A Call-by-Name CPS Hierarchy

Asami Tanaka and Yukiyoshi Kameyama

University of Tsukuba, Japan
asami@logic.cs.tsukuba.ac.jp,kameyama@acm.org

Abstract. The Continuation-Passing-Style (CPS) translation gives semantics to control operators such as exception and first-class continuations. By iterating this translation, Danvy and Filinski obtained a CPS hierarchy, and used it to specify a series of control operators, hierarchical (or layered) delimited-control operators,
We introduce a call-by-name variant of the CPS hierarchy. While most of the work on delimited-control operators is based on call-by-value calculi, call-by-name delimited-control operators are an active target of recent studies. Our strategy for developing such a hierarchy is to use the results for the call-by-value calculi as much as possible. The key tool is Hatcliff and Danvy's factorization of Plotkin's call-by-name CPS translation into a thunk translation and a call-by-value CPS translation. We show that a call-by-name CPS hierarchy can be obtained by naturally extending the factorization to the calculi with control operators, and then prove several properties for this hierarchy.

1 Introduction

Translating terms into Continuation Passing Style (CPS) is a key to define and understand control operators such as first-class continuations (Scheme's `call/cc`). By iterating the CPS translation, Danvy and Filinski [5] have obtained a hierarchy for CPS translations, and found a series of control operators indexed by natural numbers. Among the series, the first one corresponds to the delimited-control operators `shift` and `reset`, which allows to capture a *delimited continuation*, part of the rest of computation. In the last two decades, these control operators have found many applications such as partial evaluation [14], CPS translations [5], mobile computing [20], and dependently typed programming such as type-safe `printf` [1].

The second and the rest of the series correspond to the higher-level versions of `shift` and `reset`, similar to layered monads [6], and are useful when we combine two or more computational effects in a single program [4]. Although the hierarchical (layered) control operators are less expressive than the arbitrarily nested control operators [13], the former can express many interesting programs, and also the existence of the purely functional CPS transform for the former is beneficial for studying its semantics and foundational issues.

While the above mentioned work has been done for call-by-value calculi, several authors have recently studied delimited-control operators in the call-by-name calculi. Herbelin and Ghilezan [8] and Saurin [19] studied variants of

Parigot's $\lambda\mu$-calculus and interpreted their computational meaning by call-by-name delimited-control operators. Kiselyov [12] proposed a call-by-name calculus with delimited-control operators and used it in linguistic analysis. Biernacka and Biernacki [3] studied both call-by-value and call-by-name calculi for delimited-control operators in a uniform way. In our previous work [11], we gave a complete equational axiomatization for the call-by-name calculi with delimited-control operators.

In this paper, we introduce the call-by-name variant of the CPS hierarchy, in which we can find a series of delimited-control operators. The key tool we use is Hatcliff and Danvy's factorization of a CPS translation [7], which connects Plotkin's call-by-name and call-by-value CPS translations by a thunk translation. They have shown that all Plotkin's criteria for correctness can be established for the thunk translation. We use their methodology in a slightly different way: rather than defining the call-by-name CPS hierarchy independently from the call-by-value one and connect the two by the thunk translation, we use the thunk translation to *define*, or *derive*, the call-by-name CPS hierarchy from the call-by-value one. The effectiveness of our method is supported by the fact that Biernacka and Biernacki's call-by-name calculus for `shift` and `reset` [3] can be obtained by simply taking the first level of our hierarchy. Note that the thunk translation is much less complicated than the CPS translation for delimited-control operators, and we can obtain the CPS hierarchy rather smoothly.

The contribution of this paper is summarized as follows: (1) we extend Hatcliff and Danvy's thunk translation to the calculi with delimited-control operators, (2) we introduce a typed call-by-name CPS hierarchy, and (3) we obtain several properties of the hierarchy by the connection made by the thunk translation.

The rest of this paper is organized as follows. Section 2 gives informal explanation for the background of this work. Section 3 formally introduces the call-by-name calculus in a CPS hierarchy and a thunk translation. In Section 4, we derive a type system for our calculus and prove basic properties. Section 5 gives an equational theory for our calculus. Section 6 compares our work to others and state concluding remarks.

2 Preliminaries

2.1 Delimited-Control Operators and a CPS Hierarchy

Explicit manipulation of continuations allows various programming styles. Unlike the standard (unlimited) continuation, a delimited continuation represents part of the rest of computation, and delimited-control operators provide an access to delimited continuations.

We informally explain delimited control-operators in a call-by-value calculus using the following examples:

$$\langle 10 + \mathcal{S}k.(k \hookleftarrow (k \hookleftarrow 5))\rangle \rightsquigarrow \langle\langle 10 + \langle 10 + 5\rangle\rangle\rangle \rightsquigarrow^* 25$$
$$\langle 20 + \langle 10 + \mathcal{S}_1 k.(k \hookleftarrow_1 (k \hookleftarrow_1 5))\rangle_1\rangle_2 \rightsquigarrow \langle 20 + \langle\langle 10 + \langle 10 + 5\rangle_1\rangle_1\rangle_1\rangle_2 \rightsquigarrow^* 45$$
$$\langle 20 + \langle 10 + \mathcal{S}_2 k.(k \hookleftarrow_2 (k \hookleftarrow_2 5))\rangle_1\rangle_2 \rightsquigarrow \langle\langle 20 + \langle 10 + \langle 20 + \langle 10 + 5\rangle_1\rangle_2\rangle_1\rangle_2\rangle_2$$
$$\rightsquigarrow^* 65$$
$$\langle 20 + \langle 10 + \mathcal{S}_1 k.(k \hookleftarrow_1 (k \hookleftarrow_1 5))\rangle_2\rangle_1 \rightsquigarrow \langle 20 + \langle\langle 10 + \langle 10 + 5\rangle_1\rangle_1\rangle_2\rangle_1 \rightsquigarrow^* 45$$

In these examples, $\rightsquigarrow$ denotes a one-step reduction, and $\rightsquigarrow^*$ denotes a many-step reduction. $\mathcal{S}k.e$ is a `shift`-term and $\langle e\rangle$ is a `reset`-term. The `shift`-term captures the evaluation context up to the nearest `reset`, namely, a continuation delimited by a `reset`. In the first example, the (delimited) continuation is $\langle 10 + [\ ]\rangle$, which is bound to k. The captured continuation is used in the subterm $k \hookleftarrow e$. Note that `shift` eliminates the current delimited continuation, and that the `throw`-term is not abortive, unlike the continuation captured by the `call/cc` primitive in Scheme. After the first step, there remains no `shift`, and, therefore, the occurrences of `reset` are ignored.

We cannot mix two or more uses of `shift` and `reset` in one program if they do not have names to distinguish one from the other. To avoid unwanted interference between different uses of control operators, we attach natural numbers to each control operator as their indices. As the second and third examples show, `shift` chooses the nearest `reset` as the one with the same index as `shift`.

The above explanation is not completely precise: the index is not merely a name. Rather, they are linearly ordered (hence a natural number is used). In the fourth example, `shift` indexed by 1 chooses the nearest `reset` as the one indexed by 2, rather than the one by 1. In general, a higher-level `reset` delimits the continuation captured by a lower-level `shift`. Put differently, a lower-level `shift` cannot escape from a higher-level `reset`, thus control operators are layered or hierarchical.

2.2 The Thunk Translation

The thunk translation introduces a "thunk" (a lambda closure) to freeze a computation, and can be regarded as a translation from a call-by-name calculus to a call-by-value calculus. Hatcliff and Danvy [7] showed that Plotkin's call-by-name CPS translation can be factored into the thunk translation and Plotkin's call-by-value CPS translation, and that all Plotkin's correctness criteria [16] can be derived using this factorization. In this subsection, we briefly review their results to the extent that is necessary for this paper.

Fig. 1 gives the source and target calculi where c is a constant and v is a value in call-by-value. Λ denotes the set of terms defined in this figure. The reductions (β) and (β_v), resp., are the call-by-name and call-by-value β-reductions, resp. Bound and free variables are defined in the standard way, and we identify α-equivalent terms. The term $e_1\{x := e_2\}$ denotes the result of capture-avoiding substitution.

(term in Λ)	$e ::= c \mid x \mid \lambda x.e \mid ee$
(term in $\Lambda_{\mathtt{fd}}$)	$e ::= \cdots \mid \mathtt{force}(e) \mid \mathtt{delay}(e)$
(cbn-value)	$v ::= c \mid \lambda x.e$
(cbv-value)	$v ::= c \mid x \mid \lambda x.e$
(β)	$(\lambda x.e_1)e_2 \rightsquigarrow e_1\{x := e_2\}$
(β_v)	$(\lambda x.e_1)v \rightsquigarrow e_1\{x := v\}$
(fd)	$\mathtt{force}(\mathtt{delay}(e)) \rightsquigarrow e$

Fig. 1. Syntax and Reduction Rules of the Basic Calculus

$\mathcal{T}[\![c]\!] \stackrel{\text{def}}{=} c$	$\mathcal{T}[\![\lambda x.e]\!] \stackrel{\text{def}}{=} \lambda x.\mathcal{T}[\![e]\!]$
$\mathcal{T}[\![x]\!] \stackrel{\text{def}}{=} \mathtt{force}(x)$	$\mathcal{T}[\![e_1e_2]\!] \stackrel{\text{def}}{=} \mathcal{T}[\![e_1]\!]\ (\mathtt{delay}(\mathcal{T}[\![e_2]\!]))$

Fig. 2. Thunk Translation

The thunk translation gives a simulation of the call-by-name calculus in the call-by-value calculus in terms of the following two functions: (1) `delay` for creating a suspended computation as a value (thunk), and (2) `force` for re-invoking the suspended computation. For a term e, the term $\mathtt{delay}(e)$ is a value, and $\mathtt{force}(\mathtt{delay}(e))$ reduces to e. Although we can express $\mathtt{delay}(e)$ by $\lambda x.e$, and $\mathtt{force}(e)$ by ex, for a fresh variable x, it is convenient to distinguish thunks from the ordinary lambda abstractions. $\Lambda_{\mathtt{fd}}$ denotes the set of terms with `delay` and `force`.

The thunk translation is a syntactic translation from Λ to $\Lambda_{\mathtt{fd}}$ defined in Fig. 2. It is easy to see that a one-step β-reduction in the source calculus one-to-one corresponds, by the thunk translation, to a one-step β-reduction in the target calculus modulo the (fd) reduction. Also, since all the arguments of functions in $\Lambda_{\mathtt{fd}}$ are values, the reductions (β) and (β_v) coincide on $\Lambda_{\mathtt{fd}}$.

Theorem 1 (Simulation [Hatcliff and Danvy]). *Let e_1 and e_2 be terms in Λ. We have that e_1 reduces to e_2 by the (β) reduction if and only if $\mathcal{T}[\![e_1]\!]$ reduces to $\mathcal{T}[\![e_2]\!]$ by the (β_v) reduction followed by the (fd) reductions. Moreover, (β_v) may be replaced by (β) in the above sentence.*

We consider equality over terms induced by a set of reduction rules $r_1, \cdots, r_n$, which is defined as the least congruence relation that subsumes $r_1, \cdots, r_n$. We write $(r_1, \cdots, r_n) \vdash e_1 = e_2$ if $e_1 = e_2$ holds under the equality induced by $r_1, \cdots, r_n$. For instance, $(\beta) \vdash e_1 = e_2$ means that e_1 and e_2 are equal under β equality.

Fig. 3 defines the call-by-value and call-by-name CPS transformations due to Plotkin [16], where the variables κ, m, and m' are fresh.

$$\mathcal{C}^n[\![c]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa c \qquad \mathcal{C}^v[\![c]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa c$$

$$\mathcal{C}^n[\![x]\!] \stackrel{\text{def}}{=} \lambda\kappa.x\kappa \qquad \mathcal{C}^v[\![x]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa x$$

$$\mathcal{C}^n[\![\lambda x.e]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa(\lambda x.\mathcal{C}^n[\![e]\!]) \qquad \mathcal{C}^v[\![\lambda x.e]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa(\lambda x.\mathcal{C}^v[\![e]\!])$$

$$\mathcal{C}^n[\![e_1\ e_2]\!] \stackrel{\text{def}}{=} \lambda\kappa.\mathcal{C}^n[\![e_1]\!](\lambda m.m\mathcal{C}^n[\![e_2]\!]\kappa) \qquad \mathcal{C}^v[\![e_1\ e_2]\!] \stackrel{\text{def}}{=} \lambda\kappa.\mathcal{C}^v[\![e_1]\!](\lambda m.\mathcal{C}^v[\![e_2]\!](\lambda m'.mm'\kappa))$$

$$\mathcal{C}^v[\![\texttt{force}(e)]\!] \stackrel{\text{def}}{=} \lambda\kappa.\mathcal{C}^v[\![e]\!](\lambda m.m\kappa)$$

$$\mathcal{C}^v[\![\texttt{delay}(e)]\!] \stackrel{\text{def}}{=} \lambda\kappa.\kappa\mathcal{C}^v[\![e]\!]$$

Fig. 3. Call-by-Name (left) and Call-by-Value (right) CPS Translations

The CPS translations in Fig. 3 are standard except the cases for `delay` and `force`. The terms $\texttt{force}(e)$ and $\texttt{delay}(e)$ are intuitively understood as $e\ x$ and $\lambda x.e$ for a fresh variable x. Then $\mathcal{C}^v[\![\texttt{force}(e)]\!]$ is understood as $\mathcal{C}^v[\![ex]\!] = \lambda\kappa.\mathcal{C}^v[\![e]\!](\lambda m.\mathcal{C}^v[\![x]\!](\lambda n.m\ n\ k))$, which reduces to $\lambda\kappa.\mathcal{C}^v[\![e]\!](\lambda m.m\ x\ k))$. We also understand $\mathcal{C}^v[\![\texttt{delay}(e)]\!]$ as $\mathcal{C}^v[\![\lambda x.e]\!]$, which reduces to $\lambda\kappa.\kappa(\lambda x.\mathcal{C}^v[\![e]\!])$. Since x is useless, we omit it in both terms, and obtain the translation in Fig. 3.

Hatcliff and Danvy proved the following key theorem.

Theorem 2 (Factorization [Hatcliff and Danvy]). *For any term e in Λ, we have $(\beta) \vdash \mathcal{C}^n[\![e]\!] = \mathcal{C}^v[\![\mathcal{T}[\![e]\!]]\!]$.*

Note that η-equality (and η-reduction) is not preserved by the thunk translation: we have $\mathcal{T}[\![\lambda x.\ y\ x]\!] = \lambda x.\texttt{force}(y)\ (\texttt{delay}(\texttt{force}(x)))$ and $\mathcal{T}[\![y]\!] = \texttt{force}(y)$. In order to equate these terms, we need $\texttt{delay}(\texttt{force}(x)) = x$ and $\lambda x.\texttt{force}(y)x = \texttt{force}(y)$, and the latter subsumes full η-equality, but it is not admissible in the target of the thunk translation (a call-by-value calculus).

3 The Calculi: Syntax and Reduction Rules

We introduce the calculi $\lambda^n_{s/r}$ for a call-by-name CPS hierarchy where n is a natural number which denotes an upper bound of indices (or levels). In other words, the indices of `shift` and `reset` must be equal to or less than n. We fix this n throughout this paper.

We assume that there are two disjoint sets of variables, one for ordinary variables (ranged over by $x, y, z, \cdots$) and the other for continuation variables (ranged over by k). An ordinary variable is bound by lambda, and a term may be substituted for it, while a continuation variable is bound by `shift`, and a delimited continuation may be substituted for it. We also assume that each continuation variable k is (implicitly) annotated by a level i (for $1 \leq i \leq n$), and that $\mathcal{S}_i k.e$ and $k \hookleftarrow_i e$ are terms only when the (implicit) level of k is i.

Fig. 4 defines the syntax of the calculi with delimited-control operators where c is a basic constant. The term $\mathcal{S}_i k.e$ is a `shift`-term of level i, in which k is

$(\Lambda_{\mathtt{sr}})$	$e ::= c \mid x \mid \lambda x.e \mid ee \mid \mathcal{S}_i k.e \mid \langle e \rangle_i \mid k \hookleftarrow_i e$	where $1 \leq i \leq n$
$(\Lambda_{\mathtt{sr,fd}})$	$e ::= \cdots \mid \mathtt{force}(e) \mid \mathtt{delay}(e)$	

Fig. 4. Terms with Delimited-Control Operators

(cbv-value)	$v ::= c \mid x \mid \lambda x.e \mid \mathtt{delay}(e)$	
(cbv-context)	$E^i ::= [\ ] \mid E^i e \mid v E^i \mid \mathtt{force}(E^i) \mid \langle E^i \rangle_h$	where $h < i$
(β_v)	$(\lambda x.e)v \rightsquigarrow e\{x := v\}$	
(fd)	$\mathtt{force}(\mathtt{delay}(e)) \rightsquigarrow e$	
(rv_v)	$\langle v \rangle_i \rightsquigarrow v$	
(rs_v)	$\langle E^i[\mathcal{S}_i k.e] \rangle_j \rightsquigarrow \langle e\{k \Leftarrow_i \langle E^i \rangle_i\} \rangle_j$	where $i \leq j$

Fig. 5. Call-by-Value Reductions

bound. The term $\langle e \rangle_i$ is a `reset`-term of level i. The term $k \hookleftarrow_i e$ is a `throw`-term of level i which applies k to the term e. Note that the continuation variable k is free in $k \hookleftarrow_i e$.

Following Biernacka and Biernacki [3], we explicitly distinguish lambda-bound (ordinary) variables from `shift`-bound (continuation) variables. This distinction is important to get simpler definitions, and is necessary to give an axiomatization for call-by-name calculus [11].

We identify α-equivalent terms. $\mathtt{FV}(e)$ and $\mathtt{FCV}(e)$, resp., denote the set of free ordinary variables and the set of free continuation variables, resp., in e, and $e_1\{x := e_2\}$ represents the result of capture-avoiding substitution of e_2 for x in e_1. E^i is an evaluation context which does not have a level-i or higher `reset` enclosing the hole E^1 is a pure evaluation context, or a delimited continuation, which does not have `reset` around the hole. We write E instead of E^i if the index does not matter. $E[e]$ denotes the term after the hole-filling operation of a term e for a hole in E. Hole-filling of an evaluation context $E_1[E_2]$ is defined similarly.

Fig. 5 gives the reduction rules for the call-by-value calculi, which are essentially due to Biernacka and Biernacki [3] with the following difference. While their calculus reifies an evaluation context E^i, and substitute it for a continuation variable k (thus $k \hookleftarrow_i e$ becomes $E^i \hookleftarrow_i e'$), our calculus uses structural substitution $\{k \Leftarrow_i E^i\}$ defined in Fig. 6.

The thunk translation is naturally extended to this calculus as in Fig. 7.

Fig. 8 defines the reduction rules of the call-by-name calculus. The reduction (β) is standard. The reduction (rv_n) (meaning "reset-value") eliminates a `reset` if its body is a value. The only interesting reduction is (rs_n) (meaning "reset-shift") for a level-i `shift`-term. By this reduction, `shift` captures a level-i (delimited) evaluation context E^i, and substitutes it for k. The corresponding

$$
\begin{aligned}
c\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} c \\
x\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} x \\
(\lambda x.e)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} \lambda x.(e\{k \Leftarrow_i E\}) \quad \text{where } x \notin \mathtt{FV}(E) \\
(e_1\ e_2)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} (e_1\{k \Leftarrow_i E\})\ (e_2\{k \Leftarrow_i E\}) \\
(\mathcal{S}_p k'.e)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} \mathcal{S}_p k'.(e\{k \Leftarrow_i E\}) \quad \text{where } k' \notin \{k\} \cup \mathtt{FCV}(E) \\
(k \hookleftarrow_i e)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} E[e\{k \Leftarrow_i E\}] \\
(k' \hookleftarrow_p e)\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} k' \hookleftarrow_p (e\{k \Leftarrow_i E\}) \quad \text{where } k' \neq k \\
\langle e \rangle_p\{k \Leftarrow_i E\} &\stackrel{\text{def}}{=} \langle e\{k \Leftarrow_i E\} \rangle_p
\end{aligned}
$$

Fig. 6. Substitution for Continuation Variables

$$
\begin{aligned}
\mathcal{T}[\![e]\!] &\stackrel{\text{def}}{=} \cdots \text{for } e = c,\ x,\ \lambda x.e,\ e_1 e_2 & \mathcal{T}[\![\langle e \rangle_i]\!] &\stackrel{\text{def}}{=} \langle \mathcal{T}[\![e]\!] \rangle_i \\
\mathcal{T}[\![\mathcal{S}_i k.e]\!] &\stackrel{\text{def}}{=} \mathcal{S}_i k.\mathcal{T}[\![e]\!] & \mathcal{T}[\![k \hookleftarrow_i e]\!] &\stackrel{\text{def}}{=} k \hookleftarrow_i \mathcal{T}[\![e]\!]
\end{aligned}
$$

Fig. 7. Thunk Translation

`reset` for this `shift` is the nearest one which has the level i or higher, because E^i does not have such a `reset` that encloses the hole.

The reduction rules in both calculi are extended to arbitrary contexts as $e_1 \rightsquigarrow e_2$ implies $C[e_1] \rightsquigarrow C[e_2]$ for any context C. We write $(r_1, \cdots, r_i) \vdash e_1 \rightsquigarrow e_2$ if e_1 reduces to e_2 by these reductions. We also write $\rightsquigarrow^*$ for zero or more step reductions.

We can show that the notions of the reductions in call-by-name/call-by-value calculi correspond to each other via the thunk translation.

Theorem 3 (Simulation). *Let e_1 and e_2 be terms in Λ_{sr}. Then we have $(\beta, rv_n, rs_n) \vdash e_1 \rightsquigarrow^* e_2$ if and only if $(\beta, fd, rv_v, rs_v) \vdash \mathcal{T}[\![e_1]\!] \rightsquigarrow^* \mathcal{T}[\![e_2]\!]$.*

Proof. From call-by-name to call-by-value, the proof is straightforward.

For the inverse direction, we only have to consider the image of the thunk translation, namely, for any application $e_1 e_2$, the term e_2 has the form $\mathtt{delay}(e_3)$, which is a value. Then it is not difficult to prove the inverse direction.

4 Type System

The thunk translation is not only useful to investigate the operational aspect, but can be also used to design a type system. As a concrete instance, we design a type system for the call-by-name calculus with delimited-control operators from the one for the call-by-value calculus. In this process, we do not have to

(cbn-value)	$v ::= c \mid \lambda x.e$	
(cbn-context)	$E^i ::= [\ ] \mid E^i e \mid \langle E^i \rangle_h$	where $h < i$
(β)	$(\lambda x.e_1)e_2 \rightsquigarrow e_1\{x := e_2\}$	
(rv_n)	$\langle v \rangle_i \rightsquigarrow v$	
(rs_n)	$\langle E^i[\mathcal{S}_i k.e] \rangle_j \rightsquigarrow \langle e\{k \Leftarrow_i \langle E^i \rangle_i\} \rangle_j$	where $i \leq j$

Fig. 8. Call-by-Name Reductions

consult with the CPS translation. Although the hierarchical control operators are complicated, and in particular, a CPS translation for them is hard to understand, we can give the type system quite smoothly thanks to Hatcliff and Danvy's factorization.

Biernacka and Biernacki [3] proposed a call-by-name typed calculus with the level-1 `shift` and `reset`. While they developed the calculus and its properties independently from the call-by-value counterpart, they can be *derived* from the call-by-value counterpart using the thunk translation. In this section, we show that it is possible for the calculus of an arbitrarily higher level.

4.1 Type System for Call-by-Value CPS Hierarchy

We review a monomorphic type system for the call-by-value calculus with hierarchical delimited-control operators.

Murthy [15] was the first to give a type system for this calculus, which was derived from the CPS translation. The basic idea of his type system is that the typing judgment of a term e should carry the same information as its CPS image $\mathcal{C}^v[\![e]\!]$. For instance, if $n = 1$, namely, we have only one level, the CPS image of a term takes the form $\lambda\kappa_1.\lambda\kappa_2.\cdots$, which suggests the type of the CPS image is $(\sigma \to (\tau \to *) \to *) \to (\rho \to *) \to *$. Here we assume that the images of the CPS translation are in (strictly) continuation-passing style, so the (final) answer type is polymorphic [21, 22], and we write it as an anonymous type $*$. In this CPS type, σ is the type corresponding to the source term, τ and ρ are so called answer types of level 1. Murthy further assumed that, the answer types do not change, which means that τ and ρ in the above type must be the same. This assumption greatly simplifies the type system, and we only need n answer types for any term if the maximum level is n.

Answer types may be regarded as computational effects, and therefore we need to modify a function type $\sigma \to \tau$ to an effectful function type $\mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}]$, whose inhabitants are functions from σ to τ that works under the answer types $\overline{\alpha}$. Here $\overline{\alpha}$ is a sequence of types $\alpha_1, \alpha_2, \cdots, \alpha_n$.

Let us formally define the type system. The syntax of types is given by:

$$\text{(type)} \quad \sigma, \tau, \alpha ::= b \mid \mathrm{Susp}[\sigma/\overline{\alpha}] \mid \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}]$$

$$\frac{}{\Gamma, x:\sigma \vdash^{cbv} x:\sigma \mid \overline{\alpha}}\ \text{var} \qquad \frac{(c \text{ is a constant of basic type } b)}{\Gamma \vdash^{cbv} c : b \mid \overline{\alpha}}\ \text{const}$$

$$\frac{\Gamma, x:\sigma \vdash^{cbv} e:\tau \mid \overline{\alpha}}{\Gamma \vdash^{cbv} \lambda x.e : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}] \mid \overline{\beta}}\ \text{fun}$$

$$\frac{\Gamma \vdash^{cbv} e_0 : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}] \mid \overline{\alpha} \quad \Gamma \vdash^{cbv} e_1 : \sigma \mid \overline{\alpha}}{\Gamma \vdash^{cbv} e_0 e_1 : \tau \mid \overline{\alpha}}\ \text{app}$$

$$\frac{\Gamma \vdash^{cbv} e : \sigma \mid \sigma, \cdots, \sigma, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} \langle e \rangle_i : \sigma \mid \overline{\alpha}}\ \text{reset}$$

$$\frac{\Gamma, k : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}] \vdash^{cbv} e : \tau \mid \tau, \cdots, \tau, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} \mathcal{S}_i k.e : \sigma \mid \beta_1, \cdots, \beta_{i-1}, \tau, \alpha_{i+1}, \cdots, \alpha_n}\ \text{shift}$$

$$\frac{\Gamma, k : \mathrm{Fun}^{\mathrm{cbv}}[\tau \to \sigma/\sigma, \cdots, \sigma, \alpha_{i+1}, \cdots, \alpha_n] \vdash^{cbv} e : \tau \mid \sigma, \cdots, \sigma, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} k \hookleftarrow_i e : \sigma \mid \overline{\alpha}}\ \text{throw}$$

$$\frac{\Gamma \vdash^{cbv} e : \mathrm{Susp}[\sigma/\overline{\alpha}] \mid \overline{\alpha}}{\Gamma \vdash^{cbv} \texttt{force}(e) : \sigma \mid \overline{\alpha}}\ \text{force} \qquad \frac{\Gamma \vdash^{cbv} e : \sigma \mid \overline{\alpha}}{\Gamma \vdash^{cbv} \texttt{delay}(e) : \mathrm{Susp}[\sigma/\overline{\alpha}] \mid \overline{\beta}}\ \text{delay}$$

Fig. 9. Type System for the Call-by-Value Calculus (First Version)

where b is a metavariable for basic types such as integer and boolean. The type $\mathrm{Susp}[\sigma/\overline{\alpha}]$ is the one for suspended computation generated by `delay`. The type $\mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}]$ is an effectful function type. A typing context Γ is a (possibly empty) sequence consisting of the form $x : \tau$ for a lambda-bound variable x, or $k : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}]$ for a continuation variable k. A judgment takes the form $\Gamma \vdash^{cbv} e : \tau \mid \overline{\alpha}$ where Γ is a typing context, τ is a type, $\overline{\alpha}$ is a sequence of types, and e is a term. The length of the sequence $\overline{\alpha}$ must be n.

Fig. 9 gives the type system for the call-by-value calculus.

The types for a variable, a constant, lambda, and application are standard if we take into account the answer types. The typing rule for `reset` means that, for a level-i `reset`, the answer type of the body e must be the same as the type of e itself. This reflects the hierarchical nature of this calculus: as a lower-level `shift` cannot escape from a higher-level `reset`, whenever there is a level-i `reset`, the answer types of lower levels must be identical. After the level-i `reset`, the answer types of these levels can be arbitrary types, so the types $\alpha_1, \cdots, \alpha_i$ can be chosen arbitrarily in the typing judgment of $\langle e \rangle_i$. The typing rule for the `shift`-term can be understood by noting the facts that $\mathcal{S}_i k.e$ has the same denotation as $\mathcal{S}_i k.\langle e \rangle_i$, and a level-$i$ `shift`-term captures a delimited continuation whose type is roughly a function from some type to the level-i answer type. As for the typing rule for `throw`, it is instructive to know $k \hookleftarrow_i e$ may be represented by $\langle k\ e \rangle_i$ if we forget the distinction between lambda-bound and `shift`-bound variables. The typing rules for `force` and `delay` can be understood by the following intuition: `force`(e) is ex and `delay`(e) is $\lambda x.e$ for a fresh variable x.

$$\frac{\Gamma, k : \mathrm{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n] \vdash^{cbv} e : \tau \mid \tau, \cdots, \tau, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} \mathcal{S}_i k.e : \sigma \mid \beta_1, \cdots, \beta_{i-1}, \tau, \alpha_{i+1}, \cdots, \alpha_n} \text{ shift}$$

$$\frac{\Gamma, k : \mathrm{Cont}^i[\tau \to \sigma/\alpha_{i+1}, \cdots, \alpha_n] \vdash^{cbv} e : \tau \mid \sigma, \cdots, \sigma, \alpha_{i+1}, \cdots, \alpha_n}{\Gamma \vdash^{cbv} k \hookleftarrow_i e : \sigma \mid \overline{\alpha}} \text{ throw}$$

Other typing rules are the same as Fig. 9.

Fig. 10. Type System for the Call-by-Value Calculus (Second Version)

4.2 Refining the Type System

We can refine the type system to the one in Fig. 10 where a continuation variable k has a distinguished continuation type $\mathrm{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n]$ (for $1 \leq i \leq n$), and $\sigma, \tau, \alpha_{i+1}, \cdots, \alpha_n$ are types.

The continuation variable is treated differently from the ordinary functions, and also its type does not have the information of answer types of level $\leq i$, which means that it is polymorphic over these answer types. The answer type polymorphism of continuations have been studied by several authors [21, 2, 3] for the level-1 (single level) delimited-control operators, and here we use it in higher levels.

Theorem 4 (Type Soundness). *(1) If $\Gamma \vdash^{cbv} e_1 : \tau \mid \overline{\alpha}$ is derivable, and $e_1 \rightsquigarrow^* e_2$, then $\Gamma \vdash^{cbv} e_2 : \tau \mid \overline{\alpha}$ is derivable.*
(2) If $\vdash^{cbv} \langle e_1 \rangle_n : \tau \mid \overline{\alpha}$ is derivable, then there exists a term e_2 such that $\langle e_1 \rangle_n \rightsquigarrow e_2$. Moreover, if e_2 is not a value, it must be $\langle e_2' \rangle_n$ for some term e_2'.

The second part (progress) takes an unusual form since the term $\langle e_1 \rangle_n$ has an outermost reset of the maximum level n. Having such a reset is necessary, since a term with "free shift" (such as $\mathcal{S}_1 k.e$) can be a closed and typable term, but it is not a value.

The first part of this theorem is essentially due to Murthy [15], and the proof of the second part is standard.

4.3 Type System for Call-by-Name CPS Hierarchy

We now derive a type system for the call-by-name calculus from the one for call-by-value calculus. Our design principle is to have the property: a term e is typable in the former if and only if its translation $\mathcal{T}[\![e]\!]$ is typable in the latter.

First, we consider the case $\mathcal{T}[\![x]\!] = \mathtt{force}(x)$ which can be typed (and is only typed) in the call-by-value calculus as:

$$\frac{\overline{\Gamma,\ x : \mathrm{Susp}[\sigma/\overline{\alpha}] \vdash^{cbv} x : \mathrm{Susp}[\sigma/\overline{\alpha}] \mid \overline{\alpha}}}{\Gamma,\ x : \mathrm{Susp}[\sigma/\overline{\alpha}] \vdash^{cbv} \mathtt{force}(x) : \sigma \mid \overline{\alpha}}$$

Hence, in the call-by-name calculus we should have the following typing rule:

$$\overline{\Gamma,\ x : \mathrm{Susp}[\sigma/\overline{\alpha}] \vdash^{cbn} x : \sigma \mid \overline{\alpha}}$$

$$\frac{}{\Gamma, x : (\sigma \mid \overline{\alpha}) \vdash^{cbn} x : \sigma \mid \overline{\alpha}}\ \text{var}$$

$$\frac{\Gamma, (x : \sigma \mid \overline{\alpha}) \vdash^{cbn} e : \tau \mid \overline{\beta}}{\Gamma \vdash^{cbn} \lambda x.e : \mathrm{Fun}^{\mathrm{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})] \mid \overline{\gamma}}\ \text{fun}$$

$$\frac{\Gamma \vdash^{cbn} e_0 : \mathrm{Fun}^{\mathrm{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})] \mid \overline{\beta} \quad \Gamma \vdash^{cbn} e_1 : \sigma \mid \overline{\alpha}}{\Gamma \vdash^{cbn} e_0 e_1 : \tau \mid \overline{\beta}}\ \text{app}$$

The typing rules for const, `shift`, `reset` and `throw` are the same as those in Fig. 10.

Fig. 11. Type System for Call-by-Name Calculus

For notational reasons, we will write $(\sigma \mid \overline{\alpha})$ for $\mathrm{Susp}[\sigma/\overline{\alpha}]$ in typing contexts. The change in the type of a variable affects the function type: the call-by-value type $\mathrm{Fun}^{\mathrm{cbv}}[\mathrm{Susp}[\sigma/\overline{\alpha}] \to \tau/\overline{\beta}]$ will be written as the call-by-name type $\mathrm{Fun}^{\mathrm{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})]$, and the typing rule for (fun) is changed accordingly.

For the case $\mathcal{T}[\![e_0\ e_1]\!] = \mathcal{T}[\![e_0]\!](\mathtt{delay}(\mathcal{T}[\![e_1]\!]))$, we have:

$$\frac{\Gamma \vdash^{cbv} \mathcal{T}[\![e_0]\!] : \mathrm{Fun}^{\mathrm{cbv}}[\sigma \to \tau/\overline{\alpha}] \mid \overline{\alpha} \quad \dfrac{\Gamma \vdash^{cbv} \mathcal{T}[\![e_1]\!] : \rho \mid \overline{\beta}}{\Gamma \vdash^{cbv} \mathtt{delay}(\mathcal{T}[\![e_1]\!]) : \sigma \mid \overline{\alpha}}}{\Gamma \vdash^{cbv} \mathcal{T}[\![e_0]\!](\mathtt{delay}(\mathcal{T}[\![e_1]\!])) : \tau \mid \overline{\alpha}}$$

where $\sigma = \mathrm{Susp}[\rho/\overline{\beta}]$. Hence we should have the following rule:

$$\frac{\Gamma \vdash^{cbn} e_0 : \mathrm{Fun}^{\mathrm{cbv}}[\mathrm{Susp}[\rho/\overline{\beta}] \to \tau/\overline{\alpha}] \mid \overline{\alpha} \quad \Gamma \vdash^{cbn} e_1 : \rho \mid \overline{\beta}}{\Gamma \vdash^{cbn} e_0\ e_1 : \tau \mid \overline{\alpha}}$$

These changes are all what we need to do for the call-by-name type system. In particular, since the thunk translation is homomorphic for control operators, no essential changes are necessary in the typing rules for them.

To summarize, the types are defined by:

$$\text{(type)}\quad \sigma, \tau, \alpha, \beta ::= b \mid \mathrm{Fun}^{\mathrm{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})]$$
$$\text{(cont-type)}\quad \phi ::= \mathrm{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n]$$

A typing context Γ is a finite sequence of the form $x : (\sigma \mid \overline{\alpha})$ or $k : \mathrm{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n]$. A judgment in this type system takes the form of $\Gamma \vdash^{cbn} e : \sigma \mid \overline{\alpha}$, and the call-by-name type system is given in Fig. 11.

Typability is preserved by the thunk translation. To state this property formally, we define the thunk-translation for types, typing contexts, and judgment as follows (the following definitions extend to sequences naturally):

$$\mathcal{T}[\![b]\!] \stackrel{\text{def}}{=} b$$
$$\mathcal{T}[\![\mathrm{Fun}^{\mathrm{cbn}}[(\sigma/\overline{\alpha}) \to (\tau/\overline{\beta})]]\!] \stackrel{\text{def}}{=} \mathrm{Fun}^{\mathrm{cbv}}[\mathrm{Susp}[\mathcal{T}[\![\sigma]\!]/\mathcal{T}[\![\overline{\alpha}]\!]] \to \mathcal{T}[\![\tau]\!]/\mathcal{T}[\![\overline{\beta}]\!]]$$
$$\mathcal{T}[\![x : (\sigma \mid \overline{\alpha})]\!] \stackrel{\text{def}}{=} x : \mathrm{Susp}[\mathcal{T}[\![\sigma]\!]/\mathcal{T}[\![\overline{\alpha}]\!]]$$
$$\mathcal{T}[\![k : \mathrm{Cont}^i[\sigma \to \tau/\alpha_{i+1}, \cdots, \alpha_n]]\!] \stackrel{\text{def}}{=} k : \mathrm{Cont}^i[\mathcal{T}[\![\sigma]\!] \to \mathcal{T}[\![\tau]\!]/\mathcal{T}[\![\alpha_{i+1}, \cdots, \alpha_n]\!]]$$

Then we can prove the following theorem easily.

Theorem 5 (Thunk Translation Preserves Typability). *We have that $\Gamma \vdash^{cbn} e : \tau \mid \overline{\alpha}$ is derivable if and only if $\mathcal{T}[\![\Gamma]\!] \vdash^{cbv} \mathcal{T}[\![e]\!] : \mathcal{T}[\![\tau]\!] \mid \mathcal{T}[\![\overline{\alpha}]\!]$ is derivable.*

Combining the above theorem with the property of the call-by-value CPS translation, we obtain that the call-by-name CPS translation preserves typability.

Subject reduction property can be also derived easily.

Theorem 6 (Subject Reduction). *If $\Gamma \vdash^{cbn} e_1 : \tau \mid \overline{\alpha}$ is derivable, and $e_1 \rightsquigarrow^* e_2$, then $\Gamma \vdash^{cbn} e_2 : \tau \mid \overline{\alpha}$ is derivable.*

Proof. Suppose $\Gamma \vdash^{cbn} e_1 : \tau \mid \overline{\alpha}$ is derivable, and $e_1 \rightsquigarrow^* e_2$ in the call-by-name calculus. By Theorem 3, we have $\mathcal{T}[\![e_1]\!] \rightsquigarrow^* \mathcal{T}[\![e_2]\!]$. By the subject property of the call-by-value calculus and Theorem 5, we have $\mathcal{T}[\![\Gamma]\!] \vdash^{cbv} \mathcal{T}[\![e_2]\!] : \mathcal{T}[\![\tau]\!] \mid \mathcal{T}[\![\overline{\alpha}]\!]$. Again by Theorem 5, we have $\Gamma \vdash^{cbn} e_2 : \tau \mid \overline{\alpha}$.

We also have the progress property for the call-by-name calculus.

Theorem 7 (Progress). *If $\vdash^{cbn} \langle e_1 \rangle_n : \tau \mid \overline{\alpha}$ is derivable, there exists a term e_2 such that $\langle e_1 \rangle_n \rightsquigarrow e_2$. Moreover, if e_2 is not a value, it must be $\langle e_2' \rangle_n$ for some term e_2'.*

Proofs of these theorems are straightforward and omitted.

We have derived a type system for the call-by-name CPS hierarchy. Note that, by taking $n = 1$, we can reproduce Biernacka and Biernacki's call-by-name type system [3] modulo notational difference. A merit of our approach is that we do not have to directly consult the iterated CPS translation.

5 Equational Theory

This section studies an equational theory for the call-by-name CPS hierarchy in the typed setting. Sabry and Felleisen [17] first established equational axiomatization of the calculus with control operators for first-class (unlimited) continuations. Regarding delimited-control operators, Kameyama and Hasegawa [10] and Kameyama [9], resp., axiomatized level-1 and higher-level, resp, `shift` and `reset` in the call-by-value calculi. For reference, Fig. 13 in the appendix lists the latter axiomatization.

We obtain an equational theory for the call-by-name calculus in the same spirit as the previous sections: we formulate them as a back image of the thunk translation and the call-by-value counterpart. The result is given in Fig. 12.

The call-by-name axioms (Fig. 12) and the call-by-value axioms (Fig. 13) have several differences. It should also be noted that, even if the axioms rs_v and rs_n have the same form, they have different meaning as the definitions of the evaluation contexts differ from each other.

$$(\lambda x.e_1)\ e_2 = e_1\{x := e_2\} \qquad (\beta)$$

$$\langle E^i[\mathcal{S}_i k.e]\rangle_j = \langle e\{k \Leftarrow \langle E^i\rangle_i\}\rangle_j \quad \text{where} \quad i \le j \qquad (\mathrm{rs}_n)$$

$$k' \hookleftarrow_j (E^i[\mathcal{S}_i k.e]) = \langle e\{k \Leftarrow (k' \hookleftarrow_j E^i)\}\rangle_j \quad \text{where} \quad i \le j \text{ and } k \neq k' \qquad (\mathrm{ts})$$

$$\langle v\rangle_i = v \qquad (\mathrm{rv}_n)$$

$$\mathcal{S}_i k.(k \hookleftarrow_i \langle e\rangle_{i-1}) = \langle e\rangle_{i-1} \quad \text{where} \quad k \notin \mathtt{FCV}(e) \qquad (\mathrm{se})$$

$$\mathcal{S}_i k.\langle e\rangle_i = \mathcal{S}_i k.e \qquad (\mathrm{sr})$$

Fig. 12. Equational Theory for the Call-by-Name Calculus

We write $\vdash^{cbn} e_1 = e_2$ if the equation is derivable using the equations in Fig. 12. Similarly we write $\vdash^{cbv} e_1 = e_2$ for the equations in Fig. 13. It is easy to prove that the reduction semantics is subsumed by these our equations.

Theorem 8. *If* $(\beta, rv_n, rs_n) \vdash e_1 \rightsquigarrow^* e_2$*, then* $\vdash^{cbn} e_1 = e_2$ *is derivable.*

We can also prove that the call-by-name equations are sound with respect to the thunk translation, and hence the call-by-name CPS translation.

Theorem 9 (Soundness). *If* $\vdash^{cbn} e_1 = e_2$*, then* $\vdash^{cbv} \mathcal{T}[\![e_1]\!] = \mathcal{T}[\![e_2]\!]$.

This theorem can be proved by simple calculations for each equation. It immediately implies soundness of call-by-name equations with respect to the call-by-name CPS translation in the appendix.

Corollary 1. *If* $\vdash^{cbn} e_1 = e_2$*, then* $(\beta, \eta) \vdash \mathcal{C}^n[\![e_1]\!] = \mathcal{C}^n[\![e_2]\!]$.

Finally, an interesting question is whether the equations are complete with respect to the thunk translation. Unfortunately, we have not succeeded in directly proving completeness using the thunk translation, since we cannot define the inverse of the thunk translation which preserves equality[1]. However, we can use our previous results [11] to connect the call-by-name and call-by-value theories for the first level:

Theorem 10 (Correspondence in the First Level). *Suppose the maximum level* n *is* 1*, and* $e_1, e_2 \in \Lambda_{sr}$*. Then we have* $\vdash^{cbn} e_1 = e_2$ *if and only if* $\vdash^{cbv} \mathcal{T}[\![e_1]\!] = \mathcal{T}[\![e_2]\!]$.

Proof. Theorem 9 states soundness (the only-if direction). For completeness (the if direction), suppose $\vdash^{cbv} \mathcal{T}[\![e_1]\!] = \mathcal{T}[\![e_2]\!]$. By the soundness of the call-by-value equational theory, we have $(\beta, \eta) \vdash \mathcal{C}^v[\![\mathcal{T}[\![e_1]\!]]\!] = \mathcal{C}^v[\![\mathcal{T}[\![e_2]\!]]\!]$, and hence $(\beta, \eta) \vdash \mathcal{C}^n[\![e_1]\!] = \mathcal{C}^n[\![e_2]\!]$. By the completeness of the call-by-name equations for the first-level [11], we have $\vdash^{cbn} e_1 = e_2$.

[1] Note that η_v is admissible in the call-by-value calculus, while η-equality is not admissible in the call-by-name calculus.

6 Concluding Remarks

We have introduced the CPS hierarchy in call-by-name. Based on the thunk translation and factorization, we have derived a calculus and a type system, and proved several interesting properties of our system. The simplicity of the thunk translation makes it easy to treat a complex machinery such as the CPS hierarchy, and we do not have to directly consult the iterated CPS translations for most of the time.

This work builds on Danvy and Filinski's CPS Hierarchy, and is related to recent works on call-by-name delimited continuations. Among all, Saurin [18] proposed "Stream Hierarchy" as a call-by-name CPS Hierarchy, and developed a very interesting theory for this calculus, as it combines $\lambda\mu$-calculus in logic and streams in functional programming. He used a quite different CPS translation than ours, namely, η-equality is admissible in his theory, and thus his delimiter (`reset`) behaves quite differently from ours. Our theory does not admit η-equality since it badly interacts not only with the CPS translation, but also with the semantics of `reset`, since if we have full η-equality, we can convert every term to a value, which makes `reset` meaningless.[2]

As future work, we hope to relate the call-by-name calculi with the call-by-value one in the sense of duality, and also with classical logic.

References

1. K. Asai. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 22(3):275–291, 2009.
2. K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. In *APLAS, LNCS 4807*, pages 239–254, 2007.
3. M. Biernacka and D. Biernacki. Context-based Proofs of Termination for Typed Delimited-Control Operators. In *PPDP*, pages 289–300, 2009.
4. M. Biernacka, D. Biernacki, and O. Danvy. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Logical Methods in Computer Science*, 1(2), 2005.
5. O. Danvy and A. Filinski. Abstracting Control. In *LFP*, pages 151–160, 1990.
6. A. Filinski. Representing Layered Monads. In *POPL*, pages 175–188, 1999.
7. J. Hatcliff and O. Danvy. Thunks and the Lambda-Calculus. *J. Funct. Program.*, 7(3):303–319, 1997.
8. H. Herbelin and S. Ghilezan. An Approach to Call-by-Name Delimited Continuations. In *POPL*, pages 383–394, 2008.
9. Y. Kameyama. Axioms for Control Operators in the CPS Hierarchy. *Higher-Order and Symbolic Computation*, 20(4):339–369, 2007.
10. Y. Kameyama and M. Hasegawa. A Sound and Complete Axiomatization of Delimited Continuations. In *ICFP*, pages 177–188, 2003.
11. Y. Kameyama and A. Tanaka. Equational Axiomatization of Call-by-Name Delimited Control. In *PPDP*, pages 77–86, 2010.
12. O. Kiselyov. Call-by-name Linguistic Side Effects. In *ESSLLI*, 2008.

[2] Using η-equality, we can derive $\langle e \rangle_i = \langle \lambda y.ey \rangle_i = \lambda y.ey = e$.

13. O. Kiselyov, C. c. Shan, and A. Sabry. Delimited Dynamic Binding. *ICFP*, pages 26–37, 2006.
14. J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In *LFP*, pages 227–238, 1994.
15. C. Murthy. Control Operators, Hierarchies, and Pseudo-Classical Type Systems: A-Translation at Work. In *Proc. ACM Workshop on Continuations*, pages 49–71, 1992.
16. G. D. Plotkin. Call-by-Name, Call-by-Value and the Lambda-Calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
17. A. Sabry and M. Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.
18. A. Saurin. A Hierarchy for Delimited Control in Call-by-Name. In *FOSSACS*, pages 374–388, 2010.
19. A. Saurin. Standardization and Böhm Trees for $\Lambda\mu$-Calculus. In *FLOPS*, pages 134–149, 2010.
20. E. Sumii. An implementation of transparent migration on standard scheme. In *Scheme and Functional Programming*, pages 61–63, 2000.
21. H. Thielecke. From Control Effects to Typed Continuation Passing. In *POPL*, pages 139–149, 2003.
22. H. Thielecke. Answer type polymorphism in call-by-name continuation passing. In *ESOP*, pages 279–293, 2004.

A Equational Axioms for the Call-by-Value Calculus

Fig. 13 lists the axioms for the call-by-value calculus with hierarchical `shift` and `reset` due to [9] modulo small notational difference.

$$E^i ::= [\ ] \mid E^i e \mid v E^i \mid \texttt{force}(E^i) \mid k \hookleftarrow_h E^i \quad \text{where} \quad h < i$$

$$(\lambda x.e_1)\, v = e_1\{x := v\} \tag{β_v}$$

$$\lambda x.v\, x = v \quad \text{where } x \notin \texttt{FV}(v) \tag{η_v}$$

$$(\lambda x.E^1[x])e = E^1[e] \quad \text{where } x \notin \texttt{FV}(E^1) \tag{β_Ω}$$

$$\langle E^i[\mathcal{S}_i k.e]\rangle_j = \langle e\{k \Leftarrow_i \langle E^i\rangle_i\}\rangle_j \quad \text{where } i \le j \tag{rs_v}$$

$$k' \hookleftarrow_j (E^i[\mathcal{S}_i k.e]) = \langle e\{k \Leftarrow (k' \hookleftarrow_j E^i)\}\rangle_j \quad \text{where } i \le j \text{ and } k \neq k' \tag{th_v}$$

$$\langle v\rangle_i = v \tag{rv_v}$$

$$\mathcal{S}_i k.(k \hookleftarrow_i \langle e\rangle_{i-1}) = \langle e\rangle_{i-1} \quad \text{where } k \notin \texttt{FCV}(e) \tag{se}$$

$$\mathcal{S}_i k.\langle e\rangle_i = \mathcal{S}_i k.e \tag{sr}$$

$$\langle(\lambda x.e_1)\langle e_2\rangle_i\rangle_h = (\lambda x.\langle e_1\rangle_h)\langle e_2\rangle_i \quad \text{where } h \le i \tag{reset-lift}$$

$$\texttt{force}(\texttt{delay}(e)) = e \tag{fd}$$

Fig. 13. Axioms for Call-by-Value with Force/Delay