Equational Axiomatization of Call-by-Name Delimited Control

Yukiyoshi Kameyama
University of Tsukuba
Tsukuba, Japan
kameyama@acm.org

Asami Tanaka
University of Tsukuba
Tsukuba, Japan
asami@logic.cs.tsukuba.ac.jp

Abstract

Control operators for *delimited continuations* are useful in various fields such as partial evaluation, CPS translation, and representation of monadic effects. While many works in the literature study them in call-by-value, several recent works have shown call-by-name delimited control operators are also worth studying.

In this paper, we study semantic foundation of the call-by-name variant of the delimited-control operators "shift" and "reset". In particular, we give a set of direct-style equations as axioms for them, and prove that it is sound and complete with respect to the CPS translation by Biernacka and Biernacki. The key observations in our proof are (1) we need to use the linearity of certain variables in the CPS terms, and (2) we must distinguish continuation variables from ordinary variables in the source terms. We also show that our axiomatization holds for the typed calculus.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Languages,Theory,Verification

Keywords Control Operators, Delimited Continuation, Call-by-Name, Axiomatization, CPS translation

1. Introduction

Delimited continuation (or delimited control) has been proved useful in many applications from partial evaluation and Continuation Passing Style (CPS) transformation to representation of arbitrary monads to mobile computation. The traditional, unlimited continuation represents the whole rest of the computation (as the object captured by Scheme's `call/cc`), but the delimited continuation represents part of the rest of the computation.

While many works in the literature studied delimited control in call-by-value [6, 8], several recent work studied it in call-by-name. Herbelin and Ghilezan [11] related the study on classical logic with the study on delimited continuations, and proposed a call-by-name calculus for it. Kiselyov [13] proposed a call-by-name calculus with delimited-control operators and used it in linguistic analysis. Biernacka and Biernacki [4] introduced a type system for call-by-name delimited-control, which is similar to the type system for call-by-value delimited-control by Danvy and Filinski [5], and proved its strong normalization property.

In this paper, we investigate the call-by-name calculi with delimited-control operators. Specifically, we identify the semantics of such a calculus through a CPS translation, and give a set of simple equations in direct style, which is sound and complete with respect to this semantics.

We think such an axiomatization is useful and worth studying. Given a set of reductions, one can compute every program (closed term), however, one cannot optimize open terms using reductions only. For instance, in a call-by-value calculus, $(\lambda x.x)\ (y\ z)$ is equal to $y\ z$ in any contexts, but $(\lambda x.x)\ (y\ z)$ cannot be reduced using the call-by-value reductions. On the contrary, they are equal after CPS translation. Appel [1] demonstrated a way to compiler construction through CPS translation, and Flanagan et al. [10] showed that, all possible optimizations after CPS-translating source terms may be done before CPS-translating them, if we adopt a sufficiently strong set of equations as axioms. The axioms must be sound in the sense that all reductions are respected (if e_1 reduces to e_2, they must be equal by the axioms), and must be complete in the sense that all possible optimizations for CPS terms can be done for direct-style terms.

In this paper, we choose the control operators "shift" and "reset" among many delimited-control operators proposed in the literature. One of the most important merits of them is that they have a simple, functional CPS translation. Kameyama and Hasegawa [12] axiomatized the call-by-value calculus with shift and reset, namely, they identified the equational theory which coincides the CPS semantics for shift and reset. The axioms are simple enough to be used as optimizations, and can be used to reason about programs with shift and reset without converting the programs into CPS. In this paper we will carry out the same program for the call-by-name calculi.

Unlike the call-by-value calculi with delimited-control operators, there are a few choices about the semantics of the call-by-name calculi:

- Whether the calculus admits η-equality or not: full η-equality is usually assumed for call-by-name calculi, but it turns every term into a function (a value), and, therefore, interferes with the control operator reset (reset for a value does nothing). Hence, we need to restrict either of the two (at least), and we decided to abandon η-equality, since our intended operational semantics does not admit it.

  In Section 8, we will mention yet another calculus which admits full η-equality but the use of reset is restricted.

- The semantics of the target calculus after the CPS translation: the standard CPS translation for delimited-control operators translates source terms into non-CPS terms, namely, arguments

PPDP'10, July26–28, 2010, Hagenberg, Austria.

of function applications are not necessarily values, and therefore, their meaning depends on the semantics of the target calculus.

We choose the call-by-value semantics for the target calculus, because it matches the intended operational behavior of control operators.

To avoid the complicated equality theory having both call-by-name $\beta\eta$-equality and call-by-value $\beta\eta$-equality, we will translate the target terms once more by a CPS translation, following Danvy and Filinski [5].

It should be noted that, a naive adaptation of Kameyama and Hasegawa's completeness proof for the call-by-value calculus did not work for the call-by-name calculus. To overcome the difficulty, we need to refine the source calculus (before CPS translation) as well as the target calculus (after CPS translation) precisely. The key observations are that (1) the delimited continuations are linear in the call-by-name setting, and (2) we need to distinguish shift-bound variables from the ordinary, lambda-bound variables. As for (1), we note that the relationship between linearity and CPS translations was studied by Filinski as "linear continuation" [9], and by Berdine et al. as "linearly used continuations" [3], and, in this paper we need *both* kinds of linearity. As for (2), we adopt the formulation by Biernacka and Biernacki which has a special construct for the application of shift-bound variables to terms.

In this paper, we give sound and complete axiomatization for the type-free and typed call-by-name calculi with delimited-control operators, and their relationship with axiomatization for the call-by-value calculus.

The rest of this paper is organized as follows: in Section 2 we introduce the call-by-name calculus with delimited-control operators shift and reset. In Section 3 we give a CPS translation by Biernacka and Biernacki, and then a sound and complete axiomatization for the semantics in Section 4. The completeness proof is given in Section 5, and Section 6 discusses these results in typed setting. Section 7 discusses the use of linearity in our proofs, and Section 8 briefly mentions yet another CPS translation which respects η-equality. Section 9 gives concluding remarks.

2. Type-free Calculus

The calculus we study in this paper is $\lambda^{cbn}_{s/r}$, a call-by-name lambda calculus with delimited-control operators "shift" and "reset". The call-by-value version of these control operators was proposed by Danvy and Filinski [6, 7], and has been used to represent backtracking and various search, let-insertion in partial evaluation, various monads, type safe direct-style implementation of "printf", and others.

This section gives a type-free formulation of $\lambda^{cbn}_{s/r}$. Its type system will be explained later.

2.1 Syntax

In this subsection we give the syntax of terms and related expressions. First, we assume that there are two disjoint sets of variables, one for ordinary variables (denoted by $x, y, z, \cdots$) and the other for shift-bound variables (denoted by k). x is (eventually) bound by lambda and a term is substituted for x, while k is (eventually) bound by shift, and a delimited context (delimited continuation) is substituted for k.

Fig. 1 gives the syntax of $\lambda^{cbn}_{s/r}$. c is a constant. When we consider a typed calculus, we restrict c be a constant of basic types such as integer. An ordinary variable x is a term, while a shift-bound variable itself is not a term. The term $\lambda x.e$ and $e_1 e_2$ are abstraction and application as usual. The term $\mathcal{S}k.e$ is a shift-term, in which k is a bound variable. The term $k \hookleftarrow e$ is a throw-term

$e ::= c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathcal{S}k.e \mid k \hookleftarrow e \mid \langle e \rangle$	term
$E ::= [\ ] \mid Ee \mid \langle E \rangle \mid k \hookleftarrow E$	ev. ctxt.
$F ::= [\ ] \mid Fe$	pure ev. ctxt.

Figure 1. Syntax of the Source Language

$$(\lambda x.e_1)\ e_2 \rightsquigarrow e_1\{x := e_2\}$$
$$\langle F[\mathcal{S}k.e] \rangle \rightsquigarrow \langle e\{k \Rightarrow F\} \rangle$$
$$\langle v \rangle \rightsquigarrow v \qquad \text{for } v = c,\ \lambda x.e$$

Figure 2. Reduction Rules

that applies the (delimited) continuation k to a term e. The variable k in $k \hookleftarrow e$ is free. The term $\langle e \rangle$ is a reset-term.

Although the distinction between ordinary (lambda-bound) variables and continuation variables appears in the work of Parigot [15], it was Biernacka and Biernacki [4] who have made a clear distinction between lambda-bound variables and shift-bound variables in the context of call-by-name delimited control. They have also introduced the notation $k \hookleftarrow e$.

We identify α-equivalent terms. $\mathtt{FV}(e)$ denotes the set of free (ordinary and shift-bound) variables in e, and $e_1\{x := e_2\}$ represents the result of capture-avoiding substitution of e_2 for x in e_1. E represents an arbitrary evaluation context in call-by-name. F is a pure evaluation context, or a delimited evaluation context, which does not have reset's around the hole $[\ ]$. In other words, a pure evaluation context is delimited by a reset, and is exactly a delimited continuation. A general evaluation context is a continuation outside of a reset, namely, a metacontinuation. $E[e]$ (or $F[e]$) denotes the term after the hole-filling operation of a term e for a hole in E (or F). Hole-filling of another evaluation context $E_1[E_2]$ is defined similarly.

Remarks. Let us briefly argue the design of the source calculus here, namely, if the distinction between shift-bound and lambda-bound variables is necessary or not.

If we are only concerned with the operational behavior of closed terms, there is no need to distinguish them. In this case, the expression $k \hookleftarrow e$ can be represented by $(k\ e)$.

However, this distinction is necessary for this paper, since we are concerned with equality theories for open terms: to obtain complete axiomatization, we need an axiom which is valid for shift-bound variables but not valid for lambda-bound variables.

We remark that Biernacka and Biernacki introduced this distinction for a different purpose: to develop a type system for the calculus, which represents the so called answer-type polymorphism without making use of polymorphism [2]. The details may be found in [4].

In summary, two distinct classes of variables are necessary in the call-by-name calculi for delimited continuations. This is a sharp contrast with the call-by-value case, for which the above-mentioned distinction is not necessary [12].

2.2 Reduction Rules

To understand the operational behavior of shift and reset, we give the reduction rules of $\lambda^{cbn}_{s/r}$ in Fig. 2.

The reduction rules are essentially the same as those in Biernacka and Biernacki, but we have slightly changed them in the following point: their calculus has an expression $F \hookleftarrow e$ for a pure evaluation context F and an expression e. It is the result of substituting F for k in $k \hookleftarrow e$ (here we assume that k does not appear

in e freely). On the other hand, our calculus does not have such an expression, and such a substitution is realized by a meta-operation $\{k \Rightarrow F\}$.

The substitution $\{k \Rightarrow F\}$ is defined as follows:

$$
\begin{aligned}
c\{k \Rightarrow F\} &\stackrel{\text{def}}{=} c \\
x\{k \Rightarrow F\} &\stackrel{\text{def}}{=} x \\
(\lambda x.e)\{k \Rightarrow F\} &\stackrel{\text{def}}{=} \lambda x.(e\{k \Rightarrow F\}) \\
&\qquad \text{where } x \notin \mathtt{FV}(F) \\
(e_1\ e_2)\{k \Rightarrow F\} &\stackrel{\text{def}}{=} (e_1\{k \Rightarrow F\})\ (e_2\{k \Rightarrow F\}) \\
(\mathcal{S}k'.e)\{k \Rightarrow F\} &\stackrel{\text{def}}{=} \mathcal{S}k'.(e\{k \Rightarrow F\}) \\
&\qquad \text{where } k' \notin \{k\} \cup \mathtt{FV}(F) \\
(k \hookleftarrow e)\{k \Rightarrow F\} &\stackrel{\text{def}}{=} \langle F[e\{k \Rightarrow F\}]\rangle \\
(k' \hookleftarrow e)\{k \Rightarrow F\} &\stackrel{\text{def}}{=} k' \hookleftarrow (e\{k \Rightarrow F\}) \\
&\qquad \text{where } k' \notin \{k\} \cup \mathtt{FV}(F) \\
\langle e\rangle\{k \Rightarrow F\} &\stackrel{\text{def}}{=} \langle e\{k \Rightarrow F\}\rangle
\end{aligned}
$$

Let us look at the reduction rules in detail. The first reduction in Fig. 2 is the standard, call-by-name β-reduction.

In the second reduction rule, the evaluation context up to the nearest delimiter (reset) is F, and it is captured and substituted for k using the substitution $\{k \Rightarrow F\}$. The key case in this substitution is $(k \hookleftarrow e)\{k \Rightarrow F\}$, which is defined to be $\langle F[e\{k \Rightarrow F\}]\rangle$. This means that, by the substitution $\{k \Rightarrow F\}$, a functional object $\lambda x.\langle F[x]\rangle$ is substituted for k, if we identify $k \hookleftarrow e'$ with $(k\ e')$. Hence, the second reduction $\langle F[\mathcal{S}k.e]\rangle \rightsquigarrow \langle e\{k \Rightarrow F\}\rangle$ may be understood as

$$\langle F[\mathcal{S}k.e]\rangle \rightsquigarrow \langle e\{k := \lambda x.\langle F[x]\rangle\}\rangle$$

This is the same reduction rule as the one in the literature [12].

The third rule means: when a value is delimited by reset, then the delimiter is simply discarded. It may be strange to have the notion of values in the call-by-name calculus, but this rule naturally reflects the abstract-machine semantics for shift and reset in call-by-name[1]. Note that, a variable x is not a value in $\lambda^{cbn}_{s/r}$, so $\langle x\rangle$ does not reduce to x.

In Section 4, we will need to extend the above substitution to the form $\{k \Rightarrow (k' \hookleftarrow F)\}$, namely, we need to extend the substituted context F to the form $k' \hookleftarrow F$, which is not a pure evaluation context. The definition of the extended substitution remains the same except that F in the above definition may be in the form $k' \hookleftarrow F'$. For example, the key case is:

$$(k \hookleftarrow e)\{k \Rightarrow (k' \hookleftarrow F)\} \stackrel{\text{def}}{=} \langle k' \hookleftarrow (F[e\{k \Rightarrow (k' \hookleftarrow F)\}])\rangle$$

We remark that reset in the right-hand side of this definition is not necessary as $\langle k' \hookleftarrow e'\rangle = k' \hookleftarrow e'$ can be proved by our axioms introduced later.

3. Semantics based on CPS Translation

3.1 CPS Semantics

A CPS translation is a syntax-directed translation from a source calculus ($\lambda^{cbn}_{s/r}$ in this paper) to a target calculus. It is a useful theoretical tool to give a precise semantics to a calculus with control operators, and may be used as an intermediate language for compilers, since we can perform various kinds of optimizations on the target terms of a CPS translation. Using the axioms in this paper, one can perform the optimizations on the source terms.

[1] This semantics is attributed to Olivier Danvy in the literature [11].

The following three semantics are often studied for computational calculi:

- Reduction semantics: $e_1 = e_2$ if and only if they are equal up to the equality induced by the reduction rules.[2]
- CPS semantics. $e_1 = e_2$ if and only if they are translated to the same term by a CPS translation.
- Observational equivalence. $e_1 = e_2$ if and only if, for any context C such that $C[e_1]$ and $C[e_2]$ are closed, $C[e_1]$ and $C[e_2]$ both terminate, or both do not terminate under (a certain notion of) the operational semantics.

Under reasonable assumptions, the strength of the three semantics increases in the order above, namely, if two programs are equal in the reduction semantics (CPS semantics, resp.), they are equal in the CPS semantics (observational equivalence, resp.). The converse direction does not, in general, hold. For instance, $(\lambda x.xx)(\lambda x.xx)$ and $(\lambda x.xxx)(\lambda x.xxx)$ are equal under the observational equivalence, but they are not CPS-translated to equal terms for any standard CPS translations.

Although CPS semantics is weaker than observational equivalence, it is in many cases sufficient for reasoning about programs, as shown by literature (for instance, [1]). Indeed we often prefer CPS semantics than observational equivalence, since the latter is fragile in language extensions; under observational equivalence, two equal terms may not be equal after a new construct is added to the calculus, as we may be able to distinguish these terms using the new construct. On the contrary, CPS semantics is robust in the sense that, two equal terms remain equal even after a new construct is added, since their images of a CPS translation do not (usually) change.

3.2 1CPS Translation

Our task is to precisely determine the CPS translation we use, and we choose a CPS translation based on Plotkin's one, as we will argue below.

For call-by-name calculi, there are two choices for CPS translations: Plotkin's [16] and Streicher and Reus' [18]. The latter respects full η-equality (η-equal terms are translated to equal terms), while the former does not. Using full η-equality, we can infer that the control operator "reset" becomes totally useless:

$$\langle e\rangle =_\eta \langle \lambda x.ex\rangle \rightsquigarrow \lambda x.ex =_\eta e$$

Without the reset operator, our calculus does not make sense. Consequently, we use a CPS translation based on Plotkin's. We will revisit this choice in Section 8.

Fig. 3 gives the CPS translation for the calculus $\lambda^{cbn}_{s/r}$ where we assume that the variables κ and m are fresh. The CPS translation is essentially due to Biernacka and Biernacki, which is based on Plotkin's call-by-name CPS translation, and is extended to the calculus with shift and reset. The translation $[\![\_]\!]_1$ maps a term in $\lambda^{cbn}_{s/r}$ to a term in the type-free lambda calculus. We call it 1CPS translation. For terms without control operators, this CPS translation is identical to Plotkin's one. Unlike the call-by-value one, a variable x is translated to $\lambda\kappa.x\kappa$. This reflects the fact that a variable in the call-by-name calculus $\lambda^{cbn}_{s/r}$ is not a value, but represents a computation, so its CPS counterpart should receive a continuation κ. We will assume that the target calculus of the CPS translation admits η-equality for the continuation variable κ, hence $[\![x]\!]_1$ could be defined as x.

The translation for a reset-term $\langle e\rangle$ is identical to that in the call-by-value setting [6]. It is also not difficult to understand the translation of a shift-term $\mathcal{S}k.e$. It captures the current continuation

[2] The equality is the reflexive, transitive, and congruent closure of the reduction rules.

$$
\begin{aligned}
[c]_1 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa c \\
[x]_1 &\stackrel{\text{def}}{=} \lambda\kappa.x\kappa \\
[\lambda x.e]_1 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa(\lambda x.[e]_1) \\
[e_1 e_2]_1 &\stackrel{\text{def}}{=} \lambda\kappa.[e_1]_1(\lambda m.m[e_2]_1\kappa) \\
[\mathcal{S}k.e]_1 &\stackrel{\text{def}}{=} \lambda\kappa.([e]_1\{k := \kappa\})I_1 \\
[k \hookleftarrow e]_1 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa([e]_1 k) \\
[\langle e \rangle]_1 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa([e]_1 I_1) \\
I_1 &\stackrel{\text{def}}{=} \lambda m.m
\end{aligned}
$$

Figure 3. 1CPS Translation

κ, binds it to the shift-bound variable k, and then installs the identity continuation I_1. The translation of a throw-term $k \hookleftarrow e$ is to install the continuation k, compute e, and then re-activates the continuation κ by applying κ to the result of e. The translation of a reset-term is to save the current (delimited) continuation κ, and install the identity continuation I_1 (which corresponds to an empty evaluation context) as the current continuation.

3.3 Semantics of CPS terms

Our purpose is to axiomatize the CPS semantics, and in order to do so, we need to precisely define the semantics of the target calculus. In the presence of delimited-control operators in the source calculus $\lambda^{cbn}_{s/r}$, the image of 1CPS translation is not strictly in CPS. Namely, the arguments in function applications may not be values. For instance, the image of a reset-term contains a subterm $\kappa([e]_1 I_1)$, in which κ is applied to a non-value term $[e]_1 I_1$, thus the evaluation of this term is dependent on the evaluation strategy. If we evaluate it in call-by-value, $[e]_1 I_1$ is computed first, and κ is applied to its result. If we evaluate it in call-by-name, $[e]_1 I_1$ is not computed now, and κ is applied to the unevaluated subterm $[e]_1 I_1$.

As for this choice, we again follow Biernacka and Biernacki who have chosen the call-by-value semantics for the target calculus. Rather than introducing the call-by-value semantics into the target calculus directly (such as Moggi's computational lambda calculus [14]), we translate the CPS terms once again [6], by the call-by-value CPS translation.

The successive CPS translations may be represented by a single CPS translation, which we call a 2CPS translation. The images of the 2CPS translation are terms in CPS, and we no longer have to be worried about different semantics.

3.4 2CPS Translation

Fig. 4 defines the 2CPS translation for $\lambda^{cbn}_{s/r}$, which introduces two continuation variables κ and γ. These variables as well as m are assumed to be fresh. If we decompose the 2CPS translation into two CPS translations, κ is introduced by the first (call-by-name) CPS translation, and γ is introduced by the second (call-by-value) CPS translation.

The definition of the 2CPS translation looks like that of the 1CPS translation. The trick is that we η-reduce the results of 2CPS translation whenever possible. For instance, $[x]_2$ should be $\lambda\kappa.\lambda\gamma.x\kappa\gamma$, however, since we will introduce η-equality for the variable γ, it is equal to $\lambda\kappa.x\kappa$, so we define it as $[x]_2 \stackrel{\text{def}}{=} \lambda\kappa.x\kappa$ (which is in turn equal to x by η-reduction for κ). Since the images of the 1CPS translation of the throw-term $k \hookleftarrow e$ and the reset-term $\langle e \rangle$ contain non-CPS forms, their images of the 2CPS translations must use the second continuation variable γ.

$$
\begin{aligned}
[x]_2 &\stackrel{\text{def}}{=} x \\
[c]_2 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa c \\
[\lambda x.e]_2 &\stackrel{\text{def}}{=} \lambda\kappa.\kappa(\lambda x.[e]_2) \\
[e_1\ e_2]_2 &\stackrel{\text{def}}{=} \lambda\kappa.[e_1]_2(\lambda m.m[e_2]_2\kappa) \\
[\mathcal{S}k.e]_2 &\stackrel{\text{def}}{=} \lambda\kappa.([e]_2\{k := \kappa\})I_2 \\
[k \hookleftarrow e]_2 &\stackrel{\text{def}}{=} \lambda\kappa.\lambda\gamma.[e]_2 k(\kappa \oplus \gamma) \\
[\langle e \rangle]_2 &\stackrel{\text{def}}{=} \lambda\kappa.\lambda\gamma.[e]_2 I_2(\kappa \oplus \gamma) \\
I_2 &\stackrel{\text{def}}{=} \lambda m.\lambda\gamma.\gamma m \\
(\kappa \oplus \gamma) &\stackrel{\text{def}}{=} \lambda m.\kappa m\gamma
\end{aligned}
$$

Figure 4. 2CPS Translation

We also note that the identity continuation I_1 ($= \lambda m.m$) becomes I_2 ($= \lambda m.\lambda\gamma.\gamma m$), and that $(\kappa \oplus \gamma)$ (which is simply an abbreviation for $\lambda m.\kappa m\gamma$) corresponds to the push-operation for the stack of meta-continuations in the abstract machine semantics. (See [4] for details.)

In the following, we simply write $[e]$ for $[e]_2$, since we do not use the 1CPS translation.

3.5 Analysis of Target Calculus

In order to define the CPS semantics, we shall precisely define the semantics of the target calculus of the 2CPS translation.

By inspecting the images of the translation, we can easily show that terms in the target calculus are generated by the following grammar:

$$
\begin{aligned}
T &::= x \mid \lambda\kappa.P \mid VT & V &::= c \mid m \mid \lambda x.T \\
P &::= KV \mid TK \mid \lambda\gamma.A & K &::= \kappa \mid \lambda m.P \\
A &::= GV \mid PG & G &::= \gamma \mid \lambda m.A
\end{aligned}
$$

The target calculus has six sorts: T is the image of expressions in $\lambda^{cbn}_{s/r}$. P is a pre-term and A is an answer. V is the image of values (constant or abstraction). K is the image of pure evaluation contexts (delimited context). G is the image of (general) evaluation contexts. Variables belong to some specific sorts: x is a variable for T, m for V, κ for K, and γ for G.

It is important to note that $\beta\eta$-reductions in the target calculus preserve the sorts, which is easily proved.

For a term in the target calculus t_1 and t_2, we write $\vdash_{\text{CPS}} t_1 = t_2$ if they are proved equal using the following equations and the standard equality rules (reflexivity, symmetry, transitivity, and substitution):

- β-equality for x: $(\lambda x.T_1)\ T_2 = T_1\{x := T_2\}$.
- $\beta\eta$-equality for m: $(\lambda m.P)V = P\{m := V\}$, $\lambda m.Km = K$ (where $m \notin \mathtt{FV}(K)$), $(\lambda m.A)V = A\{m := V\}$, and $\lambda m.Gm = G$ (where $m \notin \mathtt{FV}(G)$).
- $\beta\eta$-equality for κ: $(\lambda\kappa.P)K = P\{\kappa := K\}$ and $\lambda\kappa.T\kappa = T$ (where $\kappa \notin \mathtt{FV}(T)$).
- $\beta\eta$-equality for γ: $(\lambda\gamma.A)G = A\{\gamma := G\}$ and $\lambda\gamma.P\gamma = P$ (where $\gamma \notin \mathtt{FV}(P)$).

Note that we have excluded η-equality for x, namely, $\lambda x.Vx = V$ is not allowed in general. This choice reflects the fact that the source calculus $\lambda^{cbn}_{s/r}$ does not allow η-equality as we argued before.

3.6 Properties of 2CPS Translation

In this subsection we show several useful properties of the 2CPS translation.

We first define $|F|$ for a pure evaluation context F. The purpose of $|\_|$ is to CPS-translate a pure evaluation context F, and it is defined as follows:

$$|[\ ]| \stackrel{\text{def}}{=} \lambda\kappa.\kappa$$
$$|F\ e| \stackrel{\text{def}}{=} \lambda\kappa.|F|\ (\lambda m.m[\![e]\!]\kappa)$$

Now we can state the following lemma which is useful in the soundness proof.

LEMMA 1. *We have the following properties for the 2CPS translation.*

(1) $\vdash_{CPS}\ [\![F[e]]\!] = \lambda\kappa.[\![e]\!](|F|\kappa)$ *where* $\kappa \notin FV(e)$.
(2) $\vdash_{CPS}\ [\![e\{x := e'\}]\!] = [\![e]\!]\{x := [\![e']\!]\}$.
(3) $\vdash_{CPS}\ [\![e\{k \Rightarrow F\}]\!] = [\![e]\!]\{k := |F|\ I_2\}$.
(4) $\vdash_{CPS}\ [\![e\{k \Rightarrow (k' \hookleftarrow F)\}]\!] = [\![e]\!]\{k := |F|\ k'\}$.

Proof. (1) The equation is proved by induction on F. When $F = [\ ]$, it is trivial. When $F = F_1\ e_1$, we can prove it as follows:

$$\begin{aligned}
[\![F[e]]\!] &\equiv [\![F_1[e]\ e_1]\!] \\
&\equiv \lambda\kappa.[\![F_1[e]]\!]\ (\lambda m.m[\![e_1]\!]\kappa) \\
&= \lambda\kappa.(\lambda\kappa'.[\![e]\!](|F_1|\kappa'))\ (\lambda m.m[\![e_1]\!]\kappa)\ \text{ by I.H.} \\
&= \lambda\kappa.[\![e]\!](|F_1|(\lambda m.m[\![e_1]\!]\kappa)) \\
&= \lambda\kappa.[\![e]\!](|F_1 e_1|\ \kappa)
\end{aligned}$$

From the second line to the third, we used induction hypothesis on F_1.

(2) The equation is easily proved by induction on e.

(3) The equation is proved by induction on e. The only interesting case is $e \equiv k \hookleftarrow e_1$, and we can prove the case as follows:

$$\begin{aligned}
&[\![(k \hookleftarrow e_1)\{k \Rightarrow F\}]\!] \\
&\equiv [\![\langle F[e_1\{k \Rightarrow F\}]\rangle]\!] \\
&\equiv \lambda\kappa.\lambda\gamma.[\![F[e_1\{k \Rightarrow F\}]]\!]I_2(\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.[\![e_1\{k \Rightarrow F\}]\!](|F|\ I_2)(\kappa \oplus \gamma)\ \text{ by (1)} \\
&= \lambda\kappa.\lambda\gamma.[\![e_1]\!]\{k := |F|\ I_2\}(|F|\ I_2)(\kappa \oplus \gamma)\ \text{ by I.H.} \\
&= (\lambda\kappa.\lambda\gamma.[\![e_1]\!]k(\kappa \oplus \gamma))\{k := |F|\ I_2\} \\
&\equiv [\![(k \hookleftarrow e_1)]\!]\{k := |F|\ I_2\}
\end{aligned}$$

(4) This case is proved in the same way as (3). Here we show the case $e \equiv k \hookleftarrow e_1$.

$$\begin{aligned}
&[\![(k \hookleftarrow e_1)\{k \Rightarrow (k' \hookleftarrow F)\}]\!] \\
&\equiv [\![k' \hookleftarrow F[e_1\{k \Rightarrow (k' \hookleftarrow F)\}]]\!] \\
&\equiv \lambda\kappa.\lambda\gamma.[\![F[e_1\{k \Rightarrow (k' \hookleftarrow F)\}]]\!]k'(\kappa \oplus \gamma) \\
&= \lambda\kappa.\lambda\gamma.[\![e_1\{k \Rightarrow (k' \hookleftarrow F)\}]\!](|F|\ k')(\kappa \oplus \gamma)\ \text{ by (1)} \\
&= \lambda\kappa.\lambda\gamma.[\![e_1]\!]\{k := |F|\ k'\}(|F|\ k')(\kappa \oplus \gamma)\ \text{ by I.H.} \\
&= (\lambda\kappa.\lambda\gamma.[\![e_1]\!]k(\kappa \oplus \gamma))\{k := |F|\ k'\} \\
&\equiv [\![k \hookleftarrow e_1]\!]\{k := |F|\ k'\}
\end{aligned}$$

4. Axiomatizing the Semantics

In this section, we provide a set of axioms for $\lambda^{cbn}_{s/r}$, and show that it is sound with respect to the CPS semantics. Its completeness will be proved in the next section.

$(\lambda x.e_1)\ e_2 = e_1\{x := e_2\}$	β
$\langle F[\mathcal{S}k.e]\rangle = \langle e\{k \Rightarrow F\}\rangle$	reset-shift
$k' \hookleftarrow (F[\mathcal{S}k.e]) = \langle e\{k \Rightarrow (k' \hookleftarrow F)\}\rangle$	throw-shift
$\langle v\rangle = v$ for $v = c,\ \lambda x.e$	reset-value
$\mathcal{S}k.(k \hookleftarrow e) = e$ where $k \notin \mathtt{FV}(e)$	shift-elim
$\mathcal{S}k.\langle e\rangle = \mathcal{S}k.e$	shift-reset

Figure 5. Axioms for $\lambda^{cbn}_{s/r}$ (call-by-name)

$(\lambda x.e_1)\ v = e_1\{x := v\}$	β_v
$\lambda x.v\ x = v$ where $x \notin \mathtt{FV}(v)$	η_v
$(\lambda x.F[x])e = F[e]$ where $x \notin \mathtt{FV}(F)$	β_Ω
$\langle F[\mathcal{S}k.e]\rangle = \langle e\{k \Rightarrow F\}\rangle$	reset-shift
$k' \hookleftarrow (F[\mathcal{S}k.e]) = \langle e\{k \Rightarrow (k' \hookleftarrow F)\}\rangle$	throw-shift
$\langle v\rangle = v$	reset-value
$\mathcal{S}k.(k \hookleftarrow e) = e$ where $k \notin \mathtt{FV}(e)$	shift-elim
$\mathcal{S}k.\langle e\rangle = \mathcal{S}k.e$	shift-reset
$\langle(\lambda x.e_1)\langle e_2\rangle\rangle = (\lambda x.\langle e_1\rangle)\langle e_2\rangle$	reset-lift

where v and F are defined as follows:

$$v ::= c \mid x \mid \lambda x.e$$
$$F ::= [\ ] \mid F\ e \mid v\ F$$

Figure 6. Axioms for $\lambda^{cbv}_{s/r}$ (call-by-value)

4.1 Axioms

Fig. 5 gives our axioms for $\lambda^{cbn}_{s/r}$. We write $\vdash^{\text{cbn}}_{\text{s/r}}\ e_1 = e_2$ if we can prove the equation using these axioms and the standard inference rules for equality.

For the purpose of comparison, Fig. 6 gives the axioms for the call-by-value calculus $\lambda^{cbv}_{s/r}$ by Kameyama and Hasegawa [12]. Although their calculus does not have the throw expression $k \hookleftarrow e$, we introduce it for the purpose of comparison. Their axioms are slightly modified accordingly. Note that the definitions for a value v and the pure evaluation context F for the call-by-value calculus are different from those for $\lambda^{cbn}_{s/r}$.

We will explain the axioms in Fig. 5 by comparing them to those in Fig. 6.

- The call-by-name calculus $\lambda^{cbn}_{s/r}$ has a single strong axiom for β, while the call-by-value one $\lambda^{cbv}_{s/r}$ has three weaker axioms β_v, β_Ω, and reset-lift. The former subsumes the latter.
- No axiom for η-equality exists in $\lambda^{cbn}_{s/r}$, while $\lambda^{cbv}_{s/r}$ has the call-by-value version of η-equality as η_v.
- The axioms shift-elim and shift-reset are contained in both calculi.
- Two axioms reset-shift and reset-value are contained in both calculi, but their actual meanings differ between the calculi, since the definitions of F and v differ.
- The axiom throw-shift is contained in both calculi.

  For the call-by-value calculus, this is only because $k \hookleftarrow e$ is a special form. If we do not distinguish shift-bound and lambda-bound variables, then $k \hookleftarrow e$ may be represented by $(k\ e)$,

and the axiom throw-shift is subsumed by reset-shift (note that $(k\ F)$ is an evaluation context in $\lambda^{cbv}_{s/r}$ if k is a variable).

On the contrary, the axiom throw-shift is necessary for the call-by-name calculus, and it seems difficult to formulate this axiom without distinguishing shift-bound variables from lambda-bound variables. Without such a distinction, throw-shift would become unsound.

We believe that our axioms for the call-by-name calculus are simple enough to be used for reasoning about open programs, and that the similarity of the axioms for the two calculi is an evidence that our axioms are natural and stable.

We emphasize that establishing this kind of axioms is not trivial even after getting to know the axiomatization for the call-by-value case, since the CPS translation of the shift term in call-by-name is not quite the same as that in call-by-value.

It is easy to prove that the reduction semantics is subsumed by our axioms.

THEOREM 1. *Let e_1 and e_2 be expressions in $\lambda^{cbn}_{s/r}$. If $e_1 \rightsquigarrow e_2$ by one of the reduction rules in Fig. 2, then $\vdash^{cbn}_{s/r} e_1 = e_2$ is derivable.*

Proof. The three reductions, respectively, are subsumed by the axioms β, reset-shift, and reset-value, respectively.

The rest of this paper is devoted to establish that the axioms in Fig. 5 are sound and complete with respect to the CPS semantics induced by the 2CPS translation.

4.2 Soundness of Axioms

Soundness of the axioms means that equal terms in the source calculus $\lambda^{cbn}_{s/r}$ are CPS-translated to equal terms. Soundness may be thought as a sort of correctness for a CPS translation.

THEOREM 2 (Soundness). *Let e_1 and e_2 be terms in $\lambda^{cbn}_{s/r}$. If $\vdash^{cbn}_{s/r} e_1 = e_2$ is derivable, then $\vdash_{CPS} [e_1] = [e_2]$ is derivable.*

Proof. It suffices to prove the theorem when $e_1 = e_2$ is obtained by one of the axioms. We list the proofs of all the cases, since they are instructive to know the details of the CPS translation.

(β)

$$\begin{aligned}[(\lambda x.e_1)\ e_2] &\equiv \lambda\kappa.[\lambda x.e_1]\ (\lambda m.m\ [e_2]\ \kappa)\\ &= \lambda\kappa.(\lambda\kappa'.\kappa'\ (\lambda x.[e_1]))\ (\lambda m.m\ [e_2]\ \kappa)\\ &= \lambda\kappa.(\lambda x.[e_1])\ [e_2]\ \kappa\\ &= \lambda\kappa.[e_1]\{x := [e_2]\}\ \kappa\\ &= [e_1]\{x := [e_2]\}\\ &= [e_1\{x := e_2\}] \text{ by Lemma 1 (2)}\end{aligned}$$

(reset-shift)

$$\begin{aligned}&[\langle F[\mathcal{S}k.e]\rangle]\\ &\equiv \lambda\kappa.\lambda\gamma.[F[\mathcal{S}k.e]]\ I_2\ (\kappa\oplus\gamma)\\ &= \lambda\kappa.\lambda\gamma.[\mathcal{S}k.e]\ (|F|\ I_2)\ (\kappa\oplus\gamma) \text{ by Lemma 1 (1)}\\ &\equiv \lambda\kappa.\lambda\gamma.((\lambda\kappa'.[e]\{k := \kappa'\})\ I_2)(|F|\ I_2)\ (\kappa\oplus\gamma)\\ &= \lambda\kappa.\lambda\gamma.([e]\{k := |F|\ I_2\})\ I_2\ (\kappa\oplus\gamma)\\ &= \lambda\kappa.\lambda\gamma.[e\{k \Rightarrow F\}]\ I_2\ (\kappa\oplus\gamma) \text{ by Lemma 1 (3)}\\ &\equiv [\langle e\{k \Rightarrow F\}\rangle]\end{aligned}$$

(throw-shift, $k' \neq k$)

$$\begin{aligned}&[k' \hookleftarrow (F[\mathcal{S}k.e])]\\ &\equiv \lambda\kappa.\lambda\gamma.[F[\mathcal{S}k.e]]\ k'\ (\kappa\oplus\gamma)\\ &= \lambda\kappa.\lambda\gamma.[\mathcal{S}k.e](|F|\ k')\ (\kappa\oplus\gamma) \text{ by Lemma 1 (1)}\\ &\equiv \lambda\kappa.\lambda\gamma.(\lambda\kappa'.[e]\{k := \kappa'\}I_2)(|F|\ k')\ (\kappa\oplus\gamma)\\ &= \lambda\kappa.\lambda\gamma.([e]\{k := |F|\ k'\})\ I_2\ (\kappa\oplus\gamma)\\ &= \lambda\kappa.\lambda\gamma.[e\{k \Rightarrow (k' \hookleftarrow F)\}]\ I_2\ (\kappa\oplus\gamma) \text{ by Lemma 1 (4)}\\ &\equiv [\langle e\{k \Rightarrow (k' \hookleftarrow F)\}\rangle]\end{aligned}$$

(reset-value) We define v^* for a value v by:

$$c^* \stackrel{\text{def}}{=} c$$

$$(\lambda x.e)^* \stackrel{\text{def}}{=} \lambda x.[e]$$

Then we can prove:

$$\begin{aligned}[\langle v\rangle] &\equiv \lambda\kappa.\lambda\gamma.(\lambda\kappa'.\kappa' v^*)\ I_2\ (\kappa\oplus\gamma)\\ &= \lambda\kappa.\lambda\gamma.I_2\ v^*\ (\kappa\oplus\gamma)\\ &= \lambda\kappa.\lambda\gamma.(\kappa\oplus\gamma)\ v^*\\ &= \lambda\kappa.\lambda\gamma.\kappa\ v^*\ \gamma\\ &= \lambda\kappa.\kappa\ v^*\\ &\equiv [v]\end{aligned}$$

(shift-elim, $k \notin \mathtt{FV}(e)$)

$$\begin{aligned}[\mathcal{S}k.(k \hookleftarrow e)] &\equiv \lambda\kappa.([k \hookleftarrow e]\{k := \kappa\})\ I_2\\ &= \lambda\kappa.(\lambda\kappa'.\lambda\gamma.[e]\ k\ (\kappa'\oplus\gamma))\{k := \kappa\})\ I_2\\ &= \lambda\kappa.\lambda\gamma.[e]\ \kappa\ (I_2\oplus\gamma)\\ &= \lambda\kappa.\lambda\gamma.[e]\ \kappa\ \gamma\\ &= [e]\end{aligned}$$

(shift-reset)

$$\begin{aligned}[\mathcal{S}k.\langle e\rangle] &= \lambda\kappa.([\langle e\rangle]\{k := \kappa\})\ I_2\\ &= \lambda\kappa.(\lambda\kappa'.\lambda\gamma.[e]\ I_2\ (\kappa'\oplus\gamma))\{k := \kappa\})\ I_2\\ &= \lambda\kappa.\lambda\gamma.([e]\ I_2\ (I_2\oplus\gamma))\{k := \kappa\}\\ &= \lambda\kappa.\lambda\gamma.([e]\ I_2\ \gamma)\{k := \kappa\}\\ &= [\mathcal{S}k.e]\end{aligned}$$

5. Completeness

Completeness is the converse of soundness: if the images of the CPS translation of two terms are equal in the target calculus, then they are equal in the source calculus. Completeness ensures that all equational reasoning for CPS terms can be done in the source calculus.

THEOREM 3 (Completeness). *Let e_1 and e_2 be (type-free) terms in $\lambda^{cbn}_{s/r}$. If $\vdash_{CPS} [e_1] = [e_2]$ is derivable, then $\vdash^{cbn}_{s/r} e_1 = e_2$ is derivable.*

Let us give an overview of the proof of completeness.

We will basically follow the general recipe by Sabry and Felleisen [17] as follows:

1. Analyze the structure of target terms of CPS translation, and define a grammar for it, which should be closed under the reductions in the target calculus.
2. Define a translation from the target to the source calculus, and prove that it is an inverse of the CPS translation.
3. Prove that equality in the target calculus is preserved by the inverse translation.

However, we encountered a problem when the grammar in Section 3.5 is used for this purpose; we could not define a suitable image of the inverse function for the variable m (the variable for the V-sort) with the desired property. Let us explain this in more detail. In the completeness proof, we will need the property $\langle v^\circ \rangle = v^\circ$ for any term v of the V-sort where v° is the image of v by the inverse function. As a special case, we need $\langle m^\circ \rangle = m^\circ$. However, if m° is a variable in $\lambda^{cbn}_{s/r}$, this property does hold in general. (Note that a variable in $\lambda^{cbn}_{s/r}$ represents an arbitrary expression, and not necessarily a value, thus $\langle x \rangle = x$ does not hold in general.) In addition, no other choice for m° seems to exist as m is a variable.

We can overcome this difficulty by investigating the target calculus more carefully. The variable m in the target calculus is not used in an arbitrary way, but is used *linearly*. Linearity of the variable m reflects the fact that a delimited continuation (a pure evaluation context) in the call-by-name calculus is linear in its hole: $F ::= [\ ] \mid F\ e$. [3]

Since m is linear, we do not have to keep the name of m through the inverse function, and we may discard it. The same technique was used by Kameyama and Hasegawa's axiomatization of shift and reset in call-by-value, where the metacontinuation variable γ is linear, and thus can be discarded by the inverse function. Here, we discard not only the name of γ, but also the name of m.

In the following proof, we do not directly formulate linear lambda calculus as the target calculus. Instead, we refine the grammar to reflect the linearity of the variable m. Therefore we do not explicitly mention the linearity in the proof, but it is embedded in the grammar of the target calculus, and the definition of the inverse translation from the target calculus. The point here is that this simple trick is sufficient to avoid the problem above.

5.1 Refined Grammar and Inverse Function

We give a refined grammar for the target calculus as:

$T ::= x \mid \lambda\kappa.P \mid VT$	term
$P ::= KV \mid TK \mid \lambda\gamma.A$	preterm
$A ::= GV \mid PG$	answer
$V ::= c \mid \lambda x.T$	value
$K ::= \kappa \mid \lambda m.m\ T\ K \mid \lambda m.\lambda\gamma.\gamma\ m$	continuation
$G ::= \gamma \mid \lambda m.K\ m\ G$	metacontinuation

The difference from the previous grammar is that we no longer regard m as a member of the V-sort. Instead, the K- and G-sorts contain a few more terms using m, and m is a linear variable.

The equality of the target calculus remains the same, however, since η-redex for m does not exist in the terms generated by the new grammar, it is discarded. In summary, the equality of the terms generated by the new grammar is induced by β-equality for the variables x and m, and $\beta\eta$-equality for the variables κ and γ.

In the following, we use T, P, A, V, K and G not only for the names of sorts, but also for meta-variables of the corresponding sorts, for instance, T is used as a meta-variable for terms of the T-sort.

We can easily prove the following lemma for the refined grammar.

LEMMA 2.

- *If e is a $\lambda^{cbn}_{s/r}$-term, then $[\![e]\!]$ belongs to the T-sort.*
- *Each of the above sorts is closed under β-reduction for x and m, and $\beta\eta$-reduction for κ and γ.*

[3] This linearity has been already mentioned in the literature, for instance, in Herbelin and Ghilezan [11].

$$
\begin{array}{ll}
x^\circ \stackrel{\text{def}}{=} x & (\lambda\kappa.P)^\circ \stackrel{\text{def}}{=} \mathcal{S}\kappa.P^\circ \\
(VT)^\circ \stackrel{\text{def}}{=} V^\circ\ T^\circ & \\
(KV)^\circ \stackrel{\text{def}}{=} K^\circ[V^\circ] & (TK)^\circ \stackrel{\text{def}}{=} K^\circ[T^\circ] \\
(\lambda\gamma.A)^\circ \stackrel{\text{def}}{=} A^\circ & \\
(PG)^\circ \stackrel{\text{def}}{=} G^\circ[\langle P^\circ \rangle] & (GV)^\circ \stackrel{\text{def}}{=} G^\circ[V^\circ] \\
c^\circ \stackrel{\text{def}}{=} c & (\lambda x.T)^\circ \stackrel{\text{def}}{=} \lambda x.T^\circ \\
\kappa^\circ \stackrel{\text{def}}{=} \kappa \hookleftarrow [\ ] & (\lambda m.m\ T\ K)^\circ \stackrel{\text{def}}{=} K^\circ[[\ ]\ T^\circ] \\
 & (\lambda m.\lambda\gamma.\gamma\ m)^\circ \stackrel{\text{def}}{=} [\ ] \\
\gamma^\circ \stackrel{\text{def}}{=} [\ ] & (\lambda m.K\ m\ G)^\circ \stackrel{\text{def}}{=} G^\circ[\langle K^\circ[\ ] \rangle]
\end{array}
$$

Figure 7. Inverse Function

We then define an inverse function $(\_)^\circ$ from the target calculus to source calculus in Fig. 7. By the inverse function, a term in the T, P, A, or V-sort is mapped to a term in $\lambda^{cbn}_{s/r}$, a term in the K or G-sort to an evaluation context.

Note that reset is introduced in $(PV)^\circ$ and $(\lambda m.K\ m\ G)^\circ$, and the names of the variables m and γ are discarded through the inverse, since these variables are linear.

We prove that the above "inverse" function is actually a (left) inverse of the 2CPS translation.

LEMMA 3 (Inverse). *Let e be an expression in $\lambda^{cbn}_{s/r}$. We have $\vdash^{cbn}_{s/r}\ [\![e]\!]^\circ = e$.*

Proof. This lemma can be proved by a straightforward induction on e. Let us show a few interesting cases.

For $e = \langle e' \rangle$, we have:

$$
\begin{aligned}
[\![e]\!]^\circ &= (\lambda\kappa.\lambda\gamma.[\![e']\!] I_2(\kappa \oplus \gamma))^\circ \\
&= \mathcal{S}\kappa.(\lambda m.\kappa\ m\ \gamma)^\circ[\langle I_2{}^\circ[[\![e']\!]^\circ] \rangle] \\
&= \mathcal{S}\kappa.\kappa \hookleftarrow \langle [\![e']\!]^\circ \rangle = \langle [\![e']\!]^\circ \rangle = \langle e' \rangle
\end{aligned}
$$

For $e = \mathcal{S}k.e'$, we have:

$$
\begin{aligned}
[\![e]\!]^\circ &= (\lambda\kappa.([\![e']\!]\{k := \kappa\}) I_2)^\circ \\
&= (\lambda k.[\![e']\!] I_2)^\circ \\
&= \mathcal{S}k.I_2{}^\circ[[\![e']\!]^\circ] = \mathcal{S}k.e'
\end{aligned}
$$

5.2 Properties of Inverse Function

We present several technical properties about substitution and the inverse.

LEMMA 4. *Substitutions for x, κ, and γ commute with the inverse function in the following sense.*

(1) $\vdash^{cbn}_{s/r}\ (t\{x := T_1\})^\circ = t^\circ\{x := T_1{}^\circ\}$ *is derivable if t is a T-, P-, A-, V-, K-, or G-term.*

(2) *Let ϕ be $\{\kappa := K_1\}$ and ψ be $\{\kappa \Rightarrow K_1{}^\circ\}$. Then we have:*
- $\vdash^{cbn}_{s/r}\ (t\phi)^\circ = t^\circ\psi$ *if t is a T- or V-term.*
- $\vdash^{cbn}_{s/r}\ \langle (t\phi)^\circ \rangle = \langle t^\circ\psi \rangle$ *if t is a P- or A-term.*
- $\vdash^{cbn}_{s/r}\ \langle (K\phi)^\circ[e] \rangle = \langle (K^\circ\psi)[e] \rangle$ *for any term e in $\lambda^{cbn}_{s/r}$.*
- $\vdash^{cbn}_{s/r}\ (G\phi)^\circ[\langle e \rangle] = (G^\circ\psi)[\langle e \rangle]$ *for any term e in $\lambda^{cbn}_{s/r}$.*

(3) $\vdash^{cbn}_{s/r}\ \langle (A\{\gamma := G_1\})^\circ \rangle = \langle G_1{}^\circ[\langle A^\circ \rangle] \rangle$.

(4) $\vdash^{cbn}_{s/r}\ (G\{\gamma := G_1\})^\circ[\langle e \rangle] = G_1{}^\circ[\langle G^\circ[\langle e \rangle] \rangle]$ *for any term e in $\lambda^{cbn}_{s/r}$.*

Proof. This lemma can be proved by simultaneous induction on each term. Note that $(P\phi)^\circ = P^\circ\psi$ does not hold in general, but it suffices to prove $\langle(P\phi)^\circ\rangle = \langle P^\circ\psi\rangle$ to make the induction go through, since P always appears immediately under reset. Similarly for K, we only have to consider the form $\langle K^\circ[e]\rangle$. We also make use of the fact that γ does not appear free in T, P, V or K.

Here, we prove a few interesting cases.

(Case $K \equiv \kappa$ for $\langle(K\phi)^\circ[e]\rangle = \langle(K^\circ\psi)[e]\rangle$)

We may assume that $\kappa \notin \mathtt{FV}(e)$. The left-hand side is $\langle {K_1}^\circ[e]\rangle$, and the right-hand side is $\langle(\kappa^\circ\psi)[e]\rangle$, which is equal to $\langle(\kappa \hookleftarrow e)\{\kappa \Rightarrow {K_1}^\circ\}\rangle$. Then we have:

$$\vdash^{\text{cbn}}_{\text{s/r}} \ \langle(\kappa \hookleftarrow e)\{\kappa \Rightarrow {K_1}^\circ\}\rangle = \langle\langle {K_1}^\circ[e]\rangle\rangle = \langle {K_1}^\circ[e]\rangle$$

(Case $G \equiv \lambda m.KmG_2$ for (4))

The left-hand side of (4) is $(G_2\{\gamma := G_1\})^\circ[\langle K^\circ[\langle e\rangle]\rangle]$, which is equal to ${G_1}^\circ[\langle {G_2}^\circ[\langle K^\circ[\langle e\rangle]\rangle]\rangle]$ by induction hypothesis, which is equal to the right-hand side.

LEMMA 5 (Inverse preserves reduction). *Suppose $\vdash_{CPS} \ t_1 = t_2$ is derivable for target terms t_1 and t_2, then we have:*

- $\vdash^{cbn}_{s/r} \ {t_1}^\circ = {t_2}^\circ$ *if t_1 and t_2 are T- or V-terms.*
- $\vdash^{cbn}_{s/r} \ \langle {t_1}^\circ\rangle = \langle {t_2}^\circ\rangle$ *if t_1 and t_2 are P- or A-terms.*
- $\vdash^{cbn}_{s/r} \ \langle {t_1}^\circ[e]\rangle = \langle {t_2}^\circ[e]\rangle$ *if t_1 and t_2 are K-terms and e is a term in $\lambda^{cbn}_{s/r}$.*
- $\vdash^{cbn}_{s/r} \ {t_1}^\circ[\langle e\rangle] = {t_2}^\circ[\langle e\rangle]$ *if t_1 and t_2 are G-terms and e is a term in $\lambda^{cbn}_{s/r}$.*

Proof. It suffices to prove the lemma when t_2 is obtained from t_1 by β or η reduction.

There are six β-redexes, and two η-redexes. For each reduction $t_1 \rightsquigarrow t_2$, it is not difficult to prove $\vdash^{\text{cbn}}_{\text{s/r}} \ {t_1}^\circ = {t_2}^\circ$.

The most interesting case is β-reduction for κ, namely, the case $\vdash^{\text{cbn}}_{\text{s/r}} \ \langle((\lambda\kappa.P)K)^\circ\rangle = \langle(P\{\kappa := K\})^\circ\rangle$. We have:

$$\langle((\lambda\kappa.P)K)^\circ\rangle = \langle K^\circ[\mathcal{S}\kappa.P^\circ]\rangle$$

and

$$\langle(P\{\kappa := K\})^\circ\rangle = \langle P^\circ\{\kappa \Rightarrow K^\circ\}\rangle$$

by Lemma 4. By inspecting the definition of K°, it is either a pure evaluation context, or the form $\kappa' \hookleftarrow F$ for some κ' and F. In the former case, we use the axiom reset-shift to prove:

$$\vdash^{\text{cbn}}_{\text{s/r}} \ \langle K^\circ[\mathcal{S}\kappa.P^\circ]\rangle = \langle P^\circ\{\kappa \Rightarrow K^\circ\}\rangle$$

while in the latter case, we use the axiom throw-shift to prove the same equation. Hence we are done.

The other cases can be proved easily.

It should be emphasized that the proof of this theorem actually needs the axiom throw-shift.

Proof. of Theorem 3 Let e and e' be terms in $\lambda^{cbn}_{s/r}$. Suppose $\vdash_{\text{CPS}} \ [\![e]\!] = [\![e']\!]$ is derivable. Since $[\![e]\!]$ and $[\![e']\!]$ are T-terms, we have $\vdash^{\text{cbn}}_{\text{s/r}} \ [\![e]\!]^\circ = [\![e']\!]^\circ$ by Lemma 5, and then $\vdash^{\text{cbn}}_{\text{s/r}} \ e = e'$ by Lemma 3.

6. Typed Calculus

We have so far studied the type-free calculus only, and a natural question is whether our axioms work for typed calculus. In this section we give a positive answer to it.

6.1 Type System

We define a type system for our call-by-name calculus with delimited control. The type system defined in this section is essentially the same as the type system by Biernacka and Biernacki modulo the following modifications.

$$\frac{}{\Gamma \cup \{x : (\alpha \mid \sigma \mid \beta)\} \mid \alpha \vdash x : \sigma \mid \beta}\ \text{var}$$

$$\frac{(c \text{ is a constant of base type } b)}{\Gamma \mid \alpha \vdash c : b \mid \alpha}\ \text{const}$$

$$\frac{\Gamma, x : (\alpha \mid \sigma \mid \beta) \mid \alpha' \vdash e : \sigma' \mid \beta'}{\Gamma \mid \gamma \vdash \lambda x.e : (\alpha \mid \sigma \mid \beta) \to (\alpha' \mid \sigma' \mid \beta') \mid \gamma}\ \text{fun}$$

$$\frac{\Gamma \mid \beta' \vdash e_0 : (\alpha \mid \sigma \mid \beta) \to (\alpha' \mid \sigma' \mid \beta') \mid \gamma \quad \Gamma \mid \alpha \vdash e_1 : \sigma \mid \beta}{\Gamma \mid \alpha' \vdash e_0 e_1 : \sigma' \mid \gamma}\ \text{app}$$

$$\frac{\Gamma \mid \alpha \vdash e : \alpha \mid \beta}{\Gamma \mid \gamma \vdash \langle e\rangle : \beta \mid \gamma}\ \text{reset}$$

$$\frac{\Gamma \cup \{k : \sigma \rhd \tau\} \mid \alpha \vdash e : \alpha \mid \beta}{\Gamma \mid \tau \vdash \mathcal{S}k.e : \sigma \mid \beta}\ \text{shift}$$

$$\frac{\Gamma \mid \tau \vdash e : \sigma \mid \beta}{\Gamma \cup \{k : \sigma \rhd \tau\} \mid \alpha \vdash k \hookleftarrow e : \beta \mid \alpha}\ \text{throw}$$

Figure 8. Type System

- Our syntax does not have an expression $F \hookleftarrow e$ for a pure evaluation context F and an expression e, so the corresponding typing rule is not contained.
- As a consequence, we do not need typing rules for delimited contexts and metacontexts, so they are omitted.
- We use a slightly different notation for function types. This is only notational difference: we write $(\alpha \mid \sigma \mid \beta) \to (\alpha' \mid \sigma' \mid \beta')$ for their type ${\sigma}^{\alpha,\beta}{}_{\alpha'} \to_{\beta'} \sigma'$.

A type (denoted by $\alpha, \beta, \sigma, \tau, \cdots$) is either a basic type b (such as integer and boolean), or a function type $(\alpha \mid \tau \mid \beta) \to (\alpha' \mid \tau' \mid \beta')$. The function type may be understood as the type of a function whose argument has the type τ with an effect of type α to β, and whose return type is τ' with an effect of type α' to β'. Here, "an effect of type α to β" is a computational effect invoked by a shift-expression, which changes the answer type of the current continuation from α to β. See [2] for the detailed discussion on the answer-type modification. If there is no effect by a shift-expression, then α and β are the same type (which can be an arbitrary type [19].)

We also define a context-type $\sigma \rhd \tau$ where σ and τ are types. A context-type is used for typing a delimited continuation, or a pure evaluation context.

A typing context Γ is a (possibly empty) set consisting of $x : (\alpha \mid \tau \mid \beta)$ and $k : \sigma \rhd \tau$. The former intuitively means that an ordinary variable x has type τ with an effect of type α to β, while the latter represents the type of a delimited continuation corresponding to k. Since a delimited continuation is always a pure function, we do not have to consider its effects, so the type of k needs only two subtypes. A judgement takes the form $\Gamma \mid \alpha \vdash e : \tau \mid \beta$ where Γ is a typing context, α, τ, β are types, and e is an expression.

Fig. 8 gives the type system.

As Biernacka and Biernacki proved, the reduction rules in Fig. 2 enjoy the subject reduction property under this type system.

6.2 Typed CPS Translation

For the purpose of this paper, namely, axiomatization, we define a CPS translation for the typed calculus. Not surprisingly, we use the same CPS translation, the 2CPS translation in Fig. 4 for expressions

in the typed calculus. The remaining task is to define the CPS translation of types and typing contexts.

For a simple type[4] A, we define $\neg A$ as $A \to \bot$ where $\bot$ is an arbitrary fixed type.

The CPS translation of a type is defined by $b^* \stackrel{\text{def}}{=} b$ and

$$((\alpha \mid \tau \mid \beta) \to (\alpha' \mid \tau' \mid \beta'))^* \stackrel{\text{def}}{=} ((\tau^* \to \neg\neg\alpha^*) \to \neg\neg\beta^*) \to ((\tau'^* \to \neg\neg\alpha'^*) \to \neg\neg\beta'^*)$$

The definition above needs several occurrences of double negation $\neg\neg\_$ since we use 2CPS translation.

Finally, we define the CPS translation of a typing context as:

$$(x : (\alpha \mid \tau \mid \beta))^* \stackrel{\text{def}}{=} x : (\tau^* \to \neg\neg\alpha^*) \to \neg\neg\beta^*$$

$$(k : \sigma \rhd \tau)^* \stackrel{\text{def}}{=} k : \sigma^* \to \neg\neg\tau^*$$

THEOREM 4. *If $\Gamma \mid \alpha \vdash e : \tau \mid \beta$ is derivable, then $\Gamma^* \vdash [e] : (\tau^* \to \neg\neg\alpha^*) \to \neg\neg\beta^*$ is derivable in the simply typed lambda calculus.*

Proof. The theorem can be proved by induction on the derivation of $\Gamma \mid \alpha \vdash e : \tau \mid \beta$.

6.3 Axiomatization

We now axiomatize the CPS semantics for the typed calculus. Perhaps surprisingly, exactly the same axioms in Fig. 5 work for the typed calculus.

THEOREM 5 (Soundness and Completeness). *Let e_1 and e_2 be typed terms in $\lambda^{cbn}_{s/r}$, then $\vdash^{cbn}_{s/r} e_1 = e_2$ if and only if $\vdash_{CPS} [e_1] = [e_2]$.*

Proof. Since the proof is mostly the same as the type-free case, we only show an overview of the proof.

- We first establish that, if the term in the left-hand side of each axiom is typable, then the term in its right-hand side is typable and has the same typing judgement.
- We must check that all intermediate terms used in the completeness proof preserve the typing judgement.
- We must also check that the inverse translation preserves typing, and also, if e is typable in the source calculus, $[e]^\circ$ has the same typing judgement.
- Finally, we must check that the reductions in the target calculus are mapped to typable equations in the source calculus.

We must check a lot of things, but each check can be done straightforwardly.

7. Discussion on the use of linearity

We briefly discuss the use of linearity in the target of a CPS translation, which plays a significant role in our proof.

A (rough) type structure of the image of the 2CPS translation can be given as:

$$\begin{aligned} \texttt{Term} &= \texttt{Cont} \to_1 \texttt{MetaCont} \to_2 \texttt{Answer} \\ \texttt{Cont} &= \texttt{Value} \to_3 \texttt{MetaCont} \to_4 \texttt{Answer} \\ \texttt{MetaCont} &= \texttt{Value} \to_5 \texttt{Answer} \end{aligned}$$

where the indices $1, 2, \ldots, 5$ are for reference, and we have omitted to write the definition of type `Value`. As we have seen, some function types can be refined to linear function types:

[4] A simple type means a type in the simply typed lambda calculus.

$$\begin{aligned} [c]^{SR}_2 &= \lambda\kappa.\kappa c \\ [x]^{SR}_2 &= x \\ [\lambda x.e]^{SR}_2 &= \lambda(x,\kappa).[e]^{SR}_2\kappa \\ [e_1\ e_2]^{SR}_2 &= \lambda\kappa.[e_1]^{SR}_2([e_2]^{SR}_2,\ \kappa) \\ [\mathcal{S}k.e]^{SR}_2 &= \lambda\kappa.([e]^{SR}_2\{k := \kappa\})I_2 \\ [k \hookleftarrow e]^{SR}_2 &= \lambda\kappa.\lambda\gamma.[e]^{SR}_2 k(\kappa \oplus \gamma) \\ [\langle e \rangle]^{SR}_2 &= \lambda\kappa.\lambda\gamma.[e]^{SR}_2 I_2(\kappa \oplus \gamma) \end{aligned}$$

Figure 9. SR-style 2CPS Translation

- $\to_2$ and $\to_4$ can be made linear function space $\multimap$, which corresponds to the metacontinuation variable γ. In the terminology of Berdine et al. [3], it is a *linearly used continuation*.
- $\to_3$ and $\to_5$ can be made linear function space $\multimap$, which corresponds to the variable m. This linearity reflects the fact that evaluation contexts are linear, and is specific to the call-by-name calculus. This is related to what Filinski called *linear continuations* [9].

In our proof, we have used both linearity, hence $\to_i$ for $i = 2, 3, 4, 5$ is linear function space. The remaining (non-linear) function space $\to_1$ corresponds to the non-linear uses of delimited continuations, which are useful for representing, e.g., backtracking. Berdine et al. [3] proposed to restrict this function space to be linear, and claimed that it is a "stylized" use of control.

8. Revisiting η-equality

We have argued the interaction between reset and full η-equality, and concluded that we should give up full η-equality, because it is unsound with respect to the intended operational semantics.

However, there can be a typed calculus which admits full η-equality if the use of reset is limited. More precisely, if we restrict the type of reset-terms to be basic types (no function types are allowed as the return type of any delimited continuation captured by shift), we can formulate a calculus in which η-equality is sound.

To develop the semantics of such a calculus, we extend Streicher and Reus' CPS translation [18]. Fig. 9 gives a 2CPS translation for $\lambda^{cbn}_{s/r}$ as an extension of Streicher and Reus' CPS translation.

The target calculus of this CPS translation is the simply typed lambda calculus augmented with pairing, denoted by $(t_1,\ t_2)$. Pairs are decomposed by an abstraction $\lambda(x,y).t$ with the additional equality $(\lambda(x,y).t_0)\ (t_1,\ t_2) = t_0\{x := t_1\}\{y := t_2\}$, and $\lambda(x,y).e(x,y) = e$ for $x, y \notin \texttt{FV}(e)$. It should be noted the CPS translation preserves η-equality as:

$$[\lambda x.ex]^{SR}_2 =_\beta \lambda(x,k).[e]^{SR}_2(x,k) =_{\text{pair}} [e]^{SR}_2$$

where $=_{\text{pair}}$ means the equality induced by the two equations for pairs.

Using the CPS translation in Fig. 9, we can develop yet another type system for $\lambda^{cbn}_{s/r}$. The type system obtained through Streicher and Reus' CPS translation is different from the one obtained by Biernacka and Biernacki CPS translation.

We could axiomatize such a calculus in a similar way to this paper. Unfortunately, the result is not really illuminating; the obtained axioms are the same axioms as those given in this paper with η-equality ($\lambda x.ex = e$ for $x \notin \texttt{FV}(e)$) being added. Note that a reset-term $\langle e \rangle$ is restricted to be of basic types, and therefore $\langle \lambda x.e \rangle$ is not well-typed in this calculus.

We did not develop this calculus in this paper, because we do not know any practical meaning of such a calculus.

9. Concluding Remarks

In this paper we have investigated the CPS semantics of the call-by-name calculi with delimited-control operators. We have obtained a sound and complete axiomatization for the calculus in the sense that:

$$\vdash^{\text{cbn}}_{\text{s/r}} e_1 = e_2 \text{ if and only if } \vdash_{\text{CPS}} [e_1] = [e_2].$$

Since η-equality interferes with reset, our axioms for the source calculus do not contain η-equality. We have proved that the same axioms work for type-free and typed cases, where the type system is (essentially) given by Biernacka and Biernacki.

Our theoretical tool was the 2CPS translation, which mixes a call-by-name translation with a call-by-value one. For the completeness proof, we need to use linearity of evaluation contexts in a non-trivial way. Also we have formulated the source calculus carefully so that lambda-bound variables and shift-bound variables must be treated differently.

We also briefly mentioned yet another typed calculus for the call-by-name shift and reset. In this calculus, η-equality is admitted with no restriction, but the type of a reset-term $\langle e \rangle$ is restricted to a basic type.

Although we do not claim that the calculus proposed by Biernacka and Biernacki (or our calculus as its small variant) is the right one for call-by-name delimited control, we believe our results can be basis for further study on call-by-name delimited-control.

Relation with Other Works. In this paper, we have concentrated on the delimited-control operators "shift" and "reset" in the style of Danvy and Filinski and of Biernacka and Biernacki, and have not considered the relationship with others.

Herbelin and Ghilezan proposed a call-by-name calculus with control operators and connected it with classical logic [11]. They have obtained the calculus from the call-by-value calculus with delimited-control operators, but the counterpart of the reset-operator behaves quite differently from those presented in this paper. In their call-by-value calculus, $\langle M \rangle$ is represented by $\mu\hat{\mathtt{tp}}.[\hat{\mathtt{tp}}]M$ where $\hat{\mathtt{tp}}$ is a variable for the top-level continuation. However, in their call-by-name calculus, this term reduces to M, so its behavior is quite different from the usual behavior of reset. Consequently, it is difficult to relate their calculus with other calculi. Their calculus is also different from the calculus in Section 8, since, in the latter calculus $\langle e \rangle = e$ does not hold if e has a basic type.

As an application of a call-by-name calculus with delimited control, Kiselyov proposed to use it in linguistic analysis [13]. His calculus has an explicit representation of evaluation contexts, and therefore it is similar to Biernacka and Biernacki's formulation. He has not given a CPS translation for his calculus, so we do not know if we can axiomatize his calculus as in this paper.

Future work. There are a number of future works, and let us list only a few of them. (1) To investigate other delimited-control operators such as Felleisen's "control" and "prompt" in call-by-name, and if possible axiomatizing them. (2) To relate the call-by-name calculi with the call-by-value one in the sense of duality, and also with classical logic. (3) To relate our calculus with logic-based calculi such as Herbelin and Ghilezan's, and give the logical account to "reset".

Acknowledgments

We would like to thank Olivier Danvy, Masahito Hasegawa and Oleg Kiselyov for insightful comments. Special thanks go to anonymous reviewers of FLOPS and PPDP, who have pointed out errors in the earlier versions of this paper.

The first author was partly supported by JSPS Grant-in-Aid for Scientific Research (B) 21300005.

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

[2] K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. In *Proc. Asian Programming Languages and Systems, LNCS 4807*, pages 239–254, Nov-Dec 2007.

[3] J. Berdine, P. O'Hearn, U. Reddy, and H. Thielecke. Linear Continuation-Passing. *Higher-Order and Symbolic Computation*, 15 (2-3):191–208, 2002.

[4] M. Biernacka and D. Biernacki. Context-based proofs of termination for typed delimited-control operators. In *PPDP 2009*, pages 289–300, 2009.

[5] O. Danvy and A. Filinski. A Functional Abstraction of Typed Contexts. Technical Report 89/12, DIKU, University of Copenhagen, July 1989.

[6] O. Danvy and A. Filinski. Abstracting Control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.

[7] O. Danvy and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4): 361–391, 1992.

[8] M. Felleisen. The Theory and Practice of First-Class Prompts. In *Proc. 15th Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[9] A. Filinski. Linear Continuations. In *POPL*, pages 27–38, 1992.

[10] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.

[11] H. Herbelin and S. Ghilezan. An Approach to Call-by-Name Delimited Continuations. In *POPL 2008*, pages 383–394, 2008.

[12] Y. Kameyama and M. Hasegawa. A sound and complete axiomatization for delimited continuations. In *ICFP*, pages 177–188, 2003.

[13] O. Kiselyov. Call-by-name Linguistic Side Effects. In *ESSLLI*, 2008.

[14] E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.

[15] M. Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992. ISBN 3-540-55727-X.

[16] G. D. Plotkin. Call-by-Name, Call-by-Value, and the λ-Calculus. *Theoretical Computer Science*, 1:125–159, 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] T. Streicher and B. Reus. Classical Logic, Continuations Semantics and Abstract Machines. *J. Functional Programming*, 8(6):543–572, 1998.

[19] H. Thielecke. From Control Effects to Typed Continuation Passing. In *POPL*, pages 139–149, New York, January 2003. ACM Press.