Typing Control Operators in the CPS Hierarchy

Małgorzata Biernacka
Institute of Computer Science
University of Wrocław
mabi@cs.uni.wroc.pl

Dariusz Biernacki
Institute of Computer Science
University of Wrocław
dabi@cs.uni.wroc.pl

Sergueï Lenglet *
Institute of Computer Science
University of Wrocław
serguei.lenglet@gmail.com

Abstract

The CPS hierarchy of Danvy and Filinski is a hierarchy of continuations that allows for expressing nested control effects characteristic of, e.g., non-deterministic programming or certain instances of normalization by evaluation. In this article, we present a comprehensive study of a typed version of the CPS hierarchy, where the typing discipline generalizes Danvy and Filinski's type system for control operators shift and reset. To this end, we define a typed family of control operators that give access to delimited continuations in the CPS hierarchy and that are slightly more flexible than Danvy and Filinski's family of control operators shift_i and reset_i, but, as we show, are equally expressive. For this type system, we prove subject reduction, soundness with respect to the CPS translation, and termination of evaluation. We also show that our results scale to a type system for even more flexible control operators expressible in the CPS hierarchy.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Languages Constructs and Features—Control Structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages, Theory

Keywords Delimited Continuation, CPS Hierarchy, Type System

1. Introduction

In the recent years delimited continuations have been recognized as an important concept in the landscape of eager functional programming, with new practical [15, 16, 18, 19], theoretical [1, 2, 4, 13, 20, 24, 27], and implementational [17, 21] advances in the field. Of the numerous control operators for delimited continuations, the so-called static control operators shift and reset introduced by Danvy and Filinski in their seminal work [8] occupy a special position, primarily due to the fact that their definition has been based on the well-known concept of the Continuation-Passing Style (CPS) [23]. As such, shift and reset have solid semantic foundations [5, 8, 10], they are fundamentally related to other computational effects [10, 11] and their use is guided by CPS [5, 8]. A typical application of shift and reset, motivating their definition, are algorithms that non-deterministically generate elements of some collection, based on the success-failure continuation model of backtracking [8].

When iterated, the CPS translation leads to a hierarchy of continuations, generalizing the concept of the continuation and meta-continuation used to define the semantics of shift and reset. In terms of so defined CPS hierarchy, Danvy and Filinski proposed a hierarchy of control operators shift_i and reset_i ($i \geq 1$) that generalize shift and reset, and that make it possible to separate computational effects that should exist independently in a program [8]. For example, in order to collect the solutions found by a backtracking algorithm implemented with shift_1 and reset_1, one has to employ shift_2 and reset_2, so that there is no interference between searching and emitting the results of the search. The CPS hierarchy was also envisaged to account for nested computations in hierarchical structures. Indeed, as shown by the first two authors and Danvy [5], the hierarchy naturally accounts for normalization by evaluation algorithms for hierarchical languages of units and products, generalizing the problem of computing disjunctive or conjunctive normal forms in propositional logic.

So far, the CPS hierarchy has been studied mainly in the untyped setting. Danvy and Filinski defined it in terms of an untyped CPS translation and a valuation function of a denotational semantics [8], Danvy and Yang introduced an operational semantics for the hierarchy and built an SML implementation of the hierarchy based on this semantics [9], Kameyama presented an axiomatization for the hierarchy that is sound and complete with respect to the CPS translation [14], and Biernacka et al. derived abstract machines and reduction semantics for the hierarchy from the definitional evaluator [5].

A byproduct of Danvy and Yang's implementation in ML is a rather restrictive type system for the hierarchy, where at each level, the answer type of the continuation is fixed once and for all. This type system generalizes Filinski's type system for shift and reset [10], but it has not been investigated on a formal ground. Formal type systems for the hierarchy appear in Murthy's [22] and in Shan's [25] work. Murthy proposes a more relaxed typing discipline than that of Danvy and Yang in that it allows the delimited continuations of level i to have varying answer types, provided the answer type agrees with the type expected by the continuation at level $i + 1$. Shan's type system, in turn, generalizes Danvy and Filinski's type system [7] which is the most expressive monomorphic type system for shift and reset. In Danvy and Filinski's type system, control effects can modify the answer type of the context (i.e., a first-order representation of the continuation) in which they occur, so statically, the answer type of the continuation at level i can be different from the argument type of the continuation at level $i+1$. Shan's work is driven by applications in linguistic theory and the hierarchy he considers is organized differently from the original CPS hierarchy of Danvy and Filinski (level 0 in his hierarchy is the highest whereas it is the lowest in the original hierarchy). Fur-

* The author is supported by the Alan Bensoussan Fellowship program.

PDPP'11, July 20–22, 2011, Odense, Denmark.

thermore, no metatheoretic properties of the presented system are considered in Shan's work.

In this article, we propose a type system which also generalizes Danvy and Filinski's type system but which has been derived directly from the iterated CPS translation that defines the original CPS hierarchy. Furthermore, the control operators we consider are slightly more flexible, although equally expressive, than the original shift$_i$ and reset$_i$ family in that they capture the subsequent continuations to separate continuation variables and allow for throwing to tuples of continuations, where the continuations may come from different captures. Such control operators arise naturally from the structure of CPS if one considers the operations of capturing continuations and throwing to captured continuations independently.

We would like to stress that it is our intention not to limit the type system for control operators in the CPS hierarchy in any way and to offer the programmer the full power of the simply-typed CPS, even though the resulting type system is rather complex. One of the goals of this work is precisely to explain the most naturally typed version of the CPS hierarchy as it is. Moreover, it has already been observed by Shan [25] that the most general types à la Danvy and Filinski are necessary in some practical applications, for instance, to deal with quantifier scope ambiguity in linguistics. Also, there exist examples that require answer type modification at the first level of the hierarchy, for instance, listing list prefixes [5] or the printf function [2], and there are potentially many more that live higher in the hierarchy waiting to be discovered. A typical scenario in which a mismatch between the answer type at level i and the type expected at level $i+1$ may arise involves a rather standard operation when programming in CPS—aborting computation of type α at level i and returning a value of a different type β to level $i+1$.

The overall goal of this article is to establish type-theoretic foundations of the CPS hierarchy and to build a general framework for studying typed control operators definable in the CPS hierarchy. The contributions of this work can be summarized as follows:

- the definition of a new family of control operators in the CPS hierarchy that are slightly more flexible than the shift$_i$ and reset$_i$ family, given in terms of a CPS translation and reduction semantics provably sound with respect to the CPS translation (Sections 3.1 and 3.2);
- a type system à la Danvy and Filinski for the proposed operators, with proofs of subject reduction and soundness of the typing with respect to the CPS translation (Section 3.3);
- a proof of termination of evaluation in reduction semantics, using a context-based method of reducibility predicates (Section 3.4);
- a simulation of the presented operators with the original family of shift$_i$ and reset$_i$ (Section 3.5);
- a generalization of the presented results to a hierarchy of even more flexible control operators expressible in the CPS hierarchy (Section 4).

2. A Programming Example

Before proceeding to the proper part of the present article, let us briefly discuss a representative example of programming in the CPS hierarchy. A typical application of the CPS hierarchy is nondeterministic programming with two layers of continuations (success and failure) on top of which there is some mechanism emitting or collecting the generated objects [8]. In order to separate searching for solutions from collecting them one uses shift$_1$/reset$_1$ for the former and shift$_2$/reset$_2$ for the latter. Here are the definitions of the standard backtracking primitives written in SML, using Danvy and Yang's implementation of the CPS hierarchy [9]:

```
fun fail ()
    = shift_1 (fn k => ())

fun amb c1 c2
    = shift_1 (fn k =>
         (reset_1 (fn () => k (c1 ()));
          reset_1 (fn () => k (c2 ()))))

fun emit v
    = shift_2 (fn k => v :: (k ()))

fun collect c
    = reset_2 (fn () =>
         let val () = reset_1 (fn () => emit (c ()))
         in nil end)
```

Given the types of `shift_i` and `reset_i`:

```
shift_i : (('a -> ans_i) -> ans_i) -> 'a
reset_i : (unit -> ans_i) -> ans_i
```

for fixed types `ans_i` for each `i`, if we fix `ans_1` to `unit` and `ans_2` to `int list list` we can, for example, write a program that lists list prefixes as follows:

```
fun prefixes xs
    = let fun walk nil
              = fail ()
            | walk (x :: xs)
              = amb (fn () => x :: nil)
                    (fn () => x :: (walk xs))
      in collect (fn () => walk xs) end
```

The semantics of the control operators shift$_i$/reset$_i$ is given by an iterated CPS translation, where the number of iterations is greater than i [8]. So, in order to see what is going on in the above code, let us CPS transform the backtracking primitives and the function `prefixes` using three layers of continuations (η-reducing them where possible to avoid clutter):

```
fun fail () k1 k2
    = k2 ()

fun amb c1 c2 k1 k2
    = c1 () k1 (fn () => c2 () k1 k2)

fun emit v k1 k2
    = k1 ()
         (fn () => fn k3 =>
                     k2 () (fn u => k3 (v :: u)))

fun collect c
    = c ()
        (fn v => emit v (fn u => fn k2 => k2 u))
        (fn () => fn k3  => k3 nil)
        (fn vs => vs)

fun prefixes xs
= let fun walk nil
          = fail ()
        | walk (x :: xs)
          = amb (fn () => fn k1 => k1 (x :: nil))
                (fn () =>
                  fn k1 => walk xs
                               (fn vs => k1 (x :: vs)))
  in collect (fn () => walk xs) end
```

In this model of backtracking the first level serves to generate the current solution, the second level remembers non-deterministic-choice points, and the third level is responsible for storing the generated solutions. Since in the above example there is always at most one choice point, we can write its simpler version, where the second level collects the prefixes and, therefore, the third level is not needed any more [5]:

```
fun prefixes xs
= let fun walk nil k1 k2
          = k2 nil
        | walk (x :: xs) k1 k2
          = k1 (x :: nil)
               (fn vs =>
                 walk xs (fn vs => k1 (x :: vs))
                         (fn vss => k2 (vs :: vss)))
  in walk xs (fn vs => fn k2 => k2 vs)
             (fn vss => vss) end
```

This simplification has an interesting consequence for the types of the continuations used in this program. We can observe that while the answer type of `k1` is `int list`, the argument type of `k2` is `int list list`. We face here a phenomenon known as answer-type modification [2, 5, 7], i.e., a continuation of answer type `int list` is used through a control effect to construct a value of type `int list list`. A direct-style counterpart of this program is the following familiar function [5]:

```
fun prefixes xs
= let fun walk nil
          = shift_1 (fn k => nil)
        | walk (x::xs)
          = shift_1
            (fn k =>
              (k (x :: nil)) ::
              (reset_1 (fn () => k (x :: (walk xs)))))
  in reset_1 (fn () => walk xs) end
```

Due to the answer-type modification, this program type-checks neither in Danvy and Yang's type system [9] nor in Murthy's type system [22]. It requires a type system à la Danvy and Filinski [7, 25], where programs have types derived from their CPS semantics and where computations can modify the answer type of continuations. The rest of this article is devoted to such a type system.

3. Flexible Control Operators

In this section, we present a hierarchy of flexible delimited-control operators and we define a type system for it. We then show that it enjoys the standard correctness properties, such as subject reduction, soundness with respect to the CPS translation, and termination of evaluation. We also discuss the link between these operators and the original hierarchy of control operators due to Danvy and Filinski [8].

3.1 Syntax

The language of terms at an arbitrary level n of the hierarchy extends the usual lambda terms with delimited-control operators *capture* $\mathcal{L}_n$, *reset* $\langle\cdot\rangle_n$, and *throw* $\hookleftarrow_n$, for $n \in \mathcal{N}_+$. At any level n, all operators inherited from lower levels $j < n$ are also available. In the following, we assume we have a set of term variables, ranged over by x, separate from n pairwise disjoint sets of continuation variables, ranged over by $k_1, \ldots, k_n$. The syntax of terms at level n is defined as follows (where $1 \leq i \leq n$):

$$\begin{array}{rcl} t & ::= & x \mid \lambda x.t \mid t\ t \mid \mathcal{L}_i(k_1,\ldots,k_i).t \mid \langle t\rangle_i \mid \\ & & (h_1,\ldots,h_i) \hookleftarrow_i t \\ h_i & ::= & k_i \mid \ulcorner E_i \urcorner \\ v & ::= & \lambda x.t \end{array}$$

and the syntax of (call-by-value) evaluation contexts is given by (where $2 \leq i \leq n+1$):

$$\begin{array}{rcl} E_1 & ::= & \bullet_1 \mid v\ E_1 \mid E_1\ t \mid (\ulcorner E_1 \urcorner, \ldots, \ulcorner E_i \urcorner) \hookleftarrow_i E_1 \\ E_i & ::= & \bullet_i \mid E_i.E_{i-1} \end{array}$$

$\mathcal{L}_i(k_1,\ldots,k_i).t$ are capture operators, each $\langle\cdot\rangle_i$ delimits the scope of the corresponding capture operator, and the throw constructs $(h_1,\ldots,h_i) \hookleftarrow_i t$ (similar to that of SML/NJ [12]) are used for applying a tuple of continuation variables or evaluation contexts to a term (or, *throwing* the term to the tuple).

In the original hierarchy of control operators [8], the shift$_i$ construct $\mathcal{S}_i k.t$ binds only one continuation variable instead of a tuple, and continuations are applied as regular functions, without any throw construct. Independently from this work, a throw construct $k \overset{\mathcal{S}}{\hookleftarrow}_i t$ can be introduced to distinguish continuation applications in the original hierarchy for typing purposes, as discussed in [4] (and in Section 3.3). Translating a term written with operators $\mathcal{S}_i$ and $\overset{\mathcal{S}}{\hookleftarrow}_i$ to fit our system amounts to replacing singular variables k by tuples of continuation variables of size i.

In a source program, a term can be thrown only to a tuple of continuation variables $(k_1,\ldots,k_i)$, and the programmer does not handle evaluation contexts explicitly. However, they can be introduced during evaluation, when some of the variables $k_1,\ldots,k_i$ are replaced by contexts captured by an operator $\mathcal{L}_j$. Therefore, we distinguish *plain terms*, i.e., terms that contain only throw constructs of the form $(k_1,\ldots,k_i) \hookleftarrow_i t$.

An abstraction $\lambda x.t$ binds x in t and a capture construct $\mathcal{L}_i(k_1,\ldots,k_i).t$ binds the variables $k_1,\ldots,k_i$ in t. The sets of free term and continuation variables are defined as usual, and we say a term is *closed* if it does not contain any free variables of any kind. A context is closed if and only if all terms occurring in it are closed. We equate terms up to α-conversion of their bound variables.

Contexts E_i can be seen as terms with a hole. We represent contexts inside-out, i.e., $\bullet_1$ represents the empty context of level 1, $v\ E_1$ represents the "term with a hole" $E_1[v\ [\ ]]$, $E_1\ t$ represents $E_1[[\ ]\ t]$, and $(\ulcorner E_1 \urcorner, \ldots, \ulcorner E_i \urcorner) \hookleftarrow_i E_1'$ represents $E_1'[(\ulcorner E_1 \urcorner, \ldots, \ulcorner E_i \urcorner) \hookleftarrow_i [\ ]]$. A context of level i for $i = 2,\ldots,n+1$ is a stack of contexts of level $i-1$ separated by a delimiter $\langle\cdot\rangle_{i-1}$. Therefore the empty context $\bullet_i$ of level i stands for the term with a hole $\langle[\ ]\rangle_{i-1}$, and $E_i.E_{i-1}$ represents $E_i[\langle E_{i-1}[\ ]\rangle_{i-1}]$. Formally, the function $plug_i$ $(1 \leq i \leq n+1)$ gives the term obtained by putting a term t within a context E_i. We define $plug_1$ as follows:

$$\begin{array}{rcl} plug_1\ (t, \bullet_1) & = & t \\ plug_1\ (t, v\ E_1) & = & plug_1\ (v\ t, E_1) \\ plug_1\ (t_0, E_1\ t_1) & = & plug_1\ (t_0\ t_1, E_1) \\ plug_1\ (t, (\ulcorner E_1 \urcorner, \ldots, \ulcorner E_i \urcorner) \hookleftarrow_i E_1') & = & \\ & & plug_1\ ((\ulcorner E_1 \urcorner, \ldots, \ulcorner E_i \urcorner) \hookleftarrow_i t, E_1') \end{array}$$

and for $i = 2,\ldots,n+1$ we define:

$$\begin{array}{rcl} plug_i\ (t, \bullet_i) & = & t \\ plug_i\ (t, E_i.E_{i-1}) & = & plug_i\ (\langle plug_{i-1}\ (t, E_{i-1})\rangle_{i-1}, E_i) \end{array}$$

We write $E_i[t]$ for the term $plug_i\ (t, E_i)$.

We choose the inside-out representation of contexts (rather than the outside-in representation) because inside-out contexts arise naturally as defunctionalized continuations, i.e., they are first-order counterparts of continuations seen as higher-order functions. Consequently, continuations can be obtained by *refunctionalizing* inside-out contexts, as shown in Figure 1.

In the following, we represent terms as *programs* in order to keep the layers of contexts delimited by the reset operators $\langle\cdot\rangle_i$ explicit. Such a representation is useful when writing reduction rules, where we have to decompose a term and locate its redex. It is also well suited for defining reducibility predicates to prove termination of well-typed terms (cf. Section 3.4).

CPS translation of terms

$$
\begin{array}{rcl}
\overline{x} & = & \lambda k_1 \ldots k_{n+1}.k_1\ x\ k_2 \ldots\ k_{n+1} \\
\overline{\lambda x.t} & = & \lambda k_1 \ldots k_{n+1}.k_1\ (\lambda x k_1' k_2' \ldots k_{n+1}'.\overline{t}\ k_1'\ k_2' \ldots k_{n+1}')\ k_2 \ldots k_{n+1} \\
\overline{t_0\ t_1} & = & \lambda k_1 \ldots k_{n+1}.\overline{t_0}\ (\lambda v_0 k_2' \ldots k_{n+1}'.\overline{t_1}\ (\lambda v_1 k_2'' \ldots k_{n+1}''.v_0\ v_1\ k_1\ k_2'' \ldots k_{n+1}'')\ k_2' \ldots k_{n+1}') \\
 & & \qquad k_2 \ldots k_{n+1} \\
\overline{\langle t \rangle_i} & = & \lambda k_1 \ldots k_{n+1}.\overline{t}\ \theta_1 \ldots \theta_i (\lambda v_0 k_{i+2}' \ldots k_{n+1}'.k_1\ v_0\ k_2 \ldots k_{i+1}\ k_{i+2}' \ldots k_{n+1}')\ k_{i+2} \ldots k_{n+1} \\
\overline{\mathcal{L}_i(k_1', \ldots, k_i').t} & = & \lambda k_1 \ldots k_{n+1}.\overline{t}\{k_1/k_1', \ldots, k_i/k_i'\}\ \theta_1 \ldots \theta_i\ k_{i+1} \ldots k_{n+1} \\
\overline{(h_1, \ldots, h_i) \hookleftarrow_i t} & = & \lambda k_1 \ldots k_{n+1}.\overline{t}\ (\lambda v_0 k_2' \ldots k_{n+1}'.[\![h_1]\!]\ v_0\ [\![h_2]\!] \ldots [\![h_i]\!] (\lambda v_1 k_{i+2}'' \ldots k_{n+1}''.k_1\ v_1\ k_2' \ldots k_{i+1}'\ k_{i+2}'' \ldots k_{n+1}'') \\
 & & \qquad k_{i+2}' \ldots k_{n+1}')\ k_2 \ldots k_{n+1} \\
\text{where} \quad \theta_i & = & \lambda x k_{i+1} \ldots k_{n+1}.k_{i+1}\ x\ k_{i+2} \ldots\ k_{n+1} \quad \text{for } i = 1, \ldots, n
\end{array}
$$

Refunctionalization of contexts

$$
\begin{array}{rcl}
[\![k_i]\!] & = & k_i \\
 & & \\
[\![\bullet_i]\!] & = & \theta_i \\
[\![E_1\ t]\!] & = & \lambda v_0 k_2 \ldots k_{n+1}.\overline{t}\ (\lambda v_1 k_2' \ldots k_{n+1}'.v_0\ v_1\ [\![E_1]\!]\ k_2' \ldots k_{n+1}')\ k_2 \ldots k_{n+1} \\
[\![v_0\ E_1]\!] & = & \lambda v_1 k_2 \ldots k_{n+1}.v_0{}^*\ v_1\ [\![E_1]\!]\ k_2 \ldots k_{n+1} \\
[\![(\ulcorner E_1 \urcorner, \ldots, \ulcorner E_i \urcorner) \hookleftarrow_i E_1']\!] & = & \lambda v_0 k_2 \ldots k_{n+1}.[\![E_1]\!]\ v_0\ [\![E_2]\!] \ldots [\![E_i]\!]\ (\lambda v_1 k_{i+2}' \ldots k_{n+1}'.[\![E_1']\!]\ v_1\ k_2 \ldots k_{i+1}\ k_{i+2}' \ldots k_{n+1}') \\
 & & \qquad k_{i+2} \ldots k_{n+1} \\
[\![E_i.E_{i-1}]\!] & = & \lambda v k_{i+1} \ldots k_{n+1}.[\![E_{i-1}]\!]\ v\ [\![E_i]\!]\ k_{i+1} \ldots k_{n+1} \\
 & & \\
\lambda x.t^* & = & \lambda x.\overline{t}
\end{array}
$$

Figure 1. CPS translation

Formally, a program at level n of the hierarchy is defined as follows:

$$p ::= \langle t, E_1, \ldots, E_{n+1} \rangle.$$

The program $\langle t, E_1, \ldots, E_{n+1} \rangle$ represents the term

$$plug_{n+1}\ (\langle \ldots \langle plug_2\ (\langle plug_1\ (t, E_1) \rangle_1, E_2) \rangle_2 \ldots \rangle_n, E_{n+1}),$$

which can also be written as

$$E_{n+1}[\langle \ldots \langle E_2[\langle E_1[t] \rangle_1] \rangle_2 \ldots \rangle_n].$$

It can be seen from this definition that each term that we represent as a program at level n, will be implicitly enclosed by the reset operators of each level from 1 to n. For example, the term $\lambda x.t$ will be represented as the program $\langle \lambda x.t, \bullet_1, \ldots, \bullet_{n+1} \rangle$ that yields $\langle \ldots \langle \lambda x.t \rangle_1 \ldots \rangle_n$ after plugging. From the operational point of view, such a sequence of delimiters surrounding a term is superfluous, since it is enough to replace them by the reset of the highest level. Therefore we introduce another definition of the *plug* function—one that introduces fewer delimiters—defined on programs, which is shown in Figure 2. The idea is that the new *plug* function when operating on entire programs can detect sequences of empty contexts when it is enough to introduce only one delimiter of the highest level, rather than all of them (this behavior is captured by the last two clauses of *plug*). This definition relies on the fact that the first context E_{i+1} that is not empty has necessarily the form $E_{i+1}.(E_i.(\ldots(E_2.E_1)\ldots))$, as it can only be obtained by pushing a sequence of contexts from level 1 to i onto E_{i+1} (as a result of decomposition or reduction). We will use this definition when reconstructing a term from a program. For technical reasons, also the definitions of $plug_i$ functions will be used in the remainder of the article.

A term (which we consider in its representation as a program) can have many different *decompositions* into the term part and the context part; consequently different programs can represent the same term. For example, for $n = 2$, the program $\langle (\lambda x.t)\ \ v, \bullet_1, E_2.E_1, E_3 \rangle$ can be also decomposed as $\langle \lambda x.t, \bullet_1\ v, E_2.E_1, E_3 \rangle$, or as $\langle \langle (\lambda x.t)\ v \rangle_1, E_1, E_2, E_3 \rangle$. We identify all decompositions of the same term by defining an equivalence relation on programs, as follows:

$$p \sim p' := plug\ (p) = plug\ (p')$$

and considering programs up to this equivalence. Informally, the way to decompose a program is to read the definition of *plug* from right to left. In particular, in order to decompose a term enclosed in a level-i *reset*, we need to push the current i contexts onto the level-$i+1$ context. For example, the term $\langle \langle \langle t \rangle_1\ s \rangle_2, \bullet_1, \bullet_2, \bullet_3 \rangle$ can be decomposed into $\langle \langle t \rangle_1\ s, \bullet_1, \bullet_2, \bullet_3.\ \bullet_2\ .\bullet_1 \rangle$ and further into $\langle \langle t \rangle_1, \bullet_1\ s, \bullet_2, \bullet_3.\bullet_2\ .\bullet_1 \rangle$ and then into $\langle t, \bullet_1, \bullet_2.\bullet_1\ \ s, \bullet_3.\bullet_2.\bullet_1 \rangle$.

3.2 CPS Translation and Semantics

We first define a CPS translation for our language, and then we derive the reduction semantics from it, using the same approach as Biernacka et al. [5]. The CPS translation, presented in Figure 1, extends the standard call-by-value CPS translation for the lambda calculus, and it uses the function $[\![\cdot]\!]$ which transforms contexts into continuations they represent (leaving continuation variables unchanged).

The CPS translation is defined with respect to a fixed (but arbitrary) level n of the hierarchy. It means that the control constructs may interact with up to n surrounding contexts, and consequently, we introduce $n+1$ layers of continuations in the translation. In the case when a CPS-translated term does not touch some of its outer contexts, it results in the introduction of a number of eta redexes that could be reduced away. However, we prefer to keep them in

$$
\begin{array}{rcl}
\mathit{plug}\ \langle t, v\ E_1, E_2, \ldots, E_{n+1}\rangle & = & \mathit{plug}\ \langle v\ t, E_1, E_2, \ldots, E_{n+1}\rangle \\
\mathit{plug}\ \langle t_0, E_1\ t_1, E_2, \ldots, E_{n+1}\rangle & = & \mathit{plug}\ \langle t_0\ t_1, E_1, E_2, \ldots, E_{n+1}\rangle \\
\mathit{plug}\ \langle t, (\ulcorner E_1 \urcorner, \ldots, \ulcorner E_i \urcorner) \hookleftarrow_i E'_1, E_2, \ldots, E_{n+1}\rangle & = & \mathit{plug}\ \langle (\ulcorner E_1 \urcorner, \ldots, \ulcorner E_i \urcorner) \hookleftarrow_i t, E'_1, E_2, \ldots, E_{n+1}\rangle \\
\mathit{plug}\ \langle t, \bullet_1, \ldots, \bullet_i, E_{i+1}.(E_i.(\ldots(E_2.E_1)\ldots)), E_{i+2}, \ldots, E_{n+1}\rangle & = & \mathit{plug}\ \langle \langle t\rangle_i, E_1, \ldots, E_i, E_{i+1}, E_{i+2}, \ldots, E_{n+1}\rangle \\
\mathit{plug}\ \langle t, \bullet_1, \ldots, \bullet_{n+1}\rangle & = & \langle t\rangle_n
\end{array}
$$

Figure 2. Plug function for programs at level n

$$
\begin{array}{lrcl}
(\beta_v) & \langle (\lambda x.t)\ v, E_1, \ldots, E_{n+1}\rangle & \to_{\mathrm{v}} & \langle t\{v/x\}, E_1, , \ldots, E_{n+1}\rangle \\
(\mathit{capture}_i) & \langle \mathcal{L}_i(k_1, \ldots, k_i).t, E_1, \ldots, E_{n+1}\rangle & \to_{\mathrm{v}} & \langle t\{\ulcorner E_1 \urcorner/k_1, \ldots, \ulcorner E_i \urcorner/k_i\}, \bullet_1, \ldots, \bullet_i, E_{i+1}, \ldots, E_{n+1}\rangle \\
(\mathit{reset}_i) & \langle \langle v\rangle_i, E_1, \ldots, E_{n+1}\rangle & \to_{\mathrm{v}} & \langle v, E_1, \ldots, E_{n+1}\rangle \\
(\mathit{throw}_i) & \langle (\ulcorner E'_1 \urcorner, \ldots, \ulcorner E'_i \urcorner) \hookleftarrow_i v, E_1, \ldots, E_{n+1}\rangle & \to_{\mathrm{v}} & \langle v, E'_1, \ldots, E'_i, E_{i+1}.(E_i.\ldots.(E_2.E_1)\ldots), E_{i+2}, \ldots, E_{n+1}\rangle \\
 & & & \text{for } 1 \le i \le n
\end{array}
$$

Figure 3. Reduction rules

order to be uniform, and to exhibit the relationship with the type system of Section 3.3.

The call-by-value reduction semantics is shown in Figure 3. We write $t\{s/x\}$ for the usual capture-avoiding substitution of s for the variable x in t, and we write $t\{E_1/k_1\}\ldots\{E_i/k_i\}$ for the simultaneous capture-avoiding substitutions of contexts $E_1, \ldots, E_i$ for variables $k_1, \ldots, k_i$ in t. Terms $(\lambda x.t)\ v$ are the standard β-redexes of the call-by-value λ-calculus (the rule (β_v)). The reduction of a term $\mathcal{L}_i(k_1, \ldots, k_i).t$ within contexts $E_1, \ldots, E_{n+1}$ consists in capturing the first i contexts $E_1, \ldots, E_i$ and substituting them for the variables $k_1, \ldots, k_i$ (the rule $(\mathit{capture}_i)$). When a value is thrown to a tuple of contexts $(E'_1, \ldots, E'_i)$ using a level-i throw construct within contexts $E_1, \ldots, E_{n+1}$, then the new contexts $E'_1, \ldots, E'_i$ are reinstated as the current contexts, and the then-current contexts $E_1, \ldots, E_i$ are stacked on the context E_{i+1} (the rule (throw_i)). Finally, if a value is surrounded by a reset of any level, then the delimiter is no longer needed and can be removed using the rule (reset_i).

A *redex* is the first component (a term) of the program occurring on the left-hand side of each of the reduction rules above. A *potential redex* is either a proper redex or a stuck term, i.e., a term that neither is a value nor can be further reduced. The type system we propose in Section 3.3 ensures that a well-typed program cannot generate a stuck term in the course of its reduction. Because of the unique-decomposition property of the calculus, the relation $\to_{\mathrm{v}}$ is deterministic.

We define the evaluation relation as the reflexive and transitive closure of $\to_{\mathrm{v}}$. The result of the evaluation is a program value of the form $p_v := \langle v, \bullet_1, \ldots, \bullet_{n+1}\rangle$. Hence, a program value is a term value surrounded by the (implicit) reset operators $\langle\cdot\rangle_i$ for $i = 1, \ldots, n$.

PROPOSITION 1 (Unique-decomposition property). *For all programs p, p either is a program value, or it decomposes uniquely into contexts $E_1, \ldots, E_{n+1}$ and a potential redex r such that*

$$p = \langle r, E_1, \ldots, E_{n+1}\rangle.$$

Note that the semantics of the original shift$_i$ can be retrieved by allowing only throws to tuples $E_1, \ldots, E_i$ captured by an operator $\mathcal{L}_i(k_1, \ldots, k_i).t$, i.e., by forbidding throws to tuples built from different captures or to tuples of size i built from a capture of size $j > i$.

Let us illustrate the evaluation with an example. Consider the program $\langle (\mathcal{L}_2(k_1, k_2).(k_1, k_2) \hookleftarrow_2 \langle v\rangle_1)\ (\lambda x.x), \bullet_1, \bullet_2, \bullet_3\rangle$. First, we decompose it and locate the redex:

$$\langle \mathcal{L}_2(k_1, k_2).(k_1, k_2) \hookleftarrow_2 \langle v\rangle_1, \bullet_1\ (\lambda x.x), \bullet_2, \bullet_3\rangle,$$

then we reduce it according to rule $(\mathit{capture}_i)$ and obtain:

$$\langle (\ulcorner \bullet_1\ (\lambda x.x) \urcorner, \ulcorner \bullet_2 \urcorner) \hookleftarrow_2 \langle v\rangle_1, \bullet_1, \bullet_2, \bullet_3\rangle.$$

Again, we decompose and find the next redex:

$$\langle \langle v\rangle_1, (\ulcorner \bullet_1\ (\lambda x.x) \urcorner, \ulcorner \bullet_2 \urcorner) \hookleftarrow_2 \bullet_1, \bullet_2, \bullet_3\rangle.$$

According to rule (reset_i) we reduce it, and decompose again:

$$\langle (\ulcorner \bullet_1\ (\lambda x.x) \urcorner, \ulcorner \bullet_2 \urcorner) \hookleftarrow_2 v, \bullet_1, \bullet_2, \bullet_3\rangle.$$

Finally, we reduce it to

$$\langle v, \bullet_1\ (\lambda x.x), \bullet_2, \bullet_3.\ \bullet_2\ .\bullet_1\rangle$$

according to rule (throw_i), and then we decompose it to find the (β_v)-redex which reduces to:

$$\langle v, \bullet_1, \bullet_2, \bullet_3.\ \bullet_2\ .\bullet_1\rangle.$$

Finally, we decompose it to

$$\langle \langle v\rangle_2, \bullet_1, \bullet_2, \bullet_3\rangle$$

and apply (reset_i) again, which yields the result

$$\langle v, \bullet_1, \bullet_2, \bullet_3\rangle.$$

We can show that reductions in the hierarchy are sound with respect to the CPS translation:

PROPOSITION 2 (Soundness of reduction wrt. CPS). *If $p \to_{\mathrm{v}} p'$, then $\overline{p} =_{\beta\eta} \overline{p'}$.*

Proposition 2 is proved using a characterization of the CPS-image of a program in terms of its CPS-translated components:

PROPOSITION 3 (Characterization of CPS image).
If $p = \langle t, E_1, \ldots, E_n\rangle$, then

$$\overline{p} =_{\beta\eta} \lambda k_1 \ldots k_{n+1}.\bar{t}\ [\![E_1]\!] \ldots [\![E_n]\!](\lambda v_0.k_1\ v_0\ k_2 \ldots k_{n+1})\ .$$

Proposition 3 states that the CPS translation of a program is convertible to the CPS term obtained first, by CPS translating the term component t of the program, then applying it to continuations obtained by refunctionalizing the successive contexts E_i, and finally applying it to a continuation of the highest level that collects all the current continuations k_i (as a refunctionalized stack of lower-level contexts/continuations).

Terms ($1 \leq i \leq n$)**:**

$$\frac{}{\Gamma,\, x\,:\,S;\Delta\ \vdash_n\ x\ :\ S \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{n+1}}$$

$$\frac{\Gamma,\, x\,:\,S;\Delta\ \vdash_n\ t\ :\ C'_1, \ldots, C'_{n+1} \qquad C_1 = T \rhd \ldots}{\Gamma;\Delta\ \vdash_n\ \lambda x.t\ :\ (S \to [C'_1, \ldots, C'_{n+1}]) \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{n+1}}$$

$$\frac{\Gamma;\Delta\ \vdash_n\ t_0\ :\ (S \to [C_1, \ldots, C_{n+1}]) \rhd C''_2 \ldots \rhd C''_{n+1}, C'_2, \ldots, C'_{n+1} \qquad \Gamma;\Delta\ \vdash_n\ t_1\ :\ S \rhd C_2 \ldots \rhd C_{n+1}, C''_2, \ldots, C''_{n+1}}{\Gamma;\Delta\ \vdash_n\ t_0\, t_1\ :\ C_1, C'_2, \ldots, C'_{n+1}}$$

$$\frac{\mathcal{I}_1(D_1) \qquad \ldots \qquad \mathcal{I}_i(D_i) \qquad \Gamma;\Delta\ \vdash_n\ t\ :\ D_1, \ldots, D_i, (S \rhd C_{i+2} \ldots \rhd C_{n+1}), C'_{i+2}, \ldots, C'_{n+1}}{\Gamma;\Delta\ \vdash_n\ \langle t \rangle_i\ :\ S \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{i+1}, C'_{i+2}, \ldots, C'_{n+1}}$$

$$\frac{\mathcal{I}_1(D_1) \qquad \ldots \qquad \mathcal{I}_i(D_i) \qquad \Gamma;\Delta,\, k_1\,:\,C_1, \ldots,\, k_i\,:\,C_i\ \vdash_n\ t\ :\ D_1, \ldots, D_i, C_{i+1}, \ldots, C_{n+1}}{\Gamma;\Delta\ \vdash_n\ \mathcal{L}_i(k_1, \ldots, k_i).t\ :\ C_1, C_2, \ldots, C_{n+1}}$$

$$\frac{\begin{array}{c} C_1 = S \rhd C_2 \ldots \rhd C_{n+1} \qquad C_{i+1} = T \rhd C'_{i+2} \ldots \rhd C'_{n+1} \\ \Gamma;\Delta\ \vdash_n\ h_1\ :\ C_1 \qquad \ldots \qquad \Gamma;\Delta\ \vdash_n\ h_i\ :\ C_i \\ \Gamma;\Delta\ \vdash_n\ t\ :\ S \rhd C''_2 \rhd \ldots C''_{i+1} \rhd C_{i+2} \rhd \ldots C_{n+1}, D_2, \ldots, D_{n+1} \end{array}}{\Gamma;\Delta\ \vdash_n\ (h_1, \ldots, h_i) \hookleftarrow_i t\ :\ T \rhd C''_2 \rhd \ldots C''_{i+1} \rhd C'_{i+2} \rhd \ldots C'_{n+1}, D_2, \ldots, D_{n+1}}$$

Contexts and continuation variables:

$$\frac{\mathcal{I}_i(C_i) \qquad 1 \leq i \leq n+1}{\Gamma;\Delta\ \vdash_n\ \bullet_i\ :\ C_i} \qquad\qquad \frac{}{\Gamma;\Delta,\, k_i\,:\,C_i\ \vdash_n\ k_i\ :\ C_i}$$

$$\frac{\Gamma;\Delta\ \vdash_n\ E_i\ :\ C_i \qquad \Gamma;\Delta\ \vdash_n\ E_{i-1}\ :\ S \rhd C_i \ldots \rhd C_{n+1} \qquad 2 \leq i \leq n+1}{\Gamma;\Delta\ \vdash_n\ E_i.E_{i-1}\ :\ S \rhd C_{i+1} \ldots \rhd C_{n+1}}$$

$$\frac{\Gamma;\Delta\ \vdash_n\ E_1\ :\ C_1 \qquad \Gamma;\Delta\ \vdash_n\ t_1\ :\ S \rhd C_2 \ldots \rhd C_{n+1}, C''_2, \ldots, C''_{n+1}}{\Gamma;\Delta\ \vdash_n\ E_1\ t\ :\ (S \to [C_1, \ldots, C_{n+1}]) \rhd C''_2 \ldots \rhd C''_{n+1}}$$

$$\frac{\Gamma;\Delta\ \vdash_n\ v\ :\ (S \to [C_1, \ldots, C_{n+1}]) \rhd C''_2 \ldots \rhd C''_{n+1}, C''_2, \ldots, C''_{n+1} \qquad \Gamma;\Delta\ \vdash_n\ E_1\ :\ C_1}{\Gamma;\Delta\ \vdash_n\ v\ E_1\ :\ S \rhd C_2 \ldots \rhd C_{n+1}}$$

$$\frac{\begin{array}{c} C_1 = S \rhd C_2 \ldots \rhd C_{n+1} \qquad C_{i+1} = T \rhd C'_{i+2} \ldots \rhd C'_{n+1} \\ \Gamma;\Delta\ \vdash_n\ E_1\ :\ C_1 \qquad \ldots \qquad \Gamma;\Delta\ \vdash_n\ E_i\ :\ C_i \\ \Gamma;\Delta\ \vdash_n\ E_1{}'\ :\ T \rhd C''_2 \rhd \ldots C''_{i+1} \rhd C'_{i+2} \rhd \ldots C'_{n+1} \end{array}}{\Gamma;\Delta\ \vdash_n\ (\ulcorner E_1 \urcorner, \ldots, \ulcorner E_i \urcorner) \hookleftarrow_i E_1{}'\ :\ S \rhd C''_2 \rhd \ldots C''_{i+1} \rhd C_{i+2} \rhd \ldots C_{n+1}}$$

Programs:

$$\frac{\Gamma;\Delta\ \vdash_n\ E_{n+1}[\langle \ldots E_1[t] \ldots \rangle_n]\ :\ S \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{n+1}}{\Gamma;\Delta\ \vdash_n\ \langle t, E_1, \ldots, E_{n+1} \rangle\ :\ S}$$

Figure 4. A type system for the level n of the CPS hierarchy

3.3 Type System

We propose a type system for the CPS hierarchy. It is a conservative extension of the type system given by Biernacka and Biernacki for the first level of the hierarchy for shift and reset [4], which is itself a refinement of the classical type system of Danvy and Filinski [7]. The typing rules have been derived from the CPS image of the language (shown in Figure 1). We assume that we have a set of base type variables, ranged over by b. We let S, T range over types for terms, and C_i, D_i range over types for contexts E_i, for all $i = 1, \ldots, n+1$. The syntax of types for terms and contexts is given below, where $1 \leq i \leq n$:

$$\begin{array}{rcl} S & ::= & b \mid S \to [C_1, \ldots, C_{n+1}] \\ C_{n+1} & ::= & \neg S \\ C_i & ::= & S \rhd C_{i+1} \ldots \rhd C_{n+1} \end{array}$$

As in Danvy and Filinski's system, arrow types contain type annotations $S \to [C_1, \ldots, C_{n+1}]$: a function with such a type can be applied to an argument of type S within contexts of types

$C_1, \ldots, C_{n+1}$. For $n = 1$, the Danvy and Filinski's type $S \ {}_U \!\!\rightarrow_V T$ corresponds to $S \rightarrow [T \rhd U, \neg V]$.

In Danvy and Filinski's original type system, continuations are treated as regular functions; they are applied without any throw operator, and they are typed with regular (annotated) arrow types. As discussed in [4], this approach is too restrictive to type some interesting examples. In particular, it lacks context answer type polymorphism, which can be retrieved by representing captured continuations as contexts, and by using an explicit throw construct. Following this idea, we assign types to contexts C_i that are not function types; a context of type $S \rhd C_{i+1} \ldots \rhd C_{n+1}$ can be plugged with a term of type S, and it can be put within contexts of types $C_{i+1}, \ldots, C_{n+1}$, respectively.

A type environment for term variables Γ is a list of pairs $x : S$, and a type environment for continuation variables Δ is a list of pairs $k_i : C_i$. We derive typing judgments of the form $\Gamma; \Delta \ \vdash_n \ t \ : \ C_1, \ldots, C_{n+1}$ for terms, and typing judgments of the form $\Gamma; \Delta \ \vdash_n \ E_i \ : \ C_i$ for contexts. If $\Gamma; \Delta \ \vdash_n \ t \ : \ C_1, \ldots, C_{n+1}$, then, under the assumptions Γ, Δ, the term t can be plugged into contexts of types $C_1, \ldots, C_{n+1}$. We do not need to mention explicitly the type of the term in the judgment, because we can retrieve it from the type C_1 of the first enclosing context, if needed. If $C_1 = S \rhd C'_2 \ldots \rhd C'_{n+1}$, then t is of type S.

Danvy and Filinski's typing judgment $\Gamma; \Delta \mid T \ \vdash \ t \ : \ S \mid U$ corresponds to the judgment $\Gamma; \Delta \vdash_1 t \ : \ S \rhd T, \neg U$ in our system.

The typing rules are presented in Figure 4. We now briefly explain them and sketch how they were derived from the CPS translation of the language (of Figure 1).

The CPS defining equations are usually of the form $\overline{op(t)} = \lambda k_1 \ldots k_{n+1}.\bar{t}\, k'_1 \ldots k'_{n+1}$ for a given operator *op*, and we want to generate a typing rule of the form

$$\frac{\Gamma'; \Delta' \ \vdash_n \ t \ : \ C'_1, \ldots, C'_{n+1}}{\Gamma; \Delta \ \vdash_n \ op(t) \ : \ C_1, \ldots, C_{n+1}}.$$

To this end, we annotate the CPS equation with the most liberal types, and we then deduce the types $C_1, \ldots, C_{n+1}$ from the types of $k_1, \ldots, k_{n+1}$, and $C'_1, \ldots, C'_{n+1}$ from the types of $k'_1, \ldots, k'_{n+1}$. For example, consider the CPS translation for term variables:

$$\overline{x} = \lambda k_1 \ldots k_{n+1}.k_1 \ x \ k_2 \ldots \ k_{n+1}$$

The type of k_1 has to match those of x, $k_2, \ldots, k_{n+1}$ to make the application typable. No other constraints on types can be deduced for this equation, so we can derive the following typing rule for term variables:

$$\overline{\Gamma, \ x \ : \ S; \Delta \ \vdash_n \ x \ : \ S \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{n+1}}$$

Consider now the CPS translation for reset:

$$\overline{\langle t \rangle_i} = \lambda k_1 \ldots k_{n+1}.\bar{t}\, \theta_1 \ldots \theta_i \ k'_{i+1} \ k_{i+2} \ldots k_{n+1}$$

with

$$k'_{i+1} = (\lambda v_0 k'_{i+2} \ldots k'_{n+1}.k_1 \ v_0 \ k_2 \ldots \ k_{i+1} \ k'_{i+2} \ldots k'_{n+1})$$

Assume that we type k_1 with $S \rhd C_2 \ldots \rhd C_{n+1}$. To be able to type the continuation k'_{i+1} we have to assign types $C_2, \ldots, C_{n+1}$ to continuations $k_2, \ldots, k_{i+1}$, $k'_{i+2}, \ldots, k'_{n+1}$ and the type S to v_0. Consequently, the continuation k'_{i+1} has type $S \rhd C_{i+2} \ldots \rhd C_{n+1}$. We do not have any constraints on the types of the continuation $k_{i+2}, \ldots, k_{n+1}$. Finally, we have to assign valid types $D_1, \ldots, D_i$ to the initial continuations $\theta_1, \ldots, \theta_i$. Let us recall the definition of the initial continuation:

$$\theta_j = \lambda x k_{j+1} \ldots k_{n+1}.k_{j+1} \ x \ k_{j+2} \ldots \ k_{n+1}.$$

Therefore, a type $D_j = T \rhd D'_{j+1} \ldots \rhd D'_{n+1}$ is valid for θ_j iff $D'_{j+1} = T \rhd D'_{j+2} \ldots \rhd D'_{n+1}$. We check this condition by defining a family of predicates $\mathcal{I}_j$ ($1 \leq j \leq n$) on context types as follows:

$$\mathcal{I}_j(C_j) := \exists S, C_{j+2}, \ldots, C_{n+1}. \\ C_j = S \rhd (S \rhd C_{j+2} \ldots \rhd C_{n+1}) \rhd C_{j+2} \rhd \ldots C_{n+1}$$

and $\mathcal{I}_{n+1}(C_{n+1}) = True$. We now have enough information to write the typing rule for $\langle \cdot \rangle_i$. Similarly, we derive typing rules for the remaining term constructors.

The typing rules for contexts can be derived by inspecting the equations defining the function $[\![\cdot]\!]$. For example, because $[\![\cdot]\!]$ translates $\bullet_i$ into the initial continuation θ_i, the empty context of level i can be typed with any type C_i provided that $\mathcal{I}_i(C_i)$ holds. Note that we use the same typing judgment for continuation variables $\Gamma; \Delta \ \vdash_n \ k_i \ : \ C_i$ as for contexts; it is just to make the typing rule for throw easier to write.

We point out that the typing rule for the original shift$_i$ $\mathcal{S}_i$, (and for reset$_i$ and throw $\overset{\mathcal{S}}{\hookleftarrow}_i$, respectively) is the same as the rule for $\mathcal{L}_i$ (and for reset$_i$, throw $\hookleftarrow_i$, respectively).

Asai and Kameyama [2] defined a notion of pure term (i.e., a term free from control effects) in the polymorphic type system they designed for the level-1 shift and reset. Using Danvy and Filinski's typing judgment, a typable term t is pure iff we can derive $\Gamma; \Delta \mid T \ \vdash \ t \ : \ S \mid T$ for any type T. We can generalize this notion to an arbitrary level n of the CPS hierarchy. We see that for $i = n$ the typing rule for reset becomes

$$\frac{\mathcal{I}_1(D_1) \quad \ldots \quad \mathcal{I}_n(D_n) \qquad \Gamma; \Delta \ \vdash_n \ t \ : \ D_1, \ldots, D_n, \neg S}{\Gamma; \Delta \ \vdash_n \ \langle t \rangle_i \ : \ S \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{n+1}}.$$

We notice that in the conclusion of this rule as well as in the conclusion of the rules for term variable and lambda abstraction, the types $C_2, \ldots, C_{n+1}$ are arbitrary, and the type of the first enclosing context is of the form $S \rhd C_2 \ldots \rhd C_{n+1}$. Therefore, we say that a typable term t is pure iff we can derive $\Gamma; \Delta \ \vdash_n \ t \ : \ S \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{n+1}$ for any $C_2, \ldots, C_{n+1}$.

We now state the main properties of the type system. First, we prove subject reduction: if a program p is typable and reduces to p', then p' is typable with the same type. To this end, we will need to derive types for t and $E_1, \ldots, E_{n+1}$ from the typing judgment $\Gamma; \Delta \ \vdash_n \ \langle t, E_1, \ldots, E_{n+1} \rangle \ : \ S$. We will need a few lemmas:

LEMMA 1. *If* $\Gamma; \Delta \ \vdash_n \ E_1[t] \ : \ D_1, C_2, , \ldots, C_{n+1}$ *and* $\mathcal{I}_1(D_1)$ *hold, then* $\Gamma; \Delta \ \vdash_n \ E_1[\langle t \rangle_1] \ : \ C_1, C_2, \ldots, C_{n+1}$ *and* $\Gamma; \Delta \ \vdash_n \ E_1 \ : \ C_1$ *hold for some* C_1.

The proof is a straightforward structural induction on E_1.

LEMMA 2. *For* $i = 2, \ldots, n$, *the following properties hold:*

1. *If* $\Gamma; \Delta \ \vdash_n \ E_i[\langle t \rangle_{i-1}] \ : \ D_1, \ldots, D_i, C_{i+1}, \ldots, C_{n+1}$ *and* $\mathcal{I}_j(D_j)$ *for all* $j = 1, \ldots, j$, *then there exists* C_i *such that* $\Gamma; \Delta \ \vdash_n \ E_i \ : \ C_i$ *and*

$$\Gamma; \Delta \ \vdash_n \ t \ : \ D'_1, \ldots, D'_{i-1}, C_i, C_{i+1}, \ldots, C_{n+1}$$

with $\mathcal{I}_l(D'_l)$ *for all* $l = 1, \ldots, i-1$.

2. *If* $\Gamma; \Delta \ \vdash_n \ E_i[\langle t \rangle_j] \ : \ D_1, \ldots, D_i, C_{i+1}, \ldots, C_{n+1}$, $\mathcal{I}_l(D_l)$ *holds for all* $l = 1, \ldots, i$, *and* $j \geq i$, *then*

$$\Gamma; \Delta \ \vdash_n \ t \ : \ D'_1, \ldots, D'_j, S \rhd C'_{j+2} \ldots \rhd C'_{n+1}, C_{j+2}, \ldots, C_{n+1}$$

with $\mathcal{I}_l(D'_l)$ *for all* $l = 1, \ldots, j$ *and*

$$\Gamma; \Delta \ \vdash_n \ E_i \ : \ S \rhd C_{i+1} \rhd \ldots \rhd C_{j+1} \rhd C'_{j+2} \rhd \ldots \rhd C'_{n+1}.$$

The main difficulty in proving subject reduction was to write down and prove Lemma 2 (and its counterpart for reconstruction, Lemma 6). The two properties stated in the lemma are proved simultaneously by induction on i and on E_i.

LEMMA 3. *If*

$$\Gamma;\Delta \vdash_n E_{n+1}[\langle t\rangle_n] : S \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{n+1}$$

and $E_{n+1} = \bullet_{n+1}.E_n^1.\ldots.E_n^m$, *then*

$$\Gamma;\Delta \vdash_n t : D_1, \ldots, D_n, \neg T$$

with $\mathcal{I}_l(D_l)$ *for all* $l = 1, \ldots, n$, $\Gamma;\Delta \vdash_n E_n^1 : T' \rhd \neg S$, *and* $\Gamma;\Delta \vdash_n E_{n+1} : \neg T$.

The proof is a straightforward structural induction on E_{n+1}. With these three lemmas, we can decompose a typed program as follows:

LEMMA 4. *If* $\Gamma;\Delta \vdash_n \langle t, E_1, \ldots, E_{n+1}\rangle : S$ *and* $E_{n+1} = \bullet_{n+1}.E_n^1.\ldots.E_n^m$, *then there exist* $C_1, \ldots, C_{n+1}$ *such that* $\Gamma;\Delta \vdash_n E_i : C_i$ *for all* $i = 1, \ldots, n+1$, *and* $\Gamma;\Delta \vdash_n t : C_1, \ldots, C_{n+1}$. *Furthermore,* $\Gamma;\Delta \vdash_n E_n^1 : T' \rhd \neg S$ *is derivable for some* T'.

We now state auxiliary lemmas needed to perform the reverse operation: from a typed term t and typed contexts $E_1, \ldots, E_{n+1}$, we want to deduce the type of the program $\langle t, E_1, \ldots, E_{n+1}\rangle$.

LEMMA 5. *If* $\Gamma;\Delta \vdash_n t : C_1, \ldots, C_{n+1}$ *and* $\Gamma;\Delta \vdash_n E_1 : C_1$, *then* $\Gamma;\Delta \vdash_n E_1[t] : D_1, C_2, \ldots, C_{n+1}$ *is derivable and* $\mathcal{I}_1(D_1)$ *holds.*

LEMMA 6. *The following properties hold:*

1. *If*

$$\Gamma;\Delta \vdash_n t : D_1, \ldots, D_{i-1}, C_i, \ldots, C_{n+1},$$

$\mathcal{I}_l(D_l)$ *hold for all* $l = 1, \ldots, i-1$, *and* $\Gamma;\Delta \vdash_n E_i : C_i$, *then*

$$\Gamma;\Delta \vdash_n E_i[\langle t\rangle_{i-1}] : D_1', \ldots, D_i', C_{i+1}, \ldots, C_{n+1}$$

and $\mathcal{I}_l(D_l')$ *hold for all* $l = 1, \ldots, i$.

2. *If*

$$\Gamma;\Delta \vdash_n t : D_1, \ldots, D_j, C_{j+1}, \ldots, C_{n+1},$$

$\mathcal{I}_l(D_l)$ *hold for all* $l = 1, \ldots, j$, $C_{j+1} = S \rhd C_{j+2}' \ldots \rhd C_{n+1}'$, *and*

$$\Gamma;\Delta \vdash_n E_i : S \rhd C_{i+1} \rhd \ldots \rhd C_{j+1} \rhd C_{j+2}' \rhd \ldots \rhd C_{n+1}',$$

then

$$\Gamma;\Delta \vdash_n E_i[\langle t\rangle_j] : D_1', \ldots, D_i', C_{i+1}, \ldots, C_{n+1}$$

and $\mathcal{I}_l(D_l')$ *hold for all* $l = 1, \ldots, i$.

LEMMA 7. *If* $\Gamma;\Delta \vdash_n t : D_1, \ldots, D_n, \neg T$, $\mathcal{I}_l(D_l)$ *hold for all* $l = 1, \ldots, n$, $E_{n+1} = \bullet_{n+1}.E_n^1.,\ldots,E_n^m$, $\Gamma;\Delta \vdash_n E_{n+1} : \neg T$, *and* $\Gamma;\Delta \vdash_n E_n^1 : T' \rhd \neg S$, *then*

$$\Gamma;\Delta \vdash_n E_{n+1}[\langle t\rangle_n] : S \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{n+1}.$$

LEMMA 8. *If* $\Gamma;\Delta \vdash_n t : C_1, \ldots, C_{n+1}$, $\Gamma;\Delta \vdash_n E_i : C_i$ *for all* $i = 1, \ldots, n+1$, $E_{n+1} = \bullet_{n+1}.E_n^1.,\ldots,E_n^m$, *and* $\Gamma;\Delta \vdash_n E_n^1 : T' \rhd \neg S$, *then* $\Gamma;\Delta \vdash_n \langle t, E_1, \ldots, E_{n+1}\rangle : S$ *is derivable.*

As usual, we need a substitution lemma to deal with the (β_v) and $(capture_i)$ reduction rules.

LEMMA 9 (Substitution lemma). *The following hold:*

1. *If* $\Gamma, x : S;\Delta \vdash_n t : C_1, \ldots, C_{n+1}$ *and* $\Gamma;\Delta \vdash_n v : S \rhd C_2' \ldots \rhd C_{n+1}'$, *then* $\Gamma;\Delta \vdash_n t\{v/x\} : C_1, \ldots, C_{n+1}$.
2. *If* $\Gamma;\Delta, K : D_1 \rhd \ldots \rhd D_i \vdash_n t : C_1, \ldots, C_{n+1}$ *and* $\Gamma;\Delta \vdash_n E_j : D_j$ *for all* $j \in 1, \ldots, i$, *then* $\Gamma;\Delta \vdash_n t\{(E_1, \ldots, E_i)/K\} : C_1, \ldots, C_{n+1}$.

Using these lemmas, we can prove subject reduction.

THEOREM 1 (Subject reduction). *If* $\Gamma;\Delta \vdash_n p : S$ *and* $p \to_{\mathrm{v}} p'$ *then* $\Gamma;\Delta \vdash_n p' : S$.

We now state the correctness of the type system with respect to the CPS translation. To this end, we first introduce a translation of the types of terms and contexts into simple types as follows:

$$\begin{array}{rcl} \overline{b} & = & b \\ \overline{S \to [C_1 \ldots C_{n+1}]} & = & \overline{S} \to \overline{C_1} \to \ldots \to \overline{C_{n+1}} \to o \\ \\ \overline{C_i} & = & \overline{S \rhd C_{i+1} \ldots \rhd C_{n+1}} \\ & = & \overline{S} \to \overline{C_{i+1}} \to \ldots \to \overline{C_{n+1}} \to o \\ \overline{C_{n+1}} & = & \overline{\neg S} = \overline{S} \to o \end{array}$$

where o is an abstract answer type.

We also define a translation on typing contexts in the usual way, i.e., $\overline{\Gamma}$ (resp., $\overline{\Delta}$) is obtained from Γ (resp., Δ) by translating all types occurring in Γ (resp., Δ).

PROPOSITION 4 (Soundness of typing wrt. CPS). *The following implications ensure the soundness of the typing of the hierarchy with respect to the CPS translation, where* $\vdash$ *denotes the standard typing judgments deriving simple types for pure lambda terms:*

1. *If* $\Gamma;\Delta \vdash_n t : C_1, \ldots, C_{n+1}$, *then* $\overline{\Gamma};\overline{\Delta} \vdash \overline{t} : \overline{C_1} \to \ldots \to \overline{C_{n+1}} \to o$.
2. *If* $\Gamma;\Delta \vdash_n E_i : C_i$, *then* $\overline{\Gamma};\overline{\Delta} \vdash [\![E_i]\!] : \overline{C_i}$, *for all* $i = 1, \ldots, n$.

3.4 Termination of Evaluation

We prove termination for call-by-value evaluation, extending the method used by Biernacka and Biernacki [4] for level-1 shift and reset to the level n of the hierarchy. The proof technique is a context-based variant of Tait's reducibility predicates [26]. For simplicity, we restrict ourselves to closed terms, but the result can be extended to open terms.

We define mutually inductive families of predicates on terms and contexts as shown in Figure 5. The predicate $\mathcal{R}_S$, indexed by term types, is defined on values, and the predicates $\mathcal{K}_{C_i}$, indexed by context types, are defined on evaluation contexts for all $i = 1, \ldots, n+1$. A value of a function type is reducible iff the program obtained by applying this value to a reducible value and put within reducible contexts normalizes (i.e., it evaluates to a program value). In turn, a context E_i of level i is reducible iff the program $\langle v, \bullet_1, \ldots, \bullet_{i-1}, E_i, E_{i+1}, \ldots, E_{n+1}\rangle$ built from any reducible value v and any reducible contexts $E_{i+1}, \ldots, E_{n+1}$ of the appropriate types normalizes. The predicate $\mathcal{N}$ is defined on closed programs: $\mathcal{N}(p)$ holds iff p evaluates to a program value in the call-by-value strategy (the strategy is enforced by the grammar of contexts E_1).

In the following, for any closed value v we write $\vdash v : S$ iff there exist $C_2, \ldots, C_{n+1}$ such that $\cdot;\cdot \vdash_n v : S \rhd C_2 \ldots \rhd C_{n+1}, C_2, \ldots, C_{n+1}$. Because v is pure, we do not care about the specific $C_2, \ldots, C_{n+1}$, as discussed in Section 3.3.

In order to prove termination, we need the following two lemmas.

LEMMA 10. *If* $\mathcal{I}_i(C_i)$, *then* $\mathcal{K}_{C_i}(\bullet_i)$.

LEMMA 11. *Let* t *be a plain term such that* $\Gamma;\Delta \vdash_n t : C_1, \ldots, C_{n+1}$, *where* $\Gamma = x_1{:}T_1, \ldots, x_n : T_n$ *and* $\Delta = k_{i_1}^1{:}D_{i_1}^1, \ldots, k_{i_m}^m : D_1^m$. *Let* $\vec{v}$ *be closed values such that* $\vdash v_i : T_i$ *and* $\mathcal{R}_{T_i}(v_i)$ *for all* $i = 1, \ldots, n$. *Let* $\vec{E_i}$ *be closed contexts such that* $\cdot;\cdot \vdash_n E_{i_j}^j : D_{i_j}^j$ *and* $\mathcal{K}_{D_{i_j}^j}(E_{i_j}^j)$. *Let* $E_1', \ldots, E_{n+1}'$

$$
\begin{array}{rcl}
\mathcal{R}_b(v) & := & True \\
\mathcal{R}_{S\to[C_1,\ldots,C_{n+1}]}(v_0) & := & \forall v_1.\,\mathcal{R}_S(v_1) \to \forall E_1.\,\mathcal{K}_{C_1}(E_1) \to \ldots \to \forall E_{n+1}.\,\mathcal{K}_{C_{n+1}}(E_{n+1}) \to \mathcal{N}(\langle v_0\ v_1, E_1, \ldots, E_{n+1}\rangle) \\
\\
\mathcal{K}_{\neg S}(E_{n+1}) & := & \forall v.\,\mathcal{R}_S(v) \to \mathcal{N}(\langle v, \bullet_1, \ldots, \bullet_n, E_{n+1}\rangle) \\
\mathcal{K}_{S\triangleright C_{i+1}\ldots\triangleright C_{n+1}}(E_i) & := & \forall v.\,\mathcal{R}_S(v) \to \forall E_{i+1}.\,\mathcal{K}_{C_{i+1}}(E_{i+1}) \to \ldots \to \forall E_{n+1}.\,\mathcal{K}_{C_{n+1}}(E_{n+1}) \\
& & \qquad\qquad \to \mathcal{N}(\langle v, \bullet_1, \ldots, \bullet_{i-1}, E_i, E_{i+1}, \ldots, E_{n+1}\rangle) \\
\mathcal{N}(p) & := & \exists v.\,p \to_{\mathrm{v}}^* \langle v, \bullet_1, \ldots, \bullet_{n+1}\rangle
\end{array}
$$

Figure 5. Reducibility predicates

be closed contexts such that $\cdot;\cdot \vdash_n E'_i : C_i$ *and* $\mathcal{K}_{C_i}(E'_i)$*. Then* $\mathcal{N}(\langle t\{\vec{v}/\vec{x}, \vec{E_i}/\vec{k_i}\}, E'_1, \ldots, E'_{n+1}\rangle)$ *holds.*

The proof of Lemma 11 is similar to the one of the analogous lemma in [4]; this lemma is used to prove the following result:

THEOREM 2 (Termination of evaluation). *Let* t *be a closed plain term such that* $\cdot;\cdot \vdash_n t : C_1, \ldots, C_{n+1}$*, and* $\mathcal{I}_i(C_i)$ *hold for all* $i = 1, \ldots, n+1$*. Then* $\mathcal{N}(\langle t, \bullet_1, \ldots, \bullet_{n+1}\rangle)$ *holds.*

Theorem 2 is stated for plain terms only, since it is only for such terms that we are able to control the reducibility property of captured contexts occurring in them (here, it can only happen by substituting a reducible context for a continuation variable).

3.5 Expressiveness

In this section, we prove that the hierarchy of operators $\mathcal{L}_i$ and $\hookleftarrow_i$ is as expressive as the hierarchy of the original operators *shift* $\mathcal{S}_i$ and throw $\overset{\mathcal{S}}{\hookleftarrow}_i$ [8]. We also consider an alternative throw operator $\overset{\mathcal{C}}{\hookleftarrow}_i$ and compare it with $\hookleftarrow_i$.

Regular shift and throw operators Because the original hierarchy of shift$_i$ and reset$_i$ operators with the addition of a throw$_i$ operator can be embedded in our hierarchy, the typing rules and the associated results carry over to the original hierarchy.

We now show how to express $\mathcal{L}_i$ and $\hookleftarrow_i$ with the regular shift and throw. We define a translation $(\cdot)^\circ$ which rewrites terms with $\mathcal{L}_i$ and $\hookleftarrow_i$ into terms with $\mathcal{S}_i$ and $\overset{\mathcal{S}}{\hookleftarrow}_i$ in the following way:

$$
\begin{array}{rcl}
x^\circ & = & x \\
(\lambda x.t)^\circ & = & \lambda x.t^\circ \\
(t_0\ t_1)^\circ & = & t_0^\circ\ t_1^\circ \\
\langle t\rangle_i^\circ & = & \langle t^\circ\rangle_i \\
(\mathcal{L}_i(k_1, \ldots, k_i).t)^\circ & = & \mathcal{S}_1 k_1^\circ.\mathcal{S}_2 k_2^\circ. \ldots \mathcal{S}_i k_i^\circ.t^\circ \\
((h_1, \ldots, h_i) \hookleftarrow_i t)^\circ & = & \\
\multicolumn{3}{r}{(\lambda x.\langle h_i^\circ \overset{\mathcal{S}}{\hookleftarrow}_i \ldots \langle h_2^\circ \overset{\mathcal{S}}{\hookleftarrow}_2 \langle h_1^\circ \overset{\mathcal{S}}{\hookleftarrow}_1 x\rangle_1\rangle_2 \ldots\rangle_i)\ t^\circ} \\
E_i^\circ & = & (\bullet_1, \bullet_2, \ldots, \bullet_{i-1}, E_i) \\
k_i^\circ & = & (k_1^i, \ldots, k_i^i)
\end{array}
$$

We assume that the translation of continuation variables k_i° is deterministic (i.e., it generates always the same tuple of variables, written $(k_i^\circ(1), \ldots, k_i^\circ(i))$) and that the translation of two different variables generates disjoint tuples. The idea of the translation is to perform successive shifts, in order to capture tuples of contexts of the form $(\bullet_1, \bullet_2, \ldots, \bullet_{i-1}, E_i)$. In the translation of $\hookleftarrow_i$, the contexts E_i are then restored successively by throwing to these tuples. Note that in the translated terms, we always throw to a tuple of contexts captured by a singular shift, therefore we respect the semantics of $\mathcal{S}_i$.

In order to prove the soundness of the translation with respect to CPS, we define a function $\widehat{\ }$, which returns the CPS translation of h_i°.

$$
\begin{array}{rcl}
\widehat{E_i^\circ} & = & [\![E_i]\!] \\
\widehat{k_i^\circ} & = & \lambda x k_{i+1} \ldots k_{n+1}.k_i^\circ(1)\ x\ k_i^\circ(2) \ldots k_i^\circ(i)\ k_{i+1} \ldots k_{n+1}
\end{array}
$$

In the following, we write $t\{(k'_1, \ldots, k'_i)/k_i^\circ\}$ as a shorthand for $t\{k'_1/k_i^\circ(1), \ldots, k'_i/k_i^\circ(i)\}$.

LEMMA 12. *The following equalities hold for all* $1 \le j \le i$*:*

$$
\begin{array}{rcl}
\overline{\mathcal{S}_j k_j^\circ. \ldots \mathcal{S}_i k_i^\circ.t} & =_{\beta\eta} & \lambda k_1 \ldots k_{n+1}. \\
& & \quad \bar{t}\{(\theta_1 \ldots \theta_{i-1}, k_i)/k_i^\circ, \ldots, (k_1 \ldots k_j)/k_j^\circ\} \\
& & \quad\quad \theta_1 \ldots \theta_i\ k_{i+1} \ldots k_{n+1} \\
\overline{\langle h_i^\circ \overset{\mathcal{S}}{\hookleftarrow}_j \ldots \langle h_1^\circ \overset{\mathcal{S}}{\hookleftarrow}_1 x\rangle_1\rangle_j} & =_{\beta\eta} & \lambda k_1 \ldots k_{n+1}.\widehat{h_1^\circ}\ x\ \widehat{h_2^\circ} \ldots \widehat{h_j^\circ} \\
& & \quad (\lambda v_0 k'_{j+2} \ldots k'_{n+1}.k_1\ v_0\ k_2 \ldots k_{j+1} k'_{j+2} \ldots k'_{n+1}) \\
& & \quad\quad k_{j+2} \ldots k_{n+1} \\
\widehat{k_i^\circ}\{(\theta_1 \ldots \theta_{i-1}, k'_i)/k_i^\circ\} & =_{\beta\eta} & k'_i
\end{array}
$$

Using this lemma, we can prove the simulation theorem below.

THEOREM 3. *Let* t *be a term (written with* $\mathcal{L}_i$ *and* $\hookleftarrow_i$*) and let* $k_{i_1}^1, \ldots, k_{i_j}^j$ *be its free continuation variables. Then* $\overline{t^\circ} =_{\beta\eta} \bar{t}\{\widehat{k_{i_1}^1{}^\circ}/k_{i_1}^1, \ldots, \widehat{k_{i_j}^j{}^\circ}/k_{i_j}^j\}$.

In particular, for closed terms we have $\overline{t^\circ} =_{\beta\eta} \bar{t}$.

The translation preserves typing judgments, except that we have to take into account the fresh variables generated by the translation of a continuation variable. To this end, we translate the type assignment $(k_1 : C_i)^\circ = k_i^\circ(1) : D_1 \ldots, k_i^\circ(i-1) : D_{i_1}, k_i^\circ(i) : C_i$, where $D_1 \ldots D_{i-1}$ are arbitrary types such that $\mathcal{I}_j(D_j)$ holds for $j \in 1 \ldots i-1$. We then have the following result.

LEMMA 13. *If* $\Gamma;\Delta \vdash_n t : C_1 \ldots C_{n+1}$ *then* $\Gamma;\Delta^\circ \vdash_n t^\circ : C_1 \ldots C_{n+1}$.

An alternative throw operator In some cases we may want to consider an alternative throw operator $\overset{\mathcal{C}}{\hookleftarrow}_i$, which restores saved contexts and discards the current ones without storing them. Formally, its CPS translation is defined as follows:

$$
\begin{array}{l}
\overline{(h_1, \ldots, h_i) \overset{\mathcal{C}}{\hookleftarrow}_i t} = \lambda k_1 \ldots k_{n+1}. \\
\quad \bar{t}\ (\lambda v_0 k'_2 \ldots k'_{n+1}.[\![h_1]\!]\ v_0\ [\![h_2]\!] \ldots [\![h_i]\!]\ k'_{i+1} \ldots k'_{n+1}) \\
\quad\quad k_2 \ldots k_{n+1}
\end{array}
$$

and the corresponding reduction rule is:

$$
\begin{array}{l}
\langle (E'_1, \ldots, E'_i) \overset{\mathcal{C}}{\hookleftarrow}_i v, E_1, \ldots, E_{n+1}\rangle \to_{\mathrm{v}} \\
\quad\quad \langle v, E'_1, \ldots, E'_i, E_{i+1}, \ldots, E_{n+1}\rangle
\end{array}
$$

The operators $\overset{\mathcal{C}}{\hookleftarrow_i}$ and $\hookleftarrow_i$ can be defined one in terms of the other as follows:

$$\begin{array}{rcl}(h_1,\ldots,h_i) \overset{\mathcal{C}}{\hookleftarrow_i} t & = & (\lambda x.\mathcal{L}_i(k'_1,\ldots,k'_i).(h_1,\ldots,h_i) \hookleftarrow_i x)\, t \\ & & \text{where } \{k'_1,\ldots,k'_i\} \cap \{h_1,\ldots,h_i\} = \emptyset \\ (h_1,\ldots,h_i) \hookleftarrow_i t & = & (\lambda x.\langle (h_1,\ldots,h_i) \overset{\mathcal{C}}{\hookleftarrow_i} x\rangle_i)\, t\end{array}$$

The idea behind the first equality is to capture and destroy the current contexts with $\mathcal{L}_i$; when the throw $\hookleftarrow_i$ is performed, only empty contexts are pushed on the context of level $i+1$. In the second equation, the delimiter $\langle\cdot\rangle_i$ effectively pushes the current contexts up to level i on the context of level $i+1$, and the throw $\overset{\mathcal{C}}{\hookleftarrow_i}$ restores the captured contexts, discarding only the empty contexts $\bullet_1,\ldots,\bullet_i$ in the process. One can check that both equations are sound with respect to the CPS translation.

We can derive a typing rule for $\overset{\mathcal{C}}{\hookleftarrow_i}$ from the type system of Figure 4 using the equation above, or directly from its CPS translation:

$$\frac{\begin{array}{c}C_1 = S \rhd C_2 \ldots \rhd C_{n+1} \\ \Gamma;\Delta \vdash_n h_1 : C_1 \quad \ldots \quad \Gamma;\Delta \vdash_n h_i : C_i \\ D'_1 = S \rhd C'_2 \rhd \ldots C'_i \rhd C_{i+1} \rhd \ldots C_{n+1} \\ \Gamma;\Delta \vdash_n t : D'_1, D_2, \ldots, D_{n+1}\end{array}}{\Gamma;\Delta \vdash_n (h_1,\ldots,h_i) \overset{\mathcal{C}}{\hookleftarrow_i} t : D_1, D_2, \ldots, D_{n+1}}$$

As in the previous case, the properties of subject reduction, soundness w.r.t. the CPS translation, and termination of evaluation hold for the new system, therefore one can use interchangeably the two throw operators.

3.6 Reflecting instead of throwing

As observed in [5], in practical applications it is often more convenient to specify continuation of the computation rather than to throw a value of this computation to the continuation. Such an operation $(h_1,\ldots,h_i) \hookrightarrow_i t$ of installing a tuple of continuations as the current continuations of a given computation can be defined via CPS translation as follows:

$$\begin{array}{l}\overline{(h_1,\ldots,h_i) \hookrightarrow_i t} = \lambda k_1 \ldots k_{n+1}.\bar{t}\, [\![h_1]\!] \ldots [\![h_i]\!] \\ \qquad (\lambda v k'_{i+2} \ldots k'_{n+1}.k_1\ v\ k_2 \ldots k_{i+1} k'_{i+2} \ldots k'_{n+1}) \\ \qquad\qquad k_{i+2} \ldots k_{n+1}\end{array}$$

Following the leads of this section it is then possible to derive the reduction and typing rules for this construct and to prove their expected properties.

4. More Flexible Control Operators

In this section we consider some variants of the operators introduced in Section 3. Instead of capturing and throwing to continuous sequences of contexts starting from 1 ($E_1,\ldots,E_i$), we allow capture and throw to any sequence of contexts $E_{i_1},\ldots,E_{i_j}$, where $1 \le i_1 < i_2 < \ldots < i_j \le n$. The syntax of terms is now defined as follows:

$$\begin{array}{rcl}t & ::= & x \mid \lambda x.t \mid t\ t \mid \mathcal{L}^*(k_{i_1},\ldots,k_{i_j}).t \mid \langle t\rangle_i \mid \\ & & (h_{i_1},\ldots,h_{i_j}) \overset{*}{\hookleftarrow} t,\end{array}$$

the syntax of level-1 contexts is adjusted accordingly:

$$E_1 \quad ::= \quad \bullet_1 \mid v\ E_1 \mid E_1\ t \mid (\ulcorner E_{i_1} \urcorner,\ldots,\ulcorner E_{i_j} \urcorner) \hookleftarrow_i E_1$$

and the remaining syntactic categories are defined as before. The CPS translation, reduction rules, and typing rules for the operators $\mathcal{L}^*$ and $\overset{*}{\hookleftarrow}$ are summarized in Figure 6. Notice that when we consider captures and throws to consecutive sets of variables starting from 1 ($k_1 \ldots k_i$), we obtain the same definitions and rules as in Section 3.

Using the same proof techniques as in the previous section, we can also prove the following results.

THEOREM 4 (Subject reduction). *If* $\Gamma;\Delta \vdash_n p : S$ *and* $p \to_{\mathrm{v}} p'$ *then* $\Gamma;\Delta \vdash_n p' : S$.

PROPOSITION 5 (Soundness of typing wrt. CPS). *The following implications hold:*

1. *If* $\Gamma;\Delta \vdash_n t : C_1,\ldots,C_{n+1}$, *then* $\overline{\Gamma};\overline{\Delta} \vdash \bar{t} : \overline{C_1} \to \ldots \to \overline{C_{n+1}} \to o$.
2. *If* $\Gamma;\Delta \vdash_n E_i : C_i$, *then* $\overline{\Gamma};\overline{\Delta} \vdash [\![E_i]\!] : \overline{C_i}$, *for all* $i = 1,\ldots,n$.

PROPOSITION 6 (Soundness of reduction wrt. CPS). *If* $p \to_{\mathrm{v}} p'$, *then* $\overline{p} =_{\beta\eta} \overline{p'}$.

THEOREM 5 (Termination of evaluation). *Let* t *be a closed plain term such that* $\cdot;\cdot \vdash_n t : C_1,\ldots,C_{n+1}$ *and* $\mathcal{I}_i(C_i)$ *hold for all* $i = 1,\ldots,n+1$. *Then* $\mathcal{N}(\langle t, \bullet_1,\ldots,\bullet_{n+1}\rangle)$ *holds.*

Expressiveness We first show how to simulate the operators $\mathcal{L}_i$ and $\overset{*}{\hookleftarrow}$ with $\mathcal{S}_i$ and $\overset{\mathcal{S}}{\hookleftarrow_i}$. We use the same translation as in Section 3.5, except that we translate $\overset{*}{\hookleftarrow}$ in the following way:

$$\begin{array}{l}((h_{i_1},\ldots,h_{i_j}) \overset{*}{\hookleftarrow} t)^\circ = \\ \qquad (\lambda x.\langle K_{i_j} \overset{\mathcal{S}}{\hookleftarrow_{i_j}} \ldots \langle K_2 \overset{\mathcal{S}}{\hookleftarrow_2} \langle K_1 \overset{\mathcal{S}}{\hookleftarrow_1} x\rangle_1\rangle_2 \ldots\rangle_{i_j})\, t^\circ\end{array}$$

with $K_l = h_l^\circ$ if $l \in \{i_1,\ldots,i_j\}$ and $K_l = (\bullet_1,\ldots,\bullet_l)$ otherwise. In the translation, if $l \notin \{h_{i_1},\ldots,h_{i_j}\}$, then we restore the empty context $\bullet_l$ as the current context of level l by throwing to $\bullet_l$ (in fact to the tuple $(\bullet_1,\ldots,\bullet_l)$, but only the last context matters). Otherwise, we throw to h_l°, as in the translation for $\hookleftarrow_i$. Because the translation of $\mathcal{L}_i$ remains unchanged, we still throw to tuples of contexts captured by a singular shift, as required by the semantics of shift. Expressing $\mathcal{L}^*$ with $\mathcal{S}_i$ seems more difficult, because $\mathcal{L}^*$ may capture some contexts and leave the first one unchanged, while $\mathcal{S}_i$ always captures a tuple of contexts, starting from the first one. We conjecture that $\mathcal{L}^*$ cannot be expressed with $\mathcal{S}_i$.

As in Section 3.5, we may consider an alternative throw operator $\overset{\mathcal{C}*}{\hookleftarrow}$, such that $(E_{i_1},\ldots,E_{i_j}) \overset{\mathcal{C}*}{\hookleftarrow} v$ replaces the current contexts at positions $i_1,\ldots,i_j$ by $E_{i_1},\ldots,E_{i_j}$, and leaves the other ones unchanged. We define this operator via its CPS translation:

$$\begin{array}{l}\overline{(h_{i_1},\ldots,h_{i_j}) \overset{\mathcal{C}*}{\hookleftarrow} t} = \lambda k_1 \ldots k_{n+1}. \\ \qquad \bar{t}\, (\lambda v_0 k'_2 \ldots k'_{n+1}.c_1\ v_0\ c_2 \ldots c_{i_j}\ k'_{i_j+1} \ldots k'_{n+1}) \\ \qquad\qquad k_2 \ldots k_{n+1}\end{array}$$

where $c_l = [\![h_l]\!]$ if $l \in \{i_1,\ldots,i_j\}$ and $c_l = k_l$ otherwise. The corresponding reduction rule is:

$$\begin{array}{l}\langle (E'_1,\ldots,E'_i) \overset{\mathcal{C}*}{\hookleftarrow} v, E_1,\ldots,E_{n+1}\rangle \to_{\mathrm{v}} \\ \qquad\qquad \langle v, E''_1,\ldots,E''_{i_j}, E_{i_j+1},\ldots,E_{n+1}\rangle\end{array}$$

where $E''_l = E'_l$ if $l \in \{i_1,\ldots,i_j\}$ and $E''_l = E_l$ otherwise. We can express $\overset{*}{\hookleftarrow}$ with $\overset{\mathcal{C}*}{\hookleftarrow}$ as follows:

$$(h_{i_1},\ldots,h_{i_j}) \overset{*}{\hookleftarrow} t = (\lambda x.\langle (h_{i_1},\ldots,h_{i_j}) \overset{\mathcal{C}*}{\hookleftarrow} x\rangle_{i_j})\, t$$

Roughly, the delimiter $\langle\cdot\rangle_{i_j}$ pushes the current contexts on E_{i_j+1}, and the captured contexts are then restored with $\overset{\mathcal{C}*}{\hookleftarrow}$. However, simulating $\overset{\mathcal{C}*}{\hookleftarrow}$ with $\overset{*}{\hookleftarrow}$ seems difficult, mainly because

CPS translation

$$\begin{array}{rcl}
\overline{\mathcal{L}^*(k'_{i_1},\ldots,k'_{i_j}).t} & = & \lambda k_1 \ldots k_{n+1}.\bar{t}\{k'_{i_1}/k_{i_1},\ldots,k'_{i_j}/k_{i_j}\}\; c_1 \ldots\; c_{i_j}\; k_{i_j+1} \ldots\; k_{n+1} \\
\text{where } c_l & = & \left\{ \begin{array}{ll} \theta_l & \text{if } j \in \{i_1,\ldots,i_l\} \\ k_l & \text{otherwise} \end{array} \right. \quad \text{for all } 1 \leq l \leq i_j \\
\overline{(h_{i_1} \ldots h_{i_j}) \overset{*}{\hookleftarrow} t} & = & \lambda k_1 \ldots k_{n+1}.\bar{t}\,(\lambda v_0 k'_2 \ldots k'_{n+1}.[\![h_{i_1}]\!]\; v_0\; d_{i_1+1} \ldots\; d_{i_j}\; (\lambda v_1 k''_{i_j+2} \ldots k''_{n+1}.k_1\; v_1\; k'_2 \ldots\; k'_{i_j+1}\; k''_{i_j+2} \ldots\; k''_{n+1}) \\
& & \qquad\qquad k'_{i_j+2} \ldots\; k'_{n+1})\; k_2 \ldots k_{n+1} \\
\text{where } d_l & = & \left\{ \begin{array}{ll} [\![h_l]\!] & \text{if } j \in \{i_1,\ldots,i_l\} \\ \theta_l & \text{otherwise} \end{array} \right. \quad \text{for all } i_1 \leq l \leq i_j
\end{array}$$

Reduction rules

$$\begin{array}{lrcl}
(\mathit{capture}*) & \langle \mathcal{L}^*(k_{i_1},\ldots,k_{i_j}).t, E_1,\ldots,E_{n+1}\rangle & \to_{\mathrm{v}} & \langle t\{E_1/k_{i_1},\ldots,E_{i_j}/k_{i_j}\}, E'_1,\ldots,E'_{i_j}, E_{i_j+1},\ldots,E_{n+1}\rangle \\
& & & \text{where } E'_l = \left\{ \begin{array}{ll} \bullet_l & \text{if } j \in \{i_1,\ldots,i_l\} \\ E_l & \text{otherwise} \end{array} \right. \quad \text{for all } 1 \leq l \leq i_j \\
(\mathit{throw}*) & \langle (E'_{i_1},\ldots,E'_{i_j}) \overset{*}{\hookleftarrow} v, E_1,\ldots,E_{n+1}\rangle & \to_{\mathrm{v}} & \langle v, E''_1,\ldots,E''_{i_j}, E_{i_j+1}.(E_{i_j}.,\ldots,(E_2.E_1),\ldots,), E_{i_j+2},\ldots,E_{n+1}\rangle \\
& & & \text{where } E''_l = \left\{ \begin{array}{ll} E'_l & \text{if } j \in \{i_1,\ldots,i_l\} \\ \bullet_l & \text{otherwise} \end{array} \right. \quad \text{for all } i_1 \leq l \leq i_j
\end{array}$$

Typing rules

$$\frac{\mathcal{I}_l(D_l) \text{ if } l \in \{i_1,\ldots,i_j\} \qquad D_l = C_l \text{ if } l \notin \{i_1,\ldots,i_j\} \qquad \Gamma;\Delta,\, k_{i_1} : C_{i_1} \ldots,\, k_{i_j} : C_{i_j} \vdash_n t : D_1,\ldots,D_{i_j}, C_{i_j+1} \ldots C_{n+1}}{\Gamma;\Delta \vdash_n \mathcal{L}^*(k_{i_1},\ldots,k_{i_j}).t : C_1, C_2, \ldots, C_{n+1}}$$

$$\frac{\begin{array}{c} C_{i_1} = S \rhd C_{i_1+1} \ldots \rhd C_{n+1} \qquad C_{i_j+1} = T \rhd C'_{i_j+2} \ldots \rhd C'_{n+1} \qquad \mathcal{I}_l(C_l) \text{ if } l \notin \{i_1,\ldots,i_j\} \text{ and } i_1 \leq l \leq i_j \\ \Gamma;\Delta \vdash_n h_{i_1} : C_{i_1} \quad \ldots \quad \Gamma;\Delta \vdash_n h_{i_j} : C_{i_j} \\ \Gamma;\Delta \vdash_n t : S \rhd C''_2 \rhd \ldots C''_{i+1} \rhd C_{i+2} \rhd \ldots C_{n+1}, D_2, \ldots, D_{n+1} \end{array}}{\Gamma;\Delta \vdash_n (h_{i_1},\ldots,h_{i_j}) \hookleftarrow_i t : T \rhd C''_2 \rhd \ldots C''_{i_j+1} \rhd C'_{i_j+2} \rhd \ldots C'_{n+1}, D_2, \ldots, D_{n+1}}$$

Figure 6. CPS translation, reduction rules, and typing rules for $\mathcal{L}^*$ and $\overset{*}{\hookleftarrow}$

$(E_{i_1},\ldots,E_{i_j}) \overset{\mathcal{C}*}{\hookleftarrow} v$ leaves contexts E_l such that $l \leq i_j$ and $l \notin \{i_1,\ldots,i_j\}$ unchanged, while $(E_{i_1},\ldots,E_{i_j}) \overset{*}{\hookleftarrow} v$ replaces them with $\bullet_1$. We conjecture that $\overset{\mathcal{C}*}{\hookleftarrow}$ cannot be expressed with $\overset{*}{\hookleftarrow}$.

5. Conclusion and Perspectives

We have developed the most expressive monomorphic type system for a family of control operators in the CPS hierarchy and for this type system we have proved subject reduction, soundness with respect to CPS, and termination of evaluation. We believe that the present article, as a sequel to the operational foundations of the CPS hierarchy built by the first two authors and Danvy, is another step towards better understanding of the CPS hierarchy, and consequently, that it can inspire new theoretical and practical applications of this beautiful but complex computational structure.

There are several directions for future research related to the present work. First of all, as opposed to the type systems of Danvy and Yang [9] and of Murthy [22] the type system presented in this work allows for computations that modify the answer type of continuations at an arbitrary level of the hierarchy, which should open new possibilities for practical applications that otherwise could only be expressed in an untyped setting.

Building an experimental implementation of the hierarchy with types à la Danvy and Filinski as presented in this article is another task. In particular, one can use the syntactic correspondence between context-based reduction semantics and abstract machines [6] to obtain an abstract machine equivalent with the reduction semantics of this article, or one could adjust the existing abstract machine for the hierarchy [5] accordingly and prove its correctness with respect to the reduction semantics. Devising a type reconstruction algorithm for the hierarchy should not pose any serious problems. A more ambitious goal is to marry the type system from this work with ML-polymorphism, which could be done along the lines presented by Asai and Kameyama [2].

Another improvement would be to allow for level polymorphism. Before typing a program with our system, we have to fix the number of hierarchy levels n, which can be problematic in practice. The whole program has to be typed using the $n+1$ levels, even if only a few parts are actually using high-level control operators. If we use a library in various programs, each with its own hierarchy level, we have to type the library several times, which goes against modularity. These issues could be fixed by allowing for level polymorphism: from a level n typing judgment, it should be possible to obtain a level $n+1$ judgment, as in Shan's type system [25].

It would be interesting to formalize the proof of termination of evaluation in the CPS hierarchy in a logical framework such as the Calculus of Inductive Constructions of Coq. As has been shown before [3, 4], normalization proofs by Tait's context-based method yield, through program extraction from proofs, non-trivial evaluators in CPS and the program extraction mechanism of Coq could be helpful for this task.

The present article focuses on the CPS hierarchy under the call-by-value reduction strategy. A natural next step is to see how the type system à la Danvy and Filinski for call-by-name shift and reset introduced by the first two authors [4] generalizes to a call-by-name hierarchy. It would be instructive to relate such a hierarchy to the one recently presented by Saurin [24].

Finally, the still open question of the logical interpretation of delimited continuations through the Curry-Howard isomorphism carries over from *shift* and *reset* to the hierarchy.

Acknowledgments

We thank the anonymous reviewers for thorough comments on the presentation of this article and for excellent suggestions for future work. This work has been supported by the MNiSW grant number N N206 357436, 2009-2011.

References

[1] Z. M. Ariola, H. Herbelin, and A. Sabry. A proof-theoretic foundation of abortive continuations. *Higher-Order and Symbolic Computation*, 20(4):403–429, 2007.

[2] K. Asai and Y. Kameyama. Polymorphic delimited continuations. In *Proceedings of the Fifth Asian symposium on Programming Languages and Systems, APLAS'07*, number 4807 in Lecture Notes in Computer Science, pages 239–254, Singapore, Dec. 2007.

[3] M. Biernacka and D. Biernacki. A context-based approach to proving termination of evaluation. In *Proceedings of the 25th Annual Conference on Mathematical Foundations of Programming Semantics(MFPS XXV)*, Oxford, UK, Apr. 2009.

[4] M. Biernacka and D. Biernacki. Context-based proofs of termination for typed delimited-control operators. In F. J. López-Fraguas, editor, *Proceedings of the 11th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, Coimbra, Portugal, Sept. 2009. ACM Press.

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

[6] M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007.

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

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

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

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

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

[12] R. Harper, B. F. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, Oct. 1993. A preliminary version was presented at the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1991).

[13] H. Herbelin and S. Ghilezan. An approach to call-by-name delimited continuations. In P. Wadler, editor, *Proceedings of the Thirty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 383–394. ACM Press, Jan. 2008.

[14] Y. Kameyama. Axioms for control operators in the CPS hierarchy. *Higher-Order and Symbolic Computation*, 20(4):339–369, 2007. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).

[15] Y. Kameyama, O. Kiselyov, and C. Shan. Shifting the stage: Staging with delimited control. In G. Puebla and G. Vidal, editors, *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2009)*, pages 111–120, Savannah, GA, Jan. 2009. ACM Press.

[16] O. Kiselyov. Call-by-name linguistic side effects. In *Proceedings of the 2008 Workshop on Symmetric Calculi and Ludics for the Semantic Interpretation*, Hamburg, Germany, Aug. 2008.

[17] O. Kiselyov. Delimited control in ocaml, abstractly and concretely: System description. In M. Blume and G. Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*, number 6009 in Lecture Notes in Computer Science, pages 304–320, Sendai, Japan, Apr. 2010. Springer.

[18] O. Kiselyov and Chung-chieh. Embedded probabilistic programming. In W. Taha, editor, *Domain-Specific Languages, DSL 2009*, number 5658 in Lecture Notes in Computer Science, pages 360–384, Oxford, UK, July 2009. Springer.

[19] O. Kiselyov and C. Shan. Delimited continuations in operating systems. In B. Kokinov, D. C. Richardson, T. R. Roth-Berghofer, and L. Vieu, editors, *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT 2007*, number 4635 in Lecture Notes in Artificial Intelligence, pages 291–302, Roskilde, Denmark, Aug. 2007. Springer.

[20] O. Kiselyov and C. Shan. A substructural type system for delimited continuations. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007*, number 4583 in Lecture Notes in Computer Science, pages 223–239, Paris, France, June 2007. Springer-Verlag.

[21] M. Masuko and K. Asai. Direct implementation of shift and reset in the MinCaml compiler. In A. Rossberg, editor, *Proceedings of the ACM SIGPLAN Workshop on ML, ML'09*, pages 49–60, Edinburgh, UK, Aug. 2009.

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

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

[24] A. Saurin. A hierarchy for delimited continuations in call-by-name. In L. Ong, editor, *Foundations of Software Science and Computation Structures, 13th International Conference, FOSSACS 2010*, number 6014 in Lecture Notes in Computer Science, pages 374–388, Paphos, Cyprus, Mar. 2010. Springer-Verlag.

[25] C. Shan. Delimited continuations in natural language: quantification and polarity sensitivity. In H. Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 55–64, Venice, Italy, Jan. 2004.

[26] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32:198–212, 1967.

[27] N. Zeilberger. Polarity and the logic of delimited continuations. In J.-P. Jouannaud, editor, *Proceedings of the 25th IEEE Symposium on Logic in Computer Science (LICS 2010)*, pages 219–227, Edinburgh, UK, July 2010. IEEE Computer Society Press.