Strong Normalizability of the Non-deterministic Catch/Throw Calculi

Yukiyoshi Kameyama[a] and Masahiko Sato[b]

[a]kameyama@kuis.kyoto-u.ac.jp
Graduate School of Informatics, Kyoto University
Kyoto, 606-8501 JAPAN

[b]masahiko@kuis.kyoto-u.ac.jp
Graduate School of Informatics, Kyoto University
Kyoto, 606-8501 JAPAN

The catch/throw mechanism in Common Lisp provides a simple control mechanism for non-local exits. We study typed calculi by Nakano and Sato which formalize the catch/throw mechanism. These calculi correspond to classical logic through the Curry-Howard isomorphism, and one of their characteristic points is that they have non-deterministic reduction rules. These calculi can represent various computational meaning of classical proofs.

This paper is mainly concerned with the strong normalizability of these calculi. Namely, we prove the strong normalizability of these calculi, which was an open problem. We first formulate a non-deterministic variant of Parigot's $\lambda\mu$-calculus, and show it is strongly normalizing. We then translate the catch/throw calculi to this variant. Since the translation preserves typing and reduction, we obtain the strong normalization of the catch/throw calculi. We also briefly consider second-order extension of the catch/throw calculi.

Keywords: catch and throw, type system, strong normalizability, classical logic

1. Introduction

The catch and throw mechanism provides a means to implement non-local exits. The following simple example written in Common Lisp [19] shows how to use the catch and throw mechanism:

```
(defun multiply (x)
    (catch 'zero (multiply2 x)))

(defun multiply2 (x)
    (if (null x) 1
       (if (= (car x) 0) (throw 'zero 0)
          (* (car x) (multiply2 (cdr x))))))
```

The first function `multiply` sets up the catch-point with the tag `zero`, and immediately calls the second function. The second one `multiply2` performs the actual computation by recursion. Given a list of integers, it calculates the multiplication of the members in the list. If 0 is found in the list, then the result must be 0 without computing any further, so it returns 0 by the `throw`-expression. The catch/throw mechanism is useful if one wants to escape from nested function calls at a time, especially in run-time errors.

Nakano [11–14] proposed calculi with inference rules which give logical interpretations of the catch/throw constructs in Lisp. His calculi differ from the actual catch/throw-constructs in Common Lisp in the following two ways.

(1) He changed the scope rule of the `catch`-construct from a dynamic one to a lexical one. In the above example, the expression `(throw 'zero 0)` is not lexically in the scope of the corresponding `catch`-expression, which indicates that the `catch`-expression has dynamic scope in Common Lisp.[1] In Nakano's calculi, tags are variables rather than constants, and the correspondence between `throw` and `catch` is represented as the ordinary variable binding mechanism, in which the scope of binders is lexical.

(2) He introduced the tag-abstraction and tag-application mechanisms which do not exist in Common Lisp.[2] The motivation of this was to recover the expressivity which was lost by changing the scope rule of the `catch`-construct.

Let us see how the above example can be written in Nakano's style:

```
(defun multiply (x)
    (catch 'zero (multiply2 x 'zero)))

(defun multiply2 (x u)
      (if (null x) 1
         (if (= (car x) 0) (throw u 0)
            (* (car x) (multiply2 (cdr x) u)))))
```

In this modified program, the `catch`-construct has lexical scope so that the scope of the tag `zero` is `(multiply2 x 'zero)` only. To throw an object from another function `multiply2`, the function is abstracted by the tag variable `u`. When using the function `multiply2` we must provide the tag `zero` as the second parameter.

Nakano also introduced a new type constructor $\lhd$ (called "otherwise") for the tag abstraction mechanism; if a is a term of type A, and u is a tag-variable of type B, then the abstraction of a by u has type $A \lhd B$.

The characteristic points in Nakano's formulation were (1) $\mathsf{L}_{c/t}$ has restriction (side-condition) in the implication-introduction rule, and it excludes terms which corresponds to classical proofs. Actually $\mathsf{L}_{c/t}$ corresponds to an intuitionistic calculus through the Curry-Howard isomorphism. (2) $\mathsf{L}_{c/t}$ allows as many reductions as possible, hence it is non-deterministic (not confluent). These two features may look strange, since classical logic is said to be essentially non-confluent, while intuitionistic logic is confluent.[3] We consider that the classical version of $\mathsf{L}_{c/t}$, which is obtained by removing the restriction, is a more natural calculus, and is suitable for extracting algorithmic meaning from classical proofs. We call $\mathsf{L}^K_{c/t}$ as the classical version of $\mathsf{L}_{c/t}$.

[1] Similarly the exception mechanism in the Standard ML has dynamic scope.
[2] The exception mechanism in the Standard ML has abstraction/application.
[3] We refer to Girard [6] and Parigot [15] for the discussion on the confluence and the classical logic.

A few years later than Nakano, the second author (Sato) proposed another formulation for the catch/throw mechanism [17]. His motivation was to eliminate the type of the tag abstraction ("otherwise") in $\mathsf{L}_{c/t}$, since it is equivalent to disjunction. By unifying the `throw`-expression and the tag-abstraction mechanism, he obtained a simpler calculus $\mathsf{NJ}_{c/t}$. He also showed that $\mathsf{L}_{c/t}$ can be interpreted in $\mathsf{NJ}_{c/t}$. $\mathsf{NJ}_{c/t}$ has essentially the same restriction in the implication-introduction rule, hence it corresponds to intuitionistic logic. He also defined $\mathsf{NK}_{c/t}$ by throwing away the restriction, and showed that it corresponds to classical logic. In summary, there are proposed four calculi for the catch/throw mechanism:

Author	Intuitionistic Logic	Classical Logic
Nakano	$\mathsf{L}_{c/t}$	$\mathsf{L}^K_{c/t}$
Sato	$\mathsf{NJ}_{c/t}$	$\mathsf{NK}_{c/t}$

In this paper, we investigate the strong normalizability (SN) of the above four calculi, in particular, $\mathsf{L}^K_{c/t}$ and $\mathsf{NK}_{c/t}$. The SN of $\mathsf{L}_{c/t}$ was proved by Nakano [14], but his proof was based on complex model-theoretic arguments. In our previous works, we proved the SN of $\mathsf{NJ}_{c/t}$ in [8], and the SN of a large fragment of $\mathsf{L}^K_{c/t}$ in [9], but the SN of the full fragments of classical calculi $\mathsf{L}^K_{c/t}$ and $\mathsf{NK}_{c/t}$ was an open problem. This paper solves this problem in an affirmative way.

We first formulate a non-deterministic variant of Parigot's $\lambda\mu$-calculus by adding several reduction rules, and prove its strong normalizability using the reducibility method. We then translate the catch/throw calculi to this variant. Since this translation preserves typing as well as reduction, we obtain a proof of the strong normalizability of all the four calculi. We finally briefly discuss second-order extension of them.

2. The Catch/Throw Calculi

2.1. Nakano's Formulation

Nakano proposed several calculi for the catch/throw mechanism. Among them, $\mathsf{L}_{c/t}$ given in [14] is the strongest one. In this paper we also study $\mathsf{L}^K_{c/t}$, an extension of $\mathsf{L}_{c/t}$. Although Nakano himself did not present $\mathsf{L}^K_{c/t}$ in published papers, the latter can be obtained from $\mathsf{L}_{c/t}$ by simply throwing away the restriction in the implication introduction rule, therefore we regard $\mathsf{L}^K_{c/t}$ as one of Nakano's calculi.

In the following, we shall define $\mathsf{L}^K_{c/t}$ and mention the difference of $\mathsf{L}^K_{c/t}$ and $\mathsf{L}_{c/t}$.

We assume that there are finitely many atomic types (we use K as a metavariable for atomic types) including $\perp$ (falsity).

Definition 2.1 (Type)

$$A, B \;::=\; \mathrm{K} \mid A \to B \mid A \wedge B \mid A \vee B \mid A \lhd B$$

In this definition, $\to$, $\wedge$, $\vee$ are the types for the function space, product, and sum. By the Curry-Howard isomorphism, we may identify them with logical connectives implication, conjunction, and disjunction. The connective $\lhd$ was introduced to give a type to tag abstraction. As usual, we abbreviate $A \to \perp$ as $\neg A$.

We assume that, for each type A, there are infinitely many individual variables x^A of type A and infinitely many tag variables u^A of type A. We use x^A, y^A, z^A for individual variables and u^A, v^A, w^A for tag variables. We regard u^A and u^B as different tag variables if $A \not\equiv B$. This implies that we may sometimes use the same variable-name for different entities (different types).

Preterms of $\mathsf{L}_{c/t}$ and $\mathsf{L}^K_{c/t}$ are defined as follows.

Definition 2.2 (Preterm)

$$
\begin{array}{rcl}
t, s, r & ::= & x^A \mid \mathtt{abort}(t) \mid \lambda x^A.t \mid \mathtt{apply}(t, s) \\
& & \mid \langle t, s\rangle \mid \mathtt{proj}_1(t) \mid \mathtt{proj}_2(t) \\
& & \mid \mathtt{inj}_1(t) \mid \mathtt{inj}_2(t) \mid \mathtt{case}(t, x^A.s, y^B.r) \\
& & \mid \mathtt{catch}(u^A, t) \mid \mathtt{throw}(u^A, t) \mid \kappa u^A.t \mid \mathtt{tapp}(t, u^A)
\end{array}
$$

Among the preterms above, the constructs `catch`, `throw`, κ, and `tapp` were introduced by Nakano to represent the catch and throw mechanism. We refer to the following table for the correspondence to similar constructs in Common Lisp and Standard ML.

$\mathsf{L}_{c/t}/\mathsf{L}^K_{c/t}$	Common Lisp	Standard ML
$\mathtt{catch}(u, t)$	`(catch 'u t)`	`t handle (u x)=>x`
$\mathtt{throw}(u, t)$	`(throw 'u t)`	`raise (u t)`

As noted in the introduction, tags in Common Lisp (exception names in Standard ML) are represented as tag-variables rather than constants. The preterm $\kappa u.t$ is the tag-abstraction mechanism like the λ-abstraction $\lambda x.t$, and the preterm $\mathtt{tapp}(t, u)$ is the tag-application mechanism[4] like the functional application $\mathtt{apply}(t, u)$.

We sometimes omit the types in variables. We also write $\mathtt{apply}(a, b)$ as ab. An individual variable is bound by the λ-construct and the `case`-construct, and a tag variable is bound by the `catch`-construct and the κ-construct. We identify two terms which are equivalent under renaming of bound individual/tag variables. $FV(t)$ and $FTV(t)$ denote the set of free individual variables and the set of free tag variables in t, respectively.

The type inference rules are given in the natural deduction style, and listed in Table 1. The inference rules are used to derive a judgment of the form $\Gamma \vdash a : A \; ; \; \Delta$ where Γ is a finite set in the form $\{x_1 : A_1, \cdots, x_n : A_n\}$, and Δ is a finite set in the form $\{u_1 : B_1, \cdots, u_n : B_n\}$. In both sets we understand each variable appears only once. Γ is a context of individual variables, and Δ is a context of tag variables.
In $\mathsf{L}_{c/t}$, the implication-introduction rule (marked (*)) has a restriction on free tag variables in b. $\mathsf{L}^K_{c/t}$ has no restriction. In the intuitionistic calculus $\mathsf{L}_{c/t}$, a preterm $\lambda x^A.b$ is well-typed only when x^A does not essentially occur in the scope of any `throw`-construct in b. One of Nakano's main results was that, this restriction neatly corresponds to intuitionistic propositional calculus through the Curry-Howard isomorphism. The actual

[4] Actually, Nakano did not use the word `tapp`. Rather, he simply wrote tu for $\mathtt{tapp}(t, u)$. In this paper, we use different function symbols for different term-construction to clarify the syntax.

Table 1
Type Inference Rules of $\mathsf{L}_{c/t}$ and $\mathsf{L}^K_{c/t}$

$$\frac{}{\Gamma \cup \{x^A : A\} \vdash x^A : A \ ; \ \Delta} \qquad \frac{\Gamma \vdash a : \bot \ ; \ \Delta}{\Gamma \vdash \mathtt{abort}(a) : A \ ; \ \Delta}$$

$$\frac{\Gamma \vdash b : B \ ; \ \Delta}{\Gamma - \{x^A : A\} \vdash \lambda x^A.b : A \to B \ ; \ \Delta}\ (*) \qquad \frac{\Gamma_1 \vdash c : A \to B \ ; \ \Delta \quad \Gamma_2 \vdash a : A \ ; \ \Delta}{\Gamma_1 \cup \Gamma_2 \vdash \mathtt{apply}(c, a) : B \ ; \ \Delta}$$

$$\frac{\Gamma_1 \vdash a : A \ ; \ \Delta \quad \Gamma_2 \vdash b : B \ ; \ \Delta}{\Gamma_1 \cup \Gamma_2 \vdash \langle a, b\rangle : A \wedge B \ ; \ \Delta} \qquad \frac{\Gamma \vdash a : A \wedge B \ ; \ \Delta}{\Gamma \vdash \mathtt{proj}_1(a) : A \ ; \ \Delta} \qquad \frac{\Gamma \vdash a : A \wedge B \ ; \ \Delta}{\Gamma \vdash \mathtt{proj}_2(a) : B \ ; \ \Delta}$$

$$\frac{\Gamma \vdash a : A \ ; \ \Delta}{\Gamma \vdash \mathtt{inj}_1(a) : A \vee B \ ; \ \Delta} \qquad \frac{\Gamma \vdash b : B \ ; \ \Delta}{\Gamma \vdash \mathtt{inj}_2(b) : A \vee B \ ; \ \Delta}$$

$$\frac{\Gamma_1 \vdash c : A \vee B \ ; \ \Delta \quad \Gamma_2 \vdash a : C \ ; \ \Delta \quad \Gamma_3 \vdash b : C \ ; \ \Delta}{\Gamma_1 \cup (\Gamma_2 - \{x : A\}) \cup (\Gamma_3 - \{y : B\}) \vdash \mathtt{case}(c, x.a, y.b) : C \ ; \ \Delta}$$

$$\frac{\Gamma \vdash a : A \ ; \ \Delta}{\Gamma \vdash \mathtt{catch}(u^A, a) : A \ ; \ \Delta - \{u^A : A\}} \qquad \frac{\Gamma \vdash a : A \ ; \ \Delta}{\Gamma \vdash \mathtt{throw}(u^A, a) : B \ ; \ \Delta \cup \{u^A : A\}}$$

$$\frac{\Gamma \vdash a : A \ ; \ \Delta}{\Gamma \vdash \kappa u^B.a : A \lhd B \ ; \ \Delta - \{u^B : B\}} \qquad \frac{\Gamma \vdash a : A \lhd B \ ; \ \Delta}{\Gamma \vdash \mathtt{tapp}(a, u^B) : A \ ; \ \Delta \cup \{u^B : B\}}$$

restriction is complex due to the existence of the case-construct. In this paper we do not give the precise definition of "essential occurrence". We refer to [11] and [14] for details.

Among the inference rules, the first ten are standard. The rules for throw and catch reflect their intended semantics, namely, $\mathtt{throw}(u^B, b)$ aborts the current context so that this term can be any type regardless of the type of b, and the type of $\mathtt{catch}(u^A, a)$ is the same as a and also the same as the type of possibly thrown terms. The term $\kappa u^B.a$ is tag-abstraction, and it is assigned a new type $A \lhd B$. Conversely, if a is of type $A \lhd B$, then applying a tag variable u^B to it generates a term of type A.

An example of the type inference is as follows (which corresponds to the double negation elimination):

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma_2 \vdash y^{\neg\neg A} \ : \ \neg\neg A \ ; \ \Delta} \quad \dfrac{\dfrac{\overline{\Gamma_1 \vdash x^A \ : \ A \ ; \ \{\}}}{\Gamma_1 \vdash \mathtt{throw}(u^A, x^A) \ : \ \bot \ ; \ \Delta}}{\{\} \vdash \lambda x^A.\mathtt{throw}(u^A, x^A) \ : \ \neg A \ ; \ \Delta}}{\Gamma_2 \vdash y^{\neg\neg A}(\lambda x^A.\mathtt{throw}(u^A, x^A)) \ : \ \bot \ ; \ \Delta}}{\Gamma_2 \vdash \mathtt{abort}(y^{\neg\neg A}(\lambda x^A.\mathtt{throw}(u^A, x^A))) \ : \ A \ ; \ \Delta}}{\Gamma_2 \vdash \mathtt{catch}(u^A, \mathtt{abort}(y^{\neg\neg A}(\lambda x^A.\mathtt{throw}(u^A, x^A)))) \ : \ A \ ; \ \{\}}}{\{\} \vdash \lambda y^{\neg\neg A}.\mathtt{catch}(u^A, \mathtt{abort}(y^{\neg\neg A}(\lambda x^A.\mathtt{throw}(u^A, x^A)))) \ : \ \neg\neg A \to A \ ; \ \{\}}$$

where $\Gamma_1 \equiv \{x^A : A\}$, $\Gamma_2 \equiv \{y^{\neg\neg A} : \neg\neg A\}$, and $\Delta \equiv \{u^A : A\}$. The above one is a type inference figure in $\mathsf{L}^K_{c/t}$, but not in $\mathsf{L}_{c/t}$. This is because, in the formation of $\lambda x^A.\mathtt{throw}(u^A, x^A)$, the abstracted variable x^A occurs free in $\mathtt{throw}(u^A, x^A)$, and this does not fit into Nakano's restriction.

Let $a, b, c, \cdots$ be metavariables for terms. If $\Gamma \vdash a : A \ ; \ \Delta$ is derived by the inference rules, we say a is a term of type A under contexts Γ and Δ.

One-step reduction rules of $\mathsf{L}_{c/t}$ and $\mathsf{L}^K_{c/t}$ are given by Table 2.

Table 2
One-Step Reduction Rules of $\mathsf{L}_{c/t}$ and $\mathsf{L}^K_{c/t}$

$C[a]$	$\rightsquigarrow_1$	$C[b]$ (if $a \rightsquigarrow_1 b$)
$(\lambda x.a)b$	$\rightsquigarrow_1$	$a[b/x]$
$\mathtt{proj}_1(\langle a, b\rangle)$	$\rightsquigarrow_1$	a
$\mathtt{proj}_2(\langle a, b\rangle)$	$\rightsquigarrow_1$	b
$\mathtt{case}(\mathtt{inj}_1(c), x.a, y.b)$	$\rightsquigarrow_1$	$a[c/x]$
$\mathtt{case}(\mathtt{inj}_2(c), x.a, y.b)$	$\rightsquigarrow_1$	$b[c/y]$
$a[\mathtt{throw}(u, b)/x]$	$\rightsquigarrow_1$	$\mathtt{throw}(u, b)$ (if $a \not\equiv x$ and $x \in FV(a)$)
$\mathtt{catch}(u, a)$	$\rightsquigarrow_1$	a (if $u \notin FTV(a)$)
$\mathtt{catch}(u, \mathtt{throw}(u, a))$	$\rightsquigarrow_1$	a (if $u \notin FTV(a)$)
$\mathtt{tapp}(\kappa u.a, v)$	$\rightsquigarrow_1$	$a[v/u]$

In this definition, $C[\ ]$ represents a context with a hole $[\ ]$ defined as usual. Also substitution $a[b/x]$ and $a[v/u]$ are defined as usual.

As an instance, we have the following reductions:

$$\mathtt{catch}(u, (\mathtt{throw}(u, x))y) \rightsquigarrow_1 \mathtt{catch}(u, \mathtt{throw}(u, x)) \rightsquigarrow_1 x$$

$$\mathtt{tapp}((\kappa v.(\mathtt{throw}(v, a))b), u) \rightsquigarrow_1 (\mathtt{throw}(u, a))b \rightsquigarrow_1 \mathtt{throw}(u, a)$$

Instead of having a one-step reduction like $\mathtt{catch}(u, a[\mathtt{throw}(u, b)/x]) \rightsquigarrow_1 b$, the catch/throw mechanism splits into two steps as follows:

$$\mathtt{catch}(u, a[\mathtt{throw}(u, b)/x]) \rightsquigarrow_1 \mathtt{catch}(u, (\mathtt{throw}(u, b))) \rightsquigarrow_1 b$$

Since we did not restrict any evaluation strategy, the reduction in $\mathsf{L}^K_{c/t}$ is non-deterministic, moreover it is not confluent. For instance, we have the following reduction sequences where we put $t \equiv \mathtt{catch}(u^A, (\mathtt{throw}(u^A, x^A))(\mathtt{throw}(u^A, y^A)))$

$$t \rightsquigarrow_1 \mathtt{catch}(u^A, \mathtt{throw}(u^A, x^A)) \rightsquigarrow_1 x^A$$

$$t \rightsquigarrow_1 \mathtt{catch}(u^A, \mathtt{throw}(u^A, y^A)) \rightsquigarrow_1 y^A$$

We define $a \rightsquigarrow b$ (zero or more step reduction), and $a \rightsquigarrow_+ b$ (one or more step reduction) as usual.

Theorem 2.1 (Nakano)
The subject reduction property holds for $\mathsf{L}_{c/t}$ and $\mathsf{L}^K_{c/t}$.

2.2. Sato's Formulation

In [17], Sato proposed another formulation of the catch/throw mechanism. His primary motivation was to get rid of the logical connective $\triangleleft$ from $\mathsf{L}^K_{c/t}$, yet to obtain a system which is as powerful as $\mathsf{L}^K_{c/t}$. From the logical point of view, $\triangleleft$ is redundant, since it is equivalent to disjunction. Sato successfully eliminated $\triangleleft$ from the calculus by unifying the two binders of tag variables, `catch` and κ.

We shall give the definition of $\mathsf{NK}_{c/t}$ in the following. $\mathsf{NJ}_{c/t}$ is obtained from $\mathsf{NK}_{c/t}$ by restricting the $\to$-introduction rule in the same way as $\mathsf{L}_{c/t}$ from $\mathsf{L}^K_{c/t}$. Types are those of $\mathsf{L}_{c/t}$ with $\triangleleft$ deleted. Preterms are defined as follows.

Definition 2.3 (Preterm)

$$\begin{array}{rcl} t, s, r & ::= & x^A \mid \mathtt{abort}(t) \mid \lambda x^A.t \mid \mathtt{apply}(t, s) \\ & & \mid \langle t, s\rangle \mid \mathtt{proj}_1(t) \mid \mathtt{proj}_2(t) \\ & & \mid \mathtt{inj}_1(t) \mid \mathtt{inj}_2(t) \mid \mathtt{case}(t, x^A.s, y^B.r) \\ & & \mid\ !u^B.t \mid\ ?u^B.t \mid \mathtt{tapply}(t, u^B) \end{array}$$

Individual variables are bound by the λ- and the `case`-constructs, and tag variables are bound by the `?`-constructs. The `?`-construct replaces `catch` and κ in $\mathsf{L}_{c/t}$, the `!`-construct replaces `throw` in $\mathsf{L}_{c/t}$, and the `tapply`-construct replaces `tapp` in $\mathsf{L}_{c/t}$.

The type inference rules for the new constructs are given by Table 3.

Table 3
Type Inference Rules of $\mathsf{NJ}_{c/t}$ and $\mathsf{NK}_{c/t}$

$$\frac{\Gamma \vdash a : A\ ;\ \Delta}{\Gamma \vdash\ ?u^B.a : A \vee B\ ;\ \Delta - \{u^B : B\}} \qquad \frac{\Gamma \vdash a : A\ ;\ \Delta}{\Gamma \vdash\ !u^A.a : B\ ;\ \Delta \cup \{u^A : A\}}$$

$$\frac{\Gamma \vdash a : A \vee B\ ;\ \Delta}{\Gamma \vdash \mathtt{tapply}(a, u^B) : A\ ;\ \Delta \cup \{u^B : B\}}$$

The inference rule for the `!`-construct is the same as that of `throw` in $\mathsf{L}_{c/t}$. The term $?u^B.a$ may be constructed even if the type of a differs from B. The meaning of $?u^B.a$ is that, if the computation of a ends normally and returns a', then it returns $\mathtt{inj}_1(a')$, and if a term b is thrown during the computation of a, then it returns $\mathtt{inj}_2(b)$. Hence $?u^B.a$ has type $A \vee B$ if a is of type A. The `tapply`-construct may be difficult to understand, but it is an inverse operation of tag abstraction. So $\mathtt{tapply}(?u^B.a, v^B)$ reduces to $a[v^B/u^B]$.

Type inference rules of other constructs are the same as before. The calculus with the restriction in the implication introduction rule is called $\mathsf{NJ}_{c/t}$, and the one without the restriction is $\mathsf{NK}_{c/t}$. The former corresponds to intuitionistic logic and the latter to classical logic.

One-step reduction rules for the new constructs are given as follows:

$$\begin{array}{|rcl|}\hline a[\,!u.b/x] & \rightsquigarrow_1 & !u.b \quad (\textit{if } a \not\equiv x) \\ ?u.a & \rightsquigarrow_1 & \mathtt{inj}_1(a) \quad (\textit{if } \ u \notin FTV(a)) \\ ?u.!u.a & \rightsquigarrow_1 & \mathtt{inj}_2(a) \quad (\textit{if } \ u \notin FTV(a)) \\ \mathtt{tapply}(\mathtt{inj}_1(a), u) & \rightsquigarrow_1 & a \\ \mathtt{tapply}(\mathtt{inj}_2(a), u) & \rightsquigarrow_1 & !u.a \\ \mathtt{tapply}(?v.a, u) & \rightsquigarrow_1 & a[u/v] \\ \hline \end{array}$$

The last reduction may look strange, but it is useful in writing concise proofs [17], and necessary to simulate the reduction $\mathtt{tapp}(\kappa v.a, u) \rightsquigarrow_1 a[u/v]$ in $\mathsf{L}_{c/t}/\mathsf{L}^K_{c/t}$.

Theorem 2.2 (Sato)
The subject reduction property holds for $\mathsf{NJ}_{c/t}$ *and* $\mathsf{NK}_{c/t}$.

2.3. Non-determinism and Classical Logic

All the four calculi for the catch/throw mechanism have non-deterministic reduction rules, and are not confluent. We do not think that this is defect because: (1) as far as the strong normalizability is concerned, it is good to have as many reduction rules as possible. As a corollary of the strong normalizability of the strongest calculus, we obtain the strong normalizability of any subcalculus, and (2) classical logic is said to be *inherently* non-deterministic. In order to *express* all possible computations in classical proofs, our calculus should be non-deterministic. Later we can choose one answer by fixing an evaluation strategy. Murthy gave examples which show classical proofs may contain multiple computational meanings [10]. The second author showed in [18] that Murthy's example can be expressed in in the $\mathsf{NK}_{c/t}$-style calculus.

3. A Non-deterministic Variant of Parigot's $\lambda\mu$

In this section, we give a non-deterministic variant of Parigot's $\lambda\mu$ as a target of translation from the catch/throw calculi.

Parigot's $\lambda\mu$-calculus [16] is a second-order propositional calculus for classical logic. It is a natural-deduction system whose sequents have multiple consequents. The $\lambda\mu$-calculus is a quite nice formulation of classical logic, and at the same time, it is computationally interesting, since various control structures can be represented by the μ-construct whose typing is given as follows:

$$\frac{\Gamma \vdash a : A \ ; \ \Delta}{\Gamma \vdash \mu\beta.[\alpha]a : B \ ; \ (\Delta - \{\beta : B\}) \cup \{\alpha : A\}}$$

The most important reduction rule for the μ-construct (called structural reduction) is:

$(\mu\alpha.a)b \rightsquigarrow \mu\alpha.\ a\{[\alpha]c := [\alpha](cb)\}$

where $a\{[\alpha]c := [\alpha](cb)\}$ is the term obtained from a by substituting $[\alpha](cb)$ for every subterm in the form $[\alpha]c$ where this α is free in a. We refer to [16] for the definition of the $\lambda\mu$-calculus.

We can simulate a simplified version of the catch/throw mechanism in $\mathsf{L}^K_{c/t}$ by the μ-construct as follows:

$\texttt{catch}(u, a)$ *as* $\mu u.[u]a$
$\texttt{throw}(u, a)$ *as* $\mu v.[u]a$ (where v does not appear in $[u]a$)

However, the catch/throw calculi we consider are not confluent. Moreover, one term reduces to different variables x^A and y^A as we saw in the previous section. Since the $\lambda\mu$ calculus is a confluent calculus, direct simulation of the catch/throw calculi by $\lambda\mu$ is not possible.

A possible solution is to add more reductions to $\lambda\mu$, for instance, the call-by-value version of the structural reduction (the *symmetric* structural reduction). However, it is not known that a system which has both the structural reduction and the symmetric structural reduction is strongly normalizing or not[5]. Instead of naively adding reduction rules, we slightly modify the $\lambda\mu$-calculus, then add non-deterministic reductions. Namely, we classify uses of μ into three cases:

(1) $\mu u.[u]a$
(2) $\mu u.[v]a$ with $u \notin FTV([v]a)$
(3) $\mu u.[v]a$ with $u \not\equiv v$ and $u \in FTV(a)$

We need (1) and (2) to simulate the `catch`-construct and the `throw`-construct, respectively. We only need to extend the reduction rules for (2), and the reduction rules for (1) remain the same. We do not need (3) to simulate the catch/throw calculi, so such a term construction will be excluded.

Another modification to the $\lambda\mu$-calculus is that we no longer have distinction of individual variables and tag variables. The named term $[u]a$ will be represented by ordinary application ua. By this modification, we can directly λ-abstract over variables which correspond to names such as $[\alpha]$. This is the key to simulate the tag-abstraction/tag-application mechanism in $\mathsf{L}^K_{c/t}$. This representation is essentially due to de Groote [3], who formalized the exception mechanism for ML. Fujita [4] recently studied a similar calculus for the exception mechanism.

As notational convenience, we write $\overline{\mu}u.a$ for the term $\mu u.[u]a$, and $\texttt{abort}(va)$ for the term $\mu u.[v]a$. We also extend reduction rules for the `abort`-construct to have non-deterministic features. We call the resulting system $\lambda\mu^{\mathbf{ND}}$.

3.1. A Non-deterministic Calculus $\lambda\mu^{\mathbf{ND}}$

The types of $\lambda\mu^{\mathbf{ND}}$ are defined as follows:

Definition 3.1 (Type)

$$A, B \ ::= \ X \mid \perp \mid A \to B \mid \forall X.A$$

[5]Recently, Fujita[5] indicated that such a system is shown to be strongly normalizing by translating it to Barbanera and Berardi's symmetric λ-calculus if we restrict the system to first-order. However, we need the second-order version in this paper.

Since $\lambda\mu^{\mathbf{ND}}$ is second-order, the type $\perp$ is redundant from the logical point of view. We, however, include $\perp$ as a primitive type, since we want to interpret $\perp$ differently from $\forall X.X$. The type variable X is bound by the type abstraction $\forall X$, and we identify two types which are identical modulo renaming of bound type variables. We abbreviate $A \rightarrow\perp$ as $\neg A$.

The preterms are as follows. Note that we adopt the Curry-style (implicit typing) for $\lambda\mu^{\mathbf{ND}}$ as in the $\lambda\mu$-calculus.[6] Hence we do not attach types to variables, and consider type-free reduction rules.

Definition 3.2 (Preterm)

$$t, s \quad ::= \quad x \mid \lambda x.t \mid \texttt{apply}(t, s) \mid \overline{\mu} u.t \mid \texttt{abort}(t)$$

Contrary to the original $\lambda\mu$, we have only one sort of variables. A variable x may be used for an ordinary variable and also for a name (a tag-variable in our sense). Also we have no distinction between ordinary terms and named terms. Variables are bound by λ and $\overline{\mu}$ constructs, and we again identify two terms which differs only in the bound variables. The preterm $\texttt{abort}(t)$ is new to $\lambda\mu^{\mathbf{ND}}$ as we explained above.

A judgment in $\lambda\mu^{\mathbf{ND}}$ is in the form $\Gamma \vdash a : A$ where Γ is a finite set in the form $\{x_1 : A_1, \ldots, x_n : A_n\}$. The type inference rules which derive judgments are shown in Table 4

Table 4
Type Inference Rules of $\lambda\mu^{\mathbf{ND}}$

$$\frac{}{\Gamma \cup \{x : A\} \vdash x : A}$$

$$\frac{\Gamma \vdash b : B}{\Gamma - \{x : A\} \vdash \lambda x.b : A \rightarrow B} \qquad \frac{\Gamma \vdash c : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash \texttt{apply}(c, a) : B}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a : \forall X.A}\ (*) \qquad \frac{\Gamma \vdash a : \forall X.A}{\Gamma \vdash a : A[B/X]}$$

$$\frac{\Gamma \vdash a : A}{\Gamma - \{u : \neg A\} \vdash \overline{\mu} u.a : A} \qquad \frac{\Gamma \vdash a : \perp}{\Gamma \vdash \texttt{abort}(a) : A}$$

In the $\forall$-introduction rule (marked (*)), X may not occur freely in Γ.

If $\Gamma \vdash a : A$ is derived using the above rules, we say a is a (typable) term of type A (sometimes written as $a : A$).

The reduction rules are derived from the $\lambda\mu$-calculus, but we added several rules for abort which makes $\lambda\mu^{\mathbf{ND}}$ non-deterministic. Since we shall use the substitution in the

[6] In [16], Parigot also defines a Church-style system.

Table 5
One-Step Reduction Rules of $\lambda\mu^{\mathbf{ND}}$

$C[a]$	$\leadsto_1$	$C[b]$	(*if* $a \leadsto_1 b$)
$(\lambda x.a)b$	$\leadsto_1$	$a[b/x]$	
$\lambda x.\mathtt{abort}(a)$	$\leadsto_1$	$\mathtt{abort}(a)$	(*if* $x \notin FV(a)$)
$(\mathtt{abort}(a))b$	$\leadsto_1$	$\mathtt{abort}(a)$	
$b(\mathtt{abort}(a))$	$\leadsto_1$	$\mathtt{abort}(a)$	
$\mathtt{abort}(\mathtt{abort}(a))$	$\leadsto_1$	$\mathtt{abort}(a)$	
$\overline{\mu}u.\mathtt{abort}(ua)$	$\leadsto_1$	$\overline{\mu}u.a$	
$\overline{\mu}u.a$	$\leadsto_1$	a	(*if* $u \notin FV(a)$)
$(\overline{\mu}u.a)b$	$\leadsto_1$	$\overline{\mu}u.a[\lambda x.u(xb)/u]b$	(*if* $x \notin FV(a) \cup FV(b) \cup \{u\}$)

form $[\lambda x.u(xb)/u]$ many times, it is abbreviated as $[b/^*u]$. When using this notation, we always assume that x is a fresh variable. We also abbreviate composition of substitutions $[b_1/^*u], \cdots [b_n/^*u]$ as $[\langle b_1, \cdots, b_n\rangle/^*u]$. We often write $\bar{b}$ for the sequence $\langle b_1, \cdots, b_n\rangle$. Hence successive application $(\cdots(ab_1)\cdots b_n)$ is abbreviated as $a\bar{b}$ and successive substitution $[b_1/^*u], \cdots [b_n/^*u]$ as $[\bar{b}/^*u]$. In the last case we assume that $b_1, \cdots, b_n$ do not contain u free. We also use simultaneous substitution $[b_1/x_1, \ldots, b_n/x_n, \bar{c_1}/^*u_1, \ldots, \bar{c_m}/^*u_m]$ where $x_1, \ldots, x_n, u_1, \ldots, u_m$ are mutually distinct and $c_1, \ldots, c_m$ do not contain $u_1, \ldots, u_m$ free.

As before, we use the notation $a \leadsto b$ and $a \leadsto_+ b$.

The following lemma can be proved easily.

Lemma 3.1
Let θ be the substitution $[b_1/x_1, \ldots, b_n/x_n, \bar{c_1}/^*u_1, \ldots, \bar{c_m}/^*u_m]$. If $a \leadsto b$, then $a\theta \leadsto b\theta$.

3.2. Strong Normalizability of $\lambda\mu^{\mathbf{ND}}$

In this subsection, we prove the strong normalizability (SN) of $\lambda\mu^{\mathbf{ND}}$. The proof is a slight modification of Parigot's original proof of the SN of $\lambda\mu$. Nevertheless, we give the proof here for completeness.

Let $\mathcal{T}$ be the set of preterms in $\lambda\mu^{\mathbf{ND}}$, and $\mathcal{SN}$ be the set of strongly normalizing preterms in $\lambda\mu^{\mathbf{ND}}$. Note that we do not restrict $\mathcal{T}$ and $\lambda\mu^{\mathbf{ND}}$ be subsets of typable terms, following [16]. $\nu(a)$ is the maximum length of reduction sequences starting from a if $a \in \mathcal{SN}$, and is undefined if $a \notin \mathcal{SN}$.

For $F \subset \mathcal{T}$, let $F^{<\omega}$ be the set of finite sequences of elements of F. Namely,

$$F^{<\omega} \;=\; \{\langle a_1, \cdots, a_n\rangle \mid n \geq 0, a_i \in F\}$$

In particular, $F^{<\omega}$ contains the empty sequence $\langle\rangle$.

Let F and G be subsets of $\mathcal{T}$, and S be a subset of $\mathcal{T}^{<\omega}$. Then we introduce the following notations:

$$F \rightarrow G \;\stackrel{\triangle}{=}\; \{a \in \mathcal{T} \mid ab \in G \text{ for any } b \in F\}$$
$$S \Rightarrow G \;\stackrel{\triangle}{=}\; \{a \in \mathcal{T} \mid a\bar{b} \in G \text{ for any } \bar{b} \in S\}$$

As a special case, we have $G \;=\; \{\langle\rangle\} \Rightarrow G$.

Definition 3.3 (Reducibility Candidate)
A reducibility candidate is a subset of $\mathcal{T}$, and is inductively defined as follows:
1. *$\mathcal{SN}$ is a reducibility candidate.*
2. *If F and G are reducibility candidates, so is $F \to G$.*
3. *If $\{F_i\}_{i\in I}$ is a family of reducibility candidates for a non-empty set I, then $\bigcap_{i\in I} F_i$ is a reducibility candidate. (Note that the index set I may be infinite.)*

The set of the reducibility candidates is denoted as $\mathcal{RC}$.

Lemma 3.2
For any $F \in \mathcal{RC}$, the following four clauses hold.
1. *$F \subset \mathcal{SN}$.*
2. *All variables are contained in F.*
3. *If $a \in \mathcal{SN}$, then $\texttt{abort}(a) \in F$.*
4. *There exists a set S such that $S \subset \mathcal{SN}^{<\omega}$ and $F = (S \Rightarrow \mathcal{SN})$.*

The clause 3 was added from Parigot's original proof. It means that $\texttt{abort}(a)$ with a strongly normalizing term a is contained in *any* reducibility candidate. The main difference of our proof and Parigot's is that, in our case, a term in the form $C[\texttt{abort}(a)]$ may reduce to $\texttt{abort}(a)$, so we should always consider $\texttt{abort}(a)$ as a reduct. However such a term is contained in any reducibility candidate if a is strongly normalizing by this lemma, and therefore we can always handle this term easily.

Proof. We prove all the four clauses simultaneously by induction on "$F \in \mathcal{RC}$".
(Case: F is $\mathcal{SN}$) Clause 4 is proved by taking $\{\langle\rangle\}$ as S, and other clauses are trivial.
(Case: F is $G \to H$) Suppose $a \in G \to H$. By induction hypothesis (abbreviated as IH), we have $x \in G$, so $ax \in H$. By IH, $H \subset \mathcal{SN}$, hence $a \in \mathcal{SN}$, which proves Clause 1.

By IH, $G \subset \mathcal{SN}$, and there exists a set $S' \subset \mathcal{SN}^{<\omega}$ such that $H = S' \Rightarrow \mathcal{SN}$. Then, by taking

$$S \equiv \{\langle a, b_1, \ldots, b_n\rangle \mid a \in G,\ \langle b_1, \ldots, b_n\rangle \in S'\}$$

we have $G \to H = S \Rightarrow \mathcal{SN}$, which proves Clause 4.

Let x be a variable, $a \in \mathcal{SN}$, $b_0 \in G$, and $\langle b_1, \ldots, b_n\rangle \in S'$. Then $xb_0b_1\cdots b_n \in \mathcal{SN}$, since all its reducts are in the form $xb'_0\cdots b'_n$ or $\texttt{abort}(d)b'_{k+1}\cdots b'_n$ where $b_i \rightsquigarrow b'_i$ for $0 \le i \le n$, $0 \le k \le n$, and $b_k \rightsquigarrow \texttt{abort}(d)$. Hence, $x \in G \to H$, proving Clause 2. We also have $\texttt{abort}(a)b_0b_1\cdots b_n \in \mathcal{SN}$, therefore $\texttt{abort}(a) \in G \to H$, proving Clause 3.
(Case: F is $\bigcap_{i\in I} G_i$) Clauses 1, 2, and 3 are easily proved from IH. Also by IH, for each $i \in I$ there is an $S_i \subset \mathcal{SN}^{<\omega}$ such that $G_i = (S_i \Rightarrow \mathcal{SN})$. Let S be $\bigcup_{i\in I} S_i$, then we have $S \subset \mathcal{SN}^{<\omega}$ and $\bigcap_{i\in I} G_i = S \Rightarrow \mathcal{SN}$, which proves Clause 4. □

From Clause 4 in the above lemma, we put $F^\perp$ as the largest such S. Namely, for any $F \in \mathcal{RC}$, we have $F = F^\perp \Rightarrow \mathcal{SN}$.

A preterm a is neutral if it is either a variable or in the form bc.

Lemma 3.3
For any $F \in \mathcal{RC}$, the following two clauses hold.
1. *For any $a \in F$, if $a \rightsquigarrow_1 a'$ then $a' \in F$.*
2. *If a is neutral, and $a' \in F$ for any a' such that $a \rightsquigarrow_1 a'$, then $a \in F$.*

Proof. This lemma is proved by induction on "$F \in \mathcal{RC}$". The key case is $F \equiv G \to H$. We shall prove Clause 2 only. Suppose a is neutral, and $a' \in G \to H$ for any a' such that $a \rightsquigarrow_1 a'$. Take an arbitrary preterm $b \in G$. We shall prove $ab \in H$ by induction on $\nu(a) + \nu(b)$ (since a and b are SN). The preterm ab reduces in one step to either one of $a'b$ (with $a \rightsquigarrow_1 a'$), ab' (with $b \rightsquigarrow_1 b'$), $\mathtt{abort}(c)$ (with $a \equiv a''[\mathtt{abort}(c)/x]$), or $\mathtt{abort}(d)$ (with $b \equiv b''[\mathtt{abort}(d)/x]$). We can easily prove that all the four terms belong to H, and by IH, we have $ab \in H$. Consequently $a \in G \to H$. □

Definition 3.4 (Interpretation of Types)
An interpretation ξ is a map from type variables to reducibility candidates.

Note that there exists an interpretation which maps all the type variables to $\mathcal{SN}$.

An interpretation is naturally extended to any types in the following way:

$$\begin{aligned} \xi(\bot) &\triangleq \mathcal{SN} \\ \xi(A \to B) &\triangleq \xi(A) \to \xi(B) \\ \xi(\forall X.A) &\triangleq \bigcap_{F \in \mathcal{RC}} (\xi[F/X])(A) \end{aligned}$$

where an interpretation $\xi[F/X]$ is defined as $(\xi[F/X])(X) = F$ and $(\xi[F/X])(Y) = \xi(Y)$ for $Y \neq X$.

Lemma 3.4
Let A, B be types, and ξ be an interpretation. Then $\xi[\xi(B)/X](A) = \xi(A[B/X])$.

This lemma can be proved by induction on the structure of A.

Lemma 3.5
Let $F \in \mathcal{RC}$, x, u be variables, a, b be terms, and $\bar{c}$ be a sequence of terms.

1. *If $a[b/x] \in F$ and b is SN, then $(\lambda x.a)b \in F$.*
2. *If $(\overline{\mu}u.\ (a[d/^*u])d)\bar{c} \in \mathcal{SN}$ and $c_1, \ldots, c_n, d \in \mathcal{SN}$, then $(\overline{\mu}u.a)d\bar{c} \in \mathcal{SN}$.*
3. *If $\bar{c} \in F^\perp$ and $x \notin FV(\bar{c})$, then $\lambda x.u(x\bar{c}) \in F \to \mathcal{SN}$.*
4. *If $a \in \mathcal{SN}$, then $\overline{\mu}u.\ a \in \mathcal{SN}$.*

Proof. 1. We can prove this clause by induction on $\nu(a[b/x]) + \nu(b)$ using Lemma 3.1 and Lemma 3.3. We must take care that the reducts of $(\lambda x.a)b$ may be of the form $\mathtt{abort}(c)$, but this case can be treated using Clause 3 of Lemma 3.2.

2. We can prove this clause by induction on $\nu((\overline{\mu}u.\ (a[d/^*u])d)\bar{c}) + \Sigma_{i=1}^n \nu(c_i) + \nu(d)$.

3. From Clause 1 above, all we have to prove is $u(b\bar{c}) \in \mathcal{SN}$ for any $b \in F$. Since $\bar{c} \in F^\perp$, we have $b\bar{c} \in \mathcal{SN}$, therefore $u(b\bar{c}) \in \mathcal{SN}$.

4. This can be proved by analyzing reduction rules. □

Theorem 3.1
Assume $\{x_1 : C_1, \cdots, x_n : C_n, u_1 : \neg D_1, \cdots, u_m : \neg D_m\} \vdash a : A$ is derived in $\lambda\mu^{\mathbf{ND}}$. Assume also that ξ is an interpretation, $b_i \in \xi(C_i)$ for $1 \leq i \leq n$, and $\bar{c_j} \in \xi(D_j)^\perp$ for $1 \leq j \leq m$, then we have

$a[b_1/x_1, \cdots, b_n/x_n, \bar{c_1}/^*u_1, \cdots, \bar{c_m}/^*u_m] \in \xi(A)$

At first look, the statement of the theorem looks ambiguous. For instance, given a proof of $x^C, y^{\neg D} \vdash a : A$, we may split the lefthand side of $\vdash$ in two ways, each of which results in the following conclusion:

- $a[b_1/x, b_2/y] \in \xi(A)$ holds for any $b_1 \in \xi(C)$ and $b_2 \in \xi(\neg D)$.
- $a[b/x, \bar{c}/^*y] \in \xi(A)$ holds for any $b \in \xi(C)$ and $\bar{c} \in \xi(D)^{\perp}$.

Actually the theorem implies that *both* hold, so no ambiguity arises. We now state the proof of the theorem.

Proof. The theorem is proved by induction on the type inference of a. Let θ be a substitution $[b_1/x_1, \ldots, b_n/x_n, \bar{c_1}/^*u_1, \ldots, \bar{c_m}/^*u_m]$. We also write $\{x_1 : C_1, \cdots, x_n : C_n, u_1 : \neg D_1, \cdots, u_m : \neg D_m\}$ as Γ.
(Case: Assumption-Rule) In this case $a \equiv x$. We have to prove $x\theta \in \xi(A)$. There are two subcases. (i) $x \equiv x_i$ for some i. Then $A \equiv C_i$ and $x\theta \equiv b_i$. By the assumption $b_i \in \xi(C_i)$, so $x\theta \in \xi(A)$. (ii) $x \equiv u_i$ for some i. Then $A \equiv \neg D_i$ and $x\theta \equiv \lambda z.u_i(z\bar{c})$. From Lemma 3.5, we have $x\theta \in \xi(D_i) \to \mathcal{SN}$, hence $x\theta \in \xi(\neg D_i)$.
(Case: $\to$-introduction) In this case $a \equiv \lambda x.\ c$. We have $A \equiv B \to C$ and c is a term of type C. By a suitable renaming, we have $a\theta \equiv \lambda x.\ c\theta$. Take any $d \in \xi(B)$. By IH, $c\theta' \in \xi(C)$ for $\theta' \equiv \theta \cup [d/x]$. Hence by Lemma 3.5, we have $(\lambda x.\ c\theta)d \in \xi(C)$, hence $\lambda x.\ c\theta \in \xi(B \to C)$.
(Case: $\to$-elimination) In this case, we have $a \equiv bc$, $b : B \to A$, and $c : B$. By IH, $b\theta \in \xi(B \to A)$ and $c\theta \in \xi(B)$. Hence $a\theta \equiv (b\theta)(c\theta) \in \xi(A)$.
(Case: $\forall$-introduction) In this case, $A \equiv \forall X.\ B$ and $a : B$ is derivable. Let $F \in \mathcal{RC}$ and $\xi' \equiv \xi[F/X]$. Since the type variable X does not occur freely in Γ, $b_i \in \xi'(C_i)$ for $1 \le i \le n$, and $\bar{c_j} \in \xi'(D_j)^{\perp}$ for $1 \le j \le m$. Hence, by IH, we have $a\theta \in \xi'(B)$. Finally $a\theta \in \xi(\forall X.\ B)$.
(Case: $\forall$-elimination) In this case, $a : \forall X.\ B$ and $A \equiv B[C/X]$. We have $a\theta \in \xi(\forall X.\ B)$ from IH, hence $a\theta \in \xi[\xi(C)/X](B)$. By Lemma 3.4, $a\theta \in \xi(B[C/X])$.
(Case: $\overline{\mu}$-introduction) In this case, $a \equiv \overline{\mu}u.b$, and $b : A$. By a suitable renaming, we have $a\theta \equiv \overline{\mu}u.\ b\theta$. We have $\xi(A) = \xi(A)^{\perp} \Rightarrow \mathcal{SN}$. Let $\bar{c} \in \xi(A)^{\perp}$. By IH, we have $b\theta' \in \xi(A)$ for $\theta' = \theta \cup [\bar{c}/^*u]$. Hence $(b\theta')\bar{c} \in \mathcal{SN}$, therefore by Clause 4 of Lemma 3.5, we have $\overline{\mu}u.\ (b\theta')\bar{c} \in \mathcal{SN}$. By Clause 2 of Lemma 3.5, $(\overline{\mu}u.\ b\theta)\bar{c} \in \mathcal{SN}$. Consequently, $\overline{\mu}u.\ b\theta \in \xi(A)$.
(Case: $\perp$-elimination) In this case, $a \equiv \mathtt{abort}(b)$. By IH, $b\theta \in \xi(\perp) = \mathcal{SN}$. By Lemma 3.2, we have $a\theta \equiv \mathtt{abort}(b\theta) \in \xi(A)$. □

By choosing $x_1, \ldots, x_n$ for $b_1, \ldots, b_n$, and $\langle\rangle$ (the empty sequence) for c_i ($1 \le i \le m$) in the theorem above, we obtain $a \in \xi(A)$ for any term a of type A and an interpretation ξ. Since there exists an interpretation ξ, and $\xi(A) \subset \mathcal{SN}$, we have the following theorem:

Corollary 3.1
$\lambda\mu^{\mathbf{ND}}$ is strongly normalizing.

4. Translation from the Catch/Throw Calculi to $\lambda\mu^{\mathbf{ND}}$

This section gives translations from the catch/throw calculi to $\lambda\mu^{\mathbf{ND}}$. In the following we give only translations from the classical catch/throw calculi $\mathsf{L}^K_{c/t}$ and $\mathsf{NK}_{c/t}$, but the translations work also for $\mathsf{L}_{c/t}$ and $\mathsf{NJ}_{c/t}$, since they are subcalculi.

4.1. Translation of Nakano's Calculus

We shall translate $\mathsf{L}^K_{c/t}$ to $\lambda\mu^{\mathbf{ND}}$. The translation is the same as the standard encoding of propositional logic in second-order logic except the catch/throw-constructs.

First, we translate types. Let $K_1, \cdots, K_n$ be atomic types (other than $\bot$) in $\mathsf{L}^K_{c/t}$, and $X_1, \cdots, X_n$ be distinct type variables in $\lambda\mu^{\mathbf{ND}}$.

$$
\begin{array}{rcl}
\overline{\mathrm{K}_i} & \equiv & X_i \ \ (1 \leq i \leq n) \\
\overline{\bot} & \equiv & \bot \\
\overline{A \to B} & \equiv & \overline{A} \to \overline{B} \\
\overline{A \wedge B} & \equiv & \forall X.((\overline{A} \to \overline{B} \to X) \to X) \\
\overline{A \vee B} & \equiv & \forall X.((\overline{A} \to X) \to (\overline{B} \to X) \to X) \\
\overline{A \lhd B} & \equiv & \neg \overline{B} \to \overline{A}
\end{array}
$$

The point here is that the type $A \lhd B$ is translated to $\neg\overline{B} \to \overline{A}$. This translation reflects our intention that the κ-abstraction is translated to the λ-abstraction.

We then translate preterms in $\mathsf{L}^K_{c/t}$ to $\lambda\mu^{\mathbf{ND}}$. We assume that, for each individual variable x^A in $\mathsf{L}^K_{c/t}$, x is a variable in $\lambda\mu^{\mathbf{ND}}$, and for each tag variable u^A in $\mathsf{L}^K_{c/t}$, u is a variable in $\lambda\mu^{\mathbf{ND}}$. We also assume that this mapping on variables are injective.

Preterms are translated as follows:

$$
\begin{array}{rcl}
\overline{x^A} & \equiv & x \\
\overline{\mathtt{abort}(a)} & \equiv & \mathtt{abort}(\overline{a}) \\
\overline{\lambda x^A.a} & \equiv & \lambda x.\overline{a} \\
\overline{ab} & \equiv & \overline{a}\overline{b} \\
\overline{\langle a, b\rangle} & \equiv & \lambda x.x\overline{a}\overline{b} \\
\overline{\mathtt{proj}_1(a)} & \equiv & \overline{a}(\lambda xy.x) \\
\overline{\mathtt{proj}_2(a)} & \equiv & \overline{a}(\lambda xy.y) \\
\overline{\mathtt{inj}_1(a)} & \equiv & \lambda xy.x\overline{a} \\
\overline{\mathtt{inj}_2(b)} & \equiv & \lambda xy.y\overline{b} \\
\overline{\mathtt{case}(c, x^A.a, y^B.b)} & \equiv & \overline{c}(\lambda x.\overline{a})(\lambda y.\overline{b}) \\
\overline{\mathtt{catch}(u^A, a)} & \equiv & \overline{\mu}u.\overline{a} \\
\overline{\mathtt{throw}(u^A, a)} & \equiv & \mathtt{abort}(u\overline{a}) \\
\overline{\kappa u^A.a} & \equiv & \lambda u.\overline{a} \\
\overline{\mathtt{tapp}(a, u^A)} & \equiv & \overline{a}u
\end{array}
$$

The translation is extended to contexts for variables in the following way. Let Γ be a context for individual variables $\{x_1^{A_1} : A_1, \ldots, x_n^{A_n} : A_n\}$ and Δ be a context for tag variables $\{u_1^{B_1} : B_1, \cdots, u_m^{B_m} : B_m\}$ in $\mathsf{L}^K_{c/t}$. Then we define:

$$\begin{array}{lcl} \overline{\Gamma} & \equiv & \{x_1 : \overline{A_1}, \ldots, x_n : \overline{A_n}\} \\ \overline{\overline{\Delta}} & \equiv & \{u_1 : \neg\overline{B_1}, \cdots, u_m : \neg\overline{B_m}\} \end{array}$$

Note that the types of tag variables $B_1, \ldots, B_m$ are negated through the translation.

The translation preserves typing and reduction as we shall see.

Lemma 4.1 (Preservation of Type Assignment)
If $\Gamma \vdash a : A$; Δ is derived in $\mathsf{L}^K_{c/t}$, then $\overline{\Gamma}, \overline{\overline{\Delta}} \vdash \overline{a} : \overline{A}$ is derived in $\lambda\mu^{\mathbf{ND}}$.

Proof. Since the translation for propositional connectives are standard, we verify the other cases only.

(catch) From IH, we have $\overline{\Gamma}, \overline{\overline{\Delta}} \vdash \overline{a} : \overline{A}$. Since $\overline{\overline{\Delta - \{u : A\}}}$ is $\overline{\overline{\Delta}} - \{u : \neg\overline{A}\}$, and $\overline{\mathtt{catch}(u^A, a)}$ is $\overline{\mu}u.\overline{a}$, we can derive $\overline{\Gamma}, \overline{\overline{\Delta - \{u^A : A\}}} \vdash \overline{\mathtt{catch}(u^A, a)} : \overline{A}$ by the $\overline{\mu}$-introduction rule.

(throw) From IH, we have $\overline{\Gamma}, \overline{\overline{\Delta}} \vdash \overline{a} : \overline{A}$. Since $\overline{\overline{\Delta \cup \{u^A : A\}}}$ is $\overline{\overline{\Delta}} \cup \{u : \neg\overline{A}\}$ and $\overline{\mathtt{throw}(u^A, a)}$ is $\mathtt{abort}(u\overline{a})$, we can derive $\overline{\Gamma}, \overline{\overline{\Delta \cup \{u^A : A\}}} \vdash \overline{\mathtt{throw}(u^A, a)} : \overline{B}$ by the $\to$-elimination rule and the $\bot$-elimination rule.

(κ) From IH, we have $\overline{\Gamma}, \overline{\overline{\Delta}} \vdash \overline{a} : \overline{A}$. Since $\overline{\overline{\Delta - \{u^B : B\}}}$ is $\overline{\overline{\Delta}} - \{u : \neg\overline{B}\}$, $\overline{\kappa u^B.a}$ is $\lambda u.\overline{a}$, and $\overline{A \lhd B}$ is $\neg\overline{B} \to \overline{A}$, we can derive $\overline{\Gamma}, \overline{\overline{\Delta - \{u^B : B\}}} \vdash \overline{\kappa u^B.a} : \overline{A \lhd B}$ by the $\to$-introduction rule.

(tapp) From IH, we have $\overline{\Gamma}, \overline{\overline{\Delta}} \vdash \overline{a} : \overline{A \lhd B}$. Since $\overline{A \lhd B}$ is $\neg\overline{B} \to \overline{A}$, $\overline{\overline{\Delta \cup \{u^B : B\}}}$ is $\overline{\overline{\Delta}} \cup \{u : \neg\overline{B}\}$, and $\overline{\mathtt{tapp}(a, u^B)}$ is $\overline{a}u$, by the $\to$-elimination rule, we can derive $\overline{\Gamma}, \overline{\overline{\Delta \cup \{u^B : B\}}} \vdash \overline{\mathtt{tapp}(a, u^B)} : \overline{A}$. □

Lemma 4.2
The translation is compatible with substitution. Namely, $\overline{a[b/x^A]} \equiv \overline{a}[\overline{b}/x]$, and $\overline{a[v^B/u^B]} \equiv \overline{a}[v/u]$.

This lemma is proved by straightforward induction on the construction of a, and is omitted.

Lemma 4.3 (Preservation of Reduction)
If a and b are typable terms and $a \rightsquigarrow_1 b$ in $\mathsf{L}^K_{c/t}$, then $\overline{a} \rightsquigarrow_+ \overline{b}$ in $\lambda\mu^{\mathbf{ND}}$.

Proof.
This lemma is proved by induction on the structure of the term a. We prove the key cases only.

1. $a[\mathtt{throw}(u^A, b)/x] \rightsquigarrow_1 \mathtt{throw}(u^A, b)$ with $a \not\equiv x$ and $x \in FV(a)$
   By Lemma 4.2, we have $\overline{a[\mathtt{throw}(u^A, b)/x]} \equiv \overline{a}[\mathtt{abort}(u\overline{b})/x]$. By induction on the term a, we have $\overline{a}[\mathtt{abort}(c)/x] \rightsquigarrow_+ \mathtt{abort}(c)$ for any c, so we have $\overline{a}[\mathtt{abort}(u\overline{b})/x] \rightsquigarrow_+ \mathtt{abort}(u\overline{b})$.

2. $\mathtt{catch}(u^A, a) \rightsquigarrow_1 a$ with $u^A \notin FTV(a)$
   Since $u \notin FV(\overline{a})$, we have $\overline{\mathtt{catch}(u^A, a)} \equiv \overline{\mu}u.\overline{a} \rightsquigarrow_1 \overline{a}$.

3. $\mathtt{catch}(u^A, \mathtt{throw}(u^A, a)) \rightsquigarrow_1 a$ with $u^A \notin FTV(a)$
   We have $\overline{\mathtt{catch}(u^A, \mathtt{throw}(u^A, a))} \equiv \overline{\mu}u.\mathtt{abort}(u\overline{a})$, and $u \notin FV(a)$.
   Then $\overline{\mathtt{catch}(u^A, \mathtt{throw}(u^A, a))} \rightsquigarrow_1 \overline{\mu}u.\overline{a} \rightsquigarrow_1 \overline{a}$.

4. $\mathtt{tapp}(\kappa u.a, v) \rightsquigarrow_1 a[v/u]$
   We have $\overline{\mathtt{tapp}(\kappa u.a, v)} \equiv (\lambda u.\overline{a})v$, and it reduces to $\overline{a}[v/u]$. By Lemma 4.2, $\overline{a}[v/u] \equiv \overline{a[v/u]}$, hence we are done. □

From the above lemmas, we have the following theorem.

Theorem 4.1
The system $\mathsf{L}^K_{c/t}$ is strongly normalizing. Hence $\mathsf{L}_{c/t}$ is strongly normalizing.

Remark. The translation from $\mathsf{L}^K_{c/t}$ to $\lambda\mu^{\mathbf{ND}}$ does not really need the second order quantifier. Namely, if we eliminate $\forall$, and add $\wedge$ and $\vee$ to $\lambda\mu^{\mathbf{ND}}$, then we can translate $\mathsf{L}^K_{c/t}$ to this modified calculus. Since we can prove the SN of this modified calculus by an elementary method as in [16], we can also prove the SN of $\mathsf{L}^K_{c/t}$ by an elementary method.
However, we need the second order quantifier $\forall$ in translating $\mathsf{NK}_{c/t}$ as we shall see in the next section. We therefore proved the SN of $\mathsf{L}^K_{c/t}$ based on the reducibility method.

4.2. Translation of Sato's Calculus

In this subsection we translate $\mathsf{NK}_{c/t}$ to $\lambda\mu^{\mathbf{ND}}$.

Before defining the translation, we try to give a naive translation from $\mathsf{NK}_{c/t}$ to $\mathsf{L}^K_{c/t}$, and explains why it fails. A natural candidate of the translation is:

$$\begin{array}{rcl}
\overline{A} & \equiv & A \quad \text{for any type } A \\
\overline{?u^B.a} & \equiv & \mathtt{catch}(u^{C\vee B}, \mathtt{inj}_1(a)) \\
\overline{!u^B.a} & \equiv & \mathtt{throw}(u^{C\vee B}, \mathtt{inj}_2(a)) \\
\overline{\mathtt{tapply}(a, u^B)} & \equiv & \mathtt{case}(\overline{a}, x^C.x, y^B.\mathtt{throw}(u^B, \mathtt{inj}_2(y)))
\end{array}$$

where the type C will be supplied from the type inference of each term. At this moment let us ignore how to obtain this C. By the above translation we can interpret all but one reduction rules in $\mathsf{NK}_{c/t}$. The only exception is the following one:

$\mathtt{tapply}(?u.a, v) \rightsquigarrow a[v/u]$

The lefthand side is interpreted as

$\overline{\mathtt{tapply}(?u.a, v)} \equiv \mathtt{case}(\mathtt{catch}(u, \mathtt{inj}_1(\overline{a})), x.x, y.\mathtt{throw}(v, \mathtt{inj}_2(y)))$

which does not necessarily reduce to $\overline{a[v/u]}$. Hence the naive translation from $\mathsf{NK}_{c/t}$ to $\mathsf{L}^K_{c/t}$ fails. Moreover, it seems very difficult to find a suitable extension of $\mathsf{L}^K_{c/t}$ which is strongly normalizing and which can reduce the above term to $\overline{a[v/u]}$.

However, the situation changes if we consider the second-order calculus where the disjunctive type is no more primitive, but is defined as $\overline{A \vee B} \equiv \forall X.((\overline{A} \to X) \to (\overline{B} \to X) \to X)$. As we shall see later, the term $\overline{\mathtt{tapply}(?u.a, v)}$ reduces to $\overline{a[v/u]}$ through this encoding.

Now let us define the translation from $\mathsf{NK}_{c/t}$ to $\lambda\mu^{\mathbf{ND}}$. The translation of types are the same as the translation from $\mathsf{L}^K_{c/t}$ to $\lambda\mu^{\mathbf{ND}}$. The translation of preterms except $?u^B.a, !u^B.a$, and $\mathtt{tapply}(a, u^B)$ are the same. The translation of the new constructs is defined as follows:

$$\begin{array}{rcl} \overline{?u^B.a} & \equiv & \overline{\mu}u.\lambda xy.\ x\overline{a} \\ \overline{!u^B.a} & \equiv & \mathtt{abort}(u(\lambda xy.\ y\overline{a})) \\ \overline{\mathtt{tapply}(a, u^B)} & \equiv & \overline{a}(\lambda x.x)(\lambda y.\mathtt{abort}(u(\lambda zw.wy))) \end{array}$$

where we assume x, y are not used in the term a. This translation may look complex, but it is the result of the second-order encoding of the above naive translation from $\mathsf{NK}_{c/t}$ to $\mathsf{L}^K_{c/t}$.

The translation is extended to a context for individual variables Γ in the same way as before.

For a context for tag variables Δ, we need to change the translation, since a tag variable of type B should be translated to a variable of type $C \vee B$ where C is the type of the body of enclosing ?-expression. In other words, we cannot determine the type C until we reach the enclosing ?-expression. To solve this problem, we introduce a mapping ψ from a set of tag variables (in $\mathsf{NK}_{c/t}$) to a set of types in $\lambda\mu^{\mathbf{ND}}$, and we make the translation of contexts for tag variables dependent on ψ. Let Δ be a context for tag variables $\{u_1^{B_1} : B_1, \cdots, u_n^{B_n} : B_n\}$, and ψ be a mapping from $\{u_1^{B_1}, \ldots, u_n^{B_n}\}$ to types in $\lambda\mu^{\mathbf{ND}}$. We define

$$\overline{\overline{\Delta}}\psi \equiv \{u_1 : \neg(\psi(u_1^{B_1}) \vee' \overline{B_1}), \ldots, u_n : \neg(\psi(u_n^{B_n}) \vee' \overline{B_n})\}$$

In this definition $\vee'$ is an abbreviation defined as $C \vee' D \equiv \forall X.\ ((C \to X) \to (D \to X) \to X)$ (this is the same as the result of the translation of $\vee$).

Lemma 4.4 (Preservation of Type Assignment)
If $\Gamma \vdash a : A\ ;\ \Delta$ is derivable in $\mathsf{NK}_{c/t}$, then, for any mapping ψ whose domain contains all the tag variables in Δ, we have $\overline{\Gamma}, \overline{\overline{\Delta}}\psi \vdash \overline{a} : \overline{A}$ is derivable in $\lambda\mu^{\mathbf{ND}}$.

Proof. This is proved by induction on the derivation of $\Gamma \vdash a : A\ ;\ \Delta$. We only verify the key cases.

($?u^B.a$) We have to derive $\overline{\Gamma}, \overline{\overline{\Delta - \{u^B : B\}}}\psi \vdash \overline{?u^B.a} : \overline{A \vee B}$ for any ψ. Fix a mapping ψ. Suppose $u^B \in \Delta$ (otherwise the proof is shorter), and $\Delta' \equiv \Delta - \{u^B : B\}$. Let ψ' be a mapping such that $\psi'(v) \equiv \psi(v)$ for $v \not\equiv u$ and $\psi'(u) \equiv \overline{A}$. From IH, we can derive $\overline{\Gamma}, \overline{\overline{\Delta}}\psi' \vdash \overline{a} : \overline{A}$. We have $\overline{\overline{\Delta}}\psi'$ is $\overline{\overline{\Delta'}}\psi \cup \{u : \neg(\overline{A} \vee' \overline{B})\}$. Also we have $\overline{?u^B.a} \equiv \overline{\mu}u.\lambda xy.x\overline{a}$, and $\overline{A \vee B} \equiv \forall X.(\overline{A} \to X) \to (\overline{B} \to X) \to X$. From these facts, we can derive the desired type inference by the $\overline{\mu}$-introduction rule.

($!u^B.a$) We have to derive $\overline{\Gamma}, \overline{\overline{\Delta \cup \{u^B : B\}}}\psi \vdash \overline{!u^B.a} : \overline{A}$ for any ψ. Fix any ψ. From IH, we can derive $\overline{\Gamma}, \overline{\overline{\Delta}}\psi \vdash \overline{a} : \overline{B}$. We have $\overline{\overline{\{u^B : B\}}}\psi$ is $\{u : \neg(\psi(u^B) \vee' \overline{B})\}$, and $\overline{!u^B.a}$ is $\mathtt{abort}(u(\lambda xy.y\overline{a}))$. We can derive the desired type inference by the $\perp$-elimination rule.

($\mathtt{tapply}(a, u^B)$) We have to derive $\overline{\Gamma}, \overline{\overline{\Delta \cup \{u^B : B\}}}\psi \vdash \overline{\mathtt{tapply}(a, u^B)} : \overline{A}$ for any ψ. From IH, we can derive $\overline{\Gamma}, \overline{\overline{\Delta}}\psi \vdash \overline{a} : \overline{A \vee B}$. We have $\overline{\overline{\{u^B : B\}}}\psi$ is $\{u : \neg(\psi(u^B) \vee' \overline{B})\}$, and $\overline{\mathtt{tapply}(a, u^B)}$ is $\overline{a}(\lambda x.x)(\lambda y.\mathtt{abort}(u(\lambda zw.wy)))$. By calculating the type of this term, we can derive the desired type inference. $\Box$

The next lemma is used in proving the preservation of reduction through the translation.

Lemma 4.5
Let a be a typable term in $\mathsf{NK}_{c/t}$. Let θ be a substitution $[\lambda x.v(xt_1t_2)/v]$ where v is the result of the translation of a tag variable v^B in $\mathsf{NK}_{c/t}$, t_1 is $\lambda x.x$, and t_2 is $\lambda y.\mathtt{abort}(u(\lambda zw.wy))$. Then we have $\overline{a}\theta \rightsquigarrow \overline{a}[u/v]$ in $\lambda\mu^{\mathbf{ND}}$.

Proof. We prove this lemma by induction on the structure of the term a. We state here the key cases only.
(Case $a \equiv !v^B.b$) We have the following reduction sequence:

$$\begin{array}{rcl}
\overline{!v^B.b}\theta & \equiv & \mathtt{abort}((\lambda x.v(xt_1t_2))(\lambda zw.w(\overline{b}\theta))) \\
& \rightsquigarrow & \mathtt{abort}(v(\mathtt{abort}(u(\lambda zw.w(\overline{b}\theta))))) \\
& \rightsquigarrow & \mathtt{abort}(u(\lambda zw.w(\overline{b}\theta))) \\
& \rightsquigarrow & \mathtt{abort}(u(\lambda zw.w(\overline{b}[u/v]))) \quad \text{(by IH)} \\
& \equiv & \overline{!v^B.b}[u/v]
\end{array}$$

(Case $a \equiv \mathtt{tapply}(b, v^B)$) We have the following reduction sequence:

$$\begin{array}{rcl}
\overline{\mathtt{tapply}(b, v^B)}\theta & \equiv & (\overline{b}\theta)(\lambda x.x)(\lambda y.\mathtt{abort}((\lambda x.v(xt_1t_2))(\lambda zw.wy))) \\
& \rightsquigarrow & (\overline{b}\theta)(\lambda x.x)(\lambda y.\mathtt{abort}(v(\mathtt{abort}(u(\lambda zw.wy))))) \\
& \rightsquigarrow & (\overline{b}\theta)(\lambda x.x)(\lambda y.\mathtt{abort}(u(\lambda zw.wy))) \\
& \rightsquigarrow & (\overline{b}[u/v])(\lambda x.x)(\lambda y.\mathtt{abort}(u(\lambda zw.wy))) \quad \text{(by IH)} \\
& \equiv & \overline{\mathtt{tapply}(b, v^B)}[u/v]
\end{array}$$

Lemma 4.6 (Preservation of Reduction)
If a and b are typable terms and $a \rightsquigarrow_1 b$ in $\lambda\mu^{\mathbf{ND}}$, then $\overline{a} \rightsquigarrow_+ \overline{b}$ in $\lambda\mu^{\mathbf{ND}}$.

Proof.
We only check the key cases (the $\mathtt{tapply}$-expression). In the following we put $t_1 \equiv \lambda x.x$, and $t_2 \equiv \lambda y.\mathtt{abort}(u(\lambda zw.wy))$.

1. $\mathtt{tapply}(\mathtt{inj}_1(a), u^B) \rightsquigarrow_1 a$
   We have $\overline{\mathtt{tapply}(\mathtt{inj}_1(a), u)} \equiv (\lambda xy.x\overline{a})t_1t_2 \rightsquigarrow_+ t_1\overline{a} \rightsquigarrow_+ \overline{a}$.

2. $\mathtt{tapply}(\mathtt{inj}_2(a), u^B) \rightsquigarrow_1 \,!\, u^B.a$

   We have $\overline{\mathtt{tapply}(\mathtt{inj}_2(a), u^B)} \equiv (\lambda xy.y\overline{a})t_1t_2 \rightsquigarrow_+ t_2\overline{a} \rightsquigarrow_+ \mathtt{abort}(u(\lambda zw.w\overline{a}))$. The last term is $\overline{!\, u^B.a}$.

3. $\mathtt{tapply}(?v.a, u) \rightsquigarrow_1 a[u/v]$

   We have:

$$
\begin{array}{rcl}
\overline{\mathtt{tapply}(?v.a, u)} & \equiv & (\overline{\mu}v.\lambda zw.z\overline{a})t_1t_2 \\
& \rightsquigarrow_+ & \overline{\mu}v.((\lambda zw.z\overline{a}[\lambda x.v(xt_1t_2)/v])t_1t_2) \\
& \rightsquigarrow_+ & \overline{\mu}v.(\overline{a}[\lambda x.v(xt_1t_2)/v]) \\
& \rightsquigarrow & \overline{\mu}v.(\overline{a}[u/v]) \quad \text{(by Lemma 4.5)} \\
& \rightsquigarrow_+ & \overline{a}[u/v] \quad (\text{since } v \notin FV(\overline{a}[u/v])) \\
& \equiv & \overline{a[u/v]}
\end{array}
$$

   Hence we have the desired property. □

From the lemmas we have the following theorem.

Theorem 4.2
The system $\mathsf{NK}_{c/t}$ (and hence $\mathsf{NJ}_{c/t}$) is strongly normalizing.

Remark. In our proof, the use of the second order quantifier $\forall$ is indispensable to give a translation of $\mathsf{NK}_{c/t}$. Since $\mathsf{NK}_{c/t}$ is a first-order system, one may think that our proof used too strong a method, and that the SN of $\mathsf{NK}_{c/t}$ could be proved by a more elementary method. At present, we do not have an answer to this question. Our trial to apply an elementary method to $\mathsf{NK}_{c/t}$ was not successful.

5. Extension of the Catch/Throw Calculi

Having given the translation to $\lambda\mu^{\mathbf{ND}}$, it is easy to introduce the second-order quantifier to the four catch/throw calculi without loss of the nice properties such as the strong normalization. Since the catch/throw calculi are formulated in the Church-style (variables are explicitly labeled with their types), we should introduce term-constructs for type abstraction/application. As usual we let $\Lambda X.a$ denote the former and aX for the latter. The typing rules are given as follows:

$$
\frac{\Gamma \vdash a : A \ ; \ \Delta}{\Gamma \vdash \Lambda X.a : \forall X.A \ ; \ \Delta}\,(*) \qquad \frac{\Gamma \vdash a : \forall X.A \ ; \ \Delta}{\Gamma \vdash aB : A[B/X] \ ; \ \Delta}
$$

In the $\forall$-introduction rule (marked (*)), X may not occur freely in Γ nor Δ.

Also the reduction rule $(\Lambda X.a)B \rightsquigarrow_1 a[B/X]$ is added. By adding these rules to $\mathsf{L}^K_{c/t}$, we obtain a second-order catch/throw calculus $\mathsf{L}^{K2}_{c/t}$. Similarly we can obtain $\mathsf{NK}^2_{c/t}$.

The calculi $\mathsf{L}^{K2}_{c/t}$ and $\mathsf{NK}^2_{c/t}$ enjoy nice properties such as the subject reduction and the strong normalization. Here we briefly mention the expressivity of these calculi. Since free structures such as the integers and the binary trees can be encoded by the second-order quantifier [6], we can define functions with the catch/throw mechanism over various data types in the extended calculi.

For instance the function `multiply` mentioned before is typed as follows:

$$
\begin{array}{rcl}
\texttt{Int} & \equiv & \forall X.\ X \to (X \to X) \to X \\
0 & \equiv & \Lambda X.\lambda x^X.\lambda y^{X\to X}.x \\
1 & \equiv & \Lambda X.\lambda x^X.\lambda y^{X\to X}.yx \\
It^{\texttt{Int}} & \equiv & \Lambda U.\lambda uft.tUuf \\
\texttt{IntList} & \equiv & \forall X.\ X \to (\texttt{Int} \to X \to X) \to X \\
It^{\texttt{IntList}} & \equiv & \Lambda W.\lambda wft.tWwf \\
\texttt{multiply} & \equiv & \lambda L^{\texttt{IntList}}.\texttt{catch}(u^{\texttt{Int}}, It^{\texttt{IntList}}(1, f, L)) \\
f & \equiv & \lambda x^{\texttt{Int}}.\lambda y^{\texttt{Int}}.It^{\texttt{Int}}(\texttt{throw}(u^{\texttt{Int}}, 0), \lambda z.\texttt{Times}(x, y), x)
\end{array}
$$

Here $\texttt{Times}(a, b)$ is the multiplication of two integers a and b and defined as usual. $It^{\texttt{Int}}$ and $It^{\texttt{IntList}}$ are iterators on the type `Int` and `IntList`. The term $It^{\texttt{Int}}(a, \lambda z.b, x)$ with $x \notin FV(b)$ is the if-then-else expression, namely it reduces to a if x is 0, and b otherwise. It is easily seen that the above function `multiply` does the same computation as one given in the introduction.

Since the above representation of free structures is not so good in computation [6], we may consider another direction of extension. Namely, we may add inductive data types. In [8], the first author already proposed to add inductive data types to $\mathsf{NJ}_{c/t}$ without loss of the SN of the calculus, and showed that higher-order functions which use the catch/throw mechanism can be represented in the extended calculus. However, we have not fully studied this direction for the classical catch/throw calculi, so it is left for future work.

6. Concluding Remarks

We have investigated the four catch/throw calculi by Nakano and Sato, in particular, the calculi which correspond to classical logic through the Curry-Howard isomorphism. We define a non-deterministic variant of Parigot's $\lambda\mu$-calculus, and proved the strong normalizability of this variant. We gave faithful translations from the catch/throw calculi to this variant, and as a corollary, we obtained the strong normalizability of the four calculi. We also discussed their extension briefly.

Recently, Fujita [4] studied λ_{exc}, a call-by-name variant of de Groote's formulation of the exception mechanism in Standard ML. His calculus is a subcalculus of the first-order version of $\lambda\mu$-calculus. Since the catch/throw mechanism and the exception mechanism are essentially the same, his motivation and ours are similar. The main differences of his calculus and our $\lambda\mu^{\mathbf{ND}}$ is that (1) his calculus is confluent, while ours are non-deterministic, so we have more computations, (2) he uses the first-order version (actually, the implicational fragment), while we use the second-order version, and (3) his calculus has two sorts of variables (reminiscent of individual variables and tag variables), while we use one sort of variables, thus we can directly abstract over *tags*.

Crolard [2] also studied a confluent calculus for the catch/throw mechanism. Since his calculus can be translated to Parigot's $\lambda\mu$-calculus, it is similar to Fujita's formulation, thus differs from the calculi studied in this paper.

Extracting algorithmic contents from classical proofs is now a quite active research area. Many researchers in this area aim at obtaining confluent calculi for classical logic. However, classical logic is said to be *inherently* non-deterministic, namely, classical proofs may contain multiple computational meanings. Therefore, if we want to represent as many computational meanings as possible, it is natural to begin with non-deterministic calculi. Our approach is to design and study non-deterministic calculi first, then to study confluent subcalculi. We believe that the catch/throw calculi presented in this paper can be good basis for this approach. Barbanera and Berardi's calculus [1] is another non-deterministic calculus for classical proofs, so their calculus could be also a good basis. Further studies on extracting computational meaning from classical proofs are left for future work.

Acknowledgement

We would like to express our heartful thanks to Hiroshi Nakano, Makoto Tatsuta, and Izumi Takeuti for helpful comments on earlier works. We also thank to Ken-etsu Fujita for pointing out references and errors, and to anonymous referees for valuable comments for improvement. The author is supported in part by Grant-in-Aid for Scientific Research from the Ministry of Education, Science and Culture of Japan, No. 09780266 and No. 10143105.

REFERENCES

1. F. Barbanera and S. Berardi, "Extracting Constructive Content from Classical Logic via Control-like Reductions", Typed Lambda Calculi and its Applications (Bezem, M., and J. F. Groote eds.) *Lecture Notes in Computer Science* **664**, (Springer, 1993) 45–59.
2. T. Crolard, "A confluent λ-calculus with a catch/throw mechanism", Journal of Functional Programming 9:6 (1999) 625–647.
3. P. de Groote, "A Simple Calculus of Exception Handling", Typed Lambda Calculi and its Applications (Dezani-Ciancaglini, M. and G. Plotkin eds.), *Lecture Notes in Computer Science* **902** (Springer, 1995) 201–215.
4. K. Fujita, "Calculus of Classical Proofs II", Transactions of Information Processing Society of Japan, 39:12 (1998) 1–13.
5. K. Fujita, personal communication, 2000.
6. J.-Y. Girard, Y. Lafont and P. Taylor, *Proofs and Types* (Cambridge University Press, 1989).
7. T. Griffin, "A Formulae-as-Types Notion of Control", Proc. 17th ACM Symposium on Principles of Programming Languages (1990) 47–58.
8. Y. Kameyama, Y., "A New Formulation of the Catch/Throw Mechanism", Second

Fuji International Workshop on Functional and Logic Programming, T. Ida, A. Ohori, and M. Takeichi eds. (World Scientific, 1997) 106–122.

9. Y. Kameyama and M. Sato, "A Classical Catch/Throw Calculus with Tag Abstractions and its Strong Normalizability", Computing Theory, Proc. the 4th Australasian Theory Symposium (X. Lin ed.), Australian Computer Science Communications, Vol. 20, Number 3, (Springer, 1998) 183–197.
10. C. Murthy, "An Evaluation Semantics for Classical Proofs", *Conf. Rec. IEEE Symposium on Logic in Computer Science*, (1991) 96–107.
11. H. Nakano, "A Constructive Formalization of the Catch and Throw Mechanism", *Conf. Rec. IEEE Symposium on Logic in Computer Science*, (1992) 82–89.
12. H. Nakano, "The Non-deterministic Catch and Throw Mechanism and Its Subject Reduction Property", (N. D. Jones, M. Hagiya, and M. Sato eds.), *Lecture Notes in Computer Science* **792** (Springer, 1994) 61–72.
13. H. Nakano, "A constructive logic behind the catch and throw mechanism", *Annals of Pure and Applied Logic*, Vol. 69 (1994) 269–301.
14. H. Nakano, "Logical Structures of the Catch and Throw Mechanism", *Ph. D Dissertation*, University of Tokyo, (1995).
15. 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** (Springer, 1992) 199–201.
16. M. Parigot, "Proofs of Strong Normalization for Second Order Classical Natural Deduction", Journal of Symbolic Logic 62:4 (1997) 1461–1479.
17. M. Sato, "Intuitionistic and Classical Natural Deduction Systems with the Catch and the Throw Rules", *Theoretical Computer Science* 175:1 (1997) 75–92.
18. M. Sato, "Classical Brouwer-Heyting-Kolmogorov Interpretation", Algorithmic Learning Theory, *Lecture Notes in Artificial Intelligence* **1316** (Springer, 1997) 176–196.
19. G. L. Steele Jr, *Common Lisp: The Language*, (Digital Press, 1984).