Towards Logical Understanding of Delimited Continuations

Yukiyoshi Kameyama
Graduate School of Informatics, Kyoto University
kameyama@kuis.kyoto-u.ac.jp

ABSTRACT

The aim of this paper is to give a *logical* understanding of control operators for delimited (partial) continuations, in particular, Danvy and Filinski's shift/reset. Starting with a simple type system for a static analogue of the shift/reset operators, we analyze the type structure in a logical way, which gives rise to control operators for delimited continuations. The resulting control operators nicely behave in the sense that it satisfies strong normalization and other nice properties, and we can use them with a higher order types. We finally compare our control operators with the original shift/reset operators.

KEYWORDS

Delimited Continuations, Type system, Classical Logic

1 Introduction

Delimited continuation (often called *partial continuation*) is a notion representing a *partial* rest of the computation. There have been many proposals of control operators for delimited continuations, but our understanding for them does not seem to reach at a satisfactory level as in the case of the `call/cc` operator.

This paper tries to give a better understanding of delimited continuations. To do so, we use types and their logical meaning as our guide, namely the Curry-Howard isomorphism is our guiding principle. We start with a simple type system for control operators of delimited continuations, then study its type structure and logical meaning, obtain suitable representation (encoding) of control operators, and finally obtain reduction rules which are induced from the types of the encoded terms.

Our result is that, the naturally arising control operators are not exactly the same as any of existing ones, but they have the same operational meaning if there is only one invocation of delimited continuations.[1] Since our study is done in the framework of well understood logical machinery with nice properties, the resulting operator is ensured to have a number of desirable properties, such as subject reduction and strong normalizability, and also a type-preserving CPS-translation is obtained without restricting the type of the reset operator to atomic.

This paper is organized as follows: Section 2 gives the calculus for the catch/throw mechanism which will be used in later sections. In Sections 3, we give a simple type-theoretic formulation of Danvy and Filinski's shift/reset operators, and in Section 4, we analyze the logical structure of their static analogue to obtain two views for them. In Section 5, we obtain two computational interpretations corresponding to the two views, and show one interpretation is a control operator for delimited continuations. In Section 6, we study the properties of the obtained control operator. Section 7 gives concluding remarks.

2 Preliminary: the Catch/Throw Calculus

This section briefly introduces the static catch/throw calculi developed by Nakano, Sato, and the author [10, 12, 8]. The catch/throw calculus is used to interpret the calculus for delimited continuations in later sections.

Remark 1. The use of the word "static" in this paper may be different from [2].

In this paper, "static" means that the corresponding `catch`-construct for a `throw`-construct is determined in a static way, that is, at the time when they are written. On the contrary, the actual catch/throw mechanism in Common Lisp (and the exception mechanism in Standard ML) is "dynamic" in the sense that the corresponding `catch`-construct is determined when `throw` is evaluated. The difference is illustrated by the following example:

[1] Of course, the invoked continuation objects can be used arbitrary times.

(Static)

$$\begin{array}{ll} & \mathtt{catch}_\alpha((\lambda x.1+\mathtt{catch}_\alpha(x0))(\lambda y.\mathtt{throw}_\alpha y)) \\ \to & \mathtt{catch}_\alpha(1+\mathtt{catch}_\beta((\lambda y.\mathtt{throw}_\alpha y)0)) \\ \to & \mathtt{catch}_\alpha(1+\mathtt{catch}_\beta(\mathtt{throw}_\alpha 0)) \\ \to & \mathtt{catch}_\alpha(\mathtt{throw}_\alpha 0) \\ \to & 0 \end{array}$$

Note that, in the static calculus the tag variable α was renamed to β to avoid the unwanted capture of the free tag variable. The corresponding `catch`-construct for `throw` is the outer one.

(Dynamic)

$$\begin{array}{ll} & \mathtt{catch}_\alpha((\lambda x.1+\mathtt{catch}_\alpha(x0))(\lambda y.\mathtt{throw}_\alpha y)) \\ \to & \mathtt{catch}_\alpha(1+\mathtt{catch}_\alpha((\lambda y.\mathtt{throw}_\alpha y)0)) \\ \to & \mathtt{catch}_\alpha(1+\mathtt{catch}_\alpha(\mathtt{throw}_\alpha 0)) \\ \to & \mathtt{catch}_\alpha(1+0) \\ \stackrel{*}{\to} & 1 \end{array}$$

In this case, the `throw`-expression is captured by the inner `catch`-construct.

Although our terminology differ from Danvy and Filinski's, ours coincides with the terminology of the dynamic/static binding of variables. Later we shall see the shift/reset operators of Danvy and Filinski are dynamic in our sense. **(End of Remark 1.)**

2.1 Type System of $\mathrm{LK}_{c/t}$ and $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$

Among several variants of static catch/throw calculi, we take here $\mathrm{LK}_{c/t}$ in [8] and its call-by-value subcalculus $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$. The formulation of $\mathrm{LK}_{c/t}$ is essentially due to Nakano [10], but since he wanted to confine himself to intuitionistic logic, he put a restriction on the formation rule of λ (the implication-introduction rule). In our earlier works, we showed that the calculus without the restriction is more natural and has more applications in higher-order programming and program extraction from classical proofs.

The type system of $\mathrm{LK}_{c/t}$ and that of $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ are the same, which we will be given below.

Types are given by the grammar:

$$A, B ::= K \mid A \to B$$

where K ranges over atomic types. Raw terms are

$$M, N ::= x \mid \lambda x.M \mid MN \mid \mathtt{catch}_\alpha M \mid \mathtt{throw}_\alpha M$$

where x ranges over individual (normal) variables, and α ranges over tag variables.[2] The individual variable x is bound in $\lambda x.M$ and the tag variable α is bound in $\mathtt{catch}_\alpha M$. The bound/free occurrences of individual/tag variables are determined from these notions in a usual way. We identify two raw terms which are the same modulo the renaming of bound individual/tag variables. For instance, $\lambda x.\mathtt{catch}_\alpha(\mathtt{throw}_\alpha x)$ and $\lambda y.\mathtt{catch}_\beta(\mathtt{throw}_\beta y)$ are identified. $FV(M)$ and $FTV(M)$ denote the set of free individual/tag variables in M, respectively. $M\{x := N\}$ and $M\{\alpha := \beta\}$ denote the usual capture-avoiding substitutions.

A judgement is in the form $\Gamma \vdash M : A\ ;\ \Delta$ where M is a raw term, A is a type, Γ is a finite set of declarations of the form $x : A$, and Δ is a set of declarations of the form $\alpha : A$. Note that Γ denotes the set of free individual variables, and Δ denotes that of free tag variables. We write Δ in the righthand side of the judgement because, when we consider the Curry-Howard correspondence, this form is more natural than the form $\Gamma\ ;\ \Delta \vdash M : A$.

We now give type inference rules in the natural-deduction style which are used to infer judgements.

$$\frac{}{\Gamma \vdash x : A\ ;\ \Delta}\ (\text{if}\ \ x : A \in \Gamma)$$

$$\frac{\Gamma \vdash M : B\ ;\ \Delta}{\Gamma - \{x : A\} \vdash \lambda x.M : A \to B\ ;\ \Delta}$$

$$\frac{\Gamma \vdash M : A \to B\ ;\ \Delta \quad \Gamma \vdash N : A\ ;\ \Delta}{\Gamma \vdash MN : B\ ;\ \Delta}$$

$$\frac{\Gamma \vdash M : A\ ;\ \Delta}{\Gamma \vdash \mathtt{throw}_\alpha M : B\ ;\ \Delta \cup \{\alpha : A\}}$$

$$\frac{\Gamma \vdash M : A\ ;\ \Delta}{\Gamma \vdash \mathtt{catch}_\alpha M : A\ ;\ \Delta - \{\alpha : A\}}$$

If $\Gamma \vdash M : A\ ;\ \Delta$ is inferred using the above inference rules, we say M is a term of type A under Γ and Δ.

2.2 The Curry-Howard Isomorphism

The Curry-Howard isomorphism relates the simply typed lambda calculus with the intuitionistic propositional logic.[3] This isomorphism can be extended to the above type system and the classical propositional logic.

To see this, let us rewrite the type inference rules for catch and throw where (i) terms are omitted, (ii) $A;\Delta$ is written as $A, B_1, \cdots, B_n$ if $\Delta = \{B_1, \cdots, B_n\}$.

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash B, A, \Delta}\ (\text{throw})$$

$$\frac{\Gamma \vdash A, \cdots A, \Delta}{\Gamma \vdash A, \Delta}\ (\text{catch})$$

The first (throw) rule is the weakening rule, and the second (catch) rule is (n-times application of) the contraction rule. Namely, the essence of the catch/throw mechanism can be captured as the well known logical inference rules; weakening and contraction. Parigot's $\lambda\mu$-calculus [11] has essentially the same inference rules as above.

2.3 Operational Semantics of $\mathrm{LK}_{c/t}$

The operational semantics of $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ is given in the evaluation-context semantics style as follows.

First, values are defined as usual:

[2] We assume that the set of individual variables and the set of tag variables are disjoint.

[3] Strictly speaking, the corresponding logic is minimal logic, since we do not have the bottom-elimination rule.

$$V \quad ::= \quad x \mid \lambda x.M$$

An evaluation context (in the call-by-value semantics) is defined as follows:

$$E \quad ::= \quad [\,] \mid EM \mid VE \mid \mathtt{throw}_\alpha E \mid \mathtt{catch}_\alpha E$$

$E[M]$ denotes the term obtained by replacing the hole $[\,]$ in E by M.

The 1-step reduction in $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ is defined as the union of the following relations:

$$\begin{array}{rcl}
E[(\lambda x.M)V] & \to & E[M\{x := V\}] \\
E[\mathtt{catch}_\alpha V] & \to & E[V] \quad \text{if } \alpha \notin FTV(V) \\
E[\mathtt{catch}_\alpha(\mathtt{throw}_\alpha V)] & \to & E[\mathtt{catch}_\alpha V] \\
E[(\mathtt{throw}_\alpha V)M] & \to & E[\mathtt{throw}_\alpha V] \\
E[V_1(\mathtt{throw}_\alpha V_2)] & \to & E[\mathtt{throw}_\alpha V_2] \\
E[\mathtt{throw}_\beta(\mathtt{throw}_\alpha V)] & \to & E[\mathtt{throw}_\alpha V] \\
E[\mathtt{catch}_\beta(\mathtt{throw}_\alpha V)] & \to & E[\mathtt{throw}_\alpha V] \quad \text{if } \beta \notin FTV(V)
\end{array}$$

where in the last two rules we assumed $\alpha \not\equiv \beta$.

The calculus $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ gives a simple formulation of the static catch/throw calculus. It is a very weak calculus in the sense that the actual (dynamic) catch/throw mechanism cannot be simulated, but we can recover the lost expressiveness to some extent by introducing the abstraction mechanism of tag variables [8]. Also, the calculus cannot reduce non-value terms such as $(\mathtt{catch}_\alpha \lambda x.M)V$ if M contains free occurrences of α, but this defect can be compromised by introducing the structural reduction in the $\lambda\mu$-calculus. In the scope of this paper, the weakness of the reduction rules do not have problems (in particular, we can give encoding of control operators in this weak calculus).

$\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ can be considered as a subcalculus of (the call-by-value variant of) Parigot's $\lambda\mu$-calculus [11], so our interpretation in later sections can be done within the $\lambda\mu$-calculus. We do not do so in this paper, since there are several variants of the call-by-value $\lambda\mu$-calculus, and few studies on the non-deterministic version of $\lambda\mu$-calculus have been studied (which contains the call-by-name and the call-by-value fragments), while the catch/throw calculus is sufficiently simple and its properties have been already studied intensively.

2.4 Operational Semantics of $\mathrm{LK}_{c/t}$

The operational semantics of $\mathrm{LK}_{c/t}$ is given by changing the reduction rules of $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ as follows:

- All the occurrences of V, V_1, V_2 are replaced by arbitrary terms M, M_1, M_2.
- The following rule is added:

$$E[\lambda x.\mathtt{throw}_\alpha M] \to E[\mathtt{throw}_\alpha M] \quad \text{if } x \notin FV(M)$$

Apparently the computation of $\mathrm{LK}_{c/t}$ is non-deterministic. Nevertheless it is strongly normalizing [8].

3 Shift/Reset Operators and Their Static Analogue

3.1 Shift/Reset Operators

Danvy and Filinski's shift/reset operators [1, 2] are the most well known (and probably the most widely used) control operators for delimited continuations. The reset operator (the delimiter) is denoted as $\#M$ (or $\langle M\rangle$ in their original notation), and the shift operator is $\xi k.M$ where the variable k is bound by this ξ. Although the operational semantics of shift/reset is defined via the CPS-transformation, we can consider the following reduction rule for shift/reset:

$$E[\#(E'[\xi k.M])] \to E[\#M\{k := \lambda u.\#(E'[u])\}]$$

The continuation up to the closest delimiter # is captured as a functional object $\lambda u.\#(E[u])$ and can be used in later computation. Note that, unlike the standard (full) continuation generated by `call/cc` in Scheme and SML/NJ, the created continuation is a partial rest of computation (it is not the whole rest of computation like $\lambda u.E[E'[u]]$). Another important difference between delimited/full continuations is that the full continuation is abortive (not composable) while the delimited continuation can be composed. It is illustrated by the following example:

$$\begin{array}{rcl}
\#(1 + (\xi k.k(k(0)))) & \to & \#(k(k(0)))\{k := \lambda u.\#1 + u\} \\
& \xrightarrow{*} & 1 + 1 + 0 \\
& \xrightarrow{*} & 2
\end{array}$$

On the contrary, the term $1+\mathbf{call/cc}\lambda k.k(k(k(0)))$ evaluates to 1, since the current continuation is aborted when k is applied to 0, and $k(k(k(0)))$ and $k(0)$ result in the same answer.

Note that, the shift/reset operators are *dynamic* in our sense, since $(\lambda x.\#(xy))(\lambda z.\xi k.M)$ reduces to $\#((\lambda z.\xi k.M)y)$, and $\#((\#\lambda x.\xi k.M)y)$ reduces to $\#((\lambda x.\xi k.M)y)$. In both cases, the reset operator corresponding to the shift operator is different from the lexically determined one.

Remark 2. In [1], the shift/reset operators are said to be *static* in comparison with Felleisen's *dynamic* operators. The functional object generated by an invocation of the shift operator is $\lambda u.\#E[u]$, while Felleisen's operator generates a functional object like $\lambda u.E[u]$. Suppose E contains another occurrence of shift. Then it is delimited by a fixed delimiter in the former case, and by an unknown outer delimiter in the latter case. This difference is the reason why Danvy and Filinski's operator enjoy a simple, elegant CPS-transformation. **(End of Remark 2)**

3.2 Type System

We then try to give types to the shift/reset operators. One may think that the following typing rules are appropriate for the shift/reset operators:

$$\frac{\Gamma, k : A \to B \vdash M : B \ ; \ \Delta}{\Gamma \vdash \xi\ k.M : A \ ; \ \Delta \cup \{B\}}\ (\text{shift})$$

$$\frac{\Gamma \vdash M : B \ ; \ \Delta}{\Gamma \vdash \#M : B \ ; \ \Delta - \{B\}}\ (\text{reset})$$

However, this type system does not work since the shift/reset operators are dynamic in our sense. In other words, this type system does not enjoy the subject reduction property, which we think is the minimum requirement for a type system being sound.

In order to overcome this problem, Murthy [9] introduced a rather elaborated type system for the hierarchy of shift/reset's. He changed the function type $A \to B$ to $((A \to B) \to T_i) \to T_i$ where T_i is the type of i-th delimiter, that is, he annotates every function type by T_i, which complicates the type structures a bit, and his type system is not a (simple) extension of the simply typed lambda calculus. Thielecke [13] also considers a type system to cope with dynamic features of control operators where he uses the notation $A \to B \vee C$, which is essentially the same as $((A \to B) \to C) \to C$.

Since our aim is to find an essential logical structure of delimited continuation operators, we do not want to treat such a complicated system. Instead, we stick to a simple type system, and by analyzing it in a purely logical way, we are interested in seeing what control operators arise from the type system.

In summary, we address the problem to find a computational interpretation of the following typing rules.

$$\frac{\Gamma, A \to B \vdash B \ ; \ \Delta}{\Gamma \vdash A \ ; \ \Delta \cup \{B\}} \ (\text{shift}) \qquad \frac{\Gamma \vdash B \ ; \ \Delta}{\Gamma \vdash B \ ; \ \Delta - \{B\}} \ (\text{reset})$$

Remark 3. Since we take the Curry-Howard isomorphism as the guide of our study, the corresponding calculus always becomes static (in our sense). Hieb et al [6] proposed control operators for subcontinuations, which is another variant of delimited continuation operators, but are static in our sense. Hence, our study is closer to subcontinuations than shift/reset.

4 Logical Analysis

We first note that the first (shift) type inference rule plays two roles at the same time; (i) it discharges the type $A \to B$ and (ii) it introduces the type A in the righthand side, that is, it works as the weakening rule. Reflecting these two roles, we split the shift rule into the two rules and obtain the following set of rules:

$$\frac{\Gamma, A \to B \vdash A \ ; \ \Delta \cup \{B\}}{\Gamma \vdash A \ ; \ \Delta \cup \{B\}} \ (*)$$

$$\frac{\Gamma \vdash B \ ; \ \Delta}{\Gamma \vdash A \ ; \ \Delta \cup \{B\}} \ (\text{throw})$$

$$\frac{\Gamma \vdash B \ ; \ \Delta}{\Gamma \vdash B \ ; \ \Delta - \{B\}} \ (\text{catch})$$

In this formulation the shift rule splits into the first and the second rules by which the shift rule is derivable.

Now it is easy to see that the second (throw) and the third (catch) rules are the same as the catch and the throw rules given in the last section, namely, the weakening and the contraction rules. These rules are well known inference rules in formulating classical propositional logic, so, we can say that the logical meaning of these rules are well understood.

Our remaining task is to understand the first rule (marked with *). Since we use the Curry-Howard isomorphism as the guiding principle, we should have a consistent logical system. If A is proved by the above set of inference rules, namely, $\{\} \vdash A \ ; \ \{\}$ is derived, then A must be valid in some suitable logic, say, classical logic. By considering how to ensure this property, we can think of the following two views:

(First View) The (*) rule is a stand-alone valid rule in classical logic. Namely, we can regard it as a natural extension of Peirce's inference rule below:

$$\frac{\Gamma, A \to B \vdash A}{\Gamma \vdash A}$$

This rule is admissible in classical logic, that is, derivable from the catch/throw rules. We give a proof of the (*) rule using the catch/throw rules as follows (where we suppress to write Γ and Δ for readability):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A;}}{A \vdash B; A}}{\vdash A \to B; A} \qquad \dfrac{\begin{array}{c}\vdots \\ A \to B \vdash A;\end{array}}{\vdash (A \to B) \to A;}}{\vdash A; A}}{\vdash A;}$$

(Second View) The (*) rule is not valid as a stand-alone rule, but with the combination of the catch rule it is valid. The situation is similar to the throw rule, which is not valid as a stand-alone rule, but is valid with the combination of the catch rule. In order to have a proof of $\{\} \vdash A \ ; \ \{\}$, if there is an application of the (*) rule with B introduced in the righthand side, then some application of the catch rule must exist under it which discharges the same B. So, each occurrence of the (*) rule must be accompanied with an occurrence of the catch rule:

$$\begin{array}{c} \vdots \\ \dfrac{\Gamma, A \to B \vdash A \ ; \ \{B\} \cup \Delta}{\Gamma \vdash A \ ; \ \{B\} \cup \Delta} \\ \vdots \\ \dfrac{\Gamma' \vdash B \ ; \ \{B\} \cup \Delta'}{\Gamma' \vdash B \ ; \ \Delta'} \end{array}$$

In [7], we showed that this combination is provable in intuitionistic logic. There could be many ways to give its proof, but the presumably simplest way is given as follows (where we again suppress to write Γ, Δ and so on):

$$\begin{array}{c} \dfrac{\dfrac{\begin{array}{c}\vdots \\ A \to B \vdash A; B\end{array}}{\vdash (A \to B) \to A; B} \qquad \dfrac{\begin{array}{c} A \vdash A; B \\ \vdots \\ A \vdash B; B\end{array}}{\vdash A \to B; B}}{\vdash A; B} \\ \vdots \\ \dfrac{\vdash B; B}{\vdash B;} \end{array}$$

Note that, the right-upper subproof of this proof is taken from the lower subproof of the same proof, so that part is duplicated. Note also that this proof is written by intuitionistic inference rules only, so it is valid in intuitionistic logic.

Corresponding to these two views, we obtain two different computational interpretations of the (*) rule, which we will exploit in the next section.

5 Computational Interpretations

So far we have been concentrating on the types of (the static analogue of) the delimited continuation operators. In this section we consider the term assignment which arises from the interpretation of typing rules, and see what would be their natural reduction rules.

As in the case of the catch/throw calculi, we attach an individual variable (denoted as $x, y, \cdots$) to each element of Γ, and a tag variable (denoted as $\alpha, \beta, \cdots$) to each element of Δ. For brevity, we use the same symbols ξ and # to the control operators obtained in this analysis, but here these operators are annotated by tag variables, hence we will have terms like $\xi_\alpha k.M$ and $\#_\alpha M$.

5.1 The First View

From the first view, we obtain the following definitions:

$$\begin{aligned} \xi_\alpha k.M &\triangleq \mathtt{catch}_{\alpha'}((\lambda k.M)(\lambda x.\mathtt{throw}_{\alpha'} x)) \\ \#_\alpha M &\triangleq \mathtt{catch}_\alpha M \end{aligned}$$

where α' is a new tag.

This interpretation is not interesting; there are no throw's with the tag α in the representation of the term $\xi_\alpha k.M$, so this operator becomes the simple `call/cc`-operator, and the reset operator becomes identity.

A slightly more interesting interpretation is the following:

$$\begin{aligned} \xi_\alpha k.M &\triangleq \mathtt{catch}_{\alpha'}(\mathtt{throw}_\alpha((\lambda k.M)(\lambda x.\mathtt{throw}_{\alpha'} x))) \\ \#_\alpha M &\triangleq \mathtt{catch}_\alpha M \end{aligned}$$

At a first sight, this may look interesting, since $\#_\alpha(E[\xi_\alpha k.M])$ reduces to $\#_\alpha M\{k := \lambda x.\mathtt{throw}_\alpha E[x]\}$, which captures the delimited continuation E. However, it turns out another representation of the `call/cc` mechanism, since the reified continuation-like object $\lambda x.\mathtt{throw}_\alpha E[x]$ contains the throw operator in it, and it discards the current continuation (recall that an important property of the delimited continuation is that it is composable like normal functions on the contrary to the abortive object generated by **call/cc**.

In either case, the first view does not lead to a proper computational interpretation for delimited continuations.

5.2 The Second View

Our second view implies the following definition:

$$\begin{aligned} \xi_\alpha k.M &\triangleq \alpha(\lambda k.\mathtt{throw}_{\alpha'} M) \\ \#_\alpha M &\triangleq \mathtt{catch}_{\alpha'}(\lambda v.v(\lambda m.m\lambda u.v(\lambda y.u)))(\lambda\alpha.M) \end{aligned}$$

where α' is a new tag.

Let E be an evaluation context which contains exactly one instance of a hole, namely the term $E[\xi_\alpha k.M]$ contains exactly one occurrence of ξ. Let R be $\lambda v.v(\lambda m.m\lambda u.v(\lambda y.u))$. Then we have the following reduction sequence:

$$\begin{aligned} & \#_\alpha E[\xi_\alpha k.M] \\ \equiv\ & \mathtt{catch}_{\alpha'}\mathtt{R}(\lambda\alpha.E[\alpha(\lambda k.\mathtt{throw}_{\alpha'} M)]) \\ \stackrel{*}{\to}\ & \mathtt{catch}_{\alpha'}(\lambda\alpha.E[\alpha(\lambda k.\mathtt{throw}_{\alpha'} M)])(\lambda m.m\lambda u.E[u]) \\ \to\ & \mathtt{catch}_{\alpha'} E[(\lambda m.m\lambda u.E[u])(\lambda k.\mathtt{throw}_{\alpha'} M)] \\ \stackrel{*}{\to}\ & \mathtt{catch}_{\alpha'} E[\mathtt{throw}_{\alpha'} M\{k := \lambda u.E[u]\}] \end{aligned}$$

Let N be the final term of the above reduction sequence. Suppose $M\{k := \lambda u.E[u]\}$ reduces to a value V, then N reduces to $\mathtt{catch}_{\alpha'} E[\mathtt{throw}_{\alpha'} V]$, and then reduces to V. Suppose $M\{k := \lambda u.E[u]\}$ reduces to $\mathtt{throw}_\beta V$. Then N also reduces to $\mathtt{throw}_\beta V$. Consequently, two terms N and $M\{k := \lambda u.E[u]\}$ are observationally equivalent. In summary, the behavior of the term $\#_\alpha E[\xi_\alpha k.M]$ is the same as that of $M\{k := \lambda u.E[u]\}$.

This result shows that the computational behavior of our new control operators are the same as that of the original shift/reset operators. In particular, the result of the above computation contains a functional object $\lambda u.E[u]$, which represents the intended delimited continuation object, thus our encoding of # and ξ can be thought as another representation of control operators for delimited continuations.

If E contains more than one hole, the result may not be the same as that of the original shift/reset. That is, our encoding generated a similar, but different control operator than the existing one.

Yet, we can argue that a large number of examples contains only one occurrence of the shift operator; Danvy's flip-flop, Danvy-Filinski's CPS-translation [2], and Thiemann's partial evaluator [14]; for these examples, our encoding can be thought as providing a sound logical foundations, by which we can reason about properties of controlful programs.

In this analysis, we used the static catch/throw calculus, but the encoding can be done by the call-by-value version of Parigot's $\lambda\mu$-calculus.

6 Implications of Our Encoding

Since our encoding of control operators uses only logically established machinery which enjoys a number of good properties, the following are obtained straightforwardly:

1. Higher-orderness; types of the delimiters were limited to atomic types for shift/reset in order to obtain a type-preserving CPS-translation, while arbitrary types are allowed in our case without loss of the type-preserving CPS-translation and the strong normalization.
2. Properties such as subject reduction, confluence, and strong normalization are obtained automatically from those for the static catch/throw calculus.
3. A simple type-preserving CPS-translation is obtained by the type-preserving CPS-translation of the call-by-value version of $\lambda\mu$-calculus.

Since most of these results are apparent, we only mention a few properties and the CPS-translation in what follows.

6.1 Subject Reduction and Strong Normalization

Our encoding of control operators is within the scope of $\mathrm{LK}^{\mathrm{CBV}}_{c/t}$, they inherit nice properties from the calculus. In [8], we proved the subject reduction and the strong normalization of the non-deterministic version of the catch/throw calculi. This version contains the call-by-value version and the call-by-name version of catch/throw calculi, so these properties for $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ is guaranteed.

Theorem 1 *The calculus for the control operators (encoded using the catch/throw operators) enjoys the subject reduction property.*

Theorem 2 *The calculus for the control operators (encoded using the catch/throw operators) are strongly normalizing.*

The latter result implies that our operators differ from the original shift/reset control operators, since we can write non-terminating reduction sequence using the shift/reset operators [3].

6.2 CPS-translation

A CPS-translation of $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ is given in a standar way. First, we give translation of a value V (denoted as V^*).

$$\begin{array}{rcl} x^* & \equiv & x \\ (\lambda x.M)^* & \equiv & \lambda x.[[M]] \end{array}$$

Then the CPS-transform of a term M (denoted as $[[M]]$) is given by:

$$\begin{array}{rcl} [[V]] & \equiv & V* \\ [[MN]] & \equiv & \lambda k.[[M]]\lambda m.[[N]]\lambda n.mnk \\ [[\mathtt{catch}_\alpha M]] & \equiv & \lambda \alpha.[[M]]\alpha \\ [[\mathtt{throw}_\alpha M]] & \equiv & \lambda k.[[M]]\alpha \end{array}$$

Considering these definitions and by simple calculation, we can easily obtain the CPS-transform of obtained control operators.

$$[[\xi_\alpha c.M]] \quad \stackrel{*}{\to} \quad \lambda k.\alpha(\lambda c.\lambda k'.[[M]]\alpha')k$$

$$\begin{array}{l} [[\#_\alpha M]] \\ \stackrel{*}{\to} \quad \lambda \alpha'.[[M]]\{\alpha := (\lambda m.m\lambda u.(\lambda \alpha.[[M]])(\lambda yL.Lu))\}\alpha' \end{array}$$

Compared to the simple CPS-translation given to the shift/reset operators, the result of ours is very complex, but we can obtain the following information:

(i) Our reset operator (#) creates a functional object $\lambda u.(\lambda \alpha.[[M]])(\lambda yL.Lu)$ which is roughly an interpretation of $\lambda u.M\{\alpha := u\}$.

(ii) $\lambda yL.Lu$ will be substituted for the α in the second occurrence of $[[M]]$, which means that the second call of our shift operator (ξ) does not really invoke the reified delimited continuation. Rather, it simply ignores it. (Recall that our encoding coincides with the real shift/reset operators only when there is only one occurrence of the shift operator.)

7 Conclusion: What did We Obtain ?

Our conclusion of this expository work is that, a static aspect of delimited continuation operators can be grasped using the sound logical system plus some encoding. We used the static catch/throw calculus as the base system in this paper, but the call-by-value variant of $\lambda\mu$-calculus can be also used (and can be better, since it has stronger reductions).

Apparently, our result does not give full understanding for shift/reset operators; our result applies only to the static variants (such as Hieb et al's subcontinuations), and also is limited to side-effect free calculi (since we duplicated the context). On the other hand, the static nature can be compromised by using the abstraction mechanism of the tag α, and we believe that this work gives a first step towards logical understanding of control operators for delimited continuations.

Acknowledgement The author would like to thank Olivier Danvy, Andrzej Filinski and Hayo Thielecke for helpful discussions. He also thanks Amr Sabry and anonymous referees of CW'01 for constructive comments. This work is supported in part by Inamori Foundation and Grant-in-Aid for Scientific Research by Ministry of Education, Japan, No. 11780213.

REFERENCES

[1] O. Danvy and A. Filinski, "Abstracting Control", Proc. 1990 ACM Conference on Lisp and Functional Programming, pp. 151-160, 1990.

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

[3] O. Danvy, private communication, 2000.

[4] P. de Groote, "A CPS-Translation of the $\lambda\mu$-Calculus", Lecture Notes in Computer Science **787**, pp. 85-99, 1994.

[5] M. Felleisen, "The Theory and Practice of First-Class Prompts", Proc. 15th ACM Symp. on Principles of Programming Languages, pp. 180-190, 1988.

[6] R. Hieb, R. Dybvig, and C. W. Anderson, "Subcontinuations", Lisp and Symbolic Computation 6, pp. 453–484, 1993.

[7] Y. Kameyama, "A Type-theoretic Study on Partial Continuations", Proc. IFIP International Conference on Theoretical Computer Science (IFIP TCS2000), Sendai, Japan, Lecture Notes in Computer Science **1872**, pp.489-504, 2000.

[8] Y. Kameyama and M. Sato, "Strong Normalizability of the Non-deterministic Catch/Throw Calculi", *Theoretical Computer Science*, to appear.

[9] C. Murthy, "Control Operators, Hierarchies, and Pseudo-Classical Type Systems: A-Translation at Work", Proc. ACM Workshop on Continuations, pp. 49–71, 1992.

[10] H. Nakano, "A Constructive Formalization of the Catch and Throw Mechanism", *Conf. Rec. IEEE Symposium on Logic in Computer Science*, pp. 82–89, 1992.

[11] M. Parigot, "$\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction", Proc. Intl Conf. on Logic Programming and Automated Reasoning, *Lecture Notes in Computer Science* **624** pp. 199–201, 1992.

[12] M. Sato, "Intuitionistic and Classical Natural Deduction Systems with the Catch and the Throw Rules", *Theoretical Computer Science* 175 (1), pp. 75-92, 1997.

[13] H. Thielecke, private communication, 2000.

[14] P. Thiemann, "Cogen in Six Lines", International Conference on Functional Programming, pp. 180-189, 1995.