An Operational Foundation for Delimited Continuations

Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy

BRICS *

Department of Computer Science

University of Aarhus †

Abstract

We derive an abstract machine that corresponds to a definitional interpreter for the control operators shift and reset. Based on this abstract machine, we construct a syntactic theory of delimited continuations.

Both the derivation and the construction scale to the family of control operators shift_n and reset_n. The definitional interpreter for shift_n and reset_n has $n+1$ layers of continuations, the corresponding abstract machine has $n+1$ layers of control stacks, and the corresponding syntactic theory has $n+1$ layers of evaluation contexts.

1 Introduction

The studies of delimited continuations can be classified in two groups: those that use continuation-passing style (CPS) and those that rely on operational intuitions about control instead. Of the latter, there is a large number [17, 20, 22, 26, 28, 29, 35, 39, 42, 43], with relatively few applications. Of the former, there is the work revolving around the control operators shift and reset [10, 11], with relatively many applications.

The original motivation for shift and reset was a continuation-based programming pattern involving several layers of continuations. The original specification relied both on a repeated CPS transformation and on a definitional interpreter with several levels of continuations (as is obtained by repeatedly transforming a direct-style interpreter into continuation-passing style). Only subsequently have shift and reset been specified operationally, by developing operational analogues of continuation semantics and of CPS transformations [15]. Beyond their original publication, shift and reset have been studied separately by Danvy and by Filinski [7, 15, 23, 24, 33], and independently by others [4, 25, 31, 36, 46, 47].

The goal of our work is to establish an operational foundation for delimited continuations by using CPS as a guideline. To this end, we start with the original definitional interpreter for shift and reset. This interpreter uses two layers of continuations: a continuation and a meta-continuation. We then defunctionalize it into an abstract machine [1] and we construct the corresponding syntactic theory [16], as pioneered by Felleisen and Friedman [19]. The construction scales to shift_n and reset_n.

This article is structured as follows. We first review the enabling technology of our work: Reynolds's defunctionalization, the observation that a defunctionalized CPS program implements an abstract machine, and the observation that Felleisen's evaluation contexts are the defunctionalized continuations of a continuation-passing evaluator; we also review related work (Section 2). We then defunctionalize the original definitional interpreter for shift and reset into an abstract machine. This abstract machine is environment-based, and we restate it as an abstract machine based on substitutions (Section 3). We analyze this abstract machine and construct the corresponding syntactic theory (Sections 4 and 5). We also present the abstract machine corresponding to the second level of the CPS hierarchy (Section 6), and we outline how the overall approach scales to higher levels (Section 7).

2 Background and related work

2.1 Defunctionalization

In his seminal work on definitional interpreters [40], Reynolds presented a generalization of closure conversion [32]: defunctionalization. This transformation amounts to representing a functional value not as a function, but as a first-order sum where each summand corresponds to a lambda-abstraction in a source program. Function introduction is thus represented as an injection, and function elimination as a case dispatch. Therefore, before defunctionalization, functional values are inhabitants of a function space and they are instances of anonymous lambda-abstractions, and after defunctionalization, functional values are inhabitants of a sum. In ML, sums are represented as a data type and injections as data-type constructors.

As a concrete example, let us consider the Fibonacci function in continuation-passing style:

```
(*  fib : int * (int -> int) -> int  *)
fun fib (0, k)
    = k 0
  | fib (1, k)
    = k 1
  | fib (n, k)
    = fib (n-1,
           fn v1 => fib (n-2,
                         fn v2 => k (v1+v2)))

(*  main : int -> int  *)
fun main n
    = fib (n, fn v => v)
```

We defunctionalize this program by representing the continuation as a data structure. All three source lambda-abstractions give rise to inhabitants of the function space `int -> int`. We specify the data structure representing the continuation as an ML data type, and we

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

† Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
E-mail: {`mbiernac,dabi,danvy`}`@brics.dk`

add an apply function to interpret elements of this data type:

```
datatype cont = CONT0
              | CONT1 of int * cont
              | CONT2 of int * cont

(*  apply_cont : cont * int -> int  *)
fun apply_cont (CONT0, v)
    = v
  | apply_cont (CONT1 (n, k), v1)
    = fib (n-2, CONT2 (v1, k))
  | apply_cont (CONT2 (v1, k), v2)
    = apply_cont (k, v1+v2)

(*  fib : int * cont -> int  *)
and fib (0, k)
    = apply_cont (k, 0)
  | fib (1, k)
    = apply_cont (k, 1)
  | fib (n, k)
    = fib (n-1, CONT1 (n, k))

(*  main : int -> int  *)
fun main n
    = fib (n, CONT0)
```

The constructor `CONT0` is constant because the initial continuation has no free variables. The constructor `CONT1` holds the values of the two free variables of the outer lambda-abstraction in the induction case, i.e., `n` and `k`, and the constructor `CONT2` holds the values of the two free variables of the inner lambda-abstraction in the induction case, i.e., `v1` and `k`. (One could have chosen to hoist the computation `n-2` from the definition of `apply_cont` to the definition of `fib`. This choice can make a difference in practice [12, 13].)

2.2 This work

The present work builds on two recent observations:

1. a defunctionalized CPS program implements an abstract machine [1, 8]; and
2. Felleisen's evaluation contexts are defunctionalized continuations [12].

Let us describe each of these observations in more detail.

2.2.1 *Abstract machines as defunctionalized CPS programs*

Plotkin's Indifference Theorem [37] states that CPS programs are independent of their evaluation order. In Reynolds's words [40], all the subterms in applications are 'trivial'; and in Moggi's words [34], these subterms are values and not computations. Furthermore, CPS programs are tail recursive [44]. Therefore, a defunctionalized CPS program implements the transition functions of an abstract machine. Each configuration is the name of a function together with its arguments.

Getting back to the example above, the defunctionalized definition of the Fibonacci function can be reformatted as the following abstract machine:

- Expressible values (integers):

$$v ::= n$$

- Evaluation contexts:

$$C ::= \mathtt{CONT0} \mid \mathtt{CONT1}\,(n, C) \mid \mathtt{CONT2}\,(v, C)$$

- Initial transition, transition rules, and final transition:

n	$\Rightarrow$	$\langle n,\, \mathtt{CONT0}\rangle_{\mathit{fib}}$
$\langle 0,\, k\rangle_{\mathit{fib}}$	$\Rightarrow$	$\langle k,\, 0\rangle_{\mathit{app}}$
$\langle 1,\, k\rangle_{\mathit{fib}}$	$\Rightarrow$	$\langle k,\, 1\rangle_{\mathit{app}}$
$\langle n,\, k\rangle_{\mathit{fib}}$	$\Rightarrow$	$\langle n-1,\, \mathtt{CONT1}\,(n,k)\rangle_{\mathit{fib}}$
$\langle \mathtt{CONT1}\,(n,k),\, v_1\rangle_{\mathit{app}}$	$\Rightarrow$	$\langle n-2,\, \mathtt{CONT2}\,(v_1,k)\rangle_{\mathit{fib}}$
$\langle \mathtt{CONT2}\,(v_1,k),\, v_2\rangle_{\mathit{app}}$	$\Rightarrow$	$\langle k,\, v_1+v_2\rangle_{\mathit{app}}$
$\langle \mathtt{CONT0},\, v\rangle_{\mathit{app}}$	$\Rightarrow$	v

Ager, Biernacki, Danvy, and Midtgaard have built on this observation to establish a functional correspondence between evaluators and abstract machines by relating them using closure conversion, CPS transformation, and defunctionalization [1, 8]. For example, Krivine's abstract machine corresponds to an ordinary call-by-name evaluator and Felleisen et al.'s CEK machine to an ordinary call-by-value evaluator. (In fact, these two machines can be derived from the *same* vanilla evaluator, resp. using a call-by-name CPS transformation and a call-by-value CPS transformation [9].) The correspondence makes it possible to exhibit the evaluators corresponding to the SECD machine [32], the CLS machine [27], and the Categorical Abstract Machine [6], and it also holds for call-by-need evaluators and lazy abstract machines [2], for computational effects [3], and for logic programming [5]. We apply it here to delimited continuations.

2.2.2 *Evaluation contexts as defunctionalized continuations*

The realization that Felleisen et al.'s evaluation contexts are defunctionalized continuations makes it possible to mechanically construct evaluation contexts. This mechanical construction contrasts with having to define evaluation contexts on a case-by-case basis [18]. Also, the ubiquitous unique-decomposition lemma follows as a corollary when one starts from a compositional evaluator [9].

2.3 Control operators for delimited continuations

The continuation-based programming pattern that motivated shift and reset has since been found to coincide with layered computational monads [24]. Several implementations have been developed: a definitional interpreter [10], a CPS transformation [11], two embeddings in Standard ML of New Jersey using call/cc and state [15, 23], and native run-time support in a Scheme system [25]. Sustained efforts have also been made to establish an equational theory of delimited continuations [30, 31] with the goal of studying their logical content.

A specificity of our work is that we use CPS as a guideline. For example, pure contexts and general evaluation contexts have long been distinguished [21, 41]. In their work [31], Kameyama and Hasegawa required this distinction. In contrast, the distinction between contexts and meta-contexts was imposed on us by CPS.

A forerunner of our work is Murthy's presentation at CW'92 [36], where he designed an abstract machine for the CPS hierarchy that actually coincides with ours. Murthy also introduced a typing system, proved it correct with respect to the CPS transla-

```
structure Syntax
= struct
    type ide = string

    datatype term = INT of int
                  | VAR of ide
                  | LAM of ide * term
                  | APP of term * term
                  | SUCC of term
                  | SHIFT of ide * term
                  | RESET of term
  end

signature ENV
= sig
    type 'a env
    val empty : 'a env
    val extend : Syntax.ide * 'a * 'a env -> 'a env
    val lookup : Syntax.ide * 'a env -> 'a
  end

functor Definitional_Interpreter (structure Env : ENV)
= struct
    datatype value = INT of int
                   | FUNC of cont0
    withtype answer = value
          and cont2 = value -> answer
          and cont1 = value * cont2 -> answer
          and cont0 = value * cont1 * cont2 -> answer

    (*  eval : Syntax.term * value Env.env * cont1 * cont2 -> answer  *)
    fun eval (Syntax.INT n, e, k1, k2)
        = k1 (INT n, k2)
      | eval (Syntax.VAR x, e, k1, k2)
        = k1 (Env.lookup (x, e), k2)
      | eval (Syntax.LAM (x, t), e, k1, k2)
        = k1 (FUNC (fn (v, k1, k2) => eval (t, Env.extend (x, v, e), k1, k2)), k2)
      | eval (Syntax.APP (t0, t1), e, k1, k2)
        = eval (t0, e, fn (v0, k2) => eval (t1, e, fn (v1, k2) => let val (FUNC f) = v0
                                                                  in f (v1, k1, k2)
                                                                  end))
      | eval (Syntax.SUCC t, e, k1, k2)
        = eval (t, e, fn (INT n, k2) => k1 (INT (n + 1), k2), k2)
      | eval (Syntax.SHIFT (k, t), e, k1, k2)
        = eval (t,
                Env.extend (k, FUNC (fn (v, k1', k2') => k1 (v, fn v' => k1' (v', k2'))), e),
                fn (v, k2) => k2 v,
                k2)
      | eval (Syntax.RESET t, e, k1, k2)
        = eval (t, e, fn (v, k2) => k2 v, fn v => k1 (v, k2))

    (*  main : Syntax.term -> value  *)
    fun main t
        = eval (t, Env.empty, fn (v, k2) => k2 v, fn v => v)
  end
```

Figure 1. An environment-based definitional interpreter for the first level of the CPS hierarchy

tion, and used it to state local reduction rules. In contrast, we mechanically derived our abstract machine using defunctionalization, and we systematically derived a syntactic theory.

3 From interpreter to abstract machine for shift and reset

We start with defining the language of the first level of the CPS hierarchy of control operators [10].

Source terms consist of integer literals, variables, λ-abstractions, function applications, applications of the successor function, shift expressions, and reset expressions:

$$t ::= \ulcorner n \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid succ\ \ t \mid \xi\, k.t \mid \langle t \rangle$$

In a shift expression $\xi\, k.t$, the variable k is bound in t.

Programs are closed terms.

- Expressible values (integers, closures and captured continuations):

$$v \quad ::= \quad \ulcorner n \urcorner \mid [x, t, e] \mid C_1$$

- Environments:

$$e \quad ::= \quad e_{empty} \mid e[x \mapsto v] \mid e[k \mapsto C_1]$$

- Evaluation contexts and meta-contexts:

$$\begin{array}{rcl} C_1 & ::= & \bullet \mid C_1\ (t, e) \mid v\, C_1 \mid succ\ \ C_1 \\ C_2 & ::= & \bullet \mid C_2 \cdot C_1 \end{array}$$

- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t, e_{empty}, \bullet, \bullet \rangle_{eval}$
$\langle \ulcorner n \urcorner, e, C_1, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle C_1, \ulcorner n \urcorner, C_2 \rangle_{cont_1}$
$\langle x, e, C_1, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle C_1, e\,(x), C_2 \rangle_{cont_1}$
$\langle \lambda x.t, e, C_1, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle C_1, [x, t, e], C_2 \rangle_{cont_1}$
$\langle t_0 t_1, e, C_1, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle t_0, e, C_1\ (t_1, e), C_2 \rangle_{eval}$
$\langle succ\ \ t, e, C_1, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle t, e, succ\ \ C_1, C_2 \rangle_{eval}$
$\langle \xi\, k.t, e, C_1, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle t, e[k \mapsto C_1], \bullet, C_2 \rangle_{eval}$
$\langle \texttt{<}t\texttt{>}, e, C_1, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle t, e, \bullet, C_2 \cdot C_1 \rangle_{eval}$
$\langle \bullet, v, C_2 \rangle_{cont_1}$	$\Rightarrow$	$\langle C_2, v \rangle_{cont_2}$
$\langle C_1\ (t, e), v, C_2 \rangle_{cont_1}$	$\Rightarrow$	$\langle t, e, v\, C_1, C_2 \rangle_{eval}$
$\langle [x, t, e]\ C_1, v, C_2 \rangle_{cont_1}$	$\Rightarrow$	$\langle t, e[x \mapsto v], C_1, C_2 \rangle_{eval}$
$\langle C_1'\ C_1, v, C_2 \rangle_{cont_1}$	$\Rightarrow$	$\langle C_1', v, C_2 \cdot C_1 \rangle_{cont_1}$
$\langle succ\ \ C_1, \ulcorner n \urcorner, C_2 \rangle_{cont_1}$	$\Rightarrow$	$\langle C_1, \ulcorner n+1 \urcorner, C_2 \rangle_{cont_1}$
$\langle C_2 \cdot C_1, v \rangle_{cont_2}$	$\Rightarrow$	$\langle C_1, v, C_2 \rangle_{cont_1}$
$\langle \bullet, v \rangle_{cont_2}$	$\Rightarrow$	v

Figure 2. An environment-based abstract machine for the first level of the CPS hierarchy

3.1 An environment-based definitional interpreter

Figure 1 displays an interpreter for the language of the first level of the CPS hierarchy. The syntax of terms is implemented in the ML structure `Syntax` as a data type. We implement the interpreter as an ML functor parameterized by the representation of an environment.

The evaluation function is defined by structural induction over the syntax of terms, and uses both a continuation `k1` and a meta-continuation `k2`. The meta-continuation intervenes to interpret reset expressions and to apply captured continuations. Otherwise, it is passively threaded to interpret literals, variables, λ-abstractions, function applications, and applications of the successor function. (If it were not for shift and reset, and if `eval` were curried, `k2` could be eta-reduced and the interpreter would be in ordinary continuation-passing style.) Stuck (i.e., ill-typed) programs raise an ML pattern-matching error.

The reset control operator is used to delimit control. A reset expression `Syntax.RESET t` is interpreted by interpreting `t` with the identity continuation and a meta-continuation on which the current continuation has been "pushed." (Indeed defunctionalizing the meta-continuation yields the data type of a stack [12].)

The shift control operator is used to abstract (delimited) control. A shift expressions `Syntax.SHIFT (k, t)` is superficially similar to Reynolds's escape expression [40]: the current (delimited) continuation is captured in `k`, and is reset to the identity continuation.

Applying a captured continuation is achieved by "pushing" the current continuation on the meta-continuation and applying the captured continuation to the new meta-continuation.

Resuming a continuation is achieved by reactivating the "pushed" continuation with the corresponding meta-continuation.

- Source syntax, including values:

$$\begin{array}{rcl} t & ::= & v \mid x \mid t_0\, t_1 \mid succ\ \ t \mid \xi\, k.t \mid \mathtt{<}t\mathtt{>} \\ v & ::= & \ulcorner n \urcorner \mid \lambda x.t \mid C_1 \end{array}$$

- Evaluation contexts and meta-contexts:

$$\begin{array}{rcl} C_1 & ::= & \bullet \mid C_1\ t \mid v\, C_1 \mid succ\ \ C_1 \\ C_2 & ::= & \bullet \mid C_2 \cdot C_1 \end{array}$$

- Initial transition, transition rules, and final transition:

$$\begin{array}{|rcl|} \hline t & \Rightarrow & \langle t, \bullet, \bullet \rangle_{eval} \\ \hline \langle \ulcorner n \urcorner, C_1, C_2 \rangle_{eval} & \Rightarrow & \langle C_1, \ulcorner n \urcorner, C_2 \rangle_{cont_1} \\ \langle \lambda x.t, C_1, C_2 \rangle_{eval} & \Rightarrow & \langle C_1, \lambda x.t, C_2 \rangle_{cont_1} \\ \langle C_1', C_1, C_2 \rangle_{eval} & \Rightarrow & \langle C_1, C_1', C_2 \rangle_{cont_1} \\ \langle t_0\, t_1, C_1, C_2 \rangle_{eval} & \Rightarrow & \langle t_0, C_1\ t_1, C_2 \rangle_{eval} \\ \langle succ\ \ t, C_1, C_2 \rangle_{eval} & \Rightarrow & \langle t, succ\ \ C_1, C_2 \rangle_{eval} \\ \langle \xi\, k.t, C_1, C_2 \rangle_{eval} & \Rightarrow & \langle t\{C_1/k\}, \bullet, C_2 \rangle_{eval} \\ \langle \mathtt{<}t\mathtt{>}, C_1, C_2 \rangle_{eval} & \Rightarrow & \langle t, \bullet, C_2 \cdot C_1 \rangle_{eval} \\ \hline \langle \bullet, v, C_2 \rangle_{cont_1} & \Rightarrow & \langle C_2, v \rangle_{cont_2} \\ \langle C_1\ t, v, C_2 \rangle_{cont_1} & \Rightarrow & \langle t, v\, C_1, C_2 \rangle_{eval} \\ \langle (\lambda x.t)\ C_1, v, C_2 \rangle_{cont_1} & \Rightarrow & \langle t\{v/x\}, C_1, C_2 \rangle_{eval} \\ \langle C_1'\ C_1, v, C_2 \rangle_{cont_1} & \Rightarrow & \langle C_1', v, C_2 \cdot C_1 \rangle_{cont_1} \\ \langle succ\ \ C_1, \ulcorner n \urcorner, C_2 \rangle_{cont_1} & \Rightarrow & \langle C_1, \ulcorner n+1 \urcorner, C_2 \rangle_{cont_1} \\ \hline \langle C_2 \cdot C_1, v \rangle_{cont_2} & \Rightarrow & \langle C_1, v, C_2 \rangle_{cont_1} \\ \hline \langle \bullet, v \rangle_{cont_2} & \Rightarrow & v \\ \hline \end{array}$$

Figure 3. A substitution-based abstract machine for the first level of the CPS hierarchy

3.2 An environment-based abstract machine

The definitional interpreter of Figure 1 is already in continuation-passing style. Therefore, we only need to defunctionalize its expressible values and its continuations to obtain an abstract machine. This abstract machine is displayed in Figure 2.

3.3 A substitution-based abstract machine

We go from the environment-based abstract machine of Figure 2 to a substitution-based abstract machine displayed in Figure 3. The equivalence of these two machines is established with a substitution lemma [45]. The substitution-based abstract machine operates on terms where "quoted" (in the sense of Lisp) contexts can occur.

4 Analysis

In this section we analyze the transitions of the substitution-based abstract machine and we identify the ones that correspond to reduction rules in the language.

The abstract machine from Figure 3 is a small-step operational semantics of the language [38]. We can think of a configuration $\langle t, C_1, C_2 \rangle_{eval}$ of the machine as the following decomposition of the initial term into a meta-context C_2, a context C_1, and an intermediate term t:

$$C_2 \,\#\, C_1[t]$$

where # separates the context and the meta-context. Notice that in most transitions the meta-context component is not used. (Similarly, most occurrences of the meta-continuation could be eta-reduced in a curried version of the interpreter, in Figure 1.)

Next, we observe that the *eval*-transitions correspond to decomposing a term: depending on its structure, a subpart of the term is chosen to be evaluated next, and the contexts are updated accordingly. Each of the $cont_1$- and $cont_2$-transitions handles a situation when a value is reached. In this case either a reduction is performed or further decomposition takes place.

Based on the distinction between decomposition and reduction, we single out the following reduction rules from the transitions of the machine:

$$
\begin{array}{ll}
\text{(succ)} & C_2 \# C_1[succ\ \ulcorner n \urcorner] \rightarrow C_2 \# C_1[\ulcorner n+1 \urcorner] \\
(\beta_\lambda) & C_2 \# C_1[(\lambda x.t)\, v] \rightarrow C_2 \# C_1[t\{v/x\}] \\
(\xi_\lambda) & C_2 \# C_1[\xi\, k.t] \rightarrow C_2 \# \bullet[t\{C_1/k\}] \\
(\beta_{ctx}) & C_2 \# C_1[C_1'\, v] \rightarrow C_2 \cdot C_1 \# C_1'[v] \\
\text{(val)} & C_2 \cdot C_1 \# \bullet[v] \rightarrow C_2 \# C_1[v] \\
\text{(val}') & \bullet \# \bullet[v] \rightarrow v
\end{array}
$$

Note that (β_λ) is the usual call-by-value β-reduction. We renamed it to indicate that the applied term is a λ-abstraction, since we can also apply a captured context, as in (β_{ctx}). The (ξ_λ) rule can be considered as applying an abstraction $\lambda k.t$ to the current context. Moreover, the (β_{ctx}) rule can be seen as performing both a reduction and a decomposition. It is a reduction because an application of a context with a hole to a value is reduced to the value plugged into the hole; and it is a decomposition because it changes the meta-context, as if the application were enclosed in a reset. Finally, the (val) rule allows us to pass the boundary of a context, when the term inside it has been reduced to a value.

The (β_{ctx}) rule and the (ξ_λ) rule give a justification for representing a captured context C_1 as a term $\lambda x.\langle C_1[x]\rangle$, as found in other work on shift and reset [31, 36]. In particular, the need for delimiting the captured context is a consequence of the (β_{ctx}) rule.

What is more, the (β_{ctx}) rule captures the set of extra reduction rules needed by Murthy to prove the representation theorem [36].

5 A syntactic theory

A syntactic theory provides a reduction relation on expressions by defining values, evaluation contexts, and redexes [16, 18, 19, 49]. In the present case,

- the values are already specified in the (substitution-based) abstract machine;
- the evaluation contexts are already specified in the abstract machine, as the data-type part of defunctionalized continuations; and
- we can read the redexes off the transitions of the abstract machine, as done in Section 4.

Furthermore, we can read the decomposition function off the *eval*-transitions of the abstract machine:

$$
\begin{array}{rcl}
\mathsf{decompose}(t) & = & \mathsf{decompose}'(t, \bullet, \bullet) \\
\mathsf{decompose}'(t_0\, t_1, C_1, C_2) & = & \mathsf{decompose}'(t_0, C_1\, t_1, C_2) \\
\mathsf{decompose}'(succ\ \ t, C_1, C_2) & = & \mathsf{decompose}'(t, succ\ \ C_1, C_2) \\
\mathsf{decompose}'(\langle t\rangle, C_1, C_2) & = & \mathsf{decompose}'(t, \bullet, C_2 \cdot C_1) \\
\mathsf{decompose}'(v, C_1\, t, C_2) & = & \mathsf{decompose}'(t, v\, C_1, C_2)
\end{array}
$$

The plug function is immediate to write:

$$
\begin{array}{rcl}
\mathsf{plug}(t, \bullet, \bullet) & = & t \\
\mathsf{plug}(t, \bullet, C_2 \cdot C_1) & = & \mathsf{plug}(\langle t\rangle, C_1, C_2) \\
\mathsf{plug}(t, C_1\, t', C_2) & = & \mathsf{plug}(t\, t', C_1, C_2) \\
\mathsf{plug}(t, v\, C_1, C_2) & = & \mathsf{plug}(v\, t, C_1, C_2) \\
\mathsf{plug}(t, succ\ \ C_1, C_2) & = & \mathsf{plug}(succ\ \ t, C_1, C_2)
\end{array}
$$

As a side benefit of starting from a compositional evaluator, the unique-decomposition lemma holds as a corollary.

All the points of this section were already made in Felleisen and Friedman's original article on control operators and abstract machines [19], except for the last one, which is new. We are currently studying how to mechanize the construction of syntactic theories from abstract machines, based on Danvy and Nielsen's converse mechanical construction [14].

6 The second level of the CPS hierarchy

We can easily generalize the results from the previous sections to an arbitrary level of the CPS hierarchy. Let us consider the second level. Starting from the standard definitional interpreter with three layers of continuations [10], we derive the corresponding environment-based abstract machine, using the same method as in Section 3.3. The equivalent substitution-based machine is presented in Figure 4. The configurations of the machine are extended with one component corresponding to the additional continuation of the interpreter. Observe that the transitions of the machine for Level 1 are "embedded" in the machine for Level 2—the extra component is threaded but not used.

Just as for the first level, the configuration of the machine $\langle t, C_1, C_2, C_3\rangle_{eval}$ corresponds to the following decomposition of the initial term:

$$C_3 \#_2 C_2 \#_1 C_1[t]$$

where the additional context C_3 represents the rest of the term outside the innermost reset_2.

Again, we can read the set of reduction rules off the transitions of the machine. The embedding of the transitions of the previous machine in the current one is materialized in the fact that all the reduction rules for Level 1 are preserved (the first five rules below), and they do not interact with the extra layer of contexts:

$$
\begin{array}{ll}
\text{(succ)} & C_3 \#_2 C_2 \#_1 C_1[succ\ \ulcorner n \urcorner] \rightarrow C_3 \#_2 C_2 \#_1 C_1[\ulcorner n+1 \urcorner] \\
(\beta_\lambda) & C_3 \#_2 C_2 \#_1 C_1[(\lambda x.t)\, v] \rightarrow C_3 \#_2 C_2 \#_1 C_1[t\{v/x\}] \\
(\xi_\lambda) & C_3 \#_2 C_2 \#_1 C_1[\xi\, k.t] \rightarrow C_3 \#_2 C_2 \#_1 \bullet[t\{C_1/k\}] \\
(\beta_{ctx}) & C_3 \#_2 C_2 \#_1 C_1[C_1'\, v] \rightarrow C_3 \#_2 C_2 \cdot C_1 \#_1 C_1'[v] \\
\text{(val)} & C_3 \#_2 C_2 \cdot C_1 \#_1 \bullet[v] \rightarrow C_3 \#_2 C_2 \#_1 C_1[v] \\
(\xi_{2\lambda}) & C_3 \#_2 C_2 \#_1 C_1[\xi_2\, k.t] \rightarrow C_3 \#_2 \bullet \#_1 \bullet[t\{C_2 \cdot C_1/k\}] \\
(\beta_{ctx2}) & C_3 \#_2 C_2 \#_1 C_1[C_2' \cdot C_1'\, v] \rightarrow C_3 \cdot (C_2 \cdot C_1) \#_2 C_2' \#_1 C_1'[v] \\
(\text{val}_2) & C_3 \cdot (C_2 \cdot C_1) \#_2 \bullet \#_1 \bullet[v] \rightarrow C_3 \#_2 C_2 \#_1 C_1[v] \\
(\text{val}_2') & \bullet \#_2 \bullet \#_1 \bullet[v] \rightarrow v
\end{array}
$$

The three new rules $(\xi_{2\lambda})$, (β_{ctx2}), and (val_2) are straightforward generalizations of their counterparts for shift and reset. Shift_2 captures not one, but two contexts (up to the nearest enclosing reset_2), and reset_2 pushes the first two contexts onto the third one. Finally, the (val_2) rule allows us to pass the boundary of a context, when the term inside it has been reduced to a value.

7 Going up in the CPS hierarchy

Having seen that much, one can write reduction rules for an arbitrary level of the hierarchy, or reconstruct the corresponding abstract machine even without repeating the whole procedure.

At the nth level of the hierarchy, all the operators shift_1, reset_1, $\ldots$, shift_n, and reset_n are available. The nth level contains $n+1$ evaluation contexts and each context C_i can be viewed as a stack of nonempty contexts C_{i-1}. The terms are decomposed as

$$C_{n+1} \#_n C_n \#_{n-1} C_{n-1} \#_{n-2} \cdots \#_2 C_2 \#_1 C_1[t],$$

- Source syntax, including values:

$$\begin{array}{rcl} t & ::= & v \mid x \mid t_0\, t_1 \mid succ\ \ t \mid \xi\, k.t \mid <t> \mid \xi_2\, k.t \mid <t>_2 \\ v & ::= & \ulcorner n \urcorner \mid \lambda x.t \mid C_1 \mid C_2 \end{array}$$

- Evaluation contexts, meta-contexts and meta-meta-contexts:

$$\begin{array}{rcl} C_1 & ::= & \bullet \mid C_1\, t \mid v\, C_1 \mid succ\ \ C_1 \\ C_2 & ::= & \bullet \mid C_2 \cdot C_1 \\ C_3 & ::= & \bullet \mid C_3 \cdot (C_2 \cdot C_1) \end{array}$$

- Initial transition, transition rules, and final transition:

$$\begin{array}{|rcl|} \hline t & \Rightarrow & \langle t, \bullet, \bullet, \bullet \rangle_{eval} \\ \hline \langle \ulcorner n \urcorner, C_1, C_2, C_3 \rangle_{eval} & \Rightarrow & \langle C_1, \ulcorner n \urcorner, C_2, C_3 \rangle_{cont_1} \\ \langle \lambda x.t, C_1, C_2, C_3 \rangle_{eval} & \Rightarrow & \langle C_1, \lambda x.t, C_2, C_3 \rangle_{cont_1} \\ \langle C_1', C_1, C_2, C_3 \rangle_{eval} & \Rightarrow & \langle C_1, C_1', C_2, C_3 \rangle_{cont_1} \\ \langle t_0\, t_1, C_1, C_2, C_3 \rangle_{eval} & \Rightarrow & \langle t_0, C_1\, t_1, C_2, C_3 \rangle_{eval} \\ \langle succ\ \ t, C_1, C_2, C_3 \rangle_{eval} & \Rightarrow & \langle t, succ\ \ C_1, C_2, C_3 \rangle_{eval} \\ \langle \xi\, k.t, C_1, C_2, C_3 \rangle_{eval} & \Rightarrow & \langle t\{C_1/k\}, \bullet, C_2, C_3 \rangle_{eval} \\ \langle <t>, C_1, C_2, C_3 \rangle_{eval} & \Rightarrow & \langle t, \bullet, C_2 \cdot C_1, C_3 \rangle_{eval} \\ \langle \xi_2\, k.t, C_1, C_2, C_3 \rangle_{eval} & \Rightarrow & \langle t\{C_2 \cdot C_1/k\}, \bullet, \bullet, C_3 \rangle_{eval} \\ \langle <t>_2, C_1, C_2, C_3 \rangle_{eval} & \Rightarrow & \langle t, \bullet, \bullet, C_3 \cdot (C_2 \cdot C_1) \rangle_{eval} \\ \hline \langle \bullet, v, C_2, C_3 \rangle_{cont_1} & \Rightarrow & \langle C_2, v, C_3 \rangle_{cont_2} \\ \langle C_1\, t, v, C_2, C_3 \rangle_{cont_1} & \Rightarrow & \langle t, v\, C_1, C_2, C_3 \rangle_{eval} \\ \langle (\lambda x.t)\, C_1, v, C_2, C_3 \rangle_{cont_1} & \Rightarrow & \langle t\{v/x\}, C_1, C_2, C_3 \rangle_{eval} \\ \langle C_1'\, C_1, v, C_2, C_3 \rangle_{cont_1} & \Rightarrow & \langle C_1', v, C_2 \cdot C_1, C_3 \rangle_{cont_1} \\ \langle (C_2' \cdot C_1')\, C_1, v, C_2, C_3 \rangle_{cont_1} & \Rightarrow & \langle C_1', v, C_2', C_3 \cdot (C_2 \cdot C_1) \rangle_{cont_1} \\ \langle succ\ \ C_1, \ulcorner n \urcorner, C_2, C_3 \rangle_{cont_1} & \Rightarrow & \langle C_1, \ulcorner n+1 \urcorner, C_2, C_3 \rangle_{cont_1} \\ \hline \langle C_2 \cdot C_1, v, C_3 \rangle_{cont_2} & \Rightarrow & \langle C_1, v, C_2, C_3 \rangle_{cont_1} \\ \langle \bullet, v, C_3 \rangle_{cont_2} & \Rightarrow & \langle C_3, v \rangle_{cont_3} \\ \hline \langle C_3 \cdot (C_2 \cdot C_1), v \rangle_{cont_3} & \Rightarrow & \langle C_1, v, C_2, C_3 \rangle_{cont_1} \\ \hline \langle \bullet, v \rangle_{cont_3} & \Rightarrow & v \\ \hline \end{array}$$

Figure 4. A substitution-based abstract machine for the second level of the CPS hierarchy

where each $\#_i$ represents a delimited context up to Level i. All the control operators that occur already at the kth level (with $k < n$) of the hierarchy do not use the contexts $k+2, \ldots, n$.

The transitions of the machine for Level k are "embedded" in the machine for Level $k+1$—the extra components are threaded but not used. The 0th level corresponds to the CEK machine and the ordinary lambda-calculus under call by value.

8 Conclusion and issues

We have used CPS as a guideline to establish an operational foundation for delimited continuations. Starting from a call-by-value evaluator for λ-terms with shift and reset, we have mechanically constructed the corresponding abstract machine. From this abstract machine, it is straightforward to construct a syntactic theory of delimited control that, by construction, is compatible with CPS—both for one-step reduction and for evaluation.

The whole approach scales seamlessly to account for the shift$_n$ and reset$_n$ family of delimited-control operators.

Defunctionalization provided a key to connect CPS and operational intuitions about control. Indeed most of the time, control stacks are defunctionalized continuations. We do not know whether CPS is the ultimate answer, but the present work shows yet another example of its usefulness. It is like nothing can go wrong with CPS.

Acknowledgments:

We are grateful to Mads Sig Ager, Julia Lawall, Jan Midtgaard, and the anonymous referees for their comments. This work is supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`).

9 References

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

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. Technical Report BRICS RS-03-24, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. Accepted for publication in Information Processing Letters.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Technical Report BRICS RS-03-35, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2003.

[4] Kenichi Asai. Online partial evaluation for shift and reset. In Peter Thiemann, editor, *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2002)*, SIGPLAN Notices, Vol. 37, No 3, pages 19–30, Portland, Oregon, March 2002. ACM Press.

[5] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. Technical Report BRICS RS-03-25, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. Presented at the 2003 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003).

[6] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[7] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[8] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report BRICS RS-03-33, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2003.

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

[10] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [48], pages 151–160.

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

[12] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press.

[13] Olivier Danvy and Lasse R. Nielsen. On one-pass CPS transformations. Technical Report BRICS RS-02-03, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 2002. Accepted for publication in the Journal of Functional Programming.

[14] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. Technical Report BRICS RS-02-04, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 2002. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[15] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[16] 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.

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

[18] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2003.

[19] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

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

[21] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[22] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snow-

bird, Utah, July 1988. ACM Press.

[23] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.

[24] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[25] Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, pages 271–282, Pittsburgh, Pennsylvania, September 2002. ACM Press.

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

[27] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

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

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

[30] Yukiyoshi Kameyama. A type-theoretic study on partial continuations. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000, Proceedings*, volume 1872 of *Lecture Notes in Computer Science*, pages 489–504, Sendai, Japan, August 2000. Springer-Verlag.

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

[32] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[33] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

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

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

[36] Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn L. Talcott, editors, *Proceedings of the First ACM SIGPLAN Workshop on Continuations (CW 1992)*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.

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

[38] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

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

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

[41] Dorai Sitaram. *Models of Control and their Implications for Programming Language Design*. PhD thesis, Computer Science Department, Rice University, Houston, Texas, April 1994.

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

[43] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [48], pages 161–175.

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

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

[46] Eijiro Sumii. An implementation of transparent migration on standard Scheme. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, pages 61–64, Montréal, Canada, September 2000. Rice Technical Report 00-368.

[47] Philip Wadler. Monads and composable continuations. In Carolyn L. Talcott, editor, *Special issue on continuations (Part II)*, Lisp and Symbolic Computation, Vol. 7, No. 1, pages 39–55, 1994.

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

[49] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.