Dynamic Control Operators in Type Theory

Yukiyoshi Kameyama

Institute of Information Sciences and Electronics, University of Tsukuba
kam@is.tsukuba.ac.jp

Abstract. We study control operators in functional programming languages. While *static* control operators such as `call/cc` in Scheme have been intensively studied and connection with classical logic have been found, *dynamic* control operators have so far attracted much less attention, since realistic dynamic control operators such as the exception mechanism in Standard ML have several bad features from the type-theoretic viewpoint; a typable program may cause run-time errors and may not terminate. This paper shows that dynamic control operators themselves are harmless if treated carefully. We present a simple calculus for dynamic exception (catch/throw) mechanism in type-theoretic setting, and show that the calculus enjoys a number of desirable properties such as subject reduction and strong normalization.

Keywords: Control operators, Exception, Type system, CPS-transform, Strong Normalization.

1 Introduction

Control operators and data types are the two most important areas in the study of programming languages. Type theory is no double the important foundation for data types, but it can also serve as a foundational theory for control operators. Since Griffin [2] observed that the Curry-Howard isomorphism can be extended to connect the type system for the call/cc operator and classical logic, there have been much research on connecting type systems for sequential control operators and classical logic [8, 3, 9, 12, 5]. Parigot's $\lambda\mu$-calculus [10] is one of the most studied logic in this area, which does correspond to the call/cc-like control operator.

However, studies so far done are mostly on the static control operators. Many existing control operators such as catch/throw in Common Lisp, exception in ML, and try-catch in JAVA are *dynamic*, and also shift/reset [1], recently invented control operators, are dynamic. There remains much room for the study on dynamic control operators.

There are at least two reasons for this situation:

(Reason 1) It is believed that the dynamic feature does not have any counterpart in ordinary logical systems, just in the same way as the dynamic binding of variables cannot have counterpart in standard logical systems.

(Reason 2) Programs with the dynamic control operators behave very badly from the viewpoint of type theory. For instance, typable programs with ML-style exception mechanism may cause "uncaught-exception" errors at run-time, and also they may not terminate. We shall show concrete examples of these behaviors later.

We consider the first reason is groundless, since CPS-transform, which is the standard device to treat control operators and does have logical meaning such as Kolomogorov embedding, can naturally encode dynamic control features.

We cannot escape from the second reason if our purpose is to formulate ML-style exception as it is. However, we usually do not want to write programs which cause "uncaught-exception" errors or loop forever, and such programs should be ruled out by type system. In other words, our question is whether it is possible to construct a typed calculus for dynamic control operators for which type soundness and strong normalization are guaranteed.[1]

[1] Of course such a calculus is meaningless if the type system is too restrictive to allow many interesting programs. This is an important issue which is not fully addressed in the present paper.

This paper shows that we can in fact construct such a calculus in the case of dynamic exception, implying the dynamic exception itself is harmless if treated carefully. We present a simple calculus for dynamic exception mechanism in type-theoretic setting, and show that the calculus enjoys a number of desirable properties such as subject reduction and strong normalization.

Our type system is a standard one except that a function type $A \to B$ is annotated with a finite set of types Δ. This Δ is used to trace types of possibly thrown values while applying some value to a function of this type. For instance, if f is a function of type $A \to B$ with annotation $\{C\}$, then the evaluation of fa may raise exception with a thrown value of type C. By making use of this annotation, we can always recover the "exceptional" types, and thus retain the subject reduction property. We can also give type-preserving CPS-transform which maps our calculus to the simply typed lambda calculus with co-products. Again, the annotation in function types plays an important role for the CPS-transform to work. To show strong normalization, we introduce a slight extension of Plotkin's colon translation, a novel technique for optimizing CPS-transform.

The idea of annotating function types with some other types is not new; Thielecke [13] proposed a similar type system for dynamic control. In fact, our type system can be viewed as a modified version of his work. However, his type system does not enjoy the subject reduction property, nor he gave any further analysis for theoretical properties such as strong normalization of the resulting system. Also, the CPS-transform for dynamic control operators given in this paper is close to those given in the literature: the SML/NJ compiler already employs two continuation parameters, Kim-Yi-Danvy proposed selective CPS-transform for dynamic control [6], and the above mentioned Thielecke's work. The emphasis of the present paper is that, we can analyze dynamic control operators in much the same way as the static counterpart.

This paper is organized as follows: Section 2 introduces static/dynamic control operators including catch/throw in Common Lisp and exception in Standard ML. In Section 3 we present a typed calculus for dynamic exception, and in Section 4 we prove subject reduction property, normal form property, Church-Rosser property and strong normalization property. Section 5 revisits dynamic control operators in real languages and briefly discuss connection with our calculus. Finally, Section 6 gives concluding remarks.

2 Static vs Dynamic Operators

This section introduces our terminology of static/dynamic control operators and state their difference.

Since control operators are functional analogue of the **go-to** statement in procedural languages, there should be two control points; the taking-off point (corresponding to **go**) and the landing point (corresponding to **label**). In some language, the binding of **go**'s and **label**'s is determined lexically (when a program is written), and in another language, it is determined in a dynamic way. The latter case splits into two cases; the binding is determined at the time **label** is evaluated, or at the time **go** is evaluated. So there are (at least) three cases of binding. We distinguish these three cases, and call the first one *static* (sometimes called *lexical*) and the third one *dynamic* in this paper. We do not treat the second one in this paper.

For instance, the first-class continuation (**call/cc**) is static; in a program **call/cc**$(\lambda f. \ldots f \ldots)$, the **label** is **call/cc**$\lambda f.$, and the **go** is free occurrences of the variable f. Since the binding is represented by the variable-binding mechanism, it ought to be lexical. The μ-construct in Parigot's $\lambda\mu$-calculus is also static. Another example of static operators is the static catch/throw mechanism studied by Nakano [9], Sato [12], and the author [5].

On the other hand, the real catch/throw mechanism in Common Lisp is dynamic; when a `throw`-expression is evaluated, it is caught by the closest `catch` with the same tag. The exception mechanism in ML and the try-catch mechanism in JAVA are also dynamic.

The difference of static/dynamic is illustrated by the following example using Lisp-style catch/throw where α is a tag for these control operators:

(Static Catch/Throw Operators)

$$\begin{aligned}
&\ \texttt{catch}_\alpha((\lambda f.1+\texttt{catch}_\alpha(f0))(\lambda y.\texttt{throw}_\alpha y))\\
&\to \texttt{catch}_\alpha(1+\texttt{catch}_\beta((\lambda y.\texttt{throw}_\alpha y)0))\\
&\to \texttt{catch}_\alpha(1+\texttt{catch}_\beta(\texttt{throw}_\alpha 0))\\
&\to \texttt{catch}_\alpha(\texttt{throw}_\alpha 0)\\
&\to 0
\end{aligned}$$

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 Catch/Throw Operators)

$$\begin{aligned}
&\ \texttt{catch}_\alpha((\lambda f.1+\texttt{catch}_\alpha(f0))(\lambda y.\texttt{throw}_\alpha y))\\
&\to \texttt{catch}_\alpha(1+\texttt{catch}_\alpha((\lambda y.\texttt{throw}_\alpha y)0))\\
&\to \texttt{catch}_\alpha(1+\texttt{catch}_\alpha(\texttt{throw}_\alpha 0))\\
&\to \texttt{catch}_\alpha(1+0)\\
&\stackrel{*}{\to} 1
\end{aligned}$$

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

In the literature, there are many results which connect type systems for static control operators to classical logic. Besides Griffin's pioneering work for the call/cc operator, Nakano, Sato, and the author studied static variant of the catch/throw operators. Independently, de Groote [3] studied a static variant of the exception mechanism in ML. For all these theories, the type systems enjoy many nice properties such as the subject reduction and strong normalizability, and correspond to classical logic. There is also a type-preserving CPS-transform which corresponds to the embedding from classical logic to intuitionistic logic. However, if we restrict the catch/throw mechanism to be static, it can be represented by the call/cc operator, or the call-by-value variant of Parigot's $\lambda\mu$-calculus.[2]

Compared to the static control operators, dynamic ones are less studied for the reasons stated earlier. This situation is quite unsatisfactory since we see many dynamic control operators in real programming languages. The purpose of the present paper is to exploit the dynamic control operators and to give a theoretical foundation in a similar way as that for static control operators.

3 Dynamic Exception Operators

In this section, we give a simple calculus for dynamic exception in a type-theoretic setting.

3.1 The Type System

Our source language is basically simply typed lambda calculus $\lambda^{\to}$, but it is modified in two ways:

For the types, the function type $A \to B$ is changed to $A \stackrel{\Delta}{\to} B$ where Δ is a finite set of types. Intuitive meaning of this type is that, given an argument of type A, it returns a value of type B or it raises an exception of type $B \in \Delta$. Note that the types in Δ may have function types again.

For the terms, two control operators `catch` and `throw` are added to term-constructs. In this simple language, the `catch`/`throw`-expressions do not have tags unlike those in Common Lisp. Instead, we use types to select the matching `catch`; when a `throw`-expression is evaluated, it will be caught by the closest `catch` which has the same type of the thrown value. Here the "closest" one is determined in a dynamic way. For instance, $\texttt{catch}_{\texttt{int}}(0+\texttt{throw}_{\texttt{int}}1)$ evaluates to 1.

[2] To be precise, the static catch/throw calculi studied by Nakano, Sato, and the author have non-deterministic reduction rules, so they cannot be represented by the call/cc-operator. We can encode the catch/throw operators by the corresponding first-class continuation operators if we fix an evaluation strategy.

The type system of this language is shown in Fig. 1. Each judgement has the form $\Gamma \vdash M : A \; ; \; \Delta$ where Γ is a finite set of variable-type declarations, M is a term, A is a type, and Δ is a finite set of types. Intuitive meaning of this judgement is that M is a term of type A under variable declarations Γ, and the evaluation of M may raise exceptions with thrown values whose types belong to Δ. Since we do not distinguish different occurrences of catch/throw-expressions of the same type, Δ is simply a set of types, rather than a multi-set or a set of types with indexes as in $\lambda\mu$-calculus.

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

$$\frac{\Gamma \cup \{x : A\} \vdash M : B \; ; \; \Delta}{\Gamma \vdash \lambda x.M : A \overset{\Delta}{\to} B \; ; \; \Sigma} \qquad \frac{\Gamma \vdash M : A \overset{\Delta}{\to} B \; ; \; \Sigma \quad \Gamma \vdash N : A \; ; \; \Sigma}{\Gamma \vdash MN : B \; ; \; \Sigma} \; (\Delta \subseteq \Sigma)$$

$$\frac{\Gamma \vdash M : A \; ; \; \Delta}{\Gamma \vdash \mathtt{throw}_A M : B \; ; \; \Delta} \; (A \in \Delta) \qquad \frac{\Gamma \vdash M : A \; ; \; \Delta \cup \{A\}}{\Gamma \vdash \mathtt{catch}_A M : A \; ; \; \Delta}$$

Fig. 1. Type System for Dynamic Exception

Note that, in the presentation of all rules in Fig. 1, we combined the weakening rule on the Γ-part and the Δ-part. For instance, the variable introduction rule (the first rule) could have the form:

$$\frac{}{\{x : A\} \vdash x : A \; ; \; \{\}}$$

but by combining this basic rule with the weakening rule, we have the first rule in Fig. 1.

In the function-introduction rule, the annotation Δ is obtained from the judgement above the vertical line. The term $\lambda x.M$ is itself a value and during its computation no exception will be thrown. So we have $\Gamma \vdash \lambda x.M : A \overset{\Delta}{\to} B \; ; \; \{\}$, and by combining this with the weakening rule, we have the second rule.

In the function-application rule, the side-condition $\Delta \subseteq \Sigma$ is essential. During the evaluation of MN, an exception of type Δ or Σ may occur, so Σ must be a superset of Δ.

In the throw-rule, we must have $A \in \Delta$, since the evaluation of $\mathtt{throw}_A M$ may throw a value of type A. On the contrary, the term $\mathtt{catch}_A M$ catches an exception of type A, so A is added to the Δ-part of the judgement above the line.

As a simple example, we have the following type inference of the term $\lambda x.\mathtt{catch}_A((\mathtt{throw}_A x)x)$ where we write $A \to B$ for $A \overset{\{\}}{\to} B$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x : A\} \vdash x : A \; ; \; \{A\}}}{\{x : A\} \vdash \mathtt{throw}_A x : A \to A \; ; \; \{A\}} \quad \dfrac{}{\{x : A\} \vdash x : A \; ; \; \{A\}}}{\{x : A\} \vdash (\mathtt{throw}_A x)x : A \; ; \; \{A\}}}{\{x : A\} \vdash \mathtt{catch}_A((\mathtt{throw}_A x)x) : A \; ; \; \{\}}}{\{\} \vdash \lambda x.\mathtt{catch}_A((\mathtt{throw}_A x)x) : A \to A \; ; \; \{\}}$$

3.2 Reduction

We define reduction rules to give an operational semantics of the language.

Noting that we are working in a call-by-value evaluation order, values are defined as:

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

The 1-step reduction is defined as the compatible closure of the union of the following relations:

$$\begin{aligned}
(\lambda x.M)V &\to_1 M[V/x] \\
\mathtt{catch}_A V &\to_1 V \\
\mathtt{catch}_A \mathtt{throw}_A V &\to_1 V \\
\mathtt{catch}_B \mathtt{throw}_A V &\to_1 \mathtt{throw}_A V \quad \text{if } A \not\equiv B \\
(\mathtt{throw}_A V)M &\to_1 \mathtt{throw}_A V \\
V_1(\mathtt{throw}_A V_2) &\to_1 \mathtt{throw}_A V_2 \\
\mathtt{throw}_B \mathtt{throw}_A V &\to_1 \mathtt{throw}_A V
\end{aligned}$$

where $M[V/x]$ denotes the usual capture-avoiding substitution.

Note that, the catch/throw mechanism ensures that the corresponding `catch` is selected at the time the `throw`-expression is evaluated, i.e., in a dynamic way. An example of reduction sequences already appeard in the previous section. The equality between two terms are the least equivalence relation which contains the 1-step reduction.

The above reduction rules define the small step semantics of our language, and we can also consider the big step semantics as follows.

First, we define the evaluation contexts as follows:

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

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

Then the big step semantics is defined by the following three rules:

$$\begin{aligned}
E[(\lambda x.M)V] &\to_1 E[M\{x := V\}] \\
E[\mathtt{catch}_A V] &\to_1 E[V] \\
E[\mathtt{catch}_A F[\mathtt{throw}_A V] &\to_1 E[V]
\end{aligned}$$

where in the last rule, F is an evaluation context such that $F \equiv F_1[\mathtt{catch}_A F_2]$ for no F_1 and F_2.

Unlike the small step semantics, the big step semantics is deterministic so is strictly weaker than the small one. However, the two semantics define essentially the same computational system as is apparent from the reduction rules.

3.3 CPS-Transform

The CPS-transform is the standard device to study typed lambda calculi with control operators. It also connects computation in the calculi with proof-reduction in the corresponding classical logic. Here we follow this tradition and give a natural CPS-transform to our calculus.

Our CPS-transform is based on the standard call-by-value CPS-transform, but we need some extension. In the case of static control operators such as `call/cc` or static catch/throw operators, we can pass the reified continuation by substitution. On the contrary, in the case of dynamic control operators, we must explicitly pass exiting continuations (to be used when exception occurs) as an argument. Hence, the target code of our CPS-transform takes multiple arguments, the first one being the usual continuation parameter, and the rest being the exiting continuation parameters. The number of the exiting continuation parameters reflect the cardinal of the set Δ. Namely, if we have $\Gamma \vdash M : A\ ;\ \{B_1, \cdots, B_n\}$, then the CPS-transform of M takes $n+1$ arguments. The extra n arguments have type $B_i^* \to *$ where $*$ is the answer type. To make ease of manipulating n arguments, we use the isomorphism:

$$(\prod_{i=1}^{n} B_i^*) \to * \quad = \quad \prod_{i=1}^{n} (B_i^* \to *)$$

(where $\coprod$ is coproduct type and $\prod$ is product type) and packs the n arguments into a single argument of type $(\coprod_{i=1}^{n} B_i^*) \to *$.[3] In summary, the target code of our CPS-transform takes two arguments, the first being the usual continuation, and the second being the packed representation of several exiting continuation.

Since we introduced the coproduct (sum) type, the target language is the simply typed lambda calculus augmented with the coproduct. Instead of defining such a language precisely, we introduce the following type inference rules for constructor/destructors for coproduct.

$$\frac{\Gamma \vdash M : B_i}{\Gamma \vdash \texttt{inl}(B_i, M) : \coprod_{i=1}^{n} B_i} \qquad \frac{\Gamma \vdash L : \coprod_{i=1}^{n} B_i \quad \Gamma \cup \{x : B_i\} \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \texttt{case}(L, \texttt{inl}(B_i, x), M, N) : A}$$

and the following reduction rules:

$$\begin{array}{l} \texttt{case}(\texttt{inl}(B_i, V), \texttt{inl}(B_i, x), M, N) \to_1 M[V/x] \\ \texttt{case}(\texttt{inl}(B_i, V), \texttt{inl}(B_j, x), M, N) \to_1 N \ \text{ if } B_i \neq B_j \end{array}$$

Having remembered this, the type-part of the CPS-transform is given as follows where $*$ represents the answer type:

$$\begin{aligned} A^* &\equiv A^* \ \text{ if } A \text{ is atomic} \\ (A \overset{\Delta}{\to} B)^* &\equiv A^* \to [\![B]\!]_{\Delta} \\ [\![A]\!]_{\Delta} &\equiv (A^* \to *) \to ((\coprod_{i=1}^{n} B_i^*) \to *) \to * \quad \text{if } \Delta = \{B_1, \cdots, B_n\} \end{aligned}$$

The term-part is given as follows:

$$\begin{aligned} x^* &\equiv kx \\ (\lambda x.M)^* &\equiv \lambda x k' p'.[\![M]\!] k' p' \end{aligned}$$

$$\begin{aligned} [\![V]\!]_{\Delta} &\equiv \lambda kp.kV^* \\ [\![MN]\!]_{\Delta} &\equiv \lambda kp.[\![M]\!]_{\Delta}(\lambda m.[\![N]\!]_{\Delta}(\lambda n.mnkp)p)p \\ [\![\texttt{catch}_A M]\!]_{\Delta} &\equiv \lambda kp.[\![M]\!]_{\Delta \cup \{A\}}(\lambda x.kx)(\lambda u.\ \texttt{case}(u, \texttt{inl}(A, x), kx, pu)) \\ [\![\texttt{throw}_A M]\!]_{\Delta} &\equiv \lambda kp.[\![M]\!](\lambda m.p(\texttt{inl}(A, m)))p \end{aligned}$$

The above CPS-transform is sound with respect to typing and reduction as shown by the following theorem. For a declaration $\Gamma = \{x_1 : A_1, \cdots, x_n : A_n\}$ we define $\Gamma^* = \{x_1 : A_1^*, \cdots, x_n : A_n^*\}$.

Theorem 1 (Soundness of CPS-transform). *1. If we have $\Gamma \vdash M : A\ ;\ \Delta$, then we can deduce $\Gamma^* \vdash [\![M]\!]_{\Delta} : [\![A]\!]_{\Delta}$ in the simply typed lambda calculus with coproduct.*

2. If we have $\Gamma \vdash M : A\ ;\ \Delta$ and $M \to N$, then $[\![M]\!]_{\Delta} = [\![N]\!]_{\Delta}$ in the simply typed lambda calculus with coproduct.

These are easily proved by structural induction. Since the target language is a standard logical system (i.e. it does not have dynamic features) we can treat our calculus for the dynamic exception through the CPS-transform.

Remark As is shown in our CPS-transform, the Δ-part of the judgment $\Gamma \vdash M : A\ ;\ \Delta$ is treated as disjunction. Consequently, the annotation Δ for function type $A \overset{\Delta}{\to} B$ is also considered as disjunction. Namely, $A \overset{\{C_1, \cdots, C_n\}}{\to} B$ is similar to $A \to (B \coprod (\coprod_{i}^{n} C_i))$ in the CPS-transform. This coincides with Thielecke's analysis, but the details are for future work.

[3] The author thanks Yasuhiko Minamide for teaching him the CPS-transform of SML/NJ.

4 Properties

In this section we study several theoretical properties of our calculus as a type system. The properties of interest are unique decomposition, normal form, subject reduction, Church-Rosser, and strong normalization.

First we consider the first two properties. We say a term M is half-closed if we have $\{\} \vdash M : A \;;\; \Delta$ for some A and Δ, namely, if it has no free variables (but may have free exceptions). We say a term M is closed if we have $\{\} \vdash M : A \;;\; \{\}$ for some A.

Theorem 2 (Unique Decomposition). *If M is a half-closed term, then M is in either one of the following forms:*

1. *a value V,*
2. *a thrown value $\mathtt{throw}_A V$, or*
3. *$E[R]$ where E is an evaluation context and R is a redex.*

Also the decomposition into $E[R]$ is unique.

The theorem is proved by induction on M. As an easy corollary of this theorem, we have the normal form property which means that the computation never stucks.

Corollary 1 (Normal Form). *If M is a closed normal term, then M is a value.*

The next theorem is one of the most important.

Theorem 3 (Subject Reduction). *If we have $\Gamma \vdash M : A \;;\; \Delta$, and $M \to M'$, then we have $\Gamma' \vdash N : A \;;\; \Delta'$ for some $\Gamma' \subseteq \Gamma$ and $\Delta' \subseteq \Delta$.*

Proof The proof goes through by induction on the length of $M \to M'$ and the derivation of $\Gamma \vdash M : A \;;\; \Delta$. The key cases are the following ones:
($M \equiv (\lambda x.N)V$ and $M' \equiv N[V/x]$)

We have

$$\dfrac{\dfrac{\begin{array}{c}\overline{\Gamma' \cup \{x : B\} \vdash x : B \;;\; \Sigma}\\ \vdots \\ \Gamma \cup \{x : B\} \vdash N : A \;;\; \Pi\end{array}}{\Gamma \vdash \lambda x.N : B \stackrel{\Pi}{\to} A \;;\; \Delta} \quad \begin{array}{c}\vdots \\ \Gamma \vdash V : B \;;\; \Delta\end{array}}{\Gamma \vdash (\lambda x.N)V : A \;;\; \Delta}$$

where $\Pi \subseteq \Delta$. By properly defining substitution on proofs, we have:

$$\begin{array}{c}\overline{\Gamma' \cup \Gamma \vdash V : B \;;\; \Sigma \cup \Delta}\\ \vdots \\ \Gamma \vdash N[V/x] : A \;;\; \Pi \cup \Delta\end{array}$$

Since $\Pi \cup \Delta = \Delta$, we are done.
($M \equiv \mathtt{catch}_A V$ and $M' \equiv V$)

We have

$$\dfrac{\begin{array}{c}\vdots \\ \Gamma \vdash V : A \;;\; \Delta \cup \{A\}\end{array}}{\Gamma \vdash \mathtt{catch}_A V : A \;;\; \Delta}$$

Since V is either a variable or a λ-abstraction, we can deduce $\Gamma \vdash V : A \;;\; \{\}$ from the above type inference, so we are done.
($M \equiv \mathtt{catch}_A \mathtt{throw}_A V$ and $M' \equiv V$)

In this case, we have

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash V : A \ ; \ \Delta \cup \{A\}\end{array}}{\Gamma \vdash \mathtt{throw}_A V : A \ ; \ \Delta \cup \{A\}}}{\Gamma \vdash \mathtt{catch}_A \mathtt{throw}_A V : A \ ; \ \Delta}$$

Again V is either a variable or a λ-abstraction, so we have $\Gamma \vdash V : A \ ; \ \{\}$, finishing the proof. □

The essential property used in the proof is that, for a value V we can always deduce $\Gamma \vdash V : A \ ; \ \{\}$.

The subject reduction property is the very minimum requirement for a type system to be theoretically sound. It is practically useful since it implies not only type soundness (together with the normal form property) but also we do not have "uncaught exception" errors in our calculus. Namely, if we have a closed term M (recall that a term M is closed if $\{\} \vdash M : A \ ; \ \{\}$ is deduced), then we will never encounter uncaught exception during the computation of M. On the other hand, as we will later show examples, the real exception mechanism in ML (and catch/throw in Common Lisp) may cause "uncaught exception" errors, and thus in our terminology such a calculus does not enjoy the subject reduction property.

Our calculus also enjoys the Church-Rosser property.

Theorem 4 (Church-Rosser). *If M is a term, $M \to L$ and $M \to N$, then there exists a term P such that $L \to P$ and $N \to P$.*

We can prove this theorem by using standard techniques such as parallel reduction.

4.1 Strong Normalization

In order to prove the strong normalization, the previously given CPS-transform does not suffice, since $\mathtt{catch}_A \mathtt{throw}_B M \to_1 \mathtt{throw}_B M$ (when $A \neq B$), but $[\![\mathtt{catch}_A \mathtt{throw}_B M]\!]$ does not reduce to $[\![\mathtt{throw}_B M]\!]$. The standard technique to overcome this situation is to apply the colon translation originally due to Plotkin [11]. We extend the original colon translation which has two arguments (a term and a continuation) to that with three arguments where the extra argument represents the exiting continuation parameter p.

For a term M, values K and P, we shall define $M : K \ ; \ P$ as follows:

$$\begin{array}{rl}
V : K \ ; \ P \equiv & KV^* \\
V_1 V_2 : K \ ; \ P \equiv & V_1^* V_2^* K P \\
VM : K \ ; \ P \equiv & M : \lambda n. V^* m K P \ ; \ P \quad \text{if } M \text{ is not a value} \\
MN : K \ ; \ P \equiv & M : (\lambda m. N : \lambda n. mnKP \ ; \ P) \ ; \ P \quad \text{if } M \text{ is not a value} \\
\mathtt{catch}_A V : K \ ; \ P \equiv & (\lambda x. Kx) V^* \\
\mathtt{catch}_A M : K \ ; \ P \equiv & M : \lambda x. Kx \ ; \ \lambda u. \ \mathtt{case}(u, \mathtt{inl}(A, x), Kx, Pu) \quad \text{if } M \text{ is not a value} \\
\mathtt{throw}_A V : K \ ; \ P \equiv & P(\mathtt{inl}(A, V^*)) \\
\mathtt{throw}_A M : K \ ; \ P \equiv & M : \lambda m. P(\mathtt{inl}(A, m)) \ ; \ P \quad \text{if } M \text{ is not a value}
\end{array}$$

We also define $\langle M \rangle$ as $\lambda kp.(M : k \ ; \ p)$.

Theorem 5. *We have*
1. $[\![M]\!]_\Delta \to \langle M \rangle$.
2. *If $M \to_1 N$, then $\langle M \rangle \to_+ \langle N \rangle$.*

From the first clause together with the subject reduction property of the target calculus, we have $\langle M \rangle$ typechecks in the target language. From the second clause together with the strong normalization property of the target calculus, we have the strong normalization of the source calculus.

Theorem 6 (Strong Normalization). *The calculus is strongly normalizing.*

5 Exception in Real Languages

Although our intention to formulate the calculus is to capture the essential core of catch/throw or ML-like exceptions, there are differences between our calculus and the dynamic control operators in real languages, some of which being desirable and others being not. In this section we compare our calculus and those *real* operators.

5.1 Exception in Standard ML

As is mentioned in the introduction, there are problems in ML-exceptions.

Problem 1. A program which typechecks may cause "uncaught exception" errors at run-time.

A toy example of this problem is:

```
let exception alpha of int in
   let val p = (fn _ => raise (alpha 0))
       handle alpha u => fn _ => u
   in
p 0
   end
end
```

The reason of this problem is obvious as seen from the example; we consider lambda abstractions as values, so a `raise`-expression may goes outside of its handlers.

This hidden possibility of raised exception is made explicit in our calculus. In the above program, we have $\{\} \vdash p : int \to int \ ; \ \{int\}$ for the locally declared p. Then the whole term is not closed but may raise exception of type *int*.

Problem 2. A typable program may not terminate.

Again, this phenomenon is illustrated by a toy example:

```
exception alpha of (unit -> int) -> int in
let val p = fn x => (let val _ = x () in fn f => f () end
                     handle alpha u => u ) x
in
 let val q = fn _ => raise (alpha p)
     in p q
 end
end
```

If we strip out computationally irrelevant part, p becomes $\lambda x.xx$ and q is essentially p, so pq is something like $(\lambda x.xx)(\lambda x.xx)$, whose computation does not terminate. Interestingly, the above expression typechecks in Standard ML. Lillibridge made use of this construction to represent Turing-computable functions [7].

In our calculus, the above term does not typecheck, so it does not contradict to the strong normalization property. To see this, let us try to type the above expression. Since we formulated the catch/throw-style, we cannot treat the handle expression in our calculus, so we translate the above term to catch/throw-style as follows:

$$P \equiv \lambda x.(\mathtt{catch}_\alpha let(z, x(), \lambda f.f()))x$$
$$Q \equiv \lambda y.\mathtt{throw}_\alpha P$$

where $let(z, M, N)$ abbreviates $(\lambda z.N)M$, and $\alpha = (Unit \overset{\Delta}{\to} Int) \overset{\Sigma}{\to} Int$ for some suitable Δ and Σ. To type PQ in our calculus, P must have type α, so Q has type $Unit \overset{\Delta}{\to} Int$. It follows that

$\Gamma \vdash \mathtt{throw}_\alpha P : Int \ ; \ \Delta$ for some Γ. By the throw-rule, we must have $\alpha \in \Delta$, which contradicts since α contains Δ as a proper subtype. So the assumption that PQ typechecks is refuted.

This analysis indicates that the above term has some circularity in its type, and it is the source of non-termination. As expected, our calculus rules out such non-well founded types and has the desired strong normalization property.

So far, we stated the merit of our approach. However there are limitations in our formulation. First of all, we did not employ the handlers for simplicity, but this is easy improved. Secondly, we have not considered polymorphism in our calculus. As is well-known, co-existence of naive let-polymorphism and the exception breaks type soundness, and the exactly same argument is applicable to our case. This means that we must introduce value-polymorphism (as in the case of SML/NJ) or some other devices to avoid the problem. Explicit polymorphism in the style of Girard's F may be promising. Thirdly, we do not have any formal evidence that our type system is not too restrictive. Some interesting programs with ML-exceptions may not typecheck in our system, and we need further investigation for expressiveness.

5.2 Catch/Throw in Common Lisp

We tried to faithfully represent the catch/throw mechanism in Common Lisp except that we omitted tag expressions for simplicity. Introducint tag expressions to our calculus may cause difficulty since we assume only finitely many different exceptions may occur, while if we allow arbitrary tag expressions we may encounter potentially infinite number of exceptions. At present, we do not know good applications for such potential infinity, and we believe our formulation captures large portions of application programs using catch/throw.

5.3 JAVA's Try/Catch

The programming language JAVA has an involved mechanism for user-invoked exception, called *checked exception*. A typical program using the checked exception is as follows where we intentionally omitted the `finally` clause, because in the absence of side-effects, the `finally` clause cannot have much meaning.

```
try {
   M
} catch (ExcName1 e) {
   N1
} catch (ExcName2 e) {
   N2
}
```

If we identify classes in JAVA as types In ML, this expression is written as:

```
M handle ExcName1(e) => N1
    |    ExcName2(e) => N2
```

An interesting fact in JAVA's checked exceptions is that we must declare the class of possible exceptions if a method may invoke exceptions as follows:

```
static void somemethod(...) throws SomeException
```

Since a method can be considered to a function this notation is very close to our annotated function type.

As in the case of ML's exception, it is very easy to modify our calculus to include exception handlers, and our calculus can be adapted also to JAVA's try/catch mechanism.

6 Conclusion

In this paper, we have formulated a very simple typed calculus for dynamic exceptions where function types have annotations, and analyzed its theoretical properties. We gave a type system, reduction rules, and an extended CPS-transform, and proved many theoretically desirable properties including strong normalization. Although dynamic control operators have been considered harmful, our type system reveals that there are harmless dynamic control operators.

This paper mainly considered the type-theoretic analysis on dynamic control operators, and the logical counterpart under the Curry-Howard isomorphism is missing. Thielecke already pointed out in [13] that dynamic control operators correspond to intuitionistic logic rather than classical logic, but the detailed analysis for our calculus is left for future work.

Acknowledgments

The author would like to thank Hayo Thielecke and Yasuhiko Minamide for insightful comments. This work is supported in part by the Grant-in-Aid for Scientific Research by Ministry of Education, Japan, No. 13680411, and University of Tsukuba Research Project.

References

1. O. Danvy and A. Filinski, "Representing Control: a Study of the CPS Transformation", *Mathematical Structures in Computer Science* 2(4), pp. 361-391, 1992.
2. T. Griffin, "A Formula-as-Types Notion of Control", *Conf. Record of 17th ACM Sympo. on Principles of Programming Languages*, pp. 47-58, 1990.
3. P. de Groote, "A CPS-Translation of the $\lambda\mu$-Calculus", *Lecture Notes in Computer Science* **787**, pp. 85-99, 1994.
4. 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.
5. Y. Kameyama and M. Sato, "Strong Normalizability of the Non-deterministic Catch/Throw Calculi", *Theoretical Computer Science*, to appear.
6. J. Kim, K. Yi and O. Danvy, "Assessing the overhead of ML exceptions by selective CPS transformation", Proc. ACM SIGPLAN Workshop on ML, pp. 103-114, 1998.
7. M. Lillibridge, "Exceptions are Strictly More Powerful than Call/CC", CMU-CS-95-178, Carnegie Mellon University, 1995.
8. C. Murthy, "Control Operators, Hierarchies, and Pseudo-Classical Type Systems: A-Translation at Work", *Proc. ACM Workshop on Continuations*, pp. 49–71, 1992.
9. H. Nakano, "A Constructive Formalization of the Catch and Throw Mechanism", *Conf. Rec. IEEE Symposium on Logic in Computer Science*, pp. 82–89, 1992.
10. 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.
11. G. Plotkin, "Call-by-name, Call-by-Value, and the λ-Calculus", *Theoretical Computer Science* 1, pp. 125–159, 1975.
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, "Comparing Control Constructs by Typing Double-Barrelled CPS Transforms", *Proc. Third ACM SIGPLAN Workshop on Continuations*, pp. 47-54, 2001.