An Operational Foundation for Delimited Continuations in the CPS Hierarchy

Małgorzata Biernacka
Dariusz Biernacki
Olivier Danvy

BRICS Report Series **RS-04-29**

ISSN 0909-0878 **December 2004**

An Operational Foundation for Delimited Continuations in the CPS Hierarchy [*]

Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy
BRICS [†]
Department of Computer Science
University of Aarhus [‡]

December 8, 2004

Abstract

We present an abstract machine and a reduction semantics for the λ-calculus extended with control operators that give access to delimited continuations in the CPS hierarchy. The abstract machine is derived from an evaluator in continuation-passing style (CPS); the reduction semantics (i.e., a small-step operational semantics with an explicit representation of evaluation contexts) is constructed from the abstract machine; and the control operators are the shift and reset family. At level n of the CPS hierarchy, programs can use the control operators shift$_i$ and reset$_i$ for $1 \leq i \leq n$, the evaluator has $n+1$ layers of continuations, the abstract machine has $n+1$ layers of control stacks, and the reduction semantics has $n+1$ layers of evaluation contexts.

We also present new applications of delimited continuations in the CPS hierarchy: finding list prefixes and normalization by evaluation for a hierarchical language of units and products.

[*] A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations [14].

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

[‡] IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
E-mail: {`mbiernac,dabi,danvy`}`@brics.dk`
Home pages: `http://www.brics.dk/~`{`dabi,mbiernac,danvy`}

Contents

List of Figures

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 proposing a variety of control operators [5, 31, 34, 35, 42, 45, 46, 56, 60, 64, 72] which have found applications in models of control, concurrency, and type-directed partial evaluation [8, 45, 65]. Of the former, there is the work revolving around the family of control operators shift and reset [22–24, 27, 36, 37, 48, 49, 57, 72] which have found applications in non-deterministic programming, code generation, partial evaluation, normalization by evaluation, computational monads, and mobile computing [6, 7, 9, 15, 18, 19, 28, 29, 38, 39, 41, 44, 50, 53, 62, 68, 69, 71].

The original motivation for shift and reset was a continuation-based programming pattern involving several layers of continuations. The original specification of these operators relied both on a repeated CPS transformation and on an evaluator with several layers of continuations (as is obtained by repeatedly transforming a direct-style evaluator into continuation-passing style). Only subsequently have shift and reset been specified operationally, by developing operational analogues of a continuation semantics and of the CPS transformation [27].

The goal of our work here is to establish a new operational foundation for delimited continuations, using CPS as a guideline. To this end, we start with the original evaluator for shift$_1$ and reset$_1$. This evaluator 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 reduction semantics [30], as pioneered by Felleisen and Friedman [33]. The development scales to shift$_n$ and reset$_n$. It is reusable for any control operators that are compatible with CPS, i.e., that can be characterized with a (possibly iterated) CPS translation or with a continuation-based evaluator. It also pinpoints where operational intuitions go beyond CPS.

This article is structured as follows. In Section 2, we 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 demonstrate this enabling technology on a simple example, arithmetic expressions. In Section 3, we illustrate the use of shift and reset with the classic example of finding list prefixes, using an ML-like programming language. In Section 4, we then present our main result: starting from the original evaluator for shift and reset, we defunctionalize it into an abstract machine; we analyze this abstract machine and construct the corresponding reduction semantics. In Section 5, we extend our main result to the CPS hierarchy. In Section 6, we illustrate the CPS hierarchy with a class of normalization functions for a hierarchical language of units and products.

2 From evaluator to reduction semantics for arithmetic expressions

We demonstrate the derivation from an evaluator to a reduction semantics. The derivation consists of the following steps:

1. we start from an evaluator for a given language; if it is in direct style, we CPS-transform it;

2. we defunctionalize the CPS evaluator, obtaining a value-based abstract machine;

3. we modify the abstract machine to make it term-based instead of value-based; in particular, if the evaluator uses an environment, then so does the corresponding value-based abstract machine, and in that case, making the machine term-based leads us to use substitutions rather than an environment;

4. we analyze the transitions of the term-based abstract machine to identify the evaluation strategy it implements and the set of reductions it performs; the result is a reduction semantics.

The first two steps are based on previous work on a functional correspondence between evaluators and abstract machines [1–3, 15, 21], which itself is based on Reynolds's seminal work on definitional interpreters [61]. The last two steps follow the lines of Felleisen and Friedman's original work on a reduction semantics for the call-by-value λ-calculus extended with control operators [33]. The last step has been studied further by Hardin, Maranget, and Pagano [43] in the context of explicit substitutions and by Danvy and Nielsen [26].

In the rest of this section, our running example is the language of arithmetic expressions, formed using natural numbers (the values) and additions (the computations):

$$\mathsf{exp} \ni e ::= \ulcorner m \urcorner \mid e_1 \mathbin{\mathsf{+}} e_2$$

2.1 The starting point: an evaluator in direct style

We define an evaluation function for arithmetic expressions by structural induction on their syntax. The resulting direct-style evaluator is displayed in Figure 1.

2.2 CPS transformation

We CPS-transform the evaluator by naming intermediate results, sequentializing their computation, and introducing an extra functional parameter, the continuation [24, 58, 66]. The resulting continuation-passing evaluator is displayed in Figure 2.

2.3 Defunctionalization

The generalization of closure conversion [52] to defunctionalization is due to Reynolds [61]. The goal is to represent a functional value with a first-order data structure. The means is to partition the function space into a first-order sum where each summand corresponds to a lambda-abstraction in the program. In a defunctionalized program, function introduction is thus represented as an injection, and function elimination as a call to a first-order apply function implementing a case dispatch. In an ML-like functional language, sums are represented as data types, injections as data-type constructors, and apply functions are defined by case over the corresponding data types [25].

Here, we defunctionalize the continuation of the continuation-passing evaluator in Figure 2. We thus need to define a first-order algebraic data type and its apply function. To this end, we enumerate the lambda-abstractions that give rise to the inhabitants of this function space; there are three: the initial continuation in evaluate and the two continuations in eval. The initial continuation is closed, and therefore the corresponding algebraic constructor is nullary. The two other continuations have two free variables, and therefore the corresponding

- Values: $\mathsf{val} \ni v ::= m$
- Evaluation function: $\mathsf{eval} : \mathsf{exp} \to \mathsf{val}$

$$\begin{array}{rcl} \mathsf{eval}\ (\ulcorner m \urcorner) & = & m \\ \mathsf{eval}\ (e_1 \mathbin{\texttt{+}} e_2) & = & \mathsf{eval}\ (e_1) + \mathsf{eval}\ (e_2) \end{array}$$

- Main function: $\mathsf{evaluate} : \mathsf{exp} \to \mathsf{val}$

$$\mathsf{evaluate}\ (e) \; = \; \mathsf{eval}\ (e)$$

Figure 1: A direct-style evaluator for arithmetic expressions

- Values: $\mathsf{val} \ni v ::= m$
- Continuations: $\mathsf{cont} = \mathsf{val} \to \mathsf{val}$
- Evaluation function: $\mathsf{eval} : \mathsf{exp} \times \mathsf{cont} \to \mathsf{val}$

$$\begin{array}{rcl} \mathsf{eval}\ (\ulcorner m \urcorner,\, k) & = & k\ m \\ \mathsf{eval}\ (e_1 \mathbin{\texttt{+}} e_2,\, k) & = & \mathsf{eval}\ (e_1,\, \lambda m_1.\, \mathsf{eval}\ (e_2,\, \lambda m_2.\, k\ (m_1 + m_2))) \end{array}$$

- Main function: $\mathsf{evaluate} : \mathsf{exp} \to \mathsf{val}$

$$\mathsf{evaluate}\ (e) \; = \; \mathsf{eval}\ (e,\, \lambda v.\, v)$$

Figure 2: A continuation-passing evaluator for arithmetic expressions

- Values: $\mathsf{val} \ni v ::= m$
- Defunctionalized continuations: $\mathsf{cont} \ni k ::= \mathit{stop} \mid \mathit{arg_2}\ (e,\ k) \mid \mathit{arg_1}\ (v,\ k)$
- Functions $\mathsf{eval} : \mathsf{exp} \times \mathsf{cont} \to \mathsf{val}$ and $\mathsf{apply\_cont} : \mathsf{cont} \times \mathsf{val} \to \mathsf{val}$:

$$\begin{array}{rcl} \mathsf{eval}\ (\ulcorner m \urcorner,\, k) & = & \mathsf{apply\_cont}\ (k,\, m) \\ \mathsf{eval}\ (e_1 \mathbin{\texttt{+}} e_2,\, k) & = & \mathsf{eval}\ (e_1,\, \mathit{arg_2}\ (e_2,\ k)) \\ \\ \mathsf{apply\_cont}\ (\mathit{stop},\, v) & = & v \\ \mathsf{apply\_cont}\ (\mathit{arg_2}\ (e_2,\ k),\, v_1) & = & \mathsf{eval}\ (e_2,\, \mathit{arg_1}\ (v_1,\ k)) \\ \mathsf{apply\_cont}\ (\mathit{arg_1}\ (m_1,\ k),\, m_2) & = & \mathsf{apply\_cont}\ (k,\, m_1 + m_2) \end{array}$$

- Main function: $\mathsf{evaluate} : \mathsf{exp} \to \mathsf{val}$

$$\mathsf{evaluate}\ (e) \; = \; \mathsf{eval}\ (e,\, \mathit{stop})$$

Figure 3: A defunctionalized continuation-passing evaluator for arithmetic expressions

constructors are binary. As for the apply function, it interprets the algebraic constructors. The resulting defunctionalized evaluator is displayed in Figure 3.

2.4 Abstract machines as defunctionalized continuation-passing programs

Elsewhere [1, 21], we have observed that a defunctionalized continuation-passing program implements an abstract machine: each configuration is the name of a function together with its arguments, and each function clause represents a transition. (As a corollary, we have also observed that the defunctionalized continuation forms what is known as an 'evaluation context' [20, 25, 33].)

Indeed Plotkin's Indifference Theorem [58] states that continuation-passing programs are independent of their evaluation order. In Reynolds's words [61], all the subterms in applications are 'trivial'; and in Moggi's words [55], these subterms are values and not computations. Furthermore, continuation-passing programs are tail recursive [66]. Therefore, since in a continuation-passing program all calls are tail calls and all subcomputations are elementary, a defunctionalized continuation-passing program implements a transition system [59], i.e., an abstract machine.

We thus reformat Figure 3 into Figure 4. The correctness of the abstract machine with respect to the initial evaluator follows from the correctness of CPS transformation and of defunctionalization.

2.5 From value-based abstract machine to term-based abstract machine

We observe that the domain of expressible values in Figure 4 can be embedded in the syntactic domain of expressions. We therefore adapt the abstract machine to work on terms rather than on values. The result is displayed in Figure 5; it is a syntactic theory [30].

2.6 From term-based abstract machine to reduction semantics

The method of deriving a reduction semantics from an abstract machine was introduced by Felleisen and Friedman [33] to give a reduction semantics for control operators. Let us demonstrate it.

We analyze the transitions of the abstract machine in Figure 5. The second component of *eval*-transitions—the stack representing "the rest of the computation"—has already been identified as the evaluation context of the currently processed expression. We thus read a configuration $\langle e, C\rangle_{eval}$ as a decomposition of some expression into a sub-expression e and an evaluation context C.

Next, we identify the reduction and decomposition rules in the transitions of the machine. Since a configuration can be read as a decomposition, we compare the left-hand side and the right-hand side of each transition. If they represent the same expression, then the given transition defines a decomposition (i.e., it searches for the next redex according to some evaluation strategy); otherwise we have found a redex. Moreover, reading the decomposition rules from right to left defines a 'plug' function that reconstructs an expression from its decomposition.

In this simple example there is only one reduction rule. This rule performs the addition of natural numbers:

$$\text{(add)} \quad C\,[\ulcorner m_1 \urcorner \mathtt{+} \ulcorner m_2 \urcorner] \quad \rightarrow \quad C\,[\ulcorner m_1 + m_2 \urcorner]$$

- Values: $v ::= m$
- Evaluation contexts: $C ::= \bullet \mid C + e \mid v + C$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow$	$\langle e, \bullet \rangle_{eval}$
$\langle \ulcorner m \urcorner, C \rangle_{eval}$	$\Rightarrow$	$\langle C, m \rangle_{cont}$
$\langle e_1 \texttt{+} e_2, C \rangle_{eval}$	$\Rightarrow$	$\langle e_1, C + e_2 \rangle_{eval}$
$\langle C + e_2, v_1 \rangle_{cont}$	$\Rightarrow$	$\langle e_2, v_1 + C \rangle_{eval}$
$\langle m_1 + C, m_2 \rangle_{cont}$	$\Rightarrow$	$\langle C, m_1 + m_2 \rangle_{cont}$
$\langle \bullet, v \rangle_{cont}$	$\Rightarrow$	v

Figure 4: A value-based abstract machine for evaluating arithmetic expressions

- Expressions and values: $e ::= v \mid e_1 \texttt{+} e_2$
  $v ::= \ulcorner m \urcorner$
- Evaluation contexts: $C ::= \bullet \mid C + e \mid v + C$
- Initial transition, transition rules, and final transition:

e	$\Rightarrow$	$\langle e, \bullet \rangle_{eval}$
$\langle \ulcorner m \urcorner, C \rangle_{eval}$	$\Rightarrow$	$\langle C, \ulcorner m \urcorner \rangle_{cont}$
$\langle e_1 \texttt{+} e_2, C \rangle_{eval}$	$\Rightarrow$	$\langle e_1, C + e_2 \rangle_{eval}$
$\langle C + e_2, v_1 \rangle_{cont}$	$\Rightarrow$	$\langle e_2, v_1 + C \rangle_{eval}$
$\langle \ulcorner m_1 \urcorner + C, \ulcorner m_2 \urcorner \rangle_{cont}$	$\Rightarrow$	$\langle C, \ulcorner m_1 + m_2 \urcorner \rangle_{cont}$
$\langle \bullet, v \rangle_{cont}$	$\Rightarrow$	v

Figure 5: A term-based abstract machine for processing arithmetic expressions

The remaining transitions decompose an expression according to the left-to-right strategy.

Danvy and Nielsen [26] have studied the converse derivation and systematized the construction of an abstract machine from a reduction semantics. The method relies on constructing a *refocus* function that optimizes the evaluation function of a reduction semantics, canonically defined as the transitive closure of one-step reduction, i.e., of decomposition (of a non-value term into a reduction context and a redex), contraction (of the redex), and plugging (of the contractum in the context); the refocus function is a deforested version of the composition of plugging and decomposition. The construction is reversible and can be used to go from an abstract machine to a refocused evaluation function and then a reduction function, systematically.

2.7 Summary and conclusion

We have demonstrated how to derive an abstract machine out of an evaluator, and how to construct the corresponding reduction semantics out of this abstract machine. In Section 4, we apply this derivation and this construction to the first level of the CPS hierarchy, and in Section 5, we apply them to an arbitrary level of the CPS hierarchy. But first, let us illustrate how to program with delimited continuations.

3 Programming with delimited continuations

We present two examples of programming with delimited continuations. Given a list xs and a predicate p, we want

1. to find the first prefix of xs whose last element satisfies p, and
2. to find all such prefixes of xs.

For example, given the predicate $\lambda m.m > 2$ and the list $[0,3,1,4,2,5]$, the first prefix is $[0,3]$ and the list of all the prefixes is $[[0,3],[0,3,1,4],[0,3,1,4,2,5]]$.

In Section 3.1, we start with a simple solution that uses a first-order accumulator. This simple solution is in defunctionalized form. In Section 3.2, we present its higher-order counterpart, which uses a functional accumulator. This functional accumulator acts as a delimited continuation. In Section 3.3, we present its direct-style counterpart (which uses shift and reset) and in Section 3.4, we present its continuation-passing counterpart (which uses two layers of continuations). In Section 3.5, we introduce the CPS hierarchy informally. We then mention a typing issue in Section 3.6 and review related work in Section 3.7.

3.1 Finding prefixes by accumulating lists

A simple solution is to accumulate the prefix of the given list in reverse order while traversing this list and testing each of its elements:

- if no element satisfies the predicate, there is no prefix and the result is the empty list;
- otherwise, the prefix is the reverse of the accumulator.

```
find_first_prefix_a (p, xs)  ≝  letrec visit (nil, a)
                                            = nil
                                          | visit (x :: xs, a)
                                            = let a' = x :: a
                                              in if p x
                                                   then reverse (a', nil)
                                                   else visit (xs, a')
                                         and reverse (nil, xs)
                                            = xs
                                          | reverse (x :: a, xs)
                                            = reverse (a, x :: xs)
                                 in visit (xs, nil)
```

```
find_all_prefixes_a (p, xs)  ≝  letrec visit (nil, a)
                                            = nil
                                          | visit (x :: xs, a)
                                            = let a' = x :: a
                                              in if p x
                                                   then (reverse (a', nil)) :: (visit (xs, a'))
                                                   else visit (xs, a')
                                         and reverse (nil, xs)
                                            = xs
                                          | reverse (x :: a, xs)
                                            = reverse (a, x :: xs)
                                 in visit (xs, nil)
```

To find the first prefix, one stops as soon as a satisfactory list element is found. To list all the prefixes, one continues the traversal, adding the current prefix to the list of the remaining prefixes.

We observe that the two solutions are in defunctionalized form [25, 61]: the accumulator has the data type of a defunctionalized function and *reverse* is its apply function. We present its higher-order counterpart next [47].

3.2 Finding prefixes by accumulating list constructors

Instead of accumulating the prefix in reverse order while traversing the given list, we accumulate a function constructing the prefix:

- if no element satisfies the predicate, the result is the empty list;
- otherwise, we apply the functional accumulator to construct the prefix.

```
$find\_first\_prefix\_c_1\,(p,\ xs) \overset{\text{def}}{=}$ $letrec\ visit\,(nil,\ k)$
                                              $= nil$
                                            $|\ visit\,(x :: xs,\ k)$
                                              $= let\ k' = \lambda vs.k\,(x :: vs)$
                                                 $in\ if\ p\ x$
                                                    $then\ k'\ nil$
                                                    $else\ visit\,(xs,\ k')$
                                       $in\ visit\,(xs,\ \lambda vs.vs)$
```

```
$find\_all\_prefixes\_c_1\,(p,\ xs) \overset{\text{def}}{=}$ $letrec\ visit\,(nil,\ k)$
                                              $= nil$
                                            $|\ visit\,(x :: xs,\ k)$
                                              $= let\ k' = \lambda vs.k\,(x :: vs)$
                                                 $in\ if\ p\ x$
                                                    $then\ (k'\ nil) :: (visit\,(xs,\ k'))$
                                                    $else\ visit\,(xs,\ k')$
                                       $in\ visit\,(xs,\ \lambda vs.vs)$
```

To find the first prefix, one applies the functional accumulator as soon as a satisfactory list element is found. To list all such prefixes, one continues the traversal, adding the current prefix to the list of the remaining prefixes.

Defunctionalizing these two definitions yields the two definitions of Section 3.1.

The functional accumulator is a delimited continuation:

- In *find_first_prefix_*c_1, *visit* is written in CPS since all calls are tail calls and all subcomputations are elementary. The continuation is initialized in the initial call to *visit*, discarded in the base case, extended in the induction case, and used if a satisfactory prefix is found.
- In *find_all_prefixes_*c_1, *visit* is almost written in CPS except that the continuation is composed if a satisfactory prefix is found: it is used twice—once where it is applied to the empty list to construct a prefix, and once in the visit of the rest of the list to construct a list of prefixes; this prefix is then prepended to the list of prefixes.

These continuation-based programming patterns (initializing a continuation, not using it, or using it more than once as if it were a composable function) have motivated the control operators shift and reset [23, 24]. Using them, in the next section, we write *visit* in direct style.

3.3 Finding prefixes in direct style

The two following local functions are the direct-style counterpart of the two local functions in Section 3.2:

```
$find\_first\_prefix\_c_0\,(p,\ xs) \overset{\text{def}}{=}$ $letrec\ \ visit\,nil$
                                            $= \mathcal{S}k.nil$
                                          $|\ visit\,(x :: xs)$
                                            $= x :: (if\ \ p\,x\ \ then\ nil\ \ else\ visit\,xs)$
                                      $in\ \langle visit\,xs \rangle$

$find\_all\_prefixes\_c_0\,(p,\ xs) \overset{\text{def}}{=}$ $letrec\ \ visit\,nil$
                                            $= \mathcal{S}k.nil$
                                          $|\ visit\,(x :: xs)$
                                            $= x :: if\ \ p\,x$
                                                   $then\ \mathcal{S}k'.(k'\,nil) :: \langle k'\,(visit\,xs) \rangle$
                                                   $else\ \ visit\,xs$
                                      $in\ \langle visit\,xs \rangle$
```

In both cases, *visit* is in direct style, i.e., it is not passed any continuation. The initial calls to *visit* are enclosed in the control delimiter reset (noted $\langle\cdot\rangle$ for conciseness). In the base cases, the current (delimited) continuation is captured with the control operator shift (noted $\mathcal{S}$), which has the effect of emptying the (delimited) context; this captured continuation is bound to an identifier k, which is not used; *nil* is then returned in the emptied context. In the induction case of $find\_all\_prefixes\_c_0$, if the predicate is satisfied, *visit* captures the current continuation and applies it twice—once to the empty list to construct a prefix, and once to the result of visiting the rest of the list to construct a list of prefixes; this prefix is then prepended to the list of prefixes.

CPS-transforming these two local functions yields the two definitions of Section 3.2 [24].

3.4 Finding prefixes in continuation-passing style

The two following local functions are the continuation-passing counterpart of the two local functions in Section 3.2:

```
$find\_first\_prefix\_c_2\,(p,\ xs) \overset{\text{def}}{=}$ $letrec\ \ visit\,(nil,\ k_1,\ k_2)$
                                          $= k_2\,nil$
                                        $|\ visit\,(x :: xs,\ k_1,\ k_2)$
                                          $= let\ k_1' = \lambda(vs,\ k_2').k_1\,(x :: vs,\ k_2')$
                                            $in\ if\ \ p\,x$
                                                $then\ k_1'\,(nil,\ k_2)$
                                                $else\ \ visit\,(xs,\ k_1',\ k_2)$
                                  $in\ visit\,(xs,\ \lambda(vs,\ k_2).k_2\,vs,\ \lambda vs.vs)$

$find\_all\_prefixes\_c_2\,(p,\ xs) \overset{\text{def}}{=}$ $letrec\ \ visit\,(nil,\ k_1,\ k_2)$
                                          $= k_2\,nil$
                                        $|\ visit\,(x :: xs,\ k_1,\ k_2)$
                                          $= let\ k_1' = \lambda(vs,\ k_2').k_1\,(x :: vs,\ k_2')$
                                            $in\ if\ \ p\,x$
                                                $then\ k_1'\,(nil,\ \lambda vs.visit\,(xs,\ k_1',\ \lambda vss.k_2\,(vs :: vss)))$
                                                $else\ \ visit\,(xs,\ k_1',\ k_2)$
                                  $in\ visit\,(xs,\ \lambda(vs,\ k_2).k_2\,vs,\ \lambda vss.vss)$
```

CPS-transforming the two local functions of Section 3.2 adds another layer of continuations and restores the syntactic characterization of all calls being tail calls and all sub-computations being elementary.

3.5 The CPS hierarchy

If k_2 were to be used non-tail recursively in a variant of the examples of Section 3.4, we could CPS-transform the definitions one more time, adding one more layer of continuations and restoring the syntactic characterization of all calls being tail calls and all sub-computations being elementary. We could also map this definition back to direct style, eliminating k_2 but accessing it with shift. If the result were mapped back to direct style one more time, k_2 would then be accessed with a new control operator, shift$_2$, and k_1 would be accessed with shift (a.k.a. shift$_1$).

All in all, successive CPS-transformations induce a CPS hierarchy [23,27], and abstracting control up to each successive layer is achieved with successive pairs of control operators shift and reset—reset to initialize the continuation up to a level, and shift to capture a delimited continuation up to this level. Each pair of control operators is indexed by the corresponding level in the hierarchy. Applying a captured continuation packages all the current layers on the next layer and restores the captured layers. When a captured continuation completes, the packaged layers are put back into place and the computation proceeds. (This informal description is made precise in Section 4.)

3.6 A note about typing

The type of $\mathit{find\_all\_prefixes\_c}_1$, in Section 3.2, is

$$(\alpha \rightarrow \mathit{bool}) \times \alpha\ \mathit{list} \rightarrow \alpha\ \mathit{list}\ \mathit{list}$$

and the type of its local function *visit* is

$$\alpha\ \mathit{list} \times (\alpha\ \mathit{list} \rightarrow \alpha\ \mathit{list}) \rightarrow \alpha\ \mathit{list}\ \mathit{list}.$$

In this example, the co-domain of the continuation is not the same as the co-domain of *visit*. Correspondingly, in Section 3.3, shift is used at type

$$((\alpha\ \mathit{list} \rightarrow \alpha\ \mathit{list}) \rightarrow \alpha\ \mathit{list}\ \mathit{list}) \rightarrow \alpha\ \mathit{list}$$

and thus $\mathit{find\_first\_prefix\_c}_0$ provides a simple example where Filinski's typing of shift [36] does not fit, since it must be used at type

$$((\beta \rightarrow \mathit{ans}) \rightarrow \mathit{ans}) \rightarrow \beta$$

for a given type *ans*. Due to a similar restriction on the type of shift, the example does not fit either in Murthy's pseudo-classical type system for the CPS hierarchy [57] and in Wadler's most general monadic type system [72, Section 3.4]. It however fits in Danvy and Filinski's original type system [22] which Ariola, Herbelin, and Sabry have recently embedded in classical subtractive logic [5].

3.7 Related work

The example considered in this section builds on the simpler function that unconditionally lists the successive prefixes of a given list. This simpler function is a traditional example of delimited continuations [17, 63]:

- In the Lisp Pointers [17], Danvy presents three versions of this function: a typed continuation-passing version (corresponding to Section 3.2), one with delimited control (corresponding to Section 3.3), and one in assembly language.
- In his PhD thesis [63, Section 6.3], Sitaram presents two versions of this function: one with an accumulator (corresponding to Section 3.1) and one with delimited control (corresponding to Section 3.3).

In Section 3.2, we have shown that the continuation-passing version mediates the version with an accumulator and the version with delimited control since defunctionalizing the continuation-passing version yields one and mapping it back to direct style yields the other.

3.8 Summary and conclusion

We have illustrated delimited continuations with the classic example of finding list prefixes, using CPS as a guideline. Direct-style programs using shift and reset can be CPS-transformed into continuation-passing programs where not all calls are tail calls and all sub-computations are elementary. One more CPS transformation establishes this syntactic property with a second layer of continuations. Further CPS transformations provide the extra layers of continuation that are characteristic of the CPS hierarchy.

In the next section, we specify the λ-calculus extended with shift and reset.

4 From evaluator to reduction semantics for delimited continuations

We derive a reduction semantics for the call-by-value λ-calculus extended with shift and reset, using the method demonstrated in Section 2. First, we transform an evaluator into an environment-based abstract machine. Then we eliminate the environment from this abstract machine, making it substitution-based. Finally, we read all the components of a reduction semantics off the substitution-based abstract machine.

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

$$t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid \mathit{succ}\ t \mid \langle t \rangle \mid \mathcal{S}k.t$$

Programs are closed terms.

This source language is a subset of the language used in the examples of Section 3. Adding the remaining constructs is a straightforward but tedious exercise, and does not contribute to our point here.

- Terms: $\mathsf{term} \ni t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid succ\; t \mid \langle t \rangle \mid \mathcal{S}k.t$
- Values: $\mathsf{val} \ni v ::= m \mid f$
- Answers, meta-continuations, continuations and functions:

$$\begin{array}{rcl} \mathsf{ans} & = & \mathsf{val} \\ k_2 \in \mathsf{cont}_2 & = & \mathsf{val} \to \mathsf{ans} \\ k_1 \in \mathsf{cont}_1 & = & \mathsf{val} \times \mathsf{cont}_2 \to \mathsf{ans} \\ f \in \mathsf{fun} & = & \mathsf{val} \times \mathsf{cont}_1 \times \mathsf{cont}_2 \to \mathsf{ans} \end{array}$$

- Initial continuation and meta-continuation: $\begin{array}{rcl} \theta_1 & = & \lambda(v,\, k_2).\, k_2\, v \\ \theta_2 & = & \lambda v.\, v \end{array}$
- Environments: $\mathsf{env} \ni e ::= e_{empty} \mid e[x \mapsto v]$
- Evaluation function: $\mathsf{eval} : \mathsf{term} \times \mathsf{env} \times \mathsf{cont}_1 \times \mathsf{cont}_2 \to \mathsf{ans}$

$$\begin{array}{rcl} \mathsf{eval}\,(\ulcorner m \urcorner,\, e,\, k_1,\, k_2) & = & k_1\,(m,\, k_2) \\ \mathsf{eval}\,(x,\, e,\, k_1,\, k_2) & = & k_1\,(e(x),\, k_2) \\ \mathsf{eval}\,(\lambda x.t,\, e,\, k_1,\, k_2) & = & k_1\,(\lambda(v,\, k_1,\, k_2).\, \mathsf{eval}\,(t,\, e[x \mapsto v],\, k_1,\, k_2),\, k_2) \\ \mathsf{eval}\,(t_0\, t_1,\, e,\, k_1,\, k_2) & = & \mathsf{eval}\,(t_0,\, e,\, \lambda(f,\, k_2).\, \mathsf{eval}\,(t_1,\, e,\, \lambda(v,\, k_2).\, f\,(v,\, k_1,\, k_2),\, k_2),\, k_2) \\ \mathsf{eval}\,(succ\; t,\, e,\, k_1,\, k_2) & = & \mathsf{eval}\,(t,\, e,\, \lambda(m,\, k_2).\, k_1\,(m+1,\, k_2),\, k_2) \\ \mathsf{eval}\,(\langle t \rangle,\, e,\, k_1,\, k_2) & = & \mathsf{eval}\,(t,\, e,\, \theta_1,\, \lambda v.\, k_1\,(v,\, k_2)) \\ \mathsf{eval}\,(\mathcal{S}k.t,\, e,\, k_1,\, k_2) & = & \mathsf{eval}\,(t,\, e[k \mapsto c],\, \theta_1,\, k_2) \\ & & \text{where } c = \lambda(v,\, k_1',\, k_2').\, k_1\,(v,\, \lambda v'.\, k_1'\,(v',\, k_2')) \end{array}$$

- Main function: $\mathsf{evaluate} : \mathsf{term} \to \mathsf{val}$

$$\mathsf{evaluate}\,(t) \;=\; \mathsf{eval}\,(t,\, e_{empty},\, \theta_1,\, \theta_2)$$

Figure 6: An environment-based evaluator for the first level of the CPS hierarchy

4.1 An environment-based evaluator

Figure 6 displays an evaluator for the language of the first level of the CPS hierarchy. This evaluation function represents the original call-by-value semantics of the λ-calculus with shift and reset [23], augmented with integer literals and applications of the successor function. It is defined by structural induction over the syntax of terms, and it makes use of an environment e, a continuation k_1, and a meta-continuation k_2.

The evaluation of a correct program (i.e., one with no ill-formed applications) yields either an integer, a function representing a λ-abstraction, or a captured continuation. Both evaluate and eval are partial functions to account for ill-formed applications and non-termination. The environment stores previously computed values of the free variables of the term under evaluation.

The meta-continuation intervenes to interpret reset expressions and to apply captured continuations. Otherwise, it is passively threaded through the evaluation of 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, k_2 could be eta-reduced and the evaluator would be in ordinary continuation-passing style.)

The reset control operator is used to delimit control. A reset expression $\langle t \rangle$ is interpreted by evaluating t with the initial continuation and a meta-continuation on which the current continuation has been "pushed." (Indeed, and as will be shown in Section 4.2, defunctionalizing the meta-continuation yields the data type of a stack [25].)

The shift control operator is used to abstract (delimited) control. A shift expression $\mathcal{S}k.t$ is interpreted by capturing the current continuation, binding it to k, and evaluating t in an environment extended with k and with a continuation reset to the initial 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.

4.2 An environment-based abstract machine

The evaluator displayed in Figure 6 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 7.

The abstract machine consists of three sets of transitions: *eval* for interpreting terms, $cont_1$ for interpreting the defunctionalized continuations (i.e., the evaluation contexts), and $cont_2$ for interpreting the defunctionalized meta-continuations (i.e., the meta-contexts). The set of possible values includes integers, closures and captured contexts. In the original evaluator, the latter two were represented as higher-order functions, but defunctionalizing expressible values of the evaluator has led them to be distinguished.

This abstract machine is an extension of the CEK machine [33] with the meta-context C_2 and its two transitions, and the two transitions for shift and reset. C_2 intervenes to process reset expressions and to apply captured continuations. Otherwise, it is passively threaded through the processing of literals, variables, λ-abstractions, function applications, and applications of the successor function. (If it were not for shift and reset, C_2 and its transitions could be omitted and the abstract machine would reduce to the CEK machine.)

Given an environment e, a context C_1, and a meta-context C_2, a reset expression $\langle t \rangle$ is processed by evaluating t with the same environment e, the empty context $\bullet$, and a meta-context where C_1 has been pushed on C_2.

Given an environment e, a context C_1, and a meta-context C_2, a shift expression $\mathcal{S}k.t$ is processed by evaluating t with an extension of e where k denotes C_1, the empty context $\bullet$, and a meta-context C_2. Applying a captured context C_1' is achieved by pushing the current context C_1 on the current meta-context C_2 and continuing with C_1'. Resuming a context C_1 is achieved by popping it off the meta-context $C_2 \cdot C_1$ and continuing with C_1.

The correctness of the abstract machine with respect to the evaluator is a consequence of the correctness of defunctionalization. In order to express it formally, we define a partial function $\mathsf{eval}^{\mathsf{e}}$ mapping a term t to a value v whenever the environment-based machine, started

- Terms: $t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid \mathit{succ}\; t \mid \langle\!\langle t \rangle\!\rangle \mid \mathcal{S}k.t$
- Values (integers, closures, and captured continuations): $v ::= m \mid [x,\, t,\, e] \mid C_1$
- Environments: $e ::= e_{\mathit{empty}} \mid e[x \mapsto v]$
- Evaluation contexts and meta-contexts: $C_1 ::= \bullet \mid C_1\; (t, e) \mid \mathit{succ}\; C_1 \mid v\; C_1$
  $C_2 ::= \bullet \mid C_2 \cdot C_1$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t,\, e_{\mathit{empty}},\, \bullet,\, \bullet \rangle_{\mathit{eval}}$
$\langle \ulcorner m \urcorner,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, m,\, C_2 \rangle_{\mathit{cont_1}}$
$\langle x,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, e\,(x),\, C_2 \rangle_{\mathit{cont_1}}$
$\langle \lambda x.t,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, [x,\, t,\, e],\, C_2 \rangle_{\mathit{cont_1}}$
$\langle t_0\, t_1,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t_0,\, e,\, C_1\; (t_1, e),\, C_2 \rangle_{\mathit{eval}}$
$\langle \mathit{succ}\; t,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t,\, e,\, \mathit{succ}\; C_1,\, C_2 \rangle_{\mathit{eval}}$
$\langle \langle\!\langle t \rangle\!\rangle,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t,\, e,\, \bullet,\, C_2 \cdot C_1 \rangle_{\mathit{eval}}$
$\langle \mathcal{S}k.t,\, e,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t,\, e[k \mapsto C_1],\, \bullet,\, C_2 \rangle_{\mathit{eval}}$
$\langle \bullet,\, v,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_2,\, v \rangle_{\mathit{cont_2}}$
$\langle C_1\; (t, e),\, v,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle t,\, e,\, v\; C_1,\, C_2 \rangle_{\mathit{eval}}$
$\langle \mathit{succ}\; C_1,\, m,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_1,\, m + 1,\, C_2 \rangle_{\mathit{cont_1}}$
$\langle [x,\, t,\, e]\; C_1,\, v,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle t,\, e[x \mapsto v],\, C_1,\, C_2 \rangle_{\mathit{eval}}$
$\langle C_1'\; C_1,\, v,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_1',\, v,\, C_2 \cdot C_1 \rangle_{\mathit{cont_1}}$
$\langle C_2 \cdot C_1,\, v \rangle_{\mathit{cont_2}}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2 \rangle_{\mathit{cont_1}}$
$\langle \bullet,\, v \rangle_{\mathit{cont_2}}$	$\Rightarrow$	v

Figure 7: An environment-based abstract machine for the first level of the CPS hierarchy

with t, stops with v. The following theorem states this correctness by relating observable results:

Theorem 1 *For any program t,* evaluate $(t) = m$ *if and only if* $\mathsf{eval}^{\mathsf{e}}\;(t) = m$.

The environment-based abstract machine can serve both as a foundation for implementing functional languages with control operators for delimited continuations and as a stepping stone in theoretical studies of shift and reset. In the rest of this section, we use it to construct a reduction semantics of shift and reset.

- Terms and values: $t ::= v \mid x \mid t_0\, t_1 \mid \mathit{succ}\ t \mid \langle t \rangle \mid \mathcal{S}k.t$
  $v ::= \ulcorner m \urcorner \mid \lambda x.t \mid C_1$
- Evaluation contexts and meta-contexts: $C_1 ::= \bullet \mid C_1\ t \mid \mathit{succ}\ C_1 \mid v\ C_1$
  $C_2 ::= \bullet \mid C_2 \cdot C_1$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t,\, \bullet,\, \bullet \rangle_{\mathit{eval}}$
$\langle \ulcorner m \urcorner,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, \ulcorner m \urcorner,\, C_2 \rangle_{\mathit{cont_1}}$
$\langle \lambda x.t,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, \lambda x.t,\, C_2 \rangle_{\mathit{cont_1}}$
$\langle C_1',\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, C_1',\, C_2 \rangle_{\mathit{cont_1}}$
$\langle t_0\, t_1,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t_0,\, C_1\ t_1,\, C_2 \rangle_{\mathit{eval}}$
$\langle \mathit{succ}\ t,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t,\, \mathit{succ}\ C_1,\, C_2 \rangle_{\mathit{eval}}$
$\langle \langle t \rangle,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t,\, \bullet,\, C_2 \cdot C_1 \rangle_{\mathit{eval}}$
$\langle \mathcal{S}k.t,\, C_1,\, C_2 \rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t\{C_1/k\},\, \bullet,\, C_2 \rangle_{\mathit{eval}}$
$\langle \bullet,\, v,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_2,\, v \rangle_{\mathit{cont_2}}$
$\langle C_1\ t,\, v,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle t,\, v\ C_1,\, C_2 \rangle_{\mathit{eval}}$
$\langle \mathit{succ}\ C_1,\, \ulcorner m \urcorner,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_1,\, \ulcorner m+1 \urcorner,\, C_2 \rangle_{\mathit{cont_1}}$
$\langle (\lambda x.t)\ C_1,\, v,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle t\{v/x\},\, C_1,\, C_2 \rangle_{\mathit{eval}}$
$\langle C_1'\ C_1,\, v,\, C_2 \rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_1',\, v,\, C_2 \cdot C_1 \rangle_{\mathit{cont_1}}$
$\langle C_2 \cdot C_1,\, v \rangle_{\mathit{cont_2}}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2 \rangle_{\mathit{cont_1}}$
$\langle \bullet,\, v \rangle_{\mathit{cont_2}}$	$\Rightarrow$	v

Figure 8: A substitution-based abstract machine for the first level of the CPS hierarchy

4.3 A substitution-based abstract machine

The environment-based abstract machine of Figure 7, on which we want to base our development, makes a distinction between terms and values. Since a reduction semantics is specified by purely syntactic operations (it gives meaning to terms by specifying their rewriting strategy and an appropriate notion of reduction, and is indeed also referred to as 'syntactic theory'), we need to embed the domain of values back into the syntax. To this end we transform the environment-based abstract machine into the substitution-based abstract machine displayed in Figure 8. The transformation is standard, except that we also need to embed evaluation contexts in the syntax; hence the substitution-based machine operates on terms where "quoted" (in the sense of Lisp) contexts can occur. (If it were not for shift and reset,

C_2 and its transitions could be omitted and the abstract machine would reduce to the CK machine [33].)

The equivalence of the two machines hinges on a substitution lemma [67]. We write $t\{v/x\}$ to denote the result of the usual capture-avoiding substitution of the value v for x in t.

Formally, the relationship between the two machines is expressed with the following simulation theorem, where evaluation with the substitution-based abstract machine is captured by the partial function $\mathsf{eval}^{\mathsf{s}}$, defined analogously to $\mathsf{eval}^{\mathsf{e}}$.

Theorem 2 *For any term t, $\mathsf{eval}^{\mathsf{s}}\ (t) = v$ if and only if $\mathsf{eval}^{\mathsf{e}}\ (t) = v'$ and $\mathcal{T}\ (v') = v$. The function $\mathcal{T}$ relates a semantic value with its syntactic representation and is defined as follows:*[1]

$$\begin{aligned}
\mathcal{T}\,(m) &= \ulcorner m \urcorner \\
\mathcal{T}\,([x,\, t,\, e]) &= \lambda x.t\{\mathcal{T}\,(e(x_1))/x_1\}\ldots\{\mathcal{T}\,(e(x_n))/x_n\},\ \textit{where}\ \{x_1,\ldots,x_n\} = FV\ (\lambda x.\,t) \\
\mathcal{T}\,(\bullet) &= \bullet \\
\mathcal{T}\,(C_1\ (t,e)) &= \mathcal{T}\,(C_1)\ t\{\mathcal{T}\,(e(x_1))/x_1\}\ldots\{\mathcal{T}\,(e(x_n))/x_n\},\ \textit{where}\ \{x_1,\ldots,x_n\} = FV\ (t) \\
\mathcal{T}\,(v\ C_1) &= \mathcal{T}\,(v)\ \mathcal{T}\,(C_1) \\
\mathcal{T}\,(succ\ C_1) &= succ\ \mathcal{T}\,(C_1)
\end{aligned}$$

We now proceed to analyze the transitions of the machine displayed in Figure 8. We can think of a configuration $\langle t,\, C_1,\, C_2\rangle_{eval}$ 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. Each transition performs either a reduction, or a decomposition in search of the next redex. Let us recall that a decomposition is performed when both sides of a transition are partitions of the same term; in that case, depending on the structure of the decomposition $C_2 \# C_1[t]$, a subpart of the term is chosen to be evaluated next, and the contexts are updated accordingly. We also observe that *eval*-transitions follow the structure of t, $cont_1$-transitions follow the structure of C_1 when the term has been reduced to a value, and $cont_2$-transitions follow the structure of C_2 when a value in the empty context has been reached.

Next we specify all the components of the reduction semantics based on the analysis of the abstract machine.

4.4 A reduction semantics

A reduction semantics provides a reduction relation on expressions by defining values, evaluation contexts, and redexes [30, 32, 33, 74]. In the present case,

- the values are already specified in the (substitution-based) abstract machine:

$$v ::= \ulcorner m \urcorner \ \mid\ \lambda x.t \ \mid\ C_1$$

[1] $\mathcal{T}$ is a generalization of Plotkin's function Real [58].

- the evaluation contexts and meta-contexts are already specified in the abstract machine, as the data-type part of defunctionalized continuations;

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

- we can read the redexes off the transitions of the abstract machine:

$$r ::= \mathit{succ}\, \ulcorner m \urcorner \mid (\lambda x.t)\, v \mid \mathcal{S}k.t \mid C_1'\, v \mid \langle v \rangle$$

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

$$\begin{array}{llcl} (\delta) & C_2 \mathbin{\#} C_1[\mathit{succ}\, \ulcorner m \urcorner] & \to & C_2 \mathbin{\#} C_1[\ulcorner m+1 \urcorner] \\ (\beta_\lambda) & C_2 \mathbin{\#} C_1[(\lambda x.t)\, v] & \to & C_2 \mathbin{\#} C_1[t\{v/x\}] \\ (\mathcal{S}_\lambda) & C_2 \mathbin{\#} C_1[\mathcal{S}k.t] & \to & C_2 \mathbin{\#} \bullet[t\{C_1/k\}] \\ (\beta_{ctx}) & C_2 \mathbin{\#} C_1[C_1'\, v] & \to & C_2 \cdot C_1 \mathbin{\#} C_1'[v] \\ (\mathrm{val}) & C_2 \mathbin{\#} C_1[\langle v \rangle] & \to & C_2 \mathbin{\#} C_1[v] \end{array}$$

(β_λ) is the usual call-by-value β-reduction; we have renamed it to indicate that the applied term is a λ-abstraction, since we can also apply a captured context, as in (β_{ctx}). $(\mathcal{S}_\lambda)$ is plausibly symmetric to (β_λ) — it can be seen as an application of the abstraction $\lambda k.t$ to the current context.[2] Moreover, (β_{ctx}) 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, (val) makes it possible to pass the boundary of a context when the term inside this context has been reduced to a value.

The β_{ctx}-rule and the $\mathcal{S}_\lambda$-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 studies of shift and reset [48, 49, 57]. In particular, the need for delimiting the captured context is a consequence of the β_{ctx}-rule.

Finally, we can read the decomposition function off the transitions of the abstract machine:

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

In the remaining cases either a value or a redex has been found:

$$\begin{array}{rcl} \mathit{decompose}'\,(v,\, \bullet,\, \bullet) & = & \bullet \mathbin{\#} \bullet[v] \\ \mathit{decompose}'\,(v,\, \bullet,\, C_2 \cdot C_1) & = & C_2 \mathbin{\#} C_1[\langle v \rangle] \\ \mathit{decompose}'\,(\mathcal{S}k.t,\, C_1,\, C_2) & = & C_2 \mathbin{\#} C_1[\mathcal{S}k.t] \\ \mathit{decompose}'\,(v,\, (\lambda x.t)\; C_1,\, C_2) & = & C_2 \mathbin{\#} C_1[(\lambda x.t)\, v] \\ \mathit{decompose}'\,(v,\, C_1'\; C_1,\, C_2) & = & C_2 \mathbin{\#} C_1[C_1'\, v] \\ \mathit{decompose}'\,(\ulcorner m \urcorner,\, \mathit{succ}\; C_1,\, C_2) & = & C_2 \mathbin{\#} C_1[\mathit{succ}\, \ulcorner m \urcorner] \end{array}$$

[2]Ariola, Herbelin, and Sabry have exploited this symmetry in their recent work [5], where they use the dual of implication (subtraction) to give a logical interpretation to languages with delimited-control operators.

An inverse of the *decompose* function, traditionally called *plug*, reconstructs a term from its decomposition:

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

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

Lemma 1 (Unique decomposition) *A term t is either a value or there exist a unique context C_1 and a unique meta-context C_2 such that $decompose(t) = C_2 \,\#\, C_1[r]$, where r is a redex.*

It is evident that evaluating a term either using the derived reduction rules or using the substitution-based abstract machine yields the same result.

Theorem 3 *For any term t, $\mathsf{eval}^{\mathsf{s}}\ (t) = v$ if and only if $t \rightarrow^* v$, where $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$.*

4.5 Beyond CPS

Alternatively to using the meta-context to compose delimited continuations, as in Figure 7, we could compose them by concatenating their representation. Such a concatenation function is defined as follows:

$$\begin{array}{rcl}
\bullet \star C_1' & = & C_1' \\
(C_1\ (t, e)) \star C_1' & = & (C_1 \star C_1')\ (t, e) \\
(v\ C_1) \star C_1' & = & v\ (C_1 \star C_1') \\
(succ\ C_1) \star C_1' & = & succ\ (C_1 \star C_1')
\end{array}$$

(For the contexts of Figure 8, the second clause would read $(C_1\ t) \star C_1' = (C_1 \star C_1')\ t$.)

Then, in Figures 7 and 8, we could replace the transition

$$\boxed{\langle C_1'\ C_1,\ v,\ C_2 \rangle_{cont_1} \quad \Rightarrow \quad \langle C_1',\ v,\ C_2 \cdot C_1 \rangle_{cont_1}}$$

by the following one:

$$\boxed{\langle C_1'\ C_1,\ v,\ C_2 \rangle_{cont_1} \quad \Rightarrow \quad \langle C_1' \star C_1,\ v,\ C_2 \rangle_{cont_1}}$$

This replacement changes the control effect of shift to that of Felleisen et al.'s $\mathcal{F}$ operator [35].

This representation of control (as a list of 'stack frames') and this implementation of composing delimited continuations (by concatenating these lists) is at the heart of virtually all non-CPS-based accounts of delimited control. However, the modified environment-based abstract machine does not correspond to a defunctionalized continuation-passing evaluator [25]; in that sense, control operators using $\star$ go beyond CPS.

4.6 Summary and conclusion

We have presented the original evaluator for the λ-calculus with shift and reset; this evaluator uses two layers of continuations. From this evaluator we have derived two abstract machines, an environment-based one and a substitution-based one; each of these machines uses two layers of evaluation contexts. Based on the substitution-based machine we have constructed a reduction semantics for the λ-calculus with shift and reset; this reduction semantics, by construction, is sound with respect to CPS.

5 From evaluator to reduction semantics for the CPS hierarchy

We construct a reduction semantics for the call-by-value λ-calculus extended with shift$_n$ and reset$_n$. As in Section 4, we go from an evaluator to an environment-based abstract machine, and from a substitution-based abstract machine to a reduction semantics. Because of the regularity of CPS, the results can be generalized from level 1 to higher levels without repeating the actual construction, based only on the original specification of the hierarchy [23].

5.1 An environment-based evaluator

At the nth level of the hierarchy, the language is extended with operators shift$_i$ and reset$_i$ for all i such that $1 \leq i \leq n$. The evaluator for this language is shown in Figures 9 and 10. If $n = 1$, it coincides with the evaluator displayed in Figure 6.

- Terms ($1 \leq i \leq n$): $\mathsf{term} \ni t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid \mathit{succ}\; t \mid \langle t \rangle_i \mid \mathcal{S}_i k.t$
- Values: $\mathsf{val} \ni v ::= m \mid f$
- Answers, continuations and functions ($1 \leq i \leq n$):

$$\begin{array}{rcllcl} & & \mathsf{ans} & = & \mathsf{val} \\ k_{n+1} & \in & \mathsf{cont}_{n+1} & = & \mathsf{val} \rightarrow \mathsf{ans} \\ k_i & \in & \mathsf{cont}_i & = & \mathsf{val} \times \mathsf{cont}_{i+1} \times \ldots \times \mathsf{cont}_{n+1} \rightarrow \mathsf{ans} \\ f & \in & \mathsf{fun} & = & \mathsf{val} \times \mathsf{cont}_1 \times \ldots \times \mathsf{cont}_{n+1} \rightarrow \mathsf{ans} \end{array}$$

- Initial continuations ($1 \leq i \leq n$):

$$\begin{array}{rcl} \theta_i & = & \lambda(v, k_{i+1}, k_{i+2}, \ldots, k_{n+1}).\, k_{i+1}\, (v, k_{i+2}, \ldots, k_{n+1}) \\ \theta_{n+1} & = & \lambda v.\, v \end{array}$$

- Environments: $\mathsf{env} \ni e ::= e_{\mathit{empty}} \mid e[x \mapsto v]$
- Evaluation function: see Figure 10

Figure 9: An environment-based evaluator for the CPS hierarchy at level n

- Evaluation function ($1 \leq i \leq n$): $\mathsf{eval}_n \ : \ \mathsf{term} \ \times \ \mathsf{env} \ \times \ \mathsf{cont}_1 \ \times \ \ldots \ \times \ \mathsf{cont}_{n+1} \ \rightarrow \ \mathsf{ans}$

$$
\begin{array}{rcl}
\mathsf{eval}_n \ (\ulcorner m \urcorner,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & k_1 \ (m,\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n \ (x,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & k_1 \ (e(x),\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n \ (\lambda x.t,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & k_1 \ (\lambda(v,\, k_1,\, k_2,\, \ldots,\, k_{n+1}).\, \mathsf{eval}_n \ (t,\, e[x \mapsto v],\, k_1,\, k_2,\, \ldots,\, k_{n+1}),\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n \ (t_0\, t_1,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n \ (t_0,\, e, \\
& & \quad \lambda(f,\, k_2,\, \ldots,\, k_{n+1}).\, \mathsf{eval}_n \ (t_1,\, e, \\
& & \qquad\qquad \lambda(v,\, k_2,\, \ldots,\, k_{n+1}).\, f \ (v,\, k_1,\, k_2,\, \ldots,\, k_{n+1}), \\
& & \qquad\qquad k_2,\, \ldots,\, k_{n+1}), \\
& & \quad k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n \ (succ\ t,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n \ (t,\, e,\, \lambda(m,\, k_2,\, \ldots,\, k_{n+1}).\, k_1 \ (m+1,\, k_2,\, \ldots,\, k_{n+1}),\, k_2,\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n \ (\langle t \rangle_i,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n \ (t,\, e,\, \theta_1,\, \ldots,\, \theta_i,\, \lambda(v,\, k'_{i+2},\, \ldots,\, k'_{n+1}).\, k_1 \ (v,\, k_2,\, \ldots,\, k_i,\, k_{i+1},\, k'_{i+2},\, \ldots,\, k'_{n+1}),\, k_{i+2},\, \ldots,\, k_{n+1}) \\
\mathsf{eval}_n \ (\mathcal{S}_i k.t,\, e,\, k_1,\, k_2,\, \ldots,\, k_{n+1}) & = & \mathsf{eval}_n \ (t,\, e[k \mapsto c_i],\, \theta_1,\, \ldots,\, \theta_i,\, k_{i+1},\, \ldots,\, k_{n+1})
\end{array}
$$

where $c_i = \lambda(v,\, k'_1,\, \ldots,\, k'_{n+1}).\, k_1 \ (v,\, k_2,\, \ldots,\, k_i,\, \lambda(v',\, k''_{i+2},\, \ldots,\, k''_{n+1}).\, k'_1 \ (v',\, k'_2,\, \ldots,\, k'_{i+1},\, k''_{i+2},\, \ldots,\, k''_{n+1}),\, k'_{i+2},\, \ldots,\, k'_{n+1})$

- Main function: $\mathsf{evaluate}_n \ : \ \mathsf{term} \ \rightarrow \ \mathsf{val}$

$$\mathsf{evaluate}_n \ (t) \ = \ \mathsf{eval}_n \ (t,\, e_{empty},\, \theta_1,\, \ldots,\, \theta_n,\, \theta_{n+1})$$

Figure 10: An environment-based evaluator for the CPS hierarchy at level n, ctd.

The evaluator uses $n+1$ layers of continuations. In the five first clauses (literal, variable, λ-abstraction, function application, and application of the successor function), the continuations $k_2, \ldots, k_{n+1}$ are passive: if the evaluator were curried, they could be eta-reduced. In the clauses defining shift_i and reset_i, the continuations $k_{i+2}, \ldots, k_{n+1}$ are also passive. Each pair of control operators is indexed by the corresponding level in the hierarchy: reset_i is used to "push" each successive continuation up to level i onto level $i+1$ and to reinitialize them with $\theta_1, \ldots, \theta_i$, which are the successive CPS counterparts of the identity function; shift_i is used to abstract control up to level i into a delimited continuation and to reinitialize the successive continuations up to level i with $\theta_1, \ldots, \theta_i$.

Applying a delimited continuation that was abstracted up to level i "pushes" each successive continuation up to level i onto level $i+1$ and restores the successive continuations that were captured in a delimited continuation. When such a delimited continuation completes, and when an expression delimited by reset_i completes, the successive continuations that were pushed onto level $i+1$ are "popped" back into place and the computation proceeds.

5.2 An environment-based abstract machine

Defunctionalizing the evaluator of Figures 9 and 10 yields the environment-based abstract machine displayed in Figures 11 and 12. If $n = 1$, it coincides with the abstract machine displayed in Figure 7.

The abstract machine consists of $n+2$ sets of transitions: *eval* for interpreting terms and $\mathit{cont}_1, \ldots, \mathit{cont}_{n+1}$ for interpreting the successive defunctionalized continuations. The set of possible values includes integers, closures and captured contexts.

This abstract machine is an extension of the abstract machine displayed in Figure 7 with $n+1$ contexts instead of 2 and the corresponding transitions for shift_i and reset_i. Each meta_{i+1}-context intervenes to process reset_i expressions and to apply captured continuations. Otherwise, the successive contexts are passively threaded to process literals, variables, λ-abstractions, function applications, and applications of the successor function.

Given an environment e and a series of successive contexts, a reset_i expression $\langle t \rangle_i$ is processed by evaluating t with the same environment e, i empty contexts, and a meta_{i+1}-context over which all the intermediate contexts have been pushed on.

Given an environment e and a series of successive contexts, a shift expression $\mathcal{S}_i k.t$ is processed by evaluating t with an extension of e where k denotes a composition of the i sur-

- Terms ($1 \leq i \leq n$): $t ::= \ulcorner m \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1 \mid \mathit{succ}\; t \mid \langle t \rangle_i \mid \mathcal{S}_i k.t$
- Values ($1 \leq i \leq n$): $v ::= m \mid [x, t, e] \mid C_i$
- Evaluation contexts ($2 \leq i \leq n+1$): $C_1 ::= \bullet \mid C_1\ (t, e) \mid \mathit{succ}\ C_1 \mid v\ C_1$
  $C_i ::= \bullet \mid C_i \cdot C_{i-1}$
- Environments: $e ::= e_{\mathit{empty}} \mid e[x \mapsto v]$
- Initial transition, transition rules, and final transition: see Figure 12

Figure 11: An environment-based abstract machine for the CPS hierarchy at level n

- Initial transition, transition rules, and final transition ($1 \leq i \leq n,\ 2 \leq j \leq n$):

t	$\Rightarrow$	$\langle t,\, e_{\mathit{empty}},\, \bullet,\, \bullet,\, \ldots,\, \bullet\rangle_{\mathit{eval}}$
$\langle \ulcorner m \urcorner,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, m,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$
$\langle x,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, e\,(x),\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$
$\langle \lambda x.t,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle C_1,\, [x,\, t,\, e],\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$
$\langle t_0\, t_1,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t_0,\, e,\, C_1\ (t_1, e),\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$
$\langle \mathit{succ}\ t,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t,\, e,\, \mathit{succ}\ C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$
$\langle \langle t\rangle_i,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t,\, e,\, \bullet,\, \ldots,\, \bullet,\, C_{i+1} \cdot (C_i \cdot \ldots (C_2 \cdot C_1) \ldots),\, C_{i+2},\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$
$\langle \mathcal{S}_i k.t,\, e,\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$	$\Rightarrow$	$\langle t,\, e[k \mapsto C_i \cdot (C_{i-1} \cdot \ldots (C_2 \cdot C_1) \ldots)],\, \bullet,\, \ldots,\, \bullet,\, C_{i+1},\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$
$\langle \bullet,\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_2,\, v,\, C_3,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_2}}$
$\langle C_1\ (t, e),\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle t,\, e,\, v\ C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$
$\langle \mathit{succ}\ C_1,\, m,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C_1,\, m+1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$
$\langle [x,\, t,\, e]\ C_1,\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle t,\, e[x \mapsto v],\, C_1,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{eval}}$
$\langle (C'_i \cdot (C'_{i-1} \cdot \ldots (C'_2 \cdot C'_1) \ldots))\ C_1,\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$	$\Rightarrow$	$\langle C'_1,\, v,\, C'_2,\, \ldots,\, C'_i,\, C_{i+1} \cdot (C_i \cdot \ldots (C_2 \cdot C_1) \ldots),\, C_{i+2},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$
$\langle \bullet,\, v,\, C_{j+1},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_j}}$	$\Rightarrow$	$\langle C_{j+1},\, v,\, C_{j+2},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_{j+1}}}$
$\langle C_j \cdot (C_{j-1} \cdot \ldots (C_2 \cdot C_1) \ldots),\, v,\, C_{j+1},\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_j}}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$
$\langle C_{n+1} \cdot (C_n \cdot \ldots (C_2 \cdot C_1) \ldots),\, v\rangle_{\mathit{cont_{n+1}}}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2,\, \ldots,\, C_{n+1}\rangle_{\mathit{cont_1}}$
$\langle \bullet,\, v\rangle_{\mathit{cont_{n+1}}}$	$\Rightarrow$	v

Figure 12: An environment-based abstract machine for the CPS hierarchy at level n, ctd.

rounding contexts, i empty contexts, and the remaining outer contexts. Applying a captured context is achieved by pushing all the current contexts on the next outer context, restoring the composition of the captured contexts, and continuing with them. Resuming a composition of captured contexts is achieved by popping them off the next outer context and continuing with them.

In order to relate the resulting abstract machine to the evaluator, we define a partial function $\mathsf{eval}^{\mathsf{e}}_n$ mapping a term t to a value v whenever the machine for level n, started with t, stops with v. The correctness of the machine with respect to the evaluator is ensured by the following theorem:

Theorem 4 *For any term t,* $\mathsf{evaluate}_n\ (t) = m$ *if and only if* $\mathsf{eval}^{\mathsf{e}}_n\ (t) = m$.

5.3 A substitution-based abstract machine

In the same fashion as in Section 4.3, we construct the substitution-based abstract machine corresponding to the environment-based abstract machine of Section 5.2. The result is displayed in Figures 13 and 14. If $n = 1$, it coincides with the abstract machine displayed in Figure 8.

The nth level contains $n+1$ evaluation contexts and each context C_i can be viewed as a stack of non-empty contexts C_{i-1}. Terms are decomposed as

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

where each $\#_i$ represents a context delimiter of level i. All the control operators that occur at the jth level (with $j < n$) of the hierarchy do not use the contexts $j+2, \ldots, n+1$.

The transitions of the machine for level j are "embedded" in the machine for level $j+1$; the extra components are threaded but not used.

We define a partial function $\mathsf{eval}^{\mathsf{s}}_n$ capturing the evaluation by the substitution-based abstract machine for an arbitrary level n, analogously to the definition of $\mathsf{eval}^{\mathsf{e}}_n$. Now we can relate evaluation with the environment-based and the substitution-based abstract machines for level n.

Theorem 5 *For any term t in the language of level n,* $\mathsf{eval}^{\mathsf{s}}_n\ (t) = v$ *if and only if* $\mathsf{eval}^{\mathsf{e}}_n\ (t) = v'$ *and* $\mathcal{T}_n\ (v') = v$.

The definition of $\mathcal{T}_n$ extends that of $\mathcal{T}$ from Theorem 2 in such a way that it is homomorphic for all the contexts C_i, with $2 \leq i \leq n$.

- Terms and values ($1 \leq i \leq n$): $t ::= v \mid x \mid t_0\, t_1 \mid \mathit{succ}\ t \mid \langle t \rangle_i \mid \mathcal{S}_i k.t$
  $v ::= \ulcorner m \urcorner \mid \lambda x.t \mid C_i$
- Evaluation contexts ($2 \leq i \leq n+1$): $C_1 ::= \bullet \mid C_1\ t \mid \mathit{succ}\ C_1 \mid v\ C_1$
  $C_i ::= \bullet \mid C_i \cdot C_{i-1}$
- Initial transition, transition rules, and final transition: see Figure 14

Figure 13: A substitution-based abstract machine for the CPS hierarchy at level n

- Initial transition, transition rules, and final transition ($1 \leq i \leq n,\ 2 \leq j \leq n$):

t	$\Rightarrow$	$\langle t, \bullet, \bullet, \ldots, \bullet\rangle_{eval}$
$\langle \ulcorner m \urcorner, C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$	$\Rightarrow$	$\langle C_1, \ulcorner m \urcorner, C_2, \ldots, C_{n+1}\rangle_{cont_1}$
$\langle \lambda x.t, C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$	$\Rightarrow$	$\langle C_1, \lambda x.t, C_2, \ldots, C_{n+1}\rangle_{cont_1}$
$\langle C'_i, C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$	$\Rightarrow$	$\langle C_1, C'_i, C_2, \ldots, C_{n+1}\rangle_{cont_1}$
$\langle t_0\, t_1, C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$	$\Rightarrow$	$\langle t_0, C_1\,(t_1, e), C_2, \ldots, C_{n+1}\rangle_{eval}$
$\langle succ\ t, C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$	$\Rightarrow$	$\langle t, succ\ C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$
$\langle \mathcal{S}_i k.t, C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$	$\Rightarrow$	$\langle t\{C_i \cdot (C_{i-1} \cdot \ldots (C_2 \cdot C_1) \ldots)/k\}, \bullet, \ldots, \bullet, C_{i+1}, \ldots, C_{n+1}\rangle_{eval}$
$\langle \langle\!\langle t \rangle\!\rangle_i, C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$	$\Rightarrow$	$\langle t, \bullet, \ldots, \bullet, C_{i+1} \cdot (C_i \cdot \ldots (C_2 \cdot C_1) \ldots), C_{i+2}, \ldots, C_{n+1}\rangle_{eval}$
$\langle \bullet, v, C_2, \ldots, C_{n+1}\rangle_{cont_1}$	$\Rightarrow$	$\langle C_2, v, C_3, \ldots, C_{n+1}\rangle_{cont_2}$
$\langle C_1\ t, v, C_2, \ldots, C_{n+1}\rangle_{cont_1}$	$\Rightarrow$	$\langle t, v\ C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$
$\langle succ\ C_1, \ulcorner m \urcorner, C_2, \ldots, C_{n+1}\rangle_{cont_1}$	$\Rightarrow$	$\langle C_1, \ulcorner m+1 \urcorner, C_2, \ldots, C_{n+1}\rangle_{cont_1}$
$\langle (\lambda x.t)\ C_1, v, C_2, \ldots, C_{n+1}\rangle_{cont_1}$	$\Rightarrow$	$\langle t\{v/x\}, C_1, C_2, \ldots, C_{n+1}\rangle_{eval}$
$\langle (C'_i \cdot (C'_{i-1} \cdot \ldots (C'_2 \cdot C'_1) \ldots))\ C_1, v, C_2, \ldots, C_{n+1}\rangle_{cont_1}$	$\Rightarrow$	$\langle C'_1, v, C'_2, \ldots, C'_i, C_{i+1} \cdot (C_i \cdot \ldots (C_2 \cdot C_1) \ldots), C_{i+2}, \ldots, C_{n+1}\rangle_{cont_1}$
$\langle \bullet, v, C_{j+1}, \ldots, C_{n+1}\rangle_{cont_j}$	$\Rightarrow$	$\langle C_{j+1}, v, C_{j+2}, \ldots, C_{n+1}\rangle_{cont_{j+1}}$
$\langle C_j \cdot (C_{j-1} \cdot \ldots (C_2 \cdot C_1) \ldots), v, C_{j+1}, \ldots, C_{n+1}\rangle_{cont_j}$	$\Rightarrow$	$\langle C_1, v, C_2, \ldots, C_{n+1}\rangle_{cont_1}$
$\langle C_{n+1} \cdot (C_n \cdot \ldots (C_2 \cdot C_1) \ldots), v\rangle_{cont_{n+1}}$	$\Rightarrow$	$\langle C_1, v, C_2, \ldots, C_{n+1}\rangle_{cont_1}$
$\langle \bullet, v\rangle_{cont_{n+1}}$	$\Rightarrow$	v

Figure 14: A substitution-based abstract machine for the the CPS hierarchy at level n, ctd.

5.4 A reduction semantics

Along the same lines as in Section 4.4, we construct the reduction semantics for the CPS hierarchy based on the abstract machine of Figures 13 and 14. For an arbitrary level n we obtain the following set of reduction rules, for all $1 \leq i \leq n$:

$$\begin{array}{ll}
(\delta) & C_{n+1} \,\#_n\, \cdots\, \#_1\, C_1[\mathit{succ}\, \ulcorner m \urcorner] \rightarrow_n C_{n+1} \,\#_n\, \cdots\, \#_1\, C_1[\ulcorner m+1 \urcorner] \\
(\beta_\lambda) & C_{n+1} \,\#_n\, \cdots\, \#_1\, C_1[(\lambda x.t)\, v] \rightarrow_n C_{n+1} \,\#_n\, \cdots\, \#_1\, C_1[t\{v/x\}] \\
(\mathcal{S}^i_\lambda) & C_{n+1} \,\#_n\, \cdots\, \#_1\, C_1[\mathcal{S}_i k.t] \rightarrow_n \\
 & C_{n+1} \,\#_n\, \cdots\, \#_{i+1}\, C_{i+1} \,\#_i\, \bullet \ldots\, \#_1\, \bullet [t\{C_i \cdot (\ldots (C_2 \cdot C_1) \ldots)/k\}] \\
(\beta^i_{ctx}) & C_{n+1} \,\#_n\, \cdots\, \#_1\, C_1[C'_i \cdot (\ldots (C'_2 \cdot C'_1) \ldots)\, v] \rightarrow_n \\
 & C_{n+1} \,\#_n\, \cdots\, \#_{i+1}\, C_{i+1} \cdot (C_i \cdot (\ldots (C_2 \cdot C_1) \ldots)) \,\#_i\, C'_i \,\#_{i-1}\, \cdots\, \#_1\, C'_1[v] \\
(\mathrm{val}^i) & C_{n+1} \,\#_n\, \cdots\, \#_1\, C_1[\langle v \rangle_i] \rightarrow_n C_{n+1} \,\#_n\, \cdots\, \#_1\, C_1[v]
\end{array}$$

Each level contains all the reductions from lower levels, and these reductions are compatible with additional layers of evaluation contexts. In particular, at level 0 there are only δ- and β_λ-reductions.

The values and evaluation contexts are already specified in the abstract machine. Moreover, the redexes are defined according to the following grammar:

$$r_n ::= \mathit{succ}\, \ulcorner m \urcorner \mid (\lambda x.t)\, v \mid \mathcal{S}_i k.t \mid (C'_i \cdot (\ldots (C'_2 \cdot C'_1) \ldots))\, v \mid \langle v \rangle_i \quad (1 \leq i \leq n)$$

Lemma 2 (Unique decomposition for level n) *Any term t is either a value or there exists a unique sequence of contexts $C_1, \ldots, C_{n+1}$ such that t can be decomposed as $\#_1\, C_1[r_n]$, where r_n is a redex.*

Evaluating a term using either the derived reduction rules or the substitution-based abstract machine from Section 5.3 yields the same result:

Theorem 6 *For any term t, $\mathsf{eval}^{\mathsf{s}}_n\,(t) = v$ if and only if $t \rightarrow^*_n v$, where $\rightarrow^*_n$ is the reflexive, transitive closure of $\rightarrow_n$.*

5.5 Beyond CPS

As in Section 4.5, one could define a concatenation function over contexts and use it to implement composable continuations in the CPS hierarchy. Again the modified environment-based abstract machine would not correspond to a defunctionalized continuation-passing evaluator. Such control operators go beyond CPS.

5.6 Summary and conclusion

We have generalized the results presented in Section 4 from level 1 to the whole CPS hierarchy of control operators shift$_n$ and reset$_n$. Starting from the original evaluator for the λ-calculus with shift$_n$ and reset$_n$ that uses $n+1$ layers of continuations, we have derived two abstract machines, an environment-based one and a substitution-based one; each of these machines use $n+1$ layers of evaluation contexts. Based on the substitution-based machine we have obtained a reduction semantics for the λ-calculus extended with shift$_n$ and reset$_n$ which, by construction, is sound with respect to CPS.

6 Programming in the CPS hierarchy

To finish, we present new examples of programming in the CPS hierarchy. The examples are normalization functions. In Sections 6.1 and 6.2, we first describe normalization by evaluation and we present the simple example of the free monoid. In Section 6.3, we present a function mapping a proposition into its disjunctive normal form; this normalization function uses delimited continuations. In Section 6.4, we generalize the normalization functions of Sections 6.2 and 6.3 to a hierarchical language of units and products, and we express the corresponding normalization function in the CPS hierarchy.

6.1 Normalization by evaluation

Normalization by evaluation is a 'reduction-free' approach to normalizing terms. Instead of reducing a term to its normal form, one evaluates this term into a non-standard model and reifies its denotation into its normal form [29]:

$$\begin{array}{rcl} \mathit{eval} & : & \mathit{term} \;\rightarrow\; \mathit{value} \\ \mathit{reify} & : & \mathit{value} \;\rightarrow\; \mathit{term}^{\mathrm{nf}} \\ \mathit{normalize} & : & \mathit{term} \;\rightarrow\; \mathit{term}^{\mathrm{nf}} \\ \mathit{normalize} & = & \mathit{reify} \circ \mathit{eval} \end{array}$$

Normalization by evaluation has been developed in intuitionistic type theory [16,54], proof theory [11,12], category theory [4], λ-definability [40], and partial evaluation [18,19], where it has emerged as a new field of application for delimited continuations [9,19,29,38,41,44,69].

6.2 The free monoid

A source term in the free monoid is either a variable, the unit element, or the product of two terms:

$$\mathit{term} \ni t ::= x \mid \varepsilon \mid t \star t'$$

The product is associative and the unit element is neutral. These properties justify the following conversion rules:

$$\begin{array}{rcl} t \star (t' \star t'') & \leftrightarrow & (t \star t') \star t'' \\ t \star \varepsilon & \leftrightarrow & t \\ \varepsilon \star t & \leftrightarrow & t \end{array}$$

We aim (for example) for list-like flat normal forms:

$$\mathit{term}^{\mathrm{nf}} \ni \widehat{t} ::= \varepsilon^{\mathrm{nf}} \mid x \star^{\mathrm{nf}} \widehat{t}$$

In a reduction-based approach to normalization, one would orient the conversion rules into reduction rules and one would apply these reduction rules until a normal form is obtained:

$$\begin{array}{rcl} t \star (t' \star t'') & \leftarrow & (t \star t') \star t'' \\ \varepsilon \star t & \rightarrow & t \end{array}$$

In a reduction-free approach to normalization, one defines a normalization function as the composition of a non-standard evaluation function and a reification function. Let us state such a normalization function.

The non-standard domain of values is the transformer

$$\mathit{value} = \mathit{term}^{\mathrm{nf}} \;\rightarrow\; \mathit{term}^{\mathrm{nf}}.$$

The evaluation function is defined by induction over the syntax of source terms, and the reification function inverts it:

$$\begin{aligned}
\mathit{eval}\; x &= \lambda t.x \star^{\mathrm{nf}} t\\
\mathit{eval}\; \varepsilon &= \lambda t.t\\
\mathit{eval}\; (t \star t') &= (\mathit{eval}\; t) \circ (\mathit{eval}\; t')\\
\mathit{reify}\; v &= v\, \varepsilon^{\mathrm{nf}}\\
\mathit{normalize}\; t &= \mathit{reify}\; (\mathit{eval}\; t)
\end{aligned}$$

In effect, *eval* is a mapping from the source monoid to the monoid of transformers (unit is mapped to unit and products are mapped to products) and the normalization function hinges on the built-in associativity of function composition. Dybjer et al. have studied its theoretical content [13, 16, 51]. From a (functional) programming standpoint, the reduction-based approach amounts to flattening a tree iteratively by reordering it, and the reduction-free approach amounts to flattening a tree with an accumulator.

6.3 A language of propositions

A source term, i.e., a proposition, is either a variable, a literal (true or false), a conjunction, or a disjunction:

$$\mathit{term} \ni t ::= x \mid \mathit{true} \mid t \wedge t' \mid \mathit{false} \mid t \vee t'$$

Conjunction and disjunction are associative and distribute over each other; *true* is neutral for conjunction and absorbant for disjunction; and *false* is neutral for disjunction and absorbant for conjunction.

We aim (for example) for list-like disjunctive normal forms:

$$\begin{aligned}
\mathit{term}^{\mathrm{nf}} &\ni \widehat{t} ::= d\\
\mathit{term}^{\mathrm{nf}}_{\mathrm{d}} &\ni d ::= \mathit{false}^{\mathrm{nf}} \mid c \vee^{\mathrm{nf}} d\\
\mathit{term}^{\mathrm{nf}}_{\mathrm{c}} &\ni c ::= \mathit{true}^{\mathrm{nf}} \mid x \wedge^{\mathrm{nf}} c
\end{aligned}$$

Our normalization function is the result of composing a non-standard evaluation function and a reification function. We state them below without proof.

Given the domains of transformers

$$\begin{aligned}
F_1 &= \mathit{term}^{\mathrm{nf}}_{\mathrm{c}} \rightarrow \mathit{term}^{\mathrm{nf}}_{\mathrm{c}}\\
F_2 &= \mathit{term}^{\mathrm{nf}}_{\mathrm{d}} \rightarrow \mathit{term}^{\mathrm{nf}}_{\mathrm{d}}
\end{aligned}$$

the non-standard domain of values is ans_1, where

$$\begin{aligned}
\mathit{ans}_2 &= F_2\\
\mathit{ans}_1 &= (F_1 \rightarrow \mathit{ans}_2) \rightarrow \mathit{ans}_2.
\end{aligned}$$

The evaluation function is defined by induction over the syntax of source terms, and the reification function inverts it:

$$\begin{aligned}
\mathit{eval}_0\, x\, k\, d &= k\,(\lambda c.x \wedge^{\mathrm{nf}} c)\, d\\
\mathit{eval}_0\, \mathit{true}\, k\, d &= k\,(\lambda c.c)\, d\\
\mathit{eval}_0\, (t \wedge t')\, k\, d &= \mathit{eval}_0\, t\, (\lambda f_1.\mathit{eval}_0\, t'\, (\lambda f_1'.k\,(f_1 \circ f_1')))\, d\\
\mathit{eval}_0\, \mathit{false}\, k\, d &= d\\
\mathit{eval}_0\, (t \vee t')\, k\, d &= \mathit{eval}_0\, t\, k\, (\mathit{eval}_0\, t'\, k\, d)\\[1ex]
\mathit{reify}_0\, v &= v\,(\lambda f_1.\lambda d.(f_1\, \mathit{true}^{\mathrm{nf}}) \vee^{\mathrm{nf}} d)\, \mathit{false}^{\mathrm{nf}}\\[1ex]
\mathit{normalize}\, t &= \mathit{reify}_0\,(\mathit{eval}_0\, t)
\end{aligned}$$

This normalization function uses a continuation k, an accumulator d to flatten disjunctions, and another one c to flatten conjunctions. The continuation is delimited: the three first clauses of eval_0 are in CPS; in the fourth, k is discarded (accounting for the fact that *false* is absorbant for conjunction); and in the last, k is duplicated and used in non-tail position (achieving the distribution of conjunctions over disjunctions). The continuation and the accumulators are initialized in the definition of reify_0.

Uncurrying the continuation and mapping eval_0 and reify_0 back to direct style yield the following definition, which lives at level 1 of the CPS hierarchy:

$$\begin{aligned}
\mathit{eval}_1\, x\; d &= (\lambda c.x \wedge^{\mathrm{nf}} c,\; d)\\
\mathit{eval}_1\, \mathit{true}\; d &= (\lambda c.c,\; d)\\
\mathit{eval}_1\, (t \wedge t')\; d &= \mathit{let}\ (f_1,\, d) = \mathit{eval}_1\, t\; d\\
&\quad\ \mathit{in}\ \mathit{let}\ (f_1',\, d) = \mathit{eval}_1\, t'\; d\\
&\qquad\ \mathit{in}\ (f_1 \circ f_1',\, d)\\
\mathit{eval}_1\, \mathit{false}\; d &= \mathcal{S}k.d\\
\mathit{eval}_1\, (t \vee t')\; d &= \mathcal{S}k.k\,(\mathit{eval}_1\, t\, \langle k\,(\mathit{eval}_1\, t'\; d)\rangle)\\[1ex]
\mathit{reify}_1\, v &= \langle \mathit{let}\ (f_1,\, d) = v\, \mathit{false}^{\mathrm{nf}}\\
&\quad\ \mathit{in}\ (f_1\, \mathit{true}^{\mathrm{nf}}) \vee^{\mathrm{nf}} d\rangle\\
\mathit{normalize}\, t &= \mathit{reify}_1\,(\mathit{eval}_1\, t)
\end{aligned}$$

The three first clauses of eval_1 are in direct style; the two others abstract control with shift. In the fourth clause, the context is discarded; and in the last clause, the context is duplicated and composed. The context and the accumulators are initialized in the definition of reify_1.

This direct-style version makes it even more clear than the CPS version that the accumulator for the disjunctions in normal form is a threaded state. A continuation-based, state-based version (or better, a monad-based one) can therefore be written—but it is out of scope here.

6.4 A hierarchical language of units and products

We consider a generalization of propositional logic where a source term is either a variable, a unit in a hierarchy of units, or a product in a hierarchy of products:

$$\begin{aligned}
\mathit{term} \ni t ::= x \mid \varepsilon_i \mid t \star_i t'\\
\mathit{where}\ 1 \leq i \leq n.
\end{aligned}$$

All the products are associative and distribute over each other. All units are neutral for products with the same index and absorbant for products with other indices.

- If $n = 1$, then the language is that of the free monoid, as in Section 6.2.

- If $n = 2$, then the language is that of propositions, as in Section 6.3: ε_1 is *true*, $\star_1$ is $\wedge$, ε_2 is *false*, and $\star_2$ is $\vee$.

We aim (for example) for a generalization of disjunctive normal forms:

$$\begin{array}{lcrcl}
\mathit{term}^{\mathrm{nf}} & \ni & \widehat{t} & ::= & t_n \\
\mathit{term}^{\mathrm{nf}}_n & \ni & t_n & ::= & \varepsilon^{\mathrm{nf}}_n \ \mid\ t_{n-1} \star^{\mathrm{nf}}_n t_n \\
 & & & \vdots & \\
\mathit{term}^{\mathrm{nf}}_1 & \ni & t_1 & ::= & \varepsilon^{\mathrm{nf}}_1 \ \mid\ t_0 \star^{\mathrm{nf}}_1 t_1 \\
\mathit{term}^{\mathrm{nf}}_0 & \ni & t_0 & ::= & x
\end{array}$$

For presentational reasons, in the remainder of this section we arbitrarily fix n to be 5.

Our normalization function is the result of composing a non-standard evaluation function and a reification function. We state them below without proof. Given the domains of transformers

$$\begin{array}{lcl}
F_1 & = & \mathit{term}^{\mathrm{nf}}_1 \to \mathit{term}^{\mathrm{nf}}_1 \\
F_2 & = & \mathit{term}^{\mathrm{nf}}_2 \to \mathit{term}^{\mathrm{nf}}_2 \\
F_3 & = & \mathit{term}^{\mathrm{nf}}_3 \to \mathit{term}^{\mathrm{nf}}_3 \\
F_4 & = & \mathit{term}^{\mathrm{nf}}_4 \to \mathit{term}^{\mathrm{nf}}_4 \\
F_5 & = & \mathit{term}^{\mathrm{nf}}_5 \to \mathit{term}^{\mathrm{nf}}_5
\end{array}$$

the non-standard domain of values is ans_1, where

$$\begin{array}{lcl}
\mathit{ans}_5 & = & F_5 \\
\mathit{ans}_4 & = & (F_4 \to \mathit{ans}_5) \to \mathit{ans}_5 \\
\mathit{ans}_3 & = & (F_3 \to \mathit{ans}_4) \to \mathit{ans}_4 \\
\mathit{ans}_2 & = & (F_2 \to \mathit{ans}_3) \to \mathit{ans}_3 \\
\mathit{ans}_1 & = & (F_1 \to \mathit{ans}_2) \to \mathit{ans}_2.
\end{array}$$

The evaluation function is defined by induction over the syntax of source terms, and the reification function inverts it:

$$\begin{array}{rcl}
\mathit{eval}_0\, x\, k_1\, k_2\, k_3\, k_4\, t_5 & = & k_1\, (\lambda t_1.x \star^{\mathrm{nf}}_1 t_1)\, k_2\, k_3\, k_4\, t_5 \\
\mathit{eval}_0\, \varepsilon_1\, k_1\, k_2\, k_3\, k_4\, t_5 & = & k_1\, (\lambda t_1.t_1)\, k_2\, k_3\, k_4\, t_5 \\
\mathit{eval}_0\, (t \star_1 t')\, k_1\, k_2\, k_3\, k_4\, t_5 & = & \mathit{eval}_0\, t\, (\lambda f_1.\mathit{eval}_0\, t'\, (\lambda f'_1.k_1\, (f_1 \circ f'_1)))\, k_2\, k_3\, k_4\, t_5 \\
\mathit{eval}_0\, \varepsilon_2\, k_1\, k_2\, k_3\, k_4\, t_5 & = & k_2\, (\lambda t_2.t_2)\, k_3\, k_4\, t_5 \\
\mathit{eval}_0\, (t \star_2 t')\, k_1\, k_2\, k_3\, k_4\, t_5 & = & \mathit{eval}_0\, t\, k_1\, (\lambda f_2.\mathit{eval}_0\, t'\, k_1\, (\lambda f'_2.k_2\, (f_2 \circ f'_2)))\, k_3\, k_4\, t_5 \\
\mathit{eval}_0\, \varepsilon_3\, k_1\, k_2\, k_3\, k_4\, t_5 & = & k_3\, (\lambda t_3.t_3)\, k_4\, t_5 \\
\mathit{eval}_0\, (t \star_3 t')\, k_1\, k_2\, k_3\, k_4\, t_5 & = & \mathit{eval}_0\, t\, k_1\, k_2\, (\lambda f_3.\mathit{eval}_0\, t'\, k_1\, k_2\, (\lambda f'_3.k_3\, (f_3 \circ f'_3)))\, k_4\, t_5 \\
\mathit{eval}_0\, \varepsilon_4\, k_1\, k_2\, k_3\, k_4\, t_5 & = & k_4\, (\lambda t_4.t_4)\, t_5 \\
\mathit{eval}_0\, (t \star_4 t')\, k_1\, k_2\, k_3\, k_4\, t_5 & = & \mathit{eval}_0\, t\, k_1\, k_2\, k_3\, (\lambda f_4.\mathit{eval}_0\, t'\, k_1\, k_2\, k_3\, (\lambda f'_4.k_4\, (f_4 \circ f'_4)))\, t_5 \\
\mathit{eval}_0\, \varepsilon_5\, k_1\, k_2\, k_3\, k_4\, t_5 & = & t_5 \\
\mathit{eval}_0\, (t \star_5 t')\, k_1\, k_2\, k_3\, k_4\, t_5 & = & \mathit{eval}_0\, t\, k_1\, k_2\, k_3\, k_4\, (\mathit{eval}_0\, t'\, k_1\, k_2\, k_3\, k_4\, t_5) \\
\mathit{reify}_0\, v & = & v\, (\lambda f_1.\lambda k_2.k_2\, (\lambda t_2.(f_1\, \varepsilon^{\mathrm{nf}}_1) \star^{\mathrm{nf}}_2 t_2)) \\
 & & (\lambda f_2.\lambda k_3.k_3\, (\lambda t_3.(f_2\, \varepsilon^{\mathrm{nf}}_2) \star^{\mathrm{nf}}_3 t_3)) \\
 & & (\lambda f_3.\lambda k_4.k_4\, (\lambda t_4.(f_3\, \varepsilon^{\mathrm{nf}}_3) \star^{\mathrm{nf}}_4 t_4)) \\
 & & (\lambda f_4.\lambda t_5.(f_4\, \varepsilon^{\mathrm{nf}}_4) \star^{\mathrm{nf}}_5 t_5) \\
 & & \varepsilon_5 \\
\mathit{normalize}\, t & = & \mathit{reify}_0\, (\mathit{eval}_0\, t)
\end{array}$$

This normalization function uses four delimited continuations k_1, k_2, k_3, k_4 and five accumulators t_1, t_2, t_3, t_4, t_5 to flatten each of the successive products. In the clause of each ε_i, the continuations $k_1, \dots, k_{i-1}$ are discarded, accounting for the fact that ε_i is absorbant for $\star_1, \dots, \star_{i-1}$, and the identity function is passed to k_i, accounting for the fact that ε_i is neutral for $\star_i$. In the clause of each $\star_{i+1}$, the continuations $k_1, \dots, k_i$ are duplicated and used in non-tail position, achieving the distribution of $\star_{i+1}$ over $\star_1, \dots, \star_i$. The continuations and the accumulators are initialized in the definition of *reify*$_0$.

This normalization function lives at level 0 of the CPS hierarchy, but we can express it at a higher level using shift and reset. For example, uncurrying k_3 and k_4 and mapping *eval*$_0$ and *reify*$_0$ back to direct style twice yield the following intermediate definition, which lives at level 2:

$$\begin{array}{rcl}
\mathit{eval}_2\; x\; k_1\; k_2\; t_5 & = & k_1\; (\lambda t_1.x \star_1^{\mathrm{nf}} t_1)\; k_2\; t_5 \\
\mathit{eval}_2\; \varepsilon_1\; k_1\; k_2\; t_5 & = & k_1\; (\lambda t_1.t_1)\; k_2\; t_5 \\
\mathit{eval}_2\; (t \star_1 t')\; k_1\; k_2\; t_5 & = & \mathit{eval}_2\; t\; (\lambda f_1.\mathit{eval}_2\; t'\; (\lambda f_1'.k_1\; (f_1 \circ f_1')))\; k_2\; t_5 \\
\mathit{eval}_2\; \varepsilon_2\; k_1\; k_2\; t_5 & = & k_2\; (\lambda t_2.t_2)\; t_5 \\
\mathit{eval}_2\; (t \star_2 t')\; k_1\; k_2\; t_5 & = & \mathit{eval}_2\; t\; k_1\; (\lambda f_2.\mathit{eval}_2\; t'\; k_1\; (\lambda f_2'.k_2\; (f_2 \circ f_2')))\; t_5 \\
\mathit{eval}_2\; \varepsilon_3\; k_1\; k_2\; t_5 & = & (\lambda t_3.t_3,\; t_5) \\
\mathit{eval}_2\; (t \star_3 t')\; k_1\; k_2\; t_5 & = & \mathit{let}\; (f_3,\; t_5) = \mathit{eval}_2\; t\; k_1\; k_2\; t_5 \\
 & & \mathit{in}\; \mathit{let}\; (f_3',\; t_5) = \mathit{eval}_2\; t'\; k_1\; k_2\; t_5 \\
 & & \quad \mathit{in}\; (f_3 \circ f_3',\; t_5) \\
\mathit{eval}_2\; \varepsilon_4\; k_1\; k_2\; t_5 & = & \mathcal{S}_1 k_3.(\lambda t_4.t_4,\; t_5) \\
\mathit{eval}_2\; (t \star_4 t')\; k_1\; k_2\; t_5 & = & \mathcal{S}_1 k_3.\mathit{let}\; (f_4,\; t_5) = \langle k_3\; (\mathit{eval}_2\; t\; k_1\; k_2\; t_5)\rangle_1 \\
 & & \qquad \mathit{in}\; \mathit{let}\; (f_4',\; t_5) = \langle k_3\; (\mathit{eval}_2\; t'\; k_1\; k_2\; t_5)\rangle_1 \\
 & & \qquad\quad \mathit{in}\; (f_4 \circ f_4',\; t_5) \\
\mathit{eval}_2\; \varepsilon_5\; k_1\; k_2\; t_5 & = & \mathcal{S}_2 k_4.t_5 \\
\mathit{eval}_2\; (t \star_5 t')\; k_1\; k_2\; t_5 & = & \mathcal{S}_1 k_3.\mathcal{S}_2 k_4.\mathit{let}\; t_5 = \langle k_4\; \langle k_3\; (\mathit{eval}_2\; t'\; k_1\; k_2\; t_5)\rangle_1\rangle_2 \\
 & & \qquad\qquad \mathit{in}\; \langle k_4\; \langle k_3\; (\mathit{eval}_2\; t\; k_1\; k_2\; t_5)\rangle_1\rangle_2 \\
\mathit{reify}_2\; v & = & \langle \mathit{let}\; (f_4,\; t_5) = \langle \mathit{let}\; (f_3,\; t_5) = v\; (\lambda f_1.\lambda k_2.k_2\; (\lambda t_2.(f_1\; \varepsilon_1^{\mathrm{nf}}) \star_2^{\mathrm{nf}} t_2)) \\
 & & \qquad\qquad\qquad\qquad\qquad\quad (\lambda f_2.\lambda t_3.(f_2\; \varepsilon_2^{\mathrm{nf}}) \star_3^{\mathrm{nf}} t_3) \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \varepsilon_5 \\
 & & \qquad\qquad \mathit{in}\; (\lambda f_4.(f_3\; \varepsilon_3^{\mathrm{nf}}) \star_4^{\mathrm{nf}} t_4,\; t_5)\rangle_1 \\
 & & \quad \mathit{in}\; (f_4\; \varepsilon_4^{\mathrm{nf}}) \star_5^{\mathrm{nf}} t_5\rangle_2 \\
\mathit{normalize}\; t & = & \mathit{reify}_2\; (\mathit{eval}_2\; t)
\end{array}$$

Whereas *eval*$_0$ had four layered continuations, *eval*$_2$ has only two layered continuations since it has been mapped back to direct style twice. Where *eval*$_0$ accesses k_3 as one of its parameters, *eval*$_2$ abstracts the first layer of control with shift$_1$, and where *eval*$_0$ accesses k_4 as one of its parameters, *eval*$_2$ abstracts the first and the second layer of control with shift$_2$.

Uncurrying k_1 and k_2 and mapping *eval*$_2$ and *reify*$_2$ back to direct style twice yield the following direct-style definition, which lives at level 4 of the CPS hierarchy:

$$\begin{array}{rcl}
\mathit{eval}_4\; x\; t_5 & = & (\lambda t_1.x \star_1^{\mathrm{nf}} t_1,\; t_5) \\
\mathit{eval}_4\; \varepsilon_1\; t_5 & = & (\lambda t_1.t_1,\; t_5) \\
\mathit{eval}_4\; (t \star_1 t')\; t_5 & = & \mathit{let}\; (f_1,\; t_5) = \mathit{eval}_4\; t\; t_5 \\
 & & \mathit{in}\; \mathit{let}\; (f_1',\; t_5) = \mathit{eval}_4\; t'\; t_5 \\
 & & \quad \mathit{in}\; (f_1 \circ f_1',\; t_5)
\end{array}$$

$$
\begin{array}{rcl}
\mathit{eval}_4\,\varepsilon_2\,t_5 & = & \mathcal{S}_1 k_1.(\lambda t_2.t_2,\ t_5)\\
\mathit{eval}_4\,(t \star_2 t')\,t_5 & = & \mathcal{S}_1 k_1.\mathit{let}\ (f_2,\ t_5) = \langle k_1\,(\mathit{eval}_4\,t\,t_5)\rangle_1\\
 & & \quad \mathit{in}\ \mathit{let}\ (f_2',\ t_5) = \langle k_1\,(\mathit{eval}_4\,t'\,t_5)\rangle_1\\
 & & \qquad \mathit{in}\ (f_2 \circ f_2',\ t_5)\\
\mathit{eval}_4\,\varepsilon_3\,t_5 & = & \mathcal{S}_2 k_2.(\lambda t_3.t_3,\ t_5)\\
\mathit{eval}_4\,(t \star_3 t')\,t_5 & = & \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\mathit{let}\ (f_3,\ t_5) = \langle k_2\,\langle k_1\,(\mathit{eval}_4\,t\,t_5)\rangle_1\rangle_2\\
 & & \quad \mathit{in}\ \mathit{let}\ (f_3',\ t_5) = \langle k_2\,\langle k_1\,(\mathit{eval}_4\,t'\,t_5)\rangle_1\rangle_2\\
 & & \qquad \mathit{in}\ (f_3 \circ f_3',\ t_5)\\
\mathit{eval}_4\,\varepsilon_4\,t_5 & = & \mathcal{S}_3 k_3.(\lambda t_4.t_4,\ t_5)\\
\mathit{eval}_4\,(t \star_4 t')\,t_5 & = & \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\mathcal{S}_3 k_3.\mathit{let}\ (f_4,\ t_5) = \langle k_3\,\langle k_2\,\langle k_1\,(\mathit{eval}_4\,t\,t_5)\rangle_1\rangle_2\rangle_3\\
 & & \quad \mathit{in}\ \mathit{let}\ (f_4',\ t_5) = \langle k_3\,\langle k_2\,\langle k_1\,(\mathit{eval}_4\,t'\,t_5)\rangle_1\rangle_2\rangle_3\\
 & & \qquad \mathit{in}\ (f_4 \circ f_4',\ t_5)\\
\mathit{eval}_4\,\varepsilon_5\,t_5 & = & \mathcal{S}_4 k_4.t_5\\
\mathit{eval}_4\,(t \star_5 t')\,t_5 & = & \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\mathcal{S}_3 k_3.\mathcal{S}_4 k_4.\mathit{let}\ t_5 = \langle k_4\,\langle k_3\,\langle k_2\,\langle k_1\,(\mathit{eval}_4\,t'\,t_5)\rangle_1\rangle_2\rangle_3\rangle_4\\
 & & \quad \mathit{in}\ \langle k_4\,\langle k_3\,\langle k_2\,\langle k_1\,(\mathit{eval}_4\,t\,t_5)\rangle_1\rangle_2\rangle_3\rangle_4
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{reify}_4\,v & = & \langle \mathit{let}\ (f_4,\ t_5) = \langle \mathit{let}\ (f_3,\ t_5) = \langle \mathit{let}\ (f_2,\ t_5) = \langle \mathit{let}\ (f_1,\ t_5) = v\,\varepsilon_5\\
 & & \qquad\qquad\qquad\qquad \mathit{in}\ (\lambda f_2.(f_1\,\varepsilon_1^{\mathrm{nf}}) \star_2^{\mathrm{nf}} t_2,\ t_5)\rangle_1\\
 & & \qquad\qquad\qquad \mathit{in}\ (\lambda f_3.(f_2\,\varepsilon_2^{\mathrm{nf}}) \star_3^{\mathrm{nf}} t_3,\ t_5)\rangle_2\\
 & & \qquad\qquad \mathit{in}\ (\lambda f_4.(f_3\,\varepsilon_3^{\mathrm{nf}}) \star_4^{\mathrm{nf}} t_4,\ t_5)\rangle_3\\
 & & \quad \mathit{in}\ (f_4\,\varepsilon_4^{\mathrm{nf}}) \star_5^{\mathrm{nf}} t_5\rangle_4\\
 & & \\
\mathit{normalize}\,t & = & \mathit{reify}_4\,(\mathit{eval}_4\,t)
\end{array}
$$

Whereas *eval*$_2$ had two layered continuations, *eval*$_4$ has none since it has been mapped back to direct style twice. Where *eval*$_2$ accesses k_1 as one of its parameters, *eval*$_4$ abstracts the first layer of control with shift$_1$, and where *eval*$_2$ accesses k_2 as one of its parameters, *eval*$_4$ abstracts the first and the second layer of control with shift$_2$. Where *eval*$_2$ uses reset$_1$ and shift$_1$, *eval*$_4$ uses reset$_3$ and shift$_3$, and where *eval*$_2$ uses reset$_2$ and shift$_2$, *eval*$_4$ uses reset$_4$ and shift$_4$.

6.5 A note about efficiency

We have implemented all the definitions of Section 6.4 as well as the intermediate versions *eval*$_1$ and *eval*$_3$, using Standard ML of New Jersey [27]. We have also implemented hierarchical normalization functions for other values than 5.

For high products (i.e., in Section 6.4, for source terms using $\star_3$ and $\star_4$), the normalization function living at level 0 of the CPS hierarchy is the most efficient one. On the other hand, for low products (i.e., in Section 6.4, for source terms using $\star_1$ and $\star_2$), the normalization functions living at a higher level of the CPS hierarchy are the most efficient ones. These relative efficiencies are explained in terms of resources:

- Accessing to a continuation as an explicit parameter is more efficient than accessing to it through a control operator.
- On the other hand, the restriction of *eval*$_4$ to source terms that only use ε_1 and $\star_1$ is in direct style, whereas the corresponding restrictions of *eval*$_2$ and *eval*$_0$ pass a number of extra parameters. These extra parameters penalize performance.

The better performance of programs in the CPS hierarchy has already been reported for level 1 in the context of continuation-based partial evaluation [53], and it has been reported for a similar "pay as you go" reason: a program that abstracts control relatively rarely is run more efficiently in direct style with a control operator rather than in continuation-passing style.

6.6 Summary and conclusion

We have illustrated the CPS hierarchy with an application of normalization by evaluation that naturally involves successive layers of continuations and that demonstrates the expressive power of shift$_n$ and reset$_n$.

The application also suggests alternative control operators that would fit better its continuation-based programming pattern. For example, instead of representing a delimited continuation as a function and apply it as such, we could represent it as a continuation and apply it with a throw operator as in MacLisp and Standard ML of New Jersey. For another example, instead of throwing a value to a continuation, we could specify the continuation of a computation, e.g., with a *reflect*$_i$ special form. For a third example, instead of abstracting control up to a layer n, we could give access to each of the successive layers up to n, e.g., with a $\mathcal{L}_n$ operator. Then instead of

$$\begin{array}{lcl} \mathit{eval}_4\,(t \star_4 t')\,t_5 & = & \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\mathcal{S}_3 k_3.\mathit{let}\,(f_4,\, t_5) = \langle k_3\,\langle k_2\,\langle k_1\,(\mathit{eval}_4\,t\,t_5)\rangle_1\rangle_2\rangle_3 \\ & & \qquad\qquad\qquad\quad \mathit{in}\ \mathit{let}\,(f_4',\, t_5) = \langle k_3\,\langle k_2\,\langle k_1\,(\mathit{eval}_4\,t'\,t_5)\rangle_1\rangle_2\rangle_3 \\ & & \qquad\qquad\qquad\qquad \mathit{in}\ (f_4 \circ f_4',\, t_5) \end{array}$$

one could write

$$\begin{array}{lcl} \mathit{eval}_4\,(t \star_4 t')\,t_5 & = & \mathcal{L}_3\,(k_1,\, k_2,\, k_3).\mathit{let}\,(f_4',\, t_5) = \mathit{reflect}_3\,(\mathit{eval}_4\,t\,t_5,\, k_1,\, k_2,\, k_3) \\ & & \qquad\qquad\qquad\quad \mathit{in}\ \mathit{let}\,(f_4',\, t_5) = \mathit{reflect}_3\,(\mathit{eval}_4\,t'\,t_5,\, k_1,\, k_2,\, k_3) \\ & & \qquad\qquad\qquad\qquad \mathit{in}\ (f_4 \circ f_4',\, t_5). \end{array}$$

Such alternative control operators can be more convenient to use, while being compatible with CPS.

7 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 derived the corresponding abstract machine. From this abstract machine, it is straightforward to obtain a reduction semantics of delimited control that, by construction, is compatible with CPS—both for one-step reduction and for evaluation. These results can also be established without the guideline of CPS, but less easily.

The whole approach scales seamlessly to account for the shift$_n$ and reset$_n$ family of delimited-control operators and more generally for any control operators that are compatible with CPS. These results would be non-trivial to establish without the guideline of CPS.

Defunctionalization provides a key for connecting continuation-passing style and operational intuitions about control. Indeed most of the time, control stacks and evaluation contexts are defunctionalized continuations. Defunctionalization also provides a key for identifying where operational intuitions about control go beyond CPS (see Section 4.5). 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 of CW'04 for their comments. The third author would also like to thank Samuel Lindley for our joint initial study of the normalization functions of Section 6.

This work is partially supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-02-0474 (for the two first authors) and Grant no. 21-03-0545 (for the third author).

A From reduction semantics to calculus for the CPS hierarchy

The original specification of shift$_n$ and reset$_n$ was based on a definitional interpreter in CPS. In the preceding sections, we have presented two alternative definitions of these control operators: the small-step operational semantics given by the abstract machine and the context-based reduction semantics using global rules, i.e., based on a decomposition of the entire term under consideration. In this appendix, we complete the picture by showing yet another approach to delimited continuations, namely a calculus for shift and reset. The calculus provides a local reduction semantics and facilitates equational reasoning about programs with delimited continuations. In the presence of Kameyama's axioms [48], the practical significance of the calculus is limited. It is, however, interesting from a theoretical viewpoint, since it enjoys most of the properties one could desire. Our development here follows Plotkin's treatment of the λ-calculus under call-by-value [58], as well as Felleisen's investigation of the λ-calculus under call-by-value and with control operators [30–33].

A.1 A calculus for shift and reset

In order to define the calculus for shift and reset we need a basic notion of reduction given in terms of local reduction rules. We immediately see that most of the global rules do not interact with their surrounding contexts, and therefore can be made into a notion of reduction simply by stripping off the context and the meta-context:

$$\begin{array}{lrcl} (\delta) & \mathit{succ}\,\ulcorner m \urcorner & \rightarrow & \ulcorner m+1 \urcorner \\ (\beta_\lambda) & (\lambda x.t)\,v & \rightarrow & t\{v/x\} \\ (\text{val}) & \langle v \rangle & \rightarrow & v \end{array}$$

The only problematic case is the $(\mathcal{S}_\lambda)$-rule which captures the entire context. Bearing in mind Felleisen et al.'s development for $\lambda_v + \mathcal{C}$ [33], one can venture to simulate the $(\mathcal{S}_\lambda)$-rule with a number of local reductions. The idea is to lift the shift-expression over the context—by consuming the context piecemeal—until a reset is found, and then remove the shift expression altogether. In order to be able to do so, we first need to eliminate explicit contexts from the syntax and represent captured contexts as terms in the ordinary λ-calculus so that we can then compose them as syntactic objects. Also, we need to merge the two rules $(\mathcal{S}_\lambda)$ and (β_{ctx}) into one:

$$(\mathcal{S}_\lambda) \quad C_2 \;\#\; C_1[\mathcal{S}k.t] \rightarrow C_2 \;\#\; \bullet[t\{\lambda x.\langle C_1[x]\rangle/k\}]$$

By analyzing the structure of contexts C_1, from $(\mathcal{S}_\lambda)$ we derive the following set of rules:

$$\begin{array}{llcl}
(\mathcal{S}_r) & (\mathcal{S}k.t)\,t' & \rightarrow & \mathcal{S}k'.t\{\lambda x.\langle k'\,(x\,t')\rangle/k\} \\
(\mathcal{S}_l) & v\,(\mathcal{S}k.t) & \rightarrow & \mathcal{S}k'.t\{\lambda x.\langle k'\,(v\,x)\rangle/k\} \\
(\mathcal{S}_s) & \mathit{succ}\,(\mathcal{S}k.t) & \rightarrow & \mathcal{S}k'.t\{\lambda x.\langle k'\,(\mathit{succ}\,x)\rangle/k\} \\
(\mathcal{S}_{\langle\rangle}) & \langle \mathcal{S}k.t\rangle & \rightarrow & \langle t\{\lambda x.\langle x\rangle/k\}\rangle
\end{array}$$

The context surrounding a shift operator—previously captured as a whole—is now represented as a composition of individual basic contexts. The composition is obtained by introducing additional β_λ-reductions, and each basic context is surrounded with a reset operator. The representation of contexts we obtain is not identical with the previous one, but they are equivalent in the sense of Theorem 8, below. Moreover, all of the local reduction rules are sound with respect to CPS.

Now, we are in a position to define the calculus for shift and reset.

Definition 1 ($\lambda\mathcal{S}$-calculus)

1. *The basic notion of reduction r is defined as follows:*

$$r \;=\; \delta \,\cup\, \beta_\lambda \,\cup\, \mathit{val} \,\cup\, \mathcal{S}_r \,\cup\, \mathcal{S}_l \,\cup\, \mathcal{S}_s \,\cup\, \mathcal{S}_{\langle\rangle}.$$

2. *The one step r-reduction, $\rightarrow_r$, is the compatible closure of r.*

3. *The reflexive-transitive closure of $\rightarrow_r$ is denoted by $\rightarrow_r^*$.*

4. *The smallest equivalence relation generated by $\rightarrow_r$ is denoted by $=_r$.*

5. *When t and t' are convertible, i.e., they are in the relation $=_r$, we write $\lambda\mathcal{S} \vdash t = t'$.*

As can be proved using a method of Tait and Martin-Löf [10], the $\lambda\mathcal{S}$-calculus is confluent:

Theorem 7 *The $\lambda\mathcal{S}$-calculus is Church-Rosser, i.e., $\rightarrow_r^*$ satisfies the diamond property.*

As a next step, we define evaluation out of reduction in the $\lambda\mathcal{S}$-calculus. To this end, we specify the leftmost-outermost call-by-value evaluation strategy given by means of evaluation contexts and a standard reduction function:

Definition 2 (Evaluation in the $\lambda\mathcal{S}$-calculus)

1. *Evaluation contexts C are defined as follows:*

$$C \;::=\; \bullet \;\mid\; C\,t \;\mid\; v\,C \;\mid\; \mathit{succ}\,C \;\mid\; \langle C\rangle$$

2. *The standard reduction function maps a term t_1 to a term t_2 ($t_1 \mapsto_r t_2$) if there are terms t_1' and t_2', and an evaluation context C, such that $(t_1', t_2') \in r$, $t_1 \equiv C\,[t_1']$, and $t_2 \equiv C\,[t_2']$.*

3. *We define the evaluation function eval_r as the partial function mapping a term t to a value v: $\mathit{eval}_r(t) = v$ iff $t \mapsto_r^* v$.*

As can be shown, the unique decomposition lemma holds for so defined contexts and redexes (given by the left-hand sides of the reduction rules), ensuring the well definedeness of the evaluation function $eval_r$.

As mentioned above, the representation of captured contexts in the calculus differs from that in the context-based reduction semantics or in the substitution-based abstract machine. The following theorem, which can be proved using Felleisen's technique [30], shows that the simulation of evaluation in the calculus is operationally faithful to evaluation with the global reduction rules ($\rightarrow^*$), and therefore, to evaluation with the abstract machine. (We assume by default that programs using shift are enclosed in a top-level reset.)

Theorem 8 *If t is a program, and $\ulcorner m \urcorner$ is an integer literal, then $eval_r(t) = \ulcorner m \urcorner$ iff $t \rightarrow^* \ulcorner m \urcorner$.*

When seen as axioms (equations), the local reduction rules can be used as optimization tools for programs with shift and reset. The following theorem ensures that any two terms that are provably equal in the theory $\lambda\mathcal{S}$ are operationally equivalent, i.e., program rewriting with the axioms is safe.

Theorem 9 *If $\lambda\mathcal{S} \vdash t_1 = t_2$ then t_1 and t_2 are operationally equivalent with respect to $eval_r$.*

Although the axioms generated by the local reduction rules are sound with respect to CPS, they are not complete. In his recent work [48], Kameyama presented a concise set of axioms for the λ-calculus with shift and reset that is sound and complete with respect to CPS. The axioms (β_λ), (val) and ($\mathcal{S}_\lambda$) are included in the Kameyama's set of axioms. Interestingly, we observe that if we localize ($\mathcal{S}_\lambda$), i.e., we replace it with ($\mathcal{S}_r$), ($\mathcal{S}_l$) and ($\mathcal{S}_{\langle\rangle}$), the soundness and completeness of such axiomatization is retained.

A.2 A calculus for shift$_n$ and reset$_n$

It is possible to transform the global reduction rules for level n into local reductions that generalize those of Section A.1, as done by Murthy [57]. Similarly to the development of Section A.1, we derive the following notions of reduction that are sound with respect to the iterated CPS transformation [23, 48]:

$$\begin{array}{ll}
(\delta) & succ\,\ulcorner m \urcorner \rightarrow \ulcorner m+1 \urcorner \\
(\beta_\lambda) & (\lambda x.t)\,v \rightarrow t\{v/x\} \\
(\mathrm{val}^n) & \langle v \rangle_i \rightarrow v \\
(\mathcal{S}^n_r) & (\mathcal{S}_i k.t)\,t' \rightarrow \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\cdots \mathcal{S}_i k.t\{\lambda x.\langle k_i \cdots \langle k_2\,\langle k_1\,(x\,t')\rangle_1\rangle_2 \cdots\rangle_i/k\} \\
(\mathcal{S}^n_r) & v\,(\mathcal{S}_i k.t) \rightarrow \mathcal{S}_1 k_1.\mathcal{S}_2 k_2.\cdots \mathcal{S}_i k.t\{\lambda x.\langle k_i \cdots \langle k_2\,\langle k_1\,(v\,x)\rangle_1\rangle_2 \cdots\rangle_i/k\} \\
(\mathcal{S}^n_\leq) & \langle \mathcal{S}_i k.t \rangle_j \rightarrow \langle t\{\lambda x.\langle x \rangle_i/k\}\rangle_j, \quad \text{if } i \leq j \\
(\mathcal{S}^n_>) & \langle \mathcal{S}_i k.t \rangle_j \rightarrow \mathcal{S}_i k.t, \quad \text{if } i > j
\end{array}$$

Again, taking the union of the above local reduction rules (seen as relations between terms) as a basic notion of reduction leads to a definition of the calculus for shift$_n$ and reset$_n$, and subsequently to an evaluation model for the CPS hierarchy.

In addition to the rules presented above, Murthy considered the generalized idempotence law for reset [57]:

$$(\langle\cdot\rangle_{\langle\rangle}) \qquad\qquad \langle\langle t\rangle_i\rangle_j \rightarrow \langle t\rangle_{max(i,j)}$$

However, it turns out that this rule is redundant both in the calculus and in Kameyama's axiomatization. In the former it is not needed for simulating evaluation, and in the latter it is derivable [48].

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. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS-RS-04-3.

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

[4] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.

[5] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A type-theoretic foundation of continuations and prompts. In Kathleen Fischer, editor, *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, pages 40–53, Snowbird, Utah, September 2004. ACM Press.

[6] 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.

[7] Kenichi Asai. Offline partial evaluation for shift and reset. In Nevin Heintze and Peter Sestoft, editors, *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2004)*, pages 3–14, Verona, Italy, August 2003. ACM Press.

[8] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, pages 64–76, Venice, Italy, January 2004. ACM Press.

[9] Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, number 2487 in Lecture Notes in Computer Science, pages 78–92, Pittsburgh, Pennsylvania, October 2002. Springer-Verlag.

[10] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics.* North-Holland, revised edition, 1984.

[11] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137, Berlin, Germany, 1998. Springer-Verlag.

[12] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[13] Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95*, number 1158 in Lecture Notes in Computer Science, pages 47–61, Torino, Italy, June 1995. Springer-Verlag.

[14] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations. In Thielecke [70], pages 25–33.

[15] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[16] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[17] Olivier Danvy. On listing list prefixes. *LISP Pointers*, 2(3-4):42–46, January 1989.

[18] 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.

[19] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[20] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Thielecke [70], pages 13–23. Invited talk.

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

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

[23] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [73], pages 151–160.

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

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

[26] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Technical Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. 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.

[27] 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.

[28] Scott Draves. Implementing bit-addressing with specialization. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, Amsterdam, The Netherlands, June 1997. ACM Press.

[29] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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, Snowbird, Utah, July 1988. ACM Press.

[36] 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.

[37] 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.

[38] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[39] 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.

[40] Mayer Goldberg. Gödelization in the λ-calculus. *Information Processing Letters*, 75(1-2):13–16, 2000.

[41] Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):173–219, 2001.

[42] 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.

[43] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[44] Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In Jieh Hsiang and Atsushi Ohori, editors, *Advances in Computing Science - ASIAN'98*, number

1538 in Lecture Notes in Computer Science, pages 188–205, Manila, The Philippines, December 1998. Springer-Verlag.

[45] 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.

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

[47] John Hughes. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, 22(3):141–144, 1986.

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

[49] 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.

[50] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[51] Yoshiki Kinoshita. A bicategorical analysis of E-categories. *Mathematica Japonica*, 47(1):157–169, 1998.

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

[53] 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, pages 227–238, Orlando, Florida, June 1994. ACM Press.

[54] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.

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

[56] 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.

[57] 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.

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

[59] 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.

[60] 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.

[61] 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.

[62] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Snowbird, Utah, September 2002.

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

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

[65] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [73], pages 161–175.

[66] 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.

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

[68] 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.

[69] Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.

[70] Hayo Thielecke, editor. *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, Venice, Italy, January 2004.

[71] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

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

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

[74] 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.