{name}} &= \lambda x.\lambda k.[\![e]\!]^{\text{1CExp}}_{\text{name}}\, k & & \text{– where } k \text{ is fresh} \\ [\![\lambda^1 x.\lambda k.e]\!]^{\text{1CTriv}}_{\text{name}} &= \lambda x.\lambda k.[\![e]\!]^{\text{1CExp}}_{\text{name}}\, k \end{aligned}$$

$$\begin{aligned} [\![\lambda v.e]\!]^{\text{1Cont}}_{\text{name}}\, k &= \lambda v.[\![e]\!]^{\text{1CExp}}_{\text{name}}\, k \\ [\![\star]\!]^{\text{1Cont}}_{\text{name}}\, k &= k \\ [\![k']\!]^{\text{1Cont}}_{\text{name}}\, k &= k' \end{aligned}$$

$$\begin{aligned} [\![\ell]\!]^{\text{1Answer}}_{\text{name}} &= \ell \\ [\![\lambda x.\lambda\star.e]\!]^{\text{1Answer}}_{\text{name}} &= \lambda x.\lambda k.[\![e]\!]^{\text{1CExp}}_{\text{name}}\, k & & \text{– where } k \text{ is fresh} \\ [\![\lambda^1 x.\lambda k.e]\!]^{\text{1Answer}}_{\text{name}} &= \lambda x.\lambda k.[\![e]\!]^{\text{1CExp}}_{\text{name}}\, k \\ [\![\text{error}]\!]^{\text{1Answer}}_{\text{name}} &= \text{error} \end{aligned}$$

Fig. 12. Translation from 1CPS to CPS – naming continuation identifiers

$$\frac{\bullet \vdash^{\text{1CExp}}_{\text{1cc}} e \hookrightarrow a}{\vdash^{\text{1CProg}}_{\text{1cc}} \lambda\star.e \hookrightarrow a} \qquad \frac{\bullet \vdash^{\text{1CExp}}_{\text{1cc}} e[\text{swap}\,\bullet/k] \hookrightarrow a}{\vdash^{\text{1CProg}}_{\text{1cc}} \lambda^1 k.e \hookrightarrow a}$$

$$\frac{}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} \ell\, t\, c \hookrightarrow \text{error}}$$

$$\frac{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} (\lambda x.\lambda\star.e)\, t\, \star \hookrightarrow a} \qquad \frac{\varphi,\, \lambda v.e' \vdash^{\text{1CExp}}_{\text{1cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} (\lambda x.\lambda\star.e)\, t\, \lambda v.e' \hookrightarrow a}$$

$$\frac{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} e[t/x,\, \text{swap}\,\varphi/k] \hookrightarrow a}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} (\lambda x.\lambda^1 k.e)\, t\, \star \hookrightarrow a} \qquad \frac{\varphi,\, \lambda v.e' \vdash^{\text{1CExp}}_{\text{1cc}} e[t/x,\, \text{swap}\,(\varphi,\, \lambda v.e')/k] \hookrightarrow a}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} (\lambda x.\lambda^1 k.e)\, t\, \lambda v.e' \hookrightarrow a}$$

$$\frac{\varphi' \vdash^{\text{1CExp}}_{\text{1cc}} e[t/x] \hookrightarrow a}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} (\lambda x.\lambda\star.e)\, t\, (\text{swap}\,\varphi') \hookrightarrow a} \qquad \frac{\varphi' \vdash^{\text{1CExp}}_{\text{1cc}} e[t/x,\, \text{swap}\,\varphi'/k] \hookrightarrow a}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} (\lambda x.\lambda^1 k.e)\, t\, (\text{swap}\,\varphi') \hookrightarrow a}$$

$$\frac{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} e[t/v] \hookrightarrow a}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} (\lambda v.e)\, t \hookrightarrow a} \qquad \frac{}{\bullet \vdash^{\text{1CExp}}_{\text{1cc}} \star t \hookrightarrow t} \qquad \frac{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} e[t/v] \hookrightarrow a}{\varphi,\, \lambda v.e \vdash^{\text{1CExp}}_{\text{1cc}} \star t \hookrightarrow a}$$

$$\frac{}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} \text{swap}\,\bullet\, t \hookrightarrow t} \qquad \frac{\varphi' \vdash^{\text{1CExp}}_{\text{1cc}} e[t/v] \hookrightarrow a}{\varphi \vdash^{\text{1CExp}}_{\text{1cc}} \text{swap}\,(\varphi',\, \lambda v.e)\, t \hookrightarrow a}$$

Fig. 13. Stack machine for 1CPS programs

Figure 13 displays a stack-based abstract machine for 1CPS programs. This machine is a version of the stack machine of Section 3 where the substitution for continuation identifiers occurring in second-class position or not occurring at all is implemented with a global control stack (as in Figure 8), and where the substitution for continuation identifiers occurring in first-class position is implemented by copying the stack into a swap form (which is new).

Calls: When a function declaring a second-class continuation is applied, its continuation is pushed on φ. When a function declaring a first-class continuation is applied, its continuation is also pushed on φ and the resulting new stack is copied into a swap form.

Returns: When a continuation is needed, it is popped from φ. If φ is empty, the intermediate result sent to the continuation is the final answer. When a swap form is encountered, its copy of φ is restored.

More formally, the judgment

$$\vdash^{\text{1CProg}}_{\text{1cc}} p \hookrightarrow a$$

is satisfied whenever a CPS program $p \in$ 1CProg evaluates to an answer $a \in$ 1Answer. The auxiliary judgment

$$\varphi \vdash^{\text{1CExp}}_{\text{1cc}} e \hookrightarrow a$$

is satisfied whenever an expression $e \in$ 1CExp evaluates to an answer a, given a control stack $\varphi \in$ 1CStack. The machine starts and stops with an empty control stack.

We prove the equivalence between the stack machine and the standard machine as in Section 3.2.

Theorem 2 (Simulation). *The stack machine of Figure 13 and the standard machine are equivalent:*

1. $\vdash^{\mathrm{CProg}}_{std} p \hookrightarrow a$ *if and only if* $\vdash^{\mathrm{1CProg}}_{1cc} [\![p]\!]^{\mathrm{CProg}}_{\mathrm{strip}} \hookrightarrow [\![a]\!]^{\mathrm{Answer}}_{\mathrm{strip}}$.
2. $\vdash^{\mathrm{CExp}}_{std} [\![e]\!]^{\mathrm{CExp}}_{\mathrm{strip}} k\{\varphi\}_1 \hookrightarrow a$ *if and only if* $\varphi \vdash^{\mathrm{1CExp}}_{1cc} [\![e]\!]^{\mathrm{CExp}}_{\mathrm{strip}} k \hookrightarrow [\![a]\!]^{\mathrm{Answer}}_{\mathrm{strip}}$, *for some* k.

Proof. Similar to the proof of Theorem 1. □

4.3 Summary and conclusion

We have formalized and proven correct a stack machine for CPS programs with first-class continuations. This machine is idealized in that, e.g., it has no provision for stack overflow. Nevertheless, it embodies the most classical implementation strategy for first-class continuations: the stack is copied at call/cc time, i.e., in the CPS world, when a first-class continuation identifier is declared; and conversely, the stack is restored at throw time, i.e., in the CPS world, when a first-class continuation identifier is invoked. This design keeps second-class continuations costless – in fact it is a zero-overhead strategy in the sense of Clinger, Hartheimer, and Ost [4, Section 3.1]: only programs using first-class continuations pay for them.

Furthermore, and as in Section 3, our representation of φ embodies its LIFO nature without committing to an actual representation. This representation can be retentive (in which case φ is implemented as a pointer into the heap) or destructive (in which case φ is implemented as, e.g., a rewriteable array) [3]. In both cases, swap φ is implemented as copying φ. Copying the pointer yields captured continuations to be shared and copying the array yields multiple representations of captured continuations.

5 A segmented stack machine for first-class continuations

Coroutines and threads are easily simulated using call/cc, but these simulations are allergic to representing control as a rewriteable array. Indeed for every switch this array is copied in the heap, yielding multiple copies to coexist without sharing, even though these copies are mostly identical.

Against this backdrop, implementations such as PC Scheme [2] segment the stack, using the top segment as a stack cache: if this cache overflows, it is flushed to the heap and the computation starts afresh with an empty cache; and if it underflows, the last flushed cache is restored. Flushed caches are linked LIFO in the heap.[3] A segmented stack accomodates call/cc and throw very simply: at call/cc time, the cache is flushed to the heap and a pointer to it is retained; and

[3] If the size of the stack cache is one, the segmented implementation coincides with a heap implementation.

at throw time, the flushed cache that is pointed to is restored. As for the bulk of the continuations, it is not copied but shared between captured continuations.

It is simple to expand the stack machine of Section 4 into a segmented stack machine. One simply needs to define the judgment

$$\Phi\,;\varphi \vdash^{\mathrm{CExp}}_{1\mathrm{cc}'} e \hookrightarrow a$$

where φ, e, and a are in Section 4 and Φ denotes a LIFO list of φ's. (One also needs an overflow predicate for φ.)

Thus equipped, it is also simple to expand the stack substitution of Section 4, and to state and prove a simulation theorem similar to Theorem 2, thereby formalizing what Clinger, Hartheimer, and Ost name the "chunked-stack strategy" [4]. Another moderate effort makes it possible to formalize the author's incremental garbage collection of unshared continuations by one-bit reference counting [5]. One is also in position to formalize "one-shot continuations" [14].

Acknowledgments: I am grateful to Belmina Dzafic and Frank Pfenning for our joint work, which forms the foundation of the present foray. Throughout, and as always, Andrzej Filinski has been a precious source of sensible comments and suggestions. This article has also benefited from the interest and comments of Lars R. Clausen, Daniel Damian, Bernd Grobauer, Niels O. Jensen, Julia L. Lawall, Lasse R. Nielsen, Morten Rhiger, and Zhe Yang. I am also grateful for the opportunity to have presented this work at Marktoberdorf, at the University of Tokyo, and at KAIST in the summer and in the fall of 1999. Finally, thanks are due to the anonymous referees for stressing the issue of retention vs. deletion.

References

1. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991.
2. David B. Bartley and John C. Jensen. The implementation of PC Scheme. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 86–93, Cambridge, Massachusetts, August 1986.
3. Daniel M. Berry. Block structure: Retention or deletion? (extended abstract). In *Conference Record of the Third Annual ACM Symposium on Theory of Computing*, pages 86–100, Shaker Heights, Ohio, May 1971.
4. William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.
5. Olivier Danvy. Memory allocation and higher-order functions. In *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, SIGPLAN Notices, Vol. 22, No 7, pages 241–252, Saint-Paul, Minnesota, June 1987.
6. Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. In *Third International Workshop on Higher-Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, pages 19–31, Paris, France, September 1999.

7. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
8. Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In *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.
9. Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1995.
10. Bruce F. Duba, Robert Harper, and David B. MacQueen. Typing first-class continuations in ML. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Orlando, Florida, January 1991.
11. Belmina Dzafic. Formalizing program transformations. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.
12. Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.
13. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993.
14. Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, 1987.
15. Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *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.
16. Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–336, 1994.
17. David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the 1986 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 21, No 7, pages 219–233, Palo Alto, California, June 1986.
18. Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
19. Drew McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped Lisp. In *Conference Record of the 1980 LISP Conference*, pages 154–162, Stanford, California, August 1980.
20. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
21. Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.
22. Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2), 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974).

Recent BRICS Report Series Publications

RS-99-51 **Olivier Danvy.** ***Formalizing Implementation Strategies for First-Class Continuations.*** **December 1999. 16 pp. Appears in Smolka, editor,** ***Programming Languages and Systems: Ninth European Symposium on Programming*****, ESOP '00 Proceedings, LNCS 1782, 2000pp. 88–103.**

RS-99-50 **Gerth Stølting Brodal and Srinivasan Venkatesh.** ***Improved Bounds for Dictionary Look-up with One Error.*** **December 1999. 5 pp. Appears in** ***Information Processing Letters*** **75(1–2):57–59, 2000.**

RS-99-49 **Alexander A. Ageev and Maxim I. Sviridenko.** ***An Approximation Algorithm for Hypergraph Max k-Cut with Given Sizes of Parts.*** **December 1999. 12 pp. Appears in Paterson, editor,** ***Eighteenth Annual European Symposiumon on Algorithms*****, ESA '00 Proceedings, LNCS 1879, 2000, pages 32–49.**

RS-99-48 **Rasmus Pagh.** ***Faster Deterministic Dictionaries.*** **December 1999. 14 pp. Appears in Shmoys, editor,** ***The Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*****, SODA '00 Proceedings, 2000, pages 487–493. Journal version in** ***Journal of Algorithms*****, 41(1):69–85, 2001, with the title** ***Deterministic Dictionaries*****.**

RS-99-47 **Peter Bro Miltersen and Vinodchandran N. Variyam.** ***Derandomizing Arthur-Merlin Games using Hitting Sets.*** **December 1999. 21 pp. Appears in Beame, editor,** ***40th Annual Symposium on Foundations of Computer Science*****, FOCS '99 Proceedings, 1999, pages 71–80.**

RS-99-46 **Peter Bro Miltersen, Vinodchandran N. Variyam, and Osamu Watanabe.** ***Super-Polynomial Versus Half-Exponential Circuit Size in the Exponential Hierarchy.*** **December 1999. 14 pp. Appears in Asano, Imai, Lee, Nakano and Tokuyama, editors,** ***Computing and Combinatorics: 5th Annual International Conference*****, COCOON '99 Proceedings, LNCS 1627, 1999, pages 210–220.**

RS-99-45 **Torben Amtoft.** ***Partial Evaluation for Constraint-Based Program Analyses.*** **December 1999. 13 pp.**