Strong Normalization of Polymorphic Calculus for Delimited Continuations

Yukiyoshi Kameyama[1] and Kenichi Asai[2]

[1] Department of Computer Science, University of Tsukuba
kameyama@acm.org
[2] Department of Information Science, Ochanomizu University
asai@is.ocha.ac.jp

Abstract. The notion of delimited continuations has been proved useful in various areas of computer programming such as partial evaluation, mobile computing, and web transaction. In our previous work, we proposed polymorphic calculi with control operators for delimited continuations. This paper presents a proof of strong normalization (SN) of these calculi based on a refined (i.e. administrative redex-free) CPS translation.
Keywords: Type System, Delimited Continuation, Control Operator, CPS Translation, Predicative Polymorphism.

1 Introduction

Control operators in functional languages allow the explicit manipulation of control flow of programs, and thus give more flexibility and expressiveness than those programs without them. Scheme has the control operator "call/cc" for continuations, which has been intensively studied from theory to implementation and practical applications. Type-theoretic studies on "call/cc" have revealed that it corresponds to classical logic [10].

Delimited continuation is a similar notion to (unlimited) continuation, but it represents *part* of the rest of the computation rather than whole rest of the computation. Since Danvy and Filinski have proposed the control operators "shift" and "reset" for delimited continuations [6], they have been proved useful in various applications such as backtracking [6], A-normalization in direct style [2], let-insertion in partial evaluation [17], type-safe "printf" [3], and web transaction [15].

This paper investigates the type structure of "shift" and "reset", and in particular, proves strong normalization of a polymorphic calculus for them. In our previous work [4], we have introduced a polymorphic type system for "shift" and "reset", and proved a number of properties such as type soundness (subject reduction and progress). The strong normalization property, however, was only mentioned as a theorem without a proof, which is the subject of the present paper.

Strong normalization (SN) is the property that no reduction sequence can be infinite, and is considered as one of the most fundamental properties of many typed lambda calculi which correspond to logical systems under the Curry-Howard isomorphism. For instance, strong normalization of Girard's System F [9] implies its (logical) consistency.

On the other hand, strong normalization of computational calculi with control operators is a subtle issue as shown by the following list:

$$
\begin{array}{lll}
v ::= c \mid x \mid \lambda x.e & & \text{value} \\
e ::= v \mid e_1 e_2 \mid \mathcal{S}k.e \mid \langle e \rangle \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & & \\
\quad \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & & \text{expression}
\end{array}
$$

Fig. 1. Syntax of $\lambda_{let}^{s/r}$.

- A typed calculus with "call/cc" is strongly normalizing, while that with exception in Standard ML is not strongly normalizing (see, for instance, [11]).
- The calculus with "shift" and "reset" under Filinski's typing[3] is not normalizing, while, under Danvy and Filinski's type system, it is strongly normalizing [1].
- A typed calculus with "control" and "prompt", the other control operators for delimited continuations, is not normalizing [13]. Similarly, a typed calculus with the control operator `cupto` is not normalizing [14].

Hence, we can say SN for the calculi with control operators is a non-trivial issue. This paper solves the problem for the case of "shift" and "reset" under the polymorphic type system.

The rest of this paper is organized as follows: Section 2 gives the syntax and semantics of the polymorphic calculus for shift and reset in [4], and Section 3 reviews the definitional CPS translation for this calculus. In Section 4 we introduce a refined CPS translation and study its properties. In Section 5, we prove strong normalization of our calculus by making use of the refined CPS translation. Section 6 gives conclusion.

2 A Polymorphic Calculus with Shift/Reset

In this section we introduce the polymorphic typed calculi $\lambda_{let}^{s/r}$ for shift and reset in [4]. Following the literature, we distinguish two versions of polymorphism: *predicative* polymorphism (or let-polymorphism) found in ML families and *impredicative* polymorphism which is based on the second order lambda calculus (Girard's System F [9]). In this paper, we concentrate on the predicative version.

2.1 Syntax and Operational Semantics

We assume that the sets of constants (denoted by c), variables (denoted by x, y, k, f), type variables (denoted by t), and basic types (denoted by b) are mutually disjoint, and that each constant is associated with a basic type. We assume `bool` is a basic type which has constants `true` and `false`.

The syntax of $\lambda_{let}^{s/r}$ is given in Figure 1. A value is either a constant, a variable or a lambda abstraction. An expression is either a value, an application, a shift expression (denoted by $\mathcal{S}k.e$), a reset expression (denoted by $\langle e \rangle$), a let expression, or a conditional. Note that we omit the fixpoint operator from the calculus in [4].

[3] Filinski did not give a type system explicitly, but his well-known implementation of "shift" and "reset" [8] specifies a certain type system.

$$
\begin{aligned}
(\lambda x.e)v &\rightsquigarrow e[v/x] \\
\langle v \rangle &\rightsquigarrow v \\
\langle F[\mathcal{S}k.e] \rangle &\rightsquigarrow \langle \texttt{let } k = \lambda x.\langle F[x] \rangle \texttt{ in } e \rangle \\
\texttt{let } x = v \texttt{ in } e &\rightsquigarrow e[v/x] \\
\texttt{if true then } e_1 \texttt{ else } e_2 &\rightsquigarrow e_1 \\
\texttt{if false then } e_1 \texttt{ else } e_2 &\rightsquigarrow e_2
\end{aligned}
$$

Fig. 2. Reduction rules for $\lambda_{let}^{s/r}$

$$
\begin{aligned}
\alpha, \beta, \gamma, \delta &::= t \mid b \mid (\alpha/\gamma \to \beta/\delta) && \text{monomorphic type} \\
A &::= \alpha \mid \forall t.A && \text{polymorphic type}
\end{aligned}
$$

Fig. 3. Types of $\lambda_{let}^{s/r}$.

Variables are bound by lambda or shift (k is bound in the expression $\mathcal{S}k.e$), and are free otherwise. $\texttt{FV}(e)$ denotes the set of free variables in e.

We give call-by-value operational semantics for $\lambda_{let}^{s/r}$. Contexts, pure evaluation contexts (abbreviated as pure e-contexts), and redexes are defined as follows:

$$
\begin{aligned}
C ::= {} & [\,] \mid \lambda x.C \mid eC \mid Ce \mid \mathcal{S}k.C \mid \langle C \rangle \mid \texttt{let } x = C \texttt{ in } e \mid \texttt{let } x = e \texttt{ in } C \\
& \mid \texttt{if } C \texttt{ then } e \texttt{ else } e \mid \texttt{if } e \texttt{ then } C \texttt{ else } e \mid \texttt{if } e \texttt{ then } e \texttt{ else } C && \text{context} \\
F ::= {} & [\,] \mid vF \mid Fe \mid \texttt{if } F \texttt{ then } e \texttt{ else } e && \text{pure e-context} \\
R ::= {} & (\lambda x.e)v \mid \langle v \rangle \mid \langle F[\mathcal{S}k.e] \rangle \mid \texttt{let } x = v \texttt{ in } e \\
& \mid \texttt{if true then } e_1 \texttt{ else } e_2 \mid \texttt{if false then } e_1 \texttt{ else } e_2 && \text{redex}
\end{aligned}
$$

A pure e-context F is an evaluation context such that no reset encloses the hole. Therefore, in the redex $\langle F[\mathcal{S}k.e] \rangle$, the outermost reset is guaranteed to be the one corresponding to this shift, i.e., no reset exists inbetween.

The notion of *one-step reduction* $\rightsquigarrow$ is defined by $C[R] \rightsquigarrow C[e]$ where C is an arbitrary context [4] and $R \rightsquigarrow e$ is an instance of reductions in Figure 2. In this figure, $e[v/x]$ denotes the ordinary capture-avoiding substitution. As usual, $\rightsquigarrow^*$ (and $\rightsquigarrow^+$, resp.) denotes the reflexive-transitive (transitive, resp.) closure of $\rightsquigarrow$.

2.2 Type System

The type system of $\lambda_{let}^{s/r}$ is an extension of Danvy and Filinski's monomorphic type system for shift and reset [5].

Types are defined by Figure 3, which are similar to those in core ML except that the function type is annotated with two answer types as $(\alpha/\gamma \to \beta/\delta)$ where γ (and δ, resp.)

[4] Note that we have slightly extended the notion of one-step reduction from our previous paper [4] where the context enclosing a redex must be an evaluation context, not a general context. Hence the SN property in this paper is slightly stronger than the one in [4].

$$\frac{(x : A \in \Gamma \text{ and } \tau \leq A)}{\Gamma \vdash_p x : \tau}\ \text{var} \qquad \frac{(c \text{ is a constant of basic type } b)}{\Gamma \vdash_p c : b}\ \text{const}$$

$$\frac{\Gamma, x : \sigma;\ \alpha \vdash e : \tau;\ \beta}{\Gamma \vdash_p \lambda x.e : (\sigma/\alpha \to \tau/\beta)}\ \text{fun}$$

$$\frac{\Gamma;\ \gamma \vdash e_1 : (\sigma/\alpha \to \tau/\beta);\ \delta \quad \Gamma;\ \beta \vdash e_2 : \sigma;\ \gamma}{\Gamma;\ \alpha \vdash e_1 e_2 : \tau;\ \delta}\ \text{app} \qquad \frac{\Gamma \vdash_p e : \tau}{\Gamma;\ \alpha \vdash e : \tau;\ \alpha}\ \text{exp}$$

$$\frac{\Gamma, k : \forall t.(\tau/t \to \alpha/t);\ \sigma \vdash e : \sigma;\ \beta}{\Gamma;\ \alpha \vdash \mathcal{S}k.e : \tau;\ \beta}\ \text{shift} \qquad \frac{\Gamma; \sigma \vdash e : \sigma;\ \tau}{\Gamma \vdash_p \langle e \rangle : \tau}\ \text{reset}$$

$$\frac{\Gamma \vdash_p e_1 : \sigma \quad \Gamma, x : \mathtt{Gen}(\sigma; \Gamma);\ \alpha \vdash e_2 : \tau;\ \beta}{\Gamma;\ \alpha \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau;\ \beta}\ \text{let}$$

$$\frac{\Gamma;\ \sigma \vdash e_1 : \mathtt{bool};\ \beta \quad \Gamma;\ \alpha \vdash e_2 : \tau;\ \sigma \quad \Gamma;\ \alpha \vdash e_3 : \tau;\ \sigma}{\Gamma;\ \alpha \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau;\ \beta}\ \text{if}$$

Fig. 4. Type Inference Rules of $\lambda_{let}^{s/r}$.

denotes the answer type before (after, resp.) the execution of the function body. See Asai and Kameyama [4] for details. A type variable is bound by the universal quantifier $\forall$ as usual, and $\mathtt{FTV}(\alpha)$ denotes the set of free type variables in α.

A *type context* (denoted by Γ) is a finite list of the form $x_1 : A_1, \cdots, x_n : A_n$ where the variables $x_1, \cdots, x_n$ are mutually distinct, and $A_1, \cdots, A_n$ are (polymorphic) types.

Judgments are either one of the following forms:

$$\begin{array}{rl} \Gamma \vdash_p e : \tau & \text{judgment for pure expression} \\ \Gamma;\ \alpha \vdash e : \tau;\ \beta & \text{judgment for general expression} \end{array}$$

The first form of the judgment signifies the expression e is a pure expression (free from control effects), and the second is for an arbitrary expression. We distinguish pure expressions from other expressions in order to present the restriction of let-polymorphism: polymorphism can be introduced only for pure expressions, as we can see it from the type inference rule for let below.

Figure 4 lists the type inference rules of $\lambda_{let}^{s/r}$ where $\tau \leq A$ in the rule (var) means the instantiation of type variables by monomorphic types. Namely, if $A \equiv \forall t_1.\cdots\forall t_n.\rho$ for some monomorphic type ρ, then $\tau \equiv \rho[\sigma_1, \cdots, \sigma_n/t_1, \cdots, t_n]$ for some monomorphic types $\sigma_1, \cdots, \sigma_n$. The type $\mathtt{Gen}(\sigma; \Gamma)$ in the rule (let) is defined by $\forall t_1.\cdots\forall t_n.\sigma$ where $\{t_1, \cdots, t_n\} = \mathtt{FTV}(\sigma) - \mathtt{FTV}(\Gamma)$.

The type inference rules are a natural extension of the monomorphic type system by Danvy and Filinski [5]. Pure expressions are defined by one of the rules (var), (const), (fun) or (reset). They can be freely turned into general expressions by the rule (exp). Pure expressions can be used polymorphically through the rule (let). It generalizes the standard let-polymorphism found in ML where the so called value restriction is adopted.[5] Finally, the rule (shift) is extended to cope with the answer type polymorphism of captured continuations: k is given a polymorphic type $\forall t.(\tau/t \to \alpha/t)$.

[5] Note that all values are pure, but pure expressions are not necessarily values.

2.3 Properties

In our previous paper [4], we claimed that our calculus provides a good foundation for studying the interaction between polymorphism and delimited continuations. To support this claim, we have presented the proofs of the following properties: Strong Type Soundness, Existence of Principal Types, and Preservation of types and equality through CPS translation. We have also stated Confluence and Strong Normalization for the calculus, but did not present the proofs.

In this subsection, we quickly review the properties which were proved in [4].[6]

Theorem 1 (Subject Reduction). *If $\Gamma;\ \alpha \vdash e_1 : \tau;\ \beta$ is derivable and $e_1 \rightsquigarrow^* e_2$, then $\Gamma;\ \alpha \vdash e_2 : \tau;\ \beta$ is derivable. Similarly, if $\Gamma \vdash_p e_1 : \tau$ is derivable and $e_1 \rightsquigarrow^* e_2$, then $\Gamma \vdash_p e_2 : \tau$ is derivable.*

Theorem 2 (Progress). *If $\vdash_p \langle e \rangle : \tau$ is derivable, then $\langle e \rangle$ can be reduced.*

By Theorems 1 and 2, we can conclude that our type system is sound (strong type soundness in the sense of [18]).

Theorem 3 (Principal Type and Type Inference). *In $\lambda_{let}^{s/r}$, principal type exists, and we can construct a sound and complete type inference algorithm as an extension of Hindley-Milner's algorithm.*

3 Definitional CPS Translation

A CPS translation is a translation from one calculus (typically with control operators) to a simpler calculus (typically without control operators). It allows us to investigate the semantic structure of the source calculus. The merit of shift and reset over other control operators for delimited continuations comes from the fact that there exists a simple, compositional CPS translation. Danvy and Filinski gave the precise semantics of shift and reset in terms of a CPS translation [6, 7], and based on their translation, various theoretical results as well as applications using shift and reset have been proposed (see, for instance, [12]).

In this section, we present an extension of Danvy and Filinski's CPS translation, namely, a CPS translation from $\lambda_{let}^{s/r}$ to a pure polymorphic lambda calculus λ_{let}. We call this CPS translation as "definitional" one, since it defines the semantics of $\lambda_{let}^{s/r}$.

In the following, we first define the target calculus λ_{let}, and then present the definitional CPS translation.

3.1 Target Calculus λ_{let}

The syntax of values and expressions in λ_{let} are the same as those in $\lambda_{let}^{s/r}$ except that λ_{let} does not have control operators shift and reset. Types of λ_{let} are standard and given by:

$$\begin{array}{rr} \alpha, \beta ::= t \mid b \mid \alpha \to \beta & \text{monomorphic type} \\ A ::= \alpha \mid \forall t.A & \text{polymorphic type} \end{array}$$

[6] Strictly speaking, these theorems are extended versions of the corresponding theorems in [4], since the notion of reduction in this paper is slightly extended.

$$\frac{(x : A \in \Gamma \text{ and } \tau \leq A)}{\Gamma \vdash x : \tau}\ \text{var} \qquad \frac{(c \text{ is a constant of basic type } b)}{\Gamma \vdash c : b}\ \text{const}$$

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \to \beta}\ \text{fun} \qquad \frac{\Gamma \vdash e_1 : \alpha \to \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1 e_2 : \beta}\ \text{app}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \mathtt{Gen}(\sigma; \Gamma) \vdash e_2 : \beta}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \beta}\ \text{let} \qquad \frac{\Gamma \vdash e_1 : \mathtt{bool} \quad \Gamma \vdash e_2 : \beta \quad \Gamma \vdash e_3 : \beta}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \beta}\ \text{if}$$

Fig. 5. Type Inference Rules of λ_{let}

$$\begin{aligned} b^* &= b \quad \text{for a basic type } b \\ t^* &= t \quad \text{for a type variable } t \\ ((\alpha/\gamma \to \beta/\delta))^* &= \alpha^* \to (\beta^* \to \gamma^*) \to \delta^* \\ (\forall t.A)^* &= \forall t.A^* \\ (\Gamma, x : A)^* &= \Gamma^*, x : A^* \end{aligned}$$

Fig. 6. CPS translation for types and type contexts.

Figure 5 defines the type inference rules of λ_{let}. Note that, in the type inference rule for (let), there is no side condition on the expression e_1. Hence, for instance, an expression $\mathtt{let}\ x = yz\ \mathtt{in}\ x$ is not typable in $\lambda_{let}^{s/r}$, but is typable in λ_{let}.

The reduction rules for λ_{let} are the same as those for $\lambda_{let}^{s/r}$ restricted to the expressions in λ_{let}, and are omitted.

3.2 Definitional CPS Translation from $\boldsymbol{\lambda_{let}^{s/r}}$ to $\boldsymbol{\lambda_{let}}$

Figures 6 and 7 define the definitional CPS translation for $\lambda_{let}^{s/r}$ where the variables κ, κ', m and n are fresh. The type $(\alpha/\gamma \to \beta/\delta)$ is translated to the type of a function which, given a parameter of type α^* and a continuation of type $\beta^* \to \gamma^*$, returns a value of type δ^*.

In [4], we proved that the CPS translation preserves types and equality.

Theorem 4 (Preservation of Types). *If $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable in $\lambda_{let}^{s/r}$, then $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \alpha^*) \to \beta^*$ is derivable in λ_{let}.*

If $\Gamma \vdash_p e : \tau$ is derivable in $\lambda_{let}^{s/r}$, then $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \gamma) \to \gamma$ is derivable for an arbitrary type γ in λ_{let}.

Theorem 5 (Preservation of Equality). *If $\Gamma;\ \alpha \vdash e_1 : \tau;\ \beta$ is derivable and $e_1 \rightsquigarrow^* e_2$ in $\lambda_{let}^{s/r}$, then $[\![e_1]\!] = [\![e_2]\!]$ in λ_{let} where $=$ is the least congruence relation containing $\rightsquigarrow$ in λ_{let}.*

Note that Theorem 5 only guarantees that the equality is preserved through the CPS translation. In fact, we cannot show that $e_1 \rightsquigarrow e_2$ implies $[\![e_1]\!] \rightsquigarrow^* [\![e_2]\!]$.

$$
\begin{aligned}
c^* &= c \\
x^* &= x \\
(\lambda x.e)^* &= \lambda x.[e] \\
[v] &= \lambda\kappa.\kappa v^* \\
[e_1 e_2] &= \lambda\kappa.[e_1](\lambda m.[e_2](\lambda n.mn\kappa)) \\
[\mathcal{S}k.e] &= \lambda\kappa.\mathtt{let}\ k = \lambda n\kappa'.\kappa'(\kappa n)\ \mathtt{in}\ [e](\lambda m.m) \\
[\langle e \rangle] &= \lambda\kappa.\kappa([e](\lambda m.m)) \\
[\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2] &= \lambda\kappa.\mathtt{let}\ x = [e_1](\lambda m.m)\ \mathtt{in}\ [e_2]\kappa \\
[\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3] &= \lambda\kappa.[e_1](\lambda m.\mathtt{if}\ m\ \mathtt{then}\ [e_2]\kappa\ \mathtt{else}\ [e_3]\kappa)
\end{aligned}
$$

Fig. 7. CPS translation for values and expressions.

$$
\begin{aligned}
e &::= c \mid x \mid \overline{\lambda}x.e \mid \underline{\lambda}x.e \mid e_1 \overline{@} e_2 \mid e_1 \underline{@} e_2 \\
&\quad \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 && \text{expression} \\
\alpha, \beta &::= t \mid b \mid \alpha \overline{\to} \beta \mid \alpha \underline{\to} \beta && \text{monomorphic type} \\
A &::= \alpha \mid \forall t.A && \text{polymorphic type}
\end{aligned}
$$

Fig. 8. Syntax of Two-Level Polymorphic Lambda Calculus.

4 Refined CPS Translation

The definitional CPS translation is useful in the semantic study of shift and reset. However, it does not preserve reductions, and hence cannot be used to prove SN. The failure of preservation of reduction is due to the fact that the CPS translation introduces a lot of administrative redexes through the translation.

To overcome this difficulty, we refine the definitional CPS translation so that it may produce fewer administrative redexes. There are several ways to define such optimized CPS translations since Plotkin proposed Colon Translation [16]. Here we use an extended version of two-level lambda calculus [7] as the target calculus of the translation, and define a refined CPS translation from $\lambda_{let}^{s/r}$ to it.

4.1 Two-Level Version of Polymorphic Lambda Calculus

In this subsection we introduce λ_{let}^{2L}, a two-level version of polymorphic typed lambda calculus (without control operators). In this calculus, function spaces are classified into two - static one and dynamic one. Accordingly, each occurrence of λ and application (explicitly denoted by "@") is annotated by overlines (static) as $\overline{\lambda}$ and $\overline{@}$, or underlines (dynamic) as $\underline{\lambda}$ and $\underline{@}$. In their original article, Danvy and Filinski classified every construct into two, but here we only classify lambda's and applications, and we assume that the other constructs are implicitly classified as dynamic ones.

Figure 8 gives the syntax of λ_{let}^{2L}, which is an annotated variant of λ_{let}.

Figure 9 gives the type system of λ_{let}^{2L}, where the type inference rules for (var), (const), (let), and (if) are the same as those in λ_{let}, and are omitted.

$$\frac{\Gamma, x:\alpha \vdash e:\beta}{\Gamma \vdash \overline{\lambda}x.e : \alpha \overline{\Rightarrow} \beta}\ \text{static fun}, x \in \mathtt{FV}(e) \qquad \frac{\Gamma, x:\alpha \vdash e:\beta}{\Gamma \vdash \underline{\lambda}x.e : \alpha \underline{\rightarrow} \beta}\ \text{dynamic fun}$$

$$\frac{\Gamma \vdash e_1 : \alpha \overline{\Rightarrow} \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1 \overline{@} e_2 : \beta}\ \text{static app} \qquad \frac{\Gamma \vdash e_1 : \alpha \underline{\rightarrow} \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1 \underline{@} e_2 : \beta}\ \text{dynamic app}$$

Fig. 9. Type Inference Rules of Two Level Polymorphic Calculus.

The crucial difference of the type system of λ^{2L}_{let} from that of λ_{let} (besides the annotations) is the side condition $x \in \mathtt{FV}(e)$ in the static function:

$$\frac{\Gamma, x:\alpha \vdash e:\beta}{\Gamma \vdash \overline{\lambda}x.e : \alpha \overline{\Rightarrow} \beta}\ \text{static fun}, x \in \mathtt{FV}(e)$$

The condition imposes that the abstracted variable x must occur freely in e.

We put this side condition by the following reason: we will use the static lambda abstraction to constitute an administrative redex (a redex which does not exist in the source expression, and is created by the CPS translation). When we prove that the CPS translation preserves reductions, it is important to guarantee that reducing an administrative redex does not discard any subexpressions, hence we put the side condition.

Note that the static lambda expression is not necessarily linear, namely, x may appear more than once in e, since CPS translating conditional expressions (if-then-else) may duplicate the arguments of continuations. Note also that the side condition is not applied to dynamic lambda abstraction which corresponds to lambda abstraction in the source expression. In other words, the actual continuations in $\lambda^{s/r}_{let}$ may discard their arguments.

The operational semantics of λ^{2L}_{let} is given as regarding the only redex as the static β-redex. Namely, the following single rule constitutes the notion of reduction in λ^{2L}_{let}:

$$(\overline{\lambda}x.e_1)\overline{@}e_2 \rightsquigarrow e_1[e_2/x]$$

Note that this is full β-reduction, rather than the call-by-value variant.

For this notion of reduction, we have subject reduction, strong normalization and confluence as follows.

Theorem 6 (Subject Reduction). *If $\Gamma \vdash e : \alpha$ is derivable in λ^{2L}_{let}, and $e \rightsquigarrow e'$ by reducing static β-redexes only, then $\Gamma \vdash e' : \alpha$ is derivable in λ^{2L}_{let}.*

Note that this theorem is not trivial, as we have a side condition in the typing rule for static lambda abstractions.

Proof. Since all static lambda abstractions $\overline{\lambda}x.e$ satisfy the side condition $x \in \mathtt{FV}(e)$, all free variables in $(\overline{\lambda}x.e)\overline{@}e'$ occur in $e[e'/x]$ freely, so the result of static β-reduction also satisfies the side condition, too.

Theorem 7. *Static reduction $\rightsquigarrow$ in λ^{2L}_{let} is confluent and strongly normalizing.*

Proof. Since we can embed λ^{2L}_{let} into the second order lambda calculus (where we only consider static reductions in λ^{2L}_{let}), strong normalization is apparent. We can easily prove that static reduction is Church-Rosser, since dynamic constructs are not reduced through the reduction.

$$
\begin{aligned}
b^* &= b \quad \text{for a basic type } b \\
t^* &= t \quad \text{for a type variable } t \\
(\alpha/\gamma \to \beta/\delta)^* &= \alpha^* \underline{\to} (\beta^* \underline{\to} \gamma^*) \underline{\to} \delta^* \\
(\forall t.A)^* &= \forall t.A^* \\
(\Gamma, x : A)^* &= \Gamma^*, x : A^*
\end{aligned}
$$

Fig. 10. Refined CPS translation for types.

By this theorem, for each expression e in λ^{2L}_{let}, its normal form uniquely exists (the normality is defined with respect to the static β-reduction). The normal form of e is denoted by $\mathtt{NF}(e)$.

4.2 Refined CPS Translation

The refined CPS translation is a syntax-directed translation from $\lambda^{s/r}_{let}$ to λ^{2L}_{let} where we use static constructs ($\overline{\lambda}$ and $\overline{@}$) for administrative redexes, and dynamic constructs ($\underline{\lambda}$, $\underline{@}$ and all other constructs) for source redexes.

Given an expression e in $\lambda^{s/r}_{let}$ and an expression K in λ^{2L}_{let}, we define an expression $[\![e, K]\!]$ in λ^{2L}_{let} as the CPS translation for e with respect to the continuation K. We first define the translation for types in Figure 10.

$$
\begin{aligned}
x^* &= x \\
c^* &= c \\
(\lambda x.e)^* &= \underline{\lambda} x.\underline{\lambda}\kappa.[\![e, \overline{\lambda} m.\ \kappa \underline{@} m]\!] \\
[\![v, K]\!] &= K \overline{@} v^* \\
[\![e_1\ e_2, K]\!] &= [\![e_1, \overline{\lambda} m_1.[\![e_2, \overline{\lambda} m_2.(m_1 \underline{@} m_2) \underline{@} (\underline{\lambda} n.\ K \overline{@} n)]\!]]\!] \\
[\![\langle e \rangle, K]\!] &= K \overline{@} [\![e, \overline{\lambda} m.m]\!] \\
[\![\mathcal{S}k.e, K]\!] &= \mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in} \\
&\qquad \mathtt{let}\ k = \underline{\lambda} n.\underline{\lambda}\kappa'.\kappa' \underline{@} (K \overline{@} n)\ \mathtt{in}\ [\![e, \overline{\lambda} m.m]\!] \\
[\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2, K]\!] &= \mathtt{let}\ x = [\![e_1, \overline{\lambda} m.m]\!]\ \mathtt{in}\ [\![e_2, K]\!] \\
[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3, K]\!] &= [\![e_1, \overline{\lambda} m.\mathtt{if}\ m\ \mathtt{then}\ [\![e_2, K]\!]\ \mathtt{else}\ [\![e_3, K]\!]]\!]
\end{aligned}
$$

Fig. 11. Refined CPS translation for expressions and values.

Figure 11 gives the CPS translation for expressions and values where m, m_1, m_2, n, κ, and κ' are fresh variables. In the definition for shift, we have added a redundant redex $\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in} \cdots$ for the purpose of SN proof.

The complete CPS transform of an expression e may be defined by $\mathcal{C}[\![e]\!] \equiv \underline{\lambda}\kappa.[\![e, \overline{\lambda} x.\kappa \underline{@} x]\!]$, though we do not need this definition in the proof of strong normalization.

Theorem 8 (Preservation of Types).

1. *Suppose* $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ *is derivable in* $\lambda_{let}^{s/r}$, $\Delta \vdash K : \tau^* \Rrightarrow \alpha^*$ *is derivable in* λ_{let}^{2L}, *and* Γ^*, Δ *is a valid type context in* λ_{let}^{2L}. *Then* $\Gamma^*, \Delta \vdash [\![e, K]\!] : \beta^*$ *is derivable in* λ_{let}^{2L}.
2. *Suppose* $\Gamma \vdash_p e : \tau$ *is derivable in* $\lambda_{let}^{s/r}$ *and* $\Delta \vdash K : \tau^* \Rrightarrow \gamma$ *is derivable in* λ_{let}^{2L}, *and* Γ^*, Δ *is a valid type context in* λ_{let}^{2L}. *Then* $\Gamma^*, \Delta \vdash [\![e, K]\!] : \gamma$ *is derivable in* λ_{let}^{2L}.

Proof. We can prove this theorem by induction on the derivation of $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ and $\Gamma \vdash_p e : \tau$. Here, we give proofs for a few cases.

(Case $e = e_1 e_2$) We assume that $\Gamma;\ \alpha \vdash e_1 e_2 : \tau;\ \beta$ is derivable in $\lambda_{let}^{s/r}$ and $\Delta \vdash K : \tau^* \Rrightarrow \alpha^*$ is derivable in λ_{let}^{2L}.

By inversion, we have

$$\Gamma;\ \gamma \vdash e_1 : (\sigma/\alpha \to \tau/\beta);\ \delta \qquad \text{in } \lambda_{let}^{s/r}$$
$$\Gamma;\ \beta \vdash e_2 : \sigma;\ \gamma \qquad \text{in } \lambda_{let}^{s/r}$$

Then by induction hypothesis on e_2, we have:

$$\Gamma^*, \Delta \vdash [\![e_2,\ \overline{\lambda} m_2.(m_1 \underline{@} m_2)\underline{@}(\underline{\lambda} n.\ K\overline{@}n)]\!] : \gamma^* \qquad \text{in } \lambda_{let}^{2L}$$

and by induction hypothesis on e_1, we have:

$$\Gamma^*, \Delta \vdash [\![e_1,\ \overline{\lambda} m_1.[\![e_2,\ \overline{\lambda} m_2.(m_1 \underline{@} m_2)\underline{@}(\underline{\lambda} n.\ K\overline{@}n)]\!]]\!] : \delta^* \qquad \text{in } \lambda_{let}^{2L}$$

Hence we are done.

(Case $\Gamma;\ \alpha \vdash \mathcal{S}k.e : \tau;\ \beta$) By inversion, we have

$$\Gamma, k : \forall t.(\tau/t \to \alpha/t);\ \sigma \vdash e : \sigma;\ \beta \qquad \text{in } \lambda_{let}^{s/r}$$

By induction hypothesis on e, we have:

$$\Gamma^*, k : \forall t.(\tau^* \underline{\to} (\alpha^* \underline{\to} t)\underline{\to} t), \Delta \vdash [\![e, \overline{\lambda} m.m]\!] : \beta^* \qquad \text{in } \lambda_{let}^{2L}$$

and then it is easy to derive:

$$\Gamma^*, \Delta \vdash \texttt{let } m_1 = \texttt{true in let } k = \underline{\lambda} n.\underline{\lambda}\kappa'.\kappa' \underline{@} (K\overline{@}n) \texttt{ in } [\![e, \overline{\lambda} m.m]\!] : \beta^* \qquad \text{in } \lambda_{let}^{2L}$$

hence we are done.

4.3 Summary of this Section

We can summarize the results in this section as the properties on the following translations:

$$\lambda_{let}^{s/r} \Longrightarrow \lambda_{let}^{2L} \Longrightarrow \lambda_{let}$$

In the first step, the refined CPS translation maps an expression in $\lambda_{let}^{s/r}$ to an expression in λ_{let}^{2L}. Theorem 8 guarantees that this step preserves the type.

In the second step, an expression in λ_{let}^{2L} is normalized to its unique normal form, which can be viewed as an expression in λ_{let} by removing all the overlines and underlines.[7] Theorem 6 guarantees that this step preserves the type.

We know that the calculus λ_{let} is strongly normalizing, since it can be embedded in, for instance, the second order lambda calculus [9]. Hence, in order to prove the strong normalizability of $\lambda_{let}^{s/r}$, it only remains to show that the composed translation from $\lambda_{let}^{s/r}$ to λ_{let} preserves reductions, which will be proved in the next section.

[7] Note that static constructs may remain in the normal forms.

5 Strong Normalization

In this section, we prove that reductions in $\lambda_{let}^{s/r}$ are preserved by the composed translation of the refined CPS translation and the static reduction in λ_{let}^{2L}.

Theorem 9 (Preservation of Reduction). *Suppose Γ; $\alpha \vdash e_1 : \tau$; β is derivable in $\lambda_{let}^{s/r}$, and $\Delta \vdash K : \tau^* \Rightarrow \alpha^*$ is derivable in λ_{let}^{2L}. Then we have:*

1. *If $e_1 \rightsquigarrow e_2$ by a reduction rule other than the reset-value reduction ($\langle v \rangle \rightsquigarrow v$), then $\mathit{NF}([\![e_1, K]\!]) \rightsquigarrow^+ \mathit{NF}([\![e_2, K]\!])$ in λ_{let}.*
2. *If $e_1 \rightsquigarrow e_2$ by the reset-value reduction ($\langle v \rangle \rightsquigarrow v$), then $\mathit{NF}([\![e_1, K]\!]) \equiv \mathit{NF}([\![e_2, K]\!])$ in λ_{let}.*

In the theorem above, we regard expressions in λ_{let}^{2L} as those in λ_{let} by erasing all overlines and underlines.

Proof. The first part of this theorem is proved by the case analysis of reduction rules used in $e_1 \rightsquigarrow e_2$.

- If the reduction is the call-by-value β reduction (the first reduction in Figure 2), or reductions for let, or conditional, then the theorem can be proved easily.
- For the reduction $\langle F[\mathcal{S}k.e] \rangle \rightsquigarrow \langle \mathtt{let}\ k = \lambda x.\langle F[x] \rangle\ \mathtt{in}\ e \rangle$, we first prove that $\mathtt{NF}([\![F[e], K]\!]) \equiv \mathtt{NF}([\![e, \overline{\lambda} m.[\![F[m], K]\!]]\!])$. This property can be easily proved by induction on F. Note that this property holds for typable expressions only.
  Then we can prove:

  $$\begin{aligned}
  &\mathtt{NF}([\![\langle F[\mathcal{S}k.e] \rangle, K]\!]) \\
  &\equiv \mathtt{NF}(K\overline{@}[\![F[\mathcal{S}k.e], \overline{\lambda} m.m]\!]) \\
  &\equiv \mathtt{NF}(K\overline{@}[\![\mathcal{S}k.e, \overline{\lambda} m'.[\![F[m'], \overline{\lambda} m.m]\!]]\!]) \\
  &\equiv \mathtt{NF}(K\overline{@}(\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in}\ \mathtt{let}\ k = \underline{\lambda} n.\underline{\lambda} \kappa'.\kappa'\underline{@}((\overline{\lambda} m'.[\![F[m'], \overline{\lambda} m.m]\!])\overline{@} n)\ \mathtt{in}\ [\![e, \overline{\lambda} m.m]\!])) \\
  &\equiv \mathtt{NF}(K\overline{@}(\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in}\ \mathtt{let}\ k = \underline{\lambda} n.\underline{\lambda} \kappa'.\kappa'\underline{@}[\![F[n], \overline{\lambda} m.m]\!]\ \mathtt{in}\ [\![e, \overline{\lambda} m.m]\!])) \\
  &\rightsquigarrow^+ \mathtt{NF}(K\overline{@}(\mathtt{let}\ k = \underline{\lambda} n.\underline{\lambda} \kappa'.\kappa'\underline{@}[\![F[n], \overline{\lambda} m.m]\!]\ \mathtt{in}\ [\![e, \overline{\lambda} m.m]\!]))
  \end{aligned}$$

  In the last step above, since K does not discard its argument by the side condition of the static lambda expression, the reduction $\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in}\ e_1 \rightsquigarrow e_1$ is preserved, and at least one step reduction occurs during this sequence. (Recall that we have added a dummy redex in the refined CPS translation of the shift expression.) We also have:

  $$\begin{aligned}
  &\mathtt{NF}([\![\langle \mathtt{let}\ k = \lambda x.\langle F[x] \rangle\ \mathtt{in}\ e \rangle, K]\!]) \\
  &\equiv \mathtt{NF}(K\overline{@}\mathtt{let}\ k = \underline{\lambda} x.\underline{\lambda} \kappa.\kappa\underline{@}[\![F[x], \overline{\lambda} m.m]\!]\ \mathtt{in}\ [\![e, \overline{\lambda} m.m]\!])
  \end{aligned}$$

  and therefore the resulting expressions are the same up to α-equivalence, hence we have:

  $$\mathtt{NF}([\![\langle F[\mathcal{S}k.e] \rangle, K]\!]) \rightsquigarrow^+ \mathtt{NF}([\![\langle \mathtt{let}\ k = \lambda x.\langle F[x] \rangle\ \mathtt{in}\ e \rangle, K]\!])$$

  The second part of this theorem is proved by a simple calculation, hence we are done.

We now give the strong normalization property for $\lambda_{let}^{s/r}$ as a theorem.

Theorem 10 (Strong Normalization). *If $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable in $\lambda_{let}^{s/r}$, then there is no infinite reduction sequence starting from e.*

Proof. Suppose there is an infinite reduction sequence $e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots$ in $\lambda_{let}^{s/r}$. Since the reset-value reduction ($\langle v \rangle \rightsquigarrow v$) cannot be applied to an expression infinitely many times, the reduction sequence must contain infinitely many reductions which are not the reset-value reduction. Then by Theorem 9, we have an infinite sequence $\mathtt{NF}(\mathcal{C}[\![e_1]\!]) \rightsquigarrow^+ \mathtt{NF}(\mathcal{C}[\![e_2]\!]) \rightsquigarrow^+ \cdots$. But, since the target calculus λ_{let} is a strongly normalizing calculus, we get contradiction.

Hence, $\lambda_{let}^{s/r}$ does not have an infinite reduction sequence.

As a corollary of strong normalization, we obtain confluence of $\lambda_{let}^{s/r}$, though it can be proved directly.

Theorem 11 (Confluence). *The notion of reduction in $\lambda_{let}^{s/r}$ is confluent.*

Proof. Since the reductions in $\lambda_{let}^{s/r}$ are not overlapping, they are weakly Church-Rosser (WCR). Church-Rosser property is subsumed by WCR and SN.

6 Conclusion

In this paper, we have presented a proof of strong normalization of the polymorphic calculus for shift and reset introduced by our previous work. The calculus allows let-polymorphism with a less restricted condition than the value restriction in ML families.

Let us emphasize that our proof is simple and easy to understand compared with the SN proofs for the calculi with call/cc and $\lambda\mu$, for which one needs more involved proof. The simplicity of our proof partly comes from the modularity of the proof, but mainly from the design of the control operators shift and reset and the naturality of the type system [4].

For future work, we plan to extend this result to the calculi with impredicative polymorphism given in [4]. Finding a better perspective of strong normalizability of calculi with various control operators is also left for future work.

Acknowledgments. We would like to thank Koji Nakazawa, Oleg Kiselyov and Chung-chieh Shan for helpful comments. The anonymous referees provided insightful comments.

The first author was partly supported by JSPS and FWF under the Japan-Austria Research Cooperative Program and by JSPS Grant-in-Aid for Scientific Research 20650003. The second author was partly supported by JSPS Grant-in-Aid for Scientific Research (C) 18500005.

References

1. Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of continuations and prompts. In *ICFP*, pages 40–53, 2004.
2. K. Asai. Logical Relations for Call-by-value Delimited Continuations. In *Trends in Functional Programming*, volume 6, pages 63–78, 2007.
3. K. Asai. On Typing Delimited Continuations: Three New Solutions to the Printf Problem. Technical Report OCHA-IS 07-1, Dept. of Information Science, Ochanomizu University, September 2007. 9 pages.
4. K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. In *APLAS*, pages 239–254, 2007.

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. A. Filinski. Representing Monads. In *POPL*, pages 446–457, 1994.
9. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
10. T. Griffin. A Formulae-as-Types Notion of Control. In *POPL*, pages 47–58, 1990.
11. R. Harper and M. Lillibridge. Explicit Polymorphism and CPS Conversion. In *POPL*, pages 206–219, 1993.
12. Y. Kameyama and M. Hasegawa. A Sound and Complete Axiomatization for Delimited Continuations. In *ICFP*, pages 177–188, 2003.
13. Y. Kameyama and T. Yonezawa. Typed Dynamic Control Operators for Delimited Continuations. In *FLOPS, LNCS 4989*, pages 239–254, 2008.
14. O. Kiselyov. Simply Typed Lambda-Calculus with a Typed-Prompt Delimited Control is not Strongly Normalizing. 2006. `http://okmij.org/ftp/Computation/Continuations.html`.
15. O. Kiselyov. Demo of Persistent Delimited Continuations in OCaml for Nested Web Transactions. In *Talk presented at Continuation Fest2008, Tokyo, Japan*, 2008.
16. G. D. Plotkin. Call-by-Name, Call-by-Value, and the λ-Calculus. *Theoretical Computer Science*, 1:125–159, 1975.
17. P. Thiemann. Combinators for Program Generation. *J. Funct. Program.*, 9(5):483–525, 1999.
18. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.