Typing control operators in the CPS hierarchy
Malgorzata Biernacka, Dariusz Biernacki, Sergueï Lenglet

To cite this version:
Malgorzata Biernacka, Dariusz Biernacki, Sergueï Lenglet. Typing control operators in the CPS hierarchy. PPDP - Principles and Practice of Declarative Programming, Jul 2011, Odense, Denmark. pp.149-160, 10.1145/2003476.2003498. hal-00903834

HAL Id: hal-00903834
https://inria.hal.science/hal-00903834
Submitted on 13 Nov 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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., nondeterministic 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, and reset, 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.

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, 26], and implementational [17, 21] advances in the field. Of the numerous control operators for delimited continuations, the 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 [8, 10, 5], 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 nondeterministically 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 metacoroutine 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, and reset, \( i \geq 1 \) 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. To our knowledge, the only work on a formal type system for the hierarchy is Murthy’s study [22], where the author proposes a more relaxed typing discipline allowing the delimited continuations of level \( i \) to have various answer types, provided the answer type agrees with the type expected by the continuation at level \( i + 1 \).

In this article, we go further and propose a type system that 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 $i + 1$. Furthermore, the control operators we consider are slightly more flexible than the original shift and reset families 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.

The overall goal of this article is to establish type-theoretic foundations of the CPS hierarchy and consequently, to shed more light on this beautiful but complex computational structure and, hopefully, inspire new theoretical and practical applications of the 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, and reset, family, given in terms of a CPS translation and reduction semantics provably sound with respect to the CPS translation (Sections 2.1 and 2.2);
- a 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 2.3);
- a proof of termination in reduction semantics, using a context-based method of reducibility predicates (Section 2.4);
- a simulation of the presented operators with the original family of shift, and reset, (Section 2.5);
- a generalization of the presented results to a hierarchy of even more flexible control operators expressible in the CPS hierarchy (Section 3).

2. 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 first hierarchy of control operators due to Danvy and Filinski [8].

2.1 Syntax

The language of terms at an arbitrary level $n$ of the hierarchy extends the usual lambda terms with delimited-control operators capture $L_n$, reset $(\cdot)_n$, and throw $\leftrightarrow_n$, for $n \in \mathbb{N}$. 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 distinct sets of continuation variables, ranged over by $k_1, \ldots, k_n$. The syntax of terms at level $n$ is defined as follows (where $1 \leq i \leq n$):

$t ::= x \mid \lambda x.t \mid t t \mid L_i(k_1, \ldots, k_i).t \mid (\cdot)_i \mid (h_1, \ldots, h_n) \leftrightarrow_i t$

$h_i ::= k_i \mid E_i$

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

$E_i ::= \bullet_i \mid v E_i \mid E_i t \mid (E_i, \ldots, E_i) \leftrightarrow_i E_i$

$L_i(k_1, \ldots, k_i).t$ are capture operators, each $(\cdot)_i$ delimits the scope of the corresponding capture operator, and the throw constructs $(h_1, \ldots, h_n) \leftrightarrow_i t$ (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 construct $S_i.k.t$ is binding 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 $k \leftrightarrow_i t$ can be introduced to distinguish continuation applications in the original hierarchy for typing purposes, as discussed in [4] (and in Section 2.3).

Translating a term written with operators $S_i$ and $\leftrightarrow_i$ 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 $L_j$. Therefore, we distinguish plain terms, i.e., terms that contain only throw constructs of the form $(k_1, \ldots, k_i) \leftrightarrow_i t$.

An abstraction $\lambda x.t$ binds $x$ in $t$ and a capture construct $L_i(k_1, \ldots, k_i).t$ 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 iff 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., $\bullet_i$ represents the empty context of level 1, $v E_i$ represents the “term with a hole” $E_i[v \cdot]$, $E_i t$ represents $E_i[[t[\cdot]]_i]$, and $(E_i, \ldots, E_i) \leftrightarrow_i E_i'$ represents $E_i'[[E_i, \ldots, E_i][\cdot]]_i$. A context of level $i$ for $i = 2, \ldots, n + 1$ is a stack of contexts of level $i - 1$ separated by a delimiter $(\cdot)_{i-1}$. Therefore the empty context $\bullet_i$, of level $i$ stands for the term with a hole $[[\cdot]]_{i-1}$, and $E_i E_i$ represents $E_i'[[E_i, \ldots, E_i][\cdot]]_i$. 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$_i$ as follows:

$$\text{plug}_i (t, \bullet_i) = t$$
$$\text{plug}_i (t, v E_i) = \text{plug}_i (v t, E_i)$$
$$\text{plug}_i (t_0, E_i t_1) = \text{plug}_i (t_0 t_1, E_i)$$
$$\text{plug}_i (t, (E_i, \ldots, E_i) \leftrightarrow_i E_i') = \text{plug}_i ((E_i, \ldots, E_i) \leftrightarrow_i t, E_i')$$
CPS translation of terms

\[\begin{align*}
\tau &= \lambda k_1 \ldots k_{n+1}. k_1 \ x \ k_2 \ldots \ k_{n+1} \\
\lambda x. t &= \lambda k_1 \ldots k_{n+1}. k_1 (\lambda x k_2 \ldots k_{n+1} \ t) k_2 \ldots k_{n+1} \\
\iota_0 \ t &= \lambda k_1 \ldots k_{n+1}. \ i_0 (\lambda k_0 k_2 \ldots k_{n+1} \ t) k_2 \ldots k_{n+1}
\end{align*}\]

\[\begin{align*}
\overline{t} &= \lambda k_1 \ldots k_{n+1}. \overline{\theta} \ t \overline{\theta} (\lambda k_0 k_2 \ldots k_{n+1} \ t) k_2 \ldots k_{n+1}. k_0 \ k_2 \ldots k_{n+1} \overline{\theta} (\lambda k_1 k_2 \ldots k_{n+1} \ t) k_2 \ldots k_{n+1}
\end{align*}\]

\[\begin{align*}
\overline{(h_1 \ldots h_i) \overline{\theta} \ t} &= \lambda k_1 \ldots k_{n+1}. \overline{\theta} (h_0 k_2 \ldots k_{n+1} h_i) v_0 [h_0] [h_i] (\lambda k_1 k_2 \ldots k_{n+1} (\lambda k_1 k_2 \ldots k_{n+1} k_i) v_1 k_2 \ldots k_{n+1} k_i)
\end{align*}\]

where \( \overline{\theta} = \lambda x k_{i+1} \ldots k_{n+1}. k_{i+1} x k_{i+2} \ldots k_{n+1} \) for \( i = 1, \ldots, n \)

Refunionalization of contexts

\[\begin{align*}
[k_1] &= \ i_1 \\
[\bullet] &= \ i_1 \ \\
[E_1] &= \ i_1 \ \iota_0 k_2 \ldots k_{n+1} \overline{\theta} (\lambda v_1 k_2 \ldots k_{n+1} v_1 [E_1] [k_2 \ldots k_{n+1}]) k_2 \ldots k_{n+1}
\end{align*}\]

\[\begin{align*}
\theta_0 [t] &= \lambda k_2 \ldots k_{n+1} \overline{\theta} (\lambda v_1 [E_1] [k_2 \ldots k_{n+1}]) \ i_1 \ [E_1] [k_2 \ldots k_{n+1}]
\end{align*}\]

\[\begin{align*}
\overline{\theta} (E_i, E_{i-1}) &= \lambda x k_{i+1} \ldots k_{n+1} \ i_1 \ v [E_1] k_{i+1} \ldots k_{n+1}
\end{align*}\]

\[\begin{align*}
\lambda x.t' &= \lambda x.t
\end{align*}\]

Figure 1: CPS translation

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

\[\begin{align*}
\text{plug}_1 (t, \bullet, ) &= \ t \\
\text{plug}_1 (t, E_i, E_{i-1}) &= \text{plug}_1 (\ i_1 \ t, E_{i-1}))
\end{align*}\]

We write \( E_i [t] \) for the term \( \text{plug}_1 (t, E_i) \).

In the following, we represent terms as programs in order to keep the layers of contexts delimited by the reset operators \( \langle \rangle \). 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 2.4).

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

\[\text{plug}_{n+1} (\ i_1 \ t, E_{n+1})\]

which can also be written as

\[E_{n+1} [\text{plug}_1 (\ i_1 \ t, E_{n+1})].\]

It can be seen from this definition that each term that we consider as a program at level \( n \) is implicitly enclosed by the reset operators of each level from 1 to \( n \). For example, the term \( \lambda x.t \) will be considered as the program representing the term \( \langle \rangle \langle \rangle \ldots \langle \rangle (\lambda x.t) \rangle \).

To shorten the notation, we write \( \text{plug} (p) \) for the term represented by the program \( p \).

Since a term may have several decompositions, different programs can represent the same term. For example, for \( n = 2 \), the program \( \langle \langle \lambda x.t \rangle \rangle \) can also be decomposed as \( \langle \lambda x.t, \bullet, v, E_2, E_1, E_0 \rangle \) or as \( \langle \langle \lambda x.t \rangle \rangle, E_1, E_2, E_3 \rangle \).

We identify all decompositions of the same term by defining an equivalence relation on programs, as follows:

\[p \sim p' \iff \text{plug} (p) = \text{plug} (p') \]

and considering programs up to this equivalence.

2.2 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 Bierne 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 \( \langle \rangle \) which transforms contexts into continuations they represent (leaving continuation variables unchanged).

The call-by-value reduction semantics is shown in Figure 2. We write \( \{s/x\} \) for the usual capture-avoiding substitution of \( s \) for the variable \( x \) in \( t \), and we write \( \{E_1/k_1\} \ldots \{E_i/k_i\} \) 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 \( \beta \)-redexes of the call-by-value \( \lambda \)-calculus (the rule \( \beta \)).
i contexts $E_1, \ldots, E_n$ and substituting them for the variables $k_1, \ldots, k_i$ (the rule \texttt{capture})). 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 contexts $E_1', \ldots, E_i'$ are reinstated as the current contexts, and the then-current contexts $E_{i+1}, \ldots, E_n$ are stacked on the context $E_{i+1}$ (the rule \texttt{throw}). 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 \texttt{reset}.

A \textit{redex} is the first component (a term) of the program occurring on the left-hand side of each of the reduction rules above. A \textit{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 2.3 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 $\rightarrow_r$ is deterministic.

\textbf{Proposition 1 (Unique-decomposition property).} For all terms $t$, $t$ either is a value, or it decomposes uniquely into contexts $E_1, \ldots, E_{n+1}$ and a potential redex $r$, i.e., $t = \texttt{plug} ((r, E_1, \ldots, E_{n+1}))$.

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

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

\section{2.3 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{align*}
S &::= b \mid S \rightarrow [C_1, \ldots, C_{n+1}] \\
C_{n+1} &::= \neg S \\
C_i &::= S \triangleright C_{i+1} \ldots \triangleright C_{n+1}
\end{align*}

As in Danvy and Filinski's system, arrow types contain type annotations $S \rightarrow [C_1, \ldots, C_{n+1}]$: 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 $S \rightarrow v \rightarrow T$ corresponds to $S \rightarrow [T \rightarrow U, \neg V]$.

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 function types; a context of type $S \triangleright C_{i+1} \ldots \triangleright C_{n+1}$ 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 $\Gamma$ is a list of pairs $\Gamma; \Delta$ of the form $\Gamma, \Delta \vdash_t t : C_1, \ldots, C_{n+1}$ for terms, and typing judgments of the form $\Gamma; \Delta \vdash_n t : C_1, \ldots, C_{n+1}$ for contexts. If $\Gamma, \Delta \vdash_n t : C_1, \ldots, C_{n+1}$, then, under the assumptions $\Gamma, \Delta$, 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 $C_1 = S \triangleright C_2 \ldots \triangleright C_{n+1}$, then $t$ is of type $S$.

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

The typing rules are presented in Figure 3. 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 $\overline{ap(t)} = (\lambda k_1 \ldots k_{n+1}. t')$, for a given operator $op$, and we want to generate a typing rule of the form

\begin{align*}
\Gamma'; \Delta \vdash_n \overline{ap(t)} &::= C_1, \ldots, C_{n+1} \\
\Gamma'; \Delta \vdash_n t &::= \lambda k_1, \ldots, k_{n+1}. t'
\end{align*}
Terms \((1 \leq i \leq n)\):
\[
\Gamma; \Delta \vdash_n t : S, \Delta \vdash_n x : S \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_{n+1}, C_2, \ldots, C_n
\]
\[
\Gamma; \Delta \vdash_n \lambda x.t : (S \Rightarrow [C_1, \ldots, C_{n+1}]) \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_{n+1}, C_2, \ldots, C_n
\]
\[
\Gamma; \Delta \vdash_n t : S \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_{n+1}, C_2, \ldots, C_n
\]
\[
\Gamma; \Delta \vdash_n (S \Rightarrow [C_1, \ldots, C_{n+1}]) \Rightarrow C''_n \Rightarrow \cdots \Rightarrow C''_{n+1}, C'_2, \ldots, C_n
\]
\[
\Gamma; \Delta \vdash_n t \in \Gamma, C_{n+1}, C_2, \ldots, C_n
\]

Contexts and continuation variables:
\[
I_i(C_i) \quad 1 \leq i \leq n + 1
\]
\[
\Gamma; \Delta \vdash_n \bullet : C_i
\]
\[
\Gamma; \Delta \vdash_n E_i : C_i \quad \Gamma; \Delta \vdash_n E_{i-1} : S \Rightarrow C_1 \Rightarrow \cdots \Rightarrow C_{n+1} \quad 2 \leq i \leq n + 1
\]
\[
\Gamma; \Delta \vdash_n E_i, E_{i-1} : S \Rightarrow C_1 \Rightarrow \cdots \Rightarrow C_{n+1}
\]
\[
\Gamma; \Delta \vdash_n E_i : C_1 \quad \Gamma; \Delta \vdash_n t_1 : S \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_{n+1}, C''_2, \ldots, C_{n+1}
\]
\[
\Gamma; \Delta \vdash_n E_i : S \Rightarrow [C_1, \ldots, C_{n+1}] \Rightarrow C''_2 \Rightarrow \cdots \Rightarrow C_{n+1}
\]
\[
\Gamma; \Delta \vdash_n t : (S \Rightarrow [C_1, \ldots, C_{n+1}]) \Rightarrow C''_2 \Rightarrow \cdots \Rightarrow C_{n+1}
\]
\[
\Gamma; \Delta \vdash_n \lambda v.t : S \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_{n+1}, C_2, \ldots, C_n
\]
\[
\Gamma; \Delta \vdash_n (E_i, \ldots, E_i) \leftarrow t, E_i : S \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_{n+1}, C_{i+1}, C_{i+2} \Rightarrow \cdots \Rightarrow C_{n+1}
\]

Programs:
\[
\Gamma; \Delta \vdash_n E_{n+1}([\ldots E_i[t] \ldots]_n) : S \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_{n+1}, E_2, \ldots, E_{n+1}
\]
\[
\Gamma; \Delta \vdash_n (t, E_1, \ldots, E_{n+1}) : S
\]

Figure 3: A type system for the level \(n\) of the CPS hierarchy

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 \(C'_1, \ldots, C'_n\) from the types of \(k'_1, \ldots, k'_{n+1}\). For example, consider the CPS translation for term variables:

\[
\mathcal{x} = \lambda x_1 \ldots x_{n+1} : k_1 \times k_2 \ldots k_{n+1}
\]

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 typng rule for term variables:

\[
\Gamma, x : S, \Delta \vdash_n x : S \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_{n+1}, C_2, \ldots, C_n
\]

Consider now the CPS translation for reset:

\[
(\theta^0_1) = \lambda \theta_1 \ldots \theta_{n+1} \theta_1 \ldots \theta_1 \cdot k_1 \ldots k_{n+1}
\]

with

\[
k_{n+1} = (\lambda \theta_1 \ldots \theta_{n+1} \ldots k_{n+1} \ldots k_1 \ldots k_{n+1})
\]

Assume that we type \(k_1\) with \(S \Rightarrow C_2 \Rightarrow \cdots \Rightarrow C_{n+1}\). To be able to type the continuation \(k'_{n+1}\) 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 $S > C_{i+2} \ldots C_{n+1}$. 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 $\theta_1, \ldots, \theta_t$. Let us recall the definition of the initial continuation:

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

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

$$I_j (C_{\cdot}) := \exists S, C_i, \ldots, C_{n+1}. \quad C_i = S \triangleright (S \triangleright C_{i+2} \ldots C_{n+1}) \triangleright C_{i+2} \ldots C_{n+1}$$

and $I_{n+1} (C_{\cdot}) = True$. We now have enough information to write the typing rule for $\langle \cdot \rangle$. 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 $\bullet$, into the initial continuation $\theta_i$, the empty context of level $i$ can be typed with any type $C_i$ provided that $I_i (C_{\cdot})$ holds. Note that we use the same typing judgment for continuation variables $\Gamma \vdash \Delta \vdash \iota : C_i$ 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, $S_i$, (and for reset, and throw $\mathcal{S}_i$, respectively) is the same as the rule for $\mathcal{L}$ (and for reset, throw $\mathcal{L}_i$, 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-shift and reset. Using Danvy and Filinski’s typing judgment, a typable term $t$ is pure iff we can derive $\Gamma, \Delta \vdash t : S \triangleright T$ 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

$$\Gamma, \Delta \vdash t : D_i, \ldots, D_n, \neg S$$

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 $E_{2, \ldots, n-1}$ are arbitrary, and the type of the first enclosing context is of the form $S \triangleright C_{2, \ldots, n-1}$. Therefore, we say that a typable term $t$ is pure iff we can derive $\Gamma, \Delta \vdash t : S \triangleright C_{2, \ldots, n-1}, C_2, \ldots, C_{n-1}$ 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 $E_i, E_{n+1}$ from the typing judgment $\Gamma, \Delta \vdash t, E_i, \ldots, E_{n+1}$ : $S$. We will need a few lemmas:

**Lemma 1.** If $\Gamma, \Delta \vdash E_i [t] : D_1, C_2, \ldots, C_{n+1}$ and $I_1 (D_{n})$ hold, then $\Gamma, \Delta \vdash E_i [t] : C_1, C_2, \ldots, C_{n+1}$ and $\Gamma, \Delta \vdash E_1 : C_1$ hold for some $C_1$.

The proof is a straightforward structural induction on $E_i$.

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

1. If

$$\Gamma, \Delta \vdash E_i [t] \vdash D_1, D_2, \ldots, D_{i-1}, C_{i+1}, \ldots, C_{n+1}$$

and $I_j (D_j)$ for all $j = 1, \ldots, j$, then there exists $C_i$ such that $\Gamma, \Delta \vdash E_i : C_i$ and

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

with $I_j (D'_j)$ for all $l = 1, \ldots, i - 1$.

2. If

$$\Gamma, \Delta \vdash E_i [t] \vdash D_1, \ldots, D_{n}, \neg S$$

$I_1 (D_i)$ holds for all $l = 1, \ldots, i$, and $j \geq i$, then

$$\Gamma, \Delta \vdash t : D'_i, \ldots, D'_j \triangleright C'_j \triangleright \ldots \triangleright C_{n+1}$

and $I_j (D'_j)$ for all $l = 1, \ldots, j$ and

$$\Gamma, \Delta \vdash E_i : S \triangleright C_{i+1} \triangleright \ldots \triangleright C_j \triangleright C'_j \triangleright \ldots \triangleright C_{n+1}$$

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

$$\Gamma, \Delta \vdash E_{n+1} [t] \vdash S \triangleright C_2 \ldots \triangleright C_{n+1}, C_2, \ldots, C_{n+1}$$

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

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

with $\mathcal{I}_l (D_l)$ for all $l = 1, \ldots, n$, $\Gamma, \Delta \vdash E_n^1 : T' \triangleright \neg S$, and

$$\Gamma, \Delta \vdash E_{n+1} : \neg T$$

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 $\Gamma, \Delta \vdash \langle t, E_1, \ldots, E_{n+1} \rangle : S \triangleright E_{n+1} = \bullet_{n+1}, E_n^1, \ldots, E_n^n$, then there exist $C_1, \ldots, C_{n+1}$ such that

$$\Gamma, \Delta \vdash E_i : C_i$$

for all $i = 1, \ldots, n + 1$, and $\Gamma, \Delta \vdash t : C_1, \ldots, C_{n+1}$. Furthermore, $\Gamma, \Delta \vdash E_n^1 : T' \triangleright \neg S$ 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 $\Gamma, \Delta \vdash t : C_1, \ldots, C_{n+1}$ and $\Gamma, \Delta \vdash E_i : C_i$, then $\Gamma, \Delta \vdash E_i [t] : D_1, C_2, \ldots, C_{n+1}$ is derivable and $I_1 (D_1)$ holds.
Lemma 6. The following properties hold:

1. If
\[ \Gamma, \Delta \vdash n \quad t : D_1, \ldots, D_{i-1}, C_i, C_{i+1}, \ldots, C_n, \]
then \( I_l(D_l) \) hold for all \( l = 1, \ldots, i-1 \), and \( \Gamma, \Delta \vdash n \quad E_i : C_i \), then
\[ \Gamma, \Delta \vdash n \quad E_{[(t)]_{i}} : D'_1, \ldots, D'_{l}, C_{i+1}, \ldots, C_{n+1} \]
and \( I_l(D'_l) \) hold for all \( l = 1, \ldots, i \).  

2. If
\[ \Gamma, \Delta \vdash n \quad t : D_1, \ldots, D_j, C_{j+1}, \ldots, C_n, \]
then \( I_l(D_l) \) hold for all \( l = 1, \ldots, j \), \( C_{j+1} = S \triangleright C'_{j+2} \triangleright \ldots \triangleright C_{n+1} \), and
\[ \Gamma, \Delta \vdash n \quad E_i : S \triangleright C_{i+1} \triangleright \ldots \triangleright C_{j+1} \triangleright C'_{j+2} \triangleright \ldots \triangleright C'_{n+1}, \]
then
\[ \Gamma, \Delta \vdash n \quad E_{[(t)]_{j}} : D'_1, \ldots, D'_{l}, C_{i+1}, \ldots, C_{n+1} \]
and \( I_l(D'_l) \) hold for all \( l = 1, \ldots, i \).

Lemma 7. If \( \Gamma, \Delta \vdash n \quad t : C_1, \ldots, C_n, \neg T \), \( I_l(D_l) \) hold for all \( l = 1, \ldots, n \), \( E_{n+1} = \bullet_{n+1}, E_{n+2}, \ldots, E_m \), \( \Gamma, \Delta \vdash n \quad E_{n+1} : \neg T, \) and \( \Gamma, \Delta \vdash n \quad E'_{n+1} : T' \triangleright \neg S, \) then
\[ \Gamma, \Delta \vdash n \quad E_{n+1+[(t)]_{n+1}} : S \triangleright C_2 \triangleright \ldots \triangleright C_{n+1}, C_{n+2}, \ldots, C_{n+1}. \]

Lemma 8. If \( \Gamma, \Delta \vdash n \quad t : C_1, \ldots, C_n+1, \Gamma, \Delta \vdash n \quad E_i : C_i \)
for all \( i = 1, \ldots, n + 1 \), \( E_{n+1} = \bullet_{n+1}, E_{n+2}, \ldots, E_m \), and
\[ \Gamma, \Delta \vdash n \quad E'_{n+1} : T' \triangleright \neg S, \) then \( \Gamma, \Delta \vdash n \quad \langle t, E_1, \ldots, E_{n+1} \rangle : S \)
is derivable.

As usual, we need a substitution lemma to deal with the \( (\beta_e) \) and \( (\text{capture}) \) reduction rules.

Lemma 9 (Substitution Lemma). The following hold:

1. If \( \Gamma, x : S; \Delta \vdash n \quad t : C_1, \ldots, C_{n+1} \) and \( \Gamma, \Delta \vdash n \quad v : S \triangleright C'_1 \triangleright \ldots \triangleright C'_{n+1} \), then \( \Gamma, \Delta \vdash n \quad t(v/x) : C'_1, \ldots, C_{n+1} \).

2. If \( \Gamma, \Delta, K : D_1 \triangleright \ldots \triangleright D_i, \Delta \vdash n \quad t : C_1, \ldots, C_{n+1} \) and \( \Gamma, \Delta \vdash n \quad E_i : D_j \) for all \( j = 1, \ldots, i \), then \( \Gamma, \Delta \vdash n \quad t\{E_1, \ldots, E_i/K\} : C_1, \ldots, C_{n+1} \).

Using these lemmas, we can prove subject reduction.

Theorem 1 (Subject Reduction). If \( \Gamma, \Delta \vdash n \quad p : S \) and \( p \to \nu \quad p' \) then \( \Gamma, \Delta \vdash n \quad p' : S \).

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:

\[ S \to [C_1 \ldots C_{n+1}] = S \to C_1 \to \ldots \to C_{n+1} \to o \]
\[ C_i \to S \to C_{i+1} \to \ldots \to C_{n+1} \to o \]
\[ \neg S = \neg S \to o \]

where \( o \) is an abstract answer type.

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

Proposition 2 (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 \( \Gamma, \Delta \vdash n \quad t : C_1, \ldots, C_{n+1}, \) then \( \Gamma, \Delta \vdash n \quad \langle t, E_1, \ldots, E_{n+1} \rangle : S \to [C_1 \ldots C_{n+1}] \to o \).
2. If \( \Gamma, \Delta \vdash n \quad E_i : C_i, \) then \( \Gamma, \Delta \vdash n \quad [E_i] : C_i, \) for all \( i = 1, \ldots, n \).

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

Proposition 3 (Soundness of Reduction wrt. CPS). If \( p \to \nu \quad p' \), then \( p = \rho_0 \cdot \theta \).

Proposition 3 is proved using a characterization of the CPS-image of a program in terms of its CPS-translated components.
Proposition 4 (Characterization of CPS image). If \( p = (t, E_1, \ldots, E_n) \), then
\[
\bar{p} = \rho \lambda k_1 \ldots k_{n+1} \mathcal{T}[E_1] \ldots [E_n](\lambda v_0.k_1 v_0 k_2 \ldots k_{n+1}).
\]

Proposition 4 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 refocusing 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 refocused stack of lower-level contexts/continuations).

2.4 Termination of Evaluation

We prove termination for call-by-value evaluation, extending the method used by Biernecka and Biernecka [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 [25]. 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 4. The predicate \( R_\Sigma \), indexed by term types, is defined on values, and the predicate \( K_C \), 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 \( E_i \)).

In the following, for any closed value \( v \) we write \( \vdash v : S \) iff there exist \( C_2, \ldots, C_{n+1} \) such that \( \vdash v : S \supset C_2 \supset \cdots \supset C_{n+1} \). Because \( v \) is pure, we do not care about the specific \( C_2, \ldots, C_{n+1} \), as discussed in Section 2.3.

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

Lemma 10. If \( \mathcal{L}(C_i) \), then \( K_{C_i}(\bullet) \).

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 \( \Delta = \{k_1^1, \ldots, k_m^1\} : D_1^1 \). Let \( \bar{v} \) be closed values such that \( \vdash \bar{v} : T_i \) and \( R_{T_i}(\bar{v}_i) \) for all \( i = 1, \ldots, n \). Let \( E_1 \) be closed contexts such that \( \vdash \bar{v} : D_1^1 \) and \( K_{C_i}(E_i^1) \). Let \( E_1', \ldots, E_{n+1}' \) be closed contexts such that \( \vdash \bar{v} : E_i', \vdash E_i' : C_i \) and \( K_{C_i}(E_i^1) \). Then \( \mathcal{N}(\langle \{\bar{v}_1, \bar{v}_2, \ldots, \bullet_{i-1}, E_i\}, E_1', \ldots, E_{n+1}' \rangle) \) 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 \( \vdash_n t : C_1, \ldots, C_{n+1} \), and \( \mathcal{L}(C_i) \) 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 of captured contexts occurring in them (here, it can only happen by substituting a reducible context for a continuation variable).

2.5 Expressiveness

In this section, we prove that the hierarchy of operators \( \mathcal{L}_i \) and \( \leftarrow_i \) is as expressive as the hierarchy of the original operators \( \mathcal{S}_i \) and \( \leftarrow_i \) [8]. We also consider an alternative throw operator \( \leftarrow_i \) and compare it with \( \leftarrow_i \).

Regular shift and throw operators. Because the original hierarchy of \( \leftarrow_i \) and \( \leftarrow_i \) with the addition of a throw, 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 \( \leftarrow_i \), with the regular shift and throw. We define a translation \( \hat{\bullet} \) which rewrites terms with \( \mathcal{L}_i \) and \( \leftarrow_i \) into terms with \( \mathcal{S}_i \) and \( \leftarrow_i \) in the following way:

\[
x = x
\]
\[
(\lambda x. t)^\hat{\bullet} = \lambda x. t^\circ
\]
\[
(t_0 t_1)^\hat{\bullet} = t_0^\circ t_1^\circ
\]
\[
\langle t \rangle_i^\circ = \langle t_i \rangle_i
\]
\[
\mathcal{L}_i(k_1, \ldots, k_i, t)^\hat{\bullet} = S_1 k_1^\circ S_2 k_2^\circ \cdots S_i k_i^\circ t^\circ
\]
\[
((h_1, \ldots, h_i) \leftarrow_i t)^\hat{\bullet} = \mathcal{S}_1 h_1 \cdots h_i \leftarrow_i t^\circ
\]

We assume that the translation of continuation variables \( k_i^\circ \) is deterministic (i.e., it generates always the same tuple of variables, written \( \langle k_1^\circ(1), \ldots, k_i^\circ(i) \rangle \)) 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 \( \langle \bullet_1, \bullet_2, \ldots, \bullet_{i-1}, E_i \rangle \). In the translation of \( \leftarrow_i \), 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 \( \leftarrow_i \).

In order to prove the soundness of the translation with respect to CPS, we define a function \( \hat{\bullet} \), which returns the CPS translation of \( h_i^\circ \):

\[
\hat{E}_i^\circ = [E_i]
\]
\[
\hat{k}_i^\circ = \lambda x k_{i+1} \ldots k_{n+1} k_i^\circ(1) x k_i^\circ(2) \cdots k_i^\circ(i) k_{i+1} \ldots k_{n+1}
\]

In the following, we write \( t \{k_i^\circ(1), \ldots, k_i^\circ(i)\} \) as a shorthand for \( t \{k_i^\circ(1), \ldots, k_i^\circ(i)\} \).
Lemma 12. The following equalities hold for all $1 \leq j \leq i$:

$$
\begin{align*}
S_j k_j^1 & = \beta_\eta \lambda k_1 \ldots k_{n+1}, \\
\bar{t}_j & = \beta_\eta \lambda (k_1 \ldots k_{n+1}) / (k_j^1 \ldots (k_1 \ldots k_{n+1} / k_j^1)) \\
\beta_\eta & = \lambda \beta_\eta \lambda x. \beta_\eta \lambda k_1 \ldots k_{n+1} \bar{t}_j \bar{t}_j
\end{align*}
$$

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

Theorem 3. Let $t$ be a term and let $k_1^i, \ldots, k_n^i$ (written with $\beta_\eta$) be its free continuation variables. Then $\bar{t} = \beta_\eta \bar{t}_i$ for all $1 \leq j \leq i$.

In particular, for closed terms we have $\bar{t} = \beta_\eta \bar{t}$. 

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

$$(h_1, \ldots, h_i) \bar{t}_i = \lambda k_1 \ldots k_{n+1}, \\
\bar{t} (\lambda x. \beta_\eta \lambda x. \beta_\eta \lambda k_1 \ldots k_{n+1} \bar{t}_i \bar{t}_j (k_1 \ldots k_{n+1} / k_j^1) k_j^1)$$

and the corresponding reduction rule is:

$$((E_1', \ldots, E_i') \bar{t}_i v, E_1, \ldots, E_{n+1}) \rightarrow_v (v, E_1', \ldots, E_i', E_{i+1}, \ldots, E_{n+1})$$

The operators $\bar{t}_i$ and $\bar{t}$ can be defined in terms of the others as follows:

$$(h_1, \ldots, h_i) \bar{t}_i = (\lambda x. \beta_\eta \lambda x. \beta_\eta \lambda k_1 \ldots k_{n+1} \bar{t}_i \bar{t}_j (k_1 \ldots k_{n+1} / k_j^1) k_j^1)$$

The idea behind the first equality is to capture and destroy the current contexts with $\beta_\eta$; when the throw is performed, only empty contexts are pushed on the context of level $i + 1$. In the second equation, the delimiter $\langle \rangle$ effectively pushes the current contexts up to level $i$ on the context of level $i + 1$, and the throw $\bar{t}_i$ 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 $\bar{t}_i$ from the type system of Figure 3 using the equation above, or directly from its CPS translation:

$$
C_i = S \Rightarrow C_{i+1} \\
\Gamma ; \Delta \vdash_n h_i : C_i \\
D'_i = S \Rightarrow C'_i \Rightarrow \ldots \Rightarrow C'_{n+1} \\
\Gamma ; \Delta \vdash_n t : D'_1, D_2, \ldots, D_{n+1} \\
\Gamma ; \Delta \vdash_n (h_1, \ldots, h_i) \bar{c}_i : D_1, D_2, \ldots, D_{n+1}
$$

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. More Flexible Control Operators

In this section we consider some variants of the operators introduced in Section 2. 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 $E_1, \ldots, E_j$, where $1 \leq i_1 < i_2 < \ldots < i_j \leq n$. The syntax of terms is now defined as follows:

$$
t ::= x | \lambda x. t | t \times t | L^b(k_1, \ldots, k_{i_j}).t | (t)_i | (h_1, \ldots, h_i) \bar{c}_i, t_i,
$$

the syntax of level-1 contexts is adjusted accordingly:

$$
E_i ::= \bullet_i | v E_i | E_i t \bar{c}_i, E_i
$$

and the remaining syntactic categories are defined as before. The CPS translation, reduction rules, and typing rules for the operators $L^b$ and $\bar{c}_i$ are summarized in Figure 5.

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 $p \rightarrow \nu p'$ 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 $\Gamma ; \Delta \vdash \bar{t} : C_1 \rightarrow \ldots \rightarrow C_{n+1} \rightarrow o.$

2. If $\Gamma ; \Delta \vdash_n E_i : C_i$, then $\Gamma ; \Delta \vdash [E_i] : C_i$, for all $i = 1, \ldots, n$.

Proposition 6 (Soundness of Reduction wrt. CPS). If $p \rightarrow \nu p'$, then $p = \beta_\eta \bar{p}$.

Theorem 5 (Termination of Evaluation). Let $t$ be a closed plain term such that $\vdash_n t : C_i, \ldots, C_{n+1}$ and $\bar{t}_i(C_i)$ hold for all $i = 1, \ldots, n+1$. Then $N((t, \bullet_1, \ldots, \bullet_{n+1}))$ holds.
CPS translation

\[ C^\ast (k_1', \ldots, k_i').t = \lambda k_1 \ldots k_{n+1}. t(k_1'/k_1, \ldots, k_i'/k_i) c_1 \ldots c_i k_{i+1} \ldots k_{n+1} \]

where \( c_l = \begin{cases} \emptyset & \text{if } j \in \{i_1, \ldots, i_l\} \\ k_i & \text{otherwise} \end{cases} \) for all \( 1 \leq l \leq i_j \)

\[ (h_1 \ldots h_{i_j}) \overset{\cdot}{\mapsto} t = \lambda k_1 \ldots k_{n+1}. t(\lambda v_0 k_2' \ldots k_{n+1}'[h_1] v_0 d_1 + \ldots + d_{i_j} (\lambda v_1 k_1' + k_2' \ldots k_{i_j+1}' k_{i_j+2} \ldots k_{n+1}')) \]

\( k_1' \ldots k_{i_j}' k_{i_j+2} \ldots k_{n+1}' \) \( k_2 \ldots k_{n+1} \)

where \( d_l = \begin{cases} [h_l] & \text{if } j \in \{i_1, \ldots, i_l\} \\ \emptyset & \text{otherwise} \end{cases} \) for all \( i_1 \leq l \leq i_j \)

Reduction rules

\[(\text{capture}) \quad \langle C^\ast (k_1, \ldots, k_{i_j}), t, E_1, \ldots, E_{n+1} \rangle \rightarrow_v \langle t(E_1 / k_1, \ldots, E_{i_j} / k_{i_j}), E_1', \ldots, E_{i_j}', E_{i_j+1}, \ldots, E_{n+1} \rangle \]

where \( E_l' = \begin{cases} \emptyset & \text{if } j \in \{i_1, \ldots, i_l\} \\ E_l & \text{otherwise} \end{cases} \) for all \( 1 \leq l \leq i_j \)

\[(\text{throw}) \quad \langle E_1', \ldots, E_{i_j}' \rangle \overset{\cdot}{\mapsto} v, E_1, \ldots, E_{n+1} \rightarrow_v \langle v, E_1'', \ldots, E_{i_j}'', E_{i_j+1}, (E_{i_j+1} \ldots (E_2, E_1), \ldots, E_{i_j+2}, \ldots, E_{n+1}) \rangle \]

where \( E_l'' = \begin{cases} E_l' & \text{if } j \in \{i_1, \ldots, i_l\} \\ \emptyset & \text{otherwise} \end{cases} \) for all \( i_1 \leq l \leq i_j \)

Typing rules

\[ \Pi_l(D_l) \text{ if } l \in \{i_1, \ldots, i_j\} \quad D_l = C_l \text{ if } l \notin \{i_1, \ldots, i_j\} \quad \Pi_l, \Delta, k_1 : C_1 \ldots k_{i_j} : C_{i_j} \vdash_n t : D_1, \ldots, D_{i_j}, C_{i_j+1} \ldots C_{n+1} \]

\[ \Pi_l, \Delta \vdash_n C(k_1, \ldots, k_j).t : C_1, C_2, \ldots, C_{n+1} \]

\[ C_{i_j} = S \triangleright C_{i_1+1} \triangleright \ldots \triangleright C_{n+1} \quad C_{i_j+1} = T \triangleright C_{i_j+2} \triangleright \ldots \triangleright C_{n+1} \]

\[ \Pi_l, \Delta \vdash_n h_{i_j} : C_{i_j} \quad \Pi_l, \Delta \vdash_n h_{i_j} : C_{i_j} \]

\[ \Pi_l, \Delta \vdash_n (h_1 \ldots h_{i_j}) \overset{\cdot}{\mapsto} t : T \triangleright C_2 \triangleright \ldots \triangleright C_{i_j+1} \triangleright C_{i_j+2} \triangleright \ldots \triangleright C_{n+1}, D_2, \ldots, D_{n+1} \]

\[ \Pi_l, \Delta \vdash_n (h_1 \ldots h_{i_j}) \overset{\cdot}{\mapsto} t : S \triangleright C_2 \triangleright \ldots \triangleright C_{i_j+1} \triangleright C_{i_j+2} \triangleright \ldots \triangleright C_{n+1}, D_2, \ldots, D_{n+1} \]

Figure 5: CPS translation, reduction rules, and typing rules for \( C^\ast \) and \( \overset{\cdot}{\mapsto} \)

Expressiveness. We first show how to simulate the operators \( \Sigma \) and \( \overset{\cdot}{\mapsto} \) with \( S \) and \( \overset{\cdot}{\mapsto} \). We use the same translation as in Section 2.5, except that we translate \( \overset{\cdot}{\mapsto} \) in the following way:

\[ ((h_1 \ldots h_{i_j}) \overset{\cdot}{\mapsto} t)^\ast = \]

\[ (Ax(K_j \overset{\cdot}{\mapsto} S_j \ldots (K_2 \overset{\cdot}{\mapsto} S_2 (K_1 \overset{\cdot}{\mapsto} S_1 x_1)_{1 \ldots j})_{1 \ldots j})) t^\ast \]

with \( K_j = h_j^i \) if \( i \in \{i_1, \ldots, i_j\} \) and \( K_j = (\emptyset, \ldots, \emptyset) \) otherwise. In the translation, if \( i \notin \{h_1 \ldots h_{i_j}\} \), then we restore the empty context \( \emptyset \) as the current context of level \( l \) by throwing to \( \emptyset \) (in fact to the tuple \( \emptyset, \ldots, \emptyset \)), but only the last context matters. Otherwise, we throw to \( h_j^i \), as in the translation for \( \overset{\cdot}{\mapsto} \). The translation of the \( \Sigma \) remains unchanged, we still throw to tuples of contexts captured by a singular shift, as required by the semantics of shift. Expressing \( \Sigma \) with \( S \) seems more difficult, because \( \Sigma \) may capture some contexts and leave the first one unchanged, while \( S \) always captures a tuple of contexts, starting from the first one. We conjecture that \( \Sigma^\ast \) cannot be expressed with \( S \). As in Section 2.5, we may consider an alternative throw operator \( \overset{\cdot}{\mapsto} \), such that \( (E_1, \ldots, E_{i_j}) \overset{\cdot}{\mapsto} v \) replaces the current contexts at positions \( i_1, \ldots, i_j \) by \( E_1, \ldots, E_{i_j} \) and leaves the other ones unchanged. We define this operator via its CPS translation:

\[ (h_1 \ldots h_{i_j}, t) \overset{\cdot}{\mapsto} t = \lambda k_1 \ldots k_{n+1}. \]

\[ (\lambda v_0 k_2' \ldots k_{n+1}'[h_1] v_0 c_1 v_2 \ldots c_{i_j} k_{i_j+1} k_{i_j+2} \ldots k_{n+1}') k_2 \ldots k_{n+1} \]

where \( c_i = [h_i] \) if \( i \in \{i_1, \ldots, i_j\} \) and \( c_i = k_i \) otherwise. The corresponding reduction rule is:

\[ ((E_1', \ldots, E_{i_j}') \overset{\cdot}{\mapsto} v, E_1, \ldots, E_{n+1}) \rightarrow_v \]

\[ (v, E_1'', \ldots, E_{i_j}'', E_{i_j+1}, \ldots, E_{n+1}) \]

where \( E''_i = E'_i \) if \( i \in \{i_1, \ldots, i_j\} \) and \( E''_i = E_i \) otherwise. We can express \( \overset{\cdot}{\mapsto} \) with \( \overset{\cdot}{\mapsto} \) as follows:

\[ (h_1 \ldots h_{i_j}, t) \overset{\cdot}{\mapsto} t = (Ax.(h_1 \ldots h_{i_j}) \overset{\cdot}{\mapsto} x_{i_j}) t \]

Roughly, the delimiter \( (\cdot) \) pushes the current contexts on
$E_{i+1}$, and the captured contexts are then restored with $\xi$. However, simulating $\xi\xi$ with $\xi$ seems difficult, mainly because $(E_{i_1}, \ldots, E_{i_j}) \xi\xi v$ leaves contexts $E_l$ such that $l \leq i_j$ and $l \not\in \{i_1, \ldots, i_j\}$ unchanged, while $(E_{i_1}, \ldots, E_{i_j}) \xi v$ replaces them with $\ast$. We conjecture that $\xi\xi\xi$ cannot be expressed with $\xi\xi$.

4. 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 that it can lead to new applications, implementations as well as theoretical results on this topic.

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].

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 iso-

morphism carries over from shift and reset to the hierarchy.

5. REFERENCES

version was presented at the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1991).


