A Classical Catch/Throw Calculus with Tag Abstractions and its Strong Normalizability

Yukiyoshi Kameyama and Masahiko Sato

Department of Information Science, Kyoto University
{kameyama,masahiko}@kuis.kyoto-u.ac.jp

Abstract. The catch and throw constructs in Common Lisp provides a means to implement non-local exits. Nakano proposed a calculus $\mathsf{L}_{c/t}$ which has inference rules for the catch and throw constructs, and whose types correspond to the intuitionistic propositional logic. He introduced the tag-abstraction/application mechanism into $\mathsf{L}_{c/t}$, which is useful to approximately represent the dynamic behavior of tags.
This paper examines the calculus $\mathsf{LK}_{c/t}$, a classicalized version of $\mathsf{L}_{c/t}$. In $\mathsf{LK}_{c/t}$, we can write many programming examples which are not expressible in $\mathsf{L}_{c/t}$, moreover, algorithmic contents can be extracted from classical proofs in $\mathsf{LK}_{c/t}$. We also prove several interesting properties of $\mathsf{LK}_{c/t}$ including the strong normalizability. We point out that, if we naively apply the well-known reducibility method, the tag abstraction/application mechanism is problematic. By introducing a missing elimination rule, we can successfully prove the strong normalizability of $\mathsf{LK}_{c/t}$.

1 Introduction

The catch and throw mechanism in Common Lisp[15] provides a means to implement non-local exits. The following simple example shows how to use the catch and throw mechanism in Common Lisp:

```
(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 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 encountered, then it immediately returns 0 without any further computation. The catch/throw mechanism is useful if one wants to escape from a nested function call at a time.

Nakano[8] proposed an intuitionistic calculus with inference rules which give logical interpretations of the catch and throw constructs. In his calculus $\mathsf{L}_{c/t}$,

tags are variables rather than constants, and a tag appears freely in the `throw`-expression, and is bound in the `catch`-expression. His calculus ensures that no uncaught `throw` may occur in a computation which begins from a closed term (a term without free tag variables).

An immediate consequence of his representation is that, if one wants to formulate a logically sound calculus, tag variables must have lexical scope. However, tags in Common Lisp have dynamic scope. Consequently, the example above cannot be written as a well-formed term since the tag `zero` in the `throw`-expression is outside of the scope of the `catch`-expression. A solution of this problem is to abstract/apply the tag `zero`, by which the example can be rewritten as follows.

```
(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)))))
```

The function `multiply2` is abstracted by the tag variable u. When called, the function is supplied an extra argument *zero* to instantiate the tag u. The key here is that the function `multiply2` no longer has free tag variables. It is easily seen that the dynamic behavior of tags in Common Lisp can be approximately represented using the tag-abstraction/application mechanism. Although the λ-abstraction was used for abstracting both x and u in the example, the two variables differ in nature. Therefore, Nakano discriminated abstraction of tag variables (denoted as $\kappa u.t$) from that of individual variables (denoted as $\lambda x.t$).

One possible defect of $\mathsf{L}_{c/t}$ is that it has a severe restriction on the λ-introduction rule; all the λ-variables must not occur in the scope of any `throw` expression. For instance, the term `(catch u (lambda (x) (throw u x)))` is not a well-typed term in $\mathsf{L}_{c/t}$ since `x` appears in the body of the `throw`-expression. Nakano puts such a restriction to $\mathsf{L}_{c/t}$, since he wanted to make the calculus intuitionistic. However, this restriction disables one to write practical examples which uses the catch/throw mechanism[5]. Moreover, the classicalized versions of the catch/throw calculi have a possibility for extracting algorithmic contents from classical proofs [13, 14].

In this paper, we examine a calculus $\mathsf{LK}_{c/t}$, which is essentially a classicalized version of Nakano's calculus $\mathsf{L}_{c/t}$. We show that programming examples such as higher-order function with the catch/throw mechanism and the classical encoding of logical connectives can be written in $\mathsf{LK}_{c/t}$, while both are not expressible in $\mathsf{L}_{c/t}$.

We also prove several theoretically interesting properties of $\mathsf{LK}_{c/t}$, in particular, the strong normalizability "any reduction sequence is finite". This result is contrast to the real programming languages such as Common Lisp and Standard ML where tags (exception names) have dynamic scope and there are non-terminating programs. The strong normalizability of $\mathsf{L}_{c/t}$ was proved in [11]

by a quite elaborate proof. We simplified the proof in our draft[4] and applied it to the second author's stronger calculus[13], but it still needs a tricky technique, and works only for the calculi with the restriction on the λ-introduction rule. In this paper, we develop a quite natural proof of the strong normalizability of the classical version $\mathsf{LK}_{c/t}$ based on Tait-Girard's reducibility method[2]. We analyzed the failure of earlier proofs, and found that the reducibility set for the tag-abstraction/application case must be strengthened. By introducing a new language primitive, we successfully define the reducibility set which works for proving the strong normalizability.

The rest of this paper is organized as follows: We introduce the calculus $\mathsf{LK}_{c/t}$ and its extension $\mathsf{LK}^{+}_{c/t}$ in Section 2, and give programming examples in Section 3. Then we turn our attention to the strong normalizability of $\mathsf{LK}^{+}_{c/t}$. We first explain the failure of direct application of reducibility method, and then give a proof in Section 4. Finally, we give concluding remarks and comparison to other works in Section 5.

2 The calculi $\mathsf{LK}_{c/t}$ and $\mathsf{LK}^{+}_{c/t}$

This section gives the calculus $\mathsf{LK}_{c/t}$ and its extension $\mathsf{LK}^{+}_{c/t}$. Before going to the definitions, we state several remarks.

The calculus $\mathsf{LK}_{c/t}$ is essentially a classicalized version of Nakano's $\mathsf{L}_{c/t}$, so its definition is almost the same as $\mathsf{L}_{c/t}$ except that there is no side-condition on the λ-introduction rule. Since our calculus can define conjunction (product type) and disjunction (sum type), we omitted them in $\mathsf{LK}_{c/t}$. The calculus $\mathsf{LK}^{+}_{c/t}$ is an extension of $\mathsf{LK}_{c/t}$ where we introduce a new construct $\circ$. $\mathsf{LK}^{+}_{c/t}$ exists only for technical purposes in proving the strong normalizability, and all the examples in this paper can be written in $\mathsf{LK}_{c/t}$.

2.1 Type Systems of $\mathsf{LK}_{c/t}$ and $\mathsf{LK}^{+}_{c/t}$

We give the type systems of $\mathsf{LK}_{c/t}$ and $\mathsf{LK}^{+}_{c/t}$, and postpone the reduction rules to the next subsection.

First, we assume that there are finitely many atomic types $\mathtt{B}_1, \cdots, \mathtt{B}_k$, including $\bot$ (falsity).

Definition 1 (Type).

$$A ::= \mathtt{B}_1 \mid \cdots \mid \mathtt{B}_k \mid A \rightarrow A \mid A \lhd A$$

In this definition, $\rightarrow$ is the type for function space and $\lhd$ is the type for tag-abstraction. The meaning of $\lhd$ will become apparent later.

We use $A, B, C, \cdots$ for metavariables for types. If a type does not contain the type-constructor $\lhd$, namely, if a typs is constructed using only atomic types and $\rightarrow$, then it is called an *implicational type*. We assume that, for each type

A, there are infinitely many individual variables of type A. We also assume that, for each implicational type A, there are infinitely many tag variables of type A. An important restriction is that the types of tag variables must be implicational. Strictly speaking, $\mathsf{LK}_{c/t}$ is not an extension of $\mathsf{L}_{c/t}$, since Nakano's original calculus $\mathsf{L}_{c/t}$ does not have this restriction. However, we believe that a tag variable of type $A \lhd B$ is meaningless, and that this restriction is harmless. At least, all the actual examples in $\mathsf{L}_{c/t}$ can be written in $\mathsf{LK}_{c/t}$.

We use metavariables 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 $\mathsf{LK}^+_{c/t}$ are defined as follows.

Definition 2 (Preterm).

$$\begin{aligned} t ::= {} & x^A \mid \lambda x^A.t \mid \mathtt{apply}(t,t) \mid \mathtt{abort}(t) \\ & \mid \mathtt{catch}(u^A,t) \mid \mathtt{throw}(u^A,t) \mid \kappa u^A.t \mid t \bullet u^A \mid t \circ t \end{aligned}$$

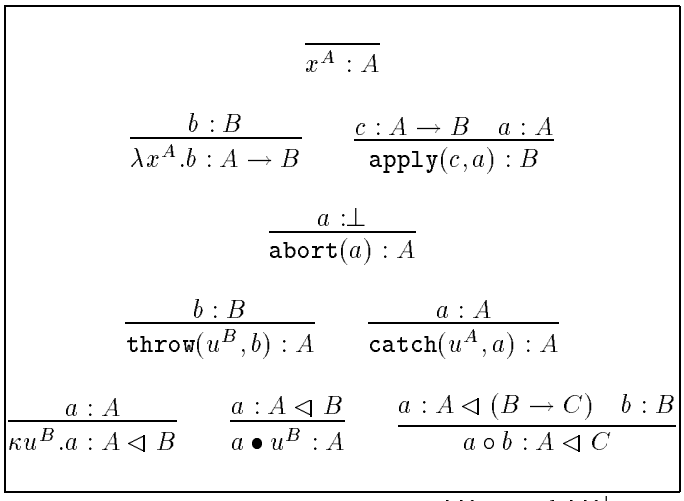
Preterms of $\mathsf{LK}_{c/t}$ are those for $\mathsf{LK}^+_{c/t}$ except the last one $t \circ t$. In the following, we sometimes omit the types in variables and preterms, for example, $\mathtt{throw}(u^B, a)$ is written as $\mathtt{throw}(u, a)$. Among the preterms above, the constructs `catch`, `throw`, κ, and $\bullet$ were introduced by Nakano to represent the catch and throw mechanism. Refer to the following table for the correspondence to similar constructs in Common Lisp and Standard ML.

$\mathsf{LK}_{c/t}/\mathsf{LK}^+_{c/t}$	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 $t \bullet u$ is the tag-application mechanism like the functional-application $\mathtt{apply}(t,t)$. The construct $\circ$ is not in Nakano's calculus $\mathsf{L}_{c/t}$ and is new to this paper. We shall explain the role of this new construct later.

An individual variable is bound by the λ-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 of $\mathsf{LK}_{c/t}$ and $\mathsf{LK}^+_{c/t}$ are given in the natural deduction style, and listed in Table 1. The inference rules are used to derive a judgement of the form $a : A$ (read "a is a term of type A").



$$\frac{}{x^A : A}$$

$$\frac{b : B}{\lambda x^A.b : A \to B} \qquad \frac{c : A \to B \quad a : A}{\mathtt{apply}(c,a) : B}$$

$$\frac{a : \bot}{\mathtt{abort}(a) : A}$$

$$\frac{b : B}{\mathtt{throw}(u^B, b) : A} \qquad \frac{a : A}{\mathtt{catch}(u^A, a) : A}$$

$$\frac{a : A}{\kappa u^B.a : A \lhd B} \qquad \frac{a : A \lhd B}{a \bullet u^B : A} \qquad \frac{a : A \lhd (B \to C) \quad b : B}{a \circ b : A \lhd C}$$

Table 1: Type Inference Rules of $\mathsf{LK}_{c/t}$ and $\mathsf{LK}^+_{c/t}$

Among the inference rules, the first four 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 a constructor, 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.

The inference rule for $\circ$ (the last one) is only for $\mathsf{LK}^+_{c/t}$. This rule is a kind of elimination rules for $A \lhd B$. Suppose a has type $A \lhd B$. If we have a tag variable u^B, we can make a term of type A, namely, $a \bullet u^B$. However, even if we have a term b of type A, we cannot make a term of type B. In this sense, the type $A \lhd B$ does not have enough destructors in $\mathsf{L}_{c/t}$ (and $\mathsf{LK}_{c/t}$), and as we shall show, this is the reason why we cannot directly prove the strong normalizability of $\mathsf{LK}_{c/t}$. In the calculus $\mathsf{LK}^+_{c/t}$, we can partly achieve such construction when B is $B_1 \to B_2$. In that case, if c is of type B_1, then the term $a \circ c$ has type $A \lhd B_2$, which is *smaller* than the type $A \lhd B$. In the following section, we shall explain how this "destructor" is used in the proof of the strong normalization.

One should note that there is no side condition in the λ-introduction rule (the second rule in Table 2). 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[1].

Let us explain the relationship between the side-condition and the intuitionistic calculus. Suppose a is a term of type A with $FV(a) = \{x^B\}$ and $FTV(a) = \{u^E\}$. Then intuitively we have $B \to (A \vee E)$. By applying the λ-formation rule to a, we obtain a term $\lambda x^B.a$ of type $B \to A$. Since $FV(\lambda x^B.a) = \{\}$ and $FTV(\lambda x^B.a) = \{u^E\}$, intuitively we have $(B \to A) \vee E$. Hence we have deduced

[1] Here we do not give the precise meaning of "essential occurrence" in $\mathsf{L}_{c/t}$. Refer to [8] and [11] for details.

$(B \to A) \vee E$ from $B \to (A \vee E)$. But this is valid only in a classical calculus, and is not valid in an intuitionistic calculus. Nakano put a restriction on this rule to obtain an intuitionistic calculus $\mathsf{L}_{c/t}$.

As an example of type inference, the following figure is a proof of the double-negation-elimination rule. Here we abbreviate $A \to \bot$ as $\neg A$.

$$\dfrac{\dfrac{\dfrac{}{y^{\neg\neg A} : \neg\neg A} \quad \dfrac{\dfrac{\dfrac{}{x^A : A}}{\mathtt{throw}(u^A, x^A) : \bot}}{\lambda x^A.\mathtt{throw}(u^A, x^A) : \neg A}}{\mathtt{apply}(y^{\neg\neg A}, \lambda x^A.\mathtt{throw}(u^A, x^A)) : \bot}}{\mathtt{abort}(\mathtt{apply}(y^{\neg\neg A}, \lambda x^A.\mathtt{throw}(u^A, x^A))) : A}}{\mathtt{catch}(u^A, \mathtt{abort}(\mathtt{apply}(y^{\neg\neg A}, \lambda x^A.\mathtt{throw}(u^A, x^A)))) : A}}{\lambda y^{\neg\neg A}.\mathtt{catch}(u^A, \mathtt{abort}(\mathtt{apply}(y^{\neg\neg A}, \lambda x^A.\mathtt{throw}(u^A, x^A)))) : \neg\neg A \to A}$$

Note that, this is a proof in $\mathsf{LK}_{c/t}$, but not a proof in $\mathsf{L}_{c/t}$, since in the application of the λ-rule (the formation of $\lambda x^A.\mathtt{throw}(u^A, x^A)$), the abstracted variable x^A occurs free in $\mathtt{throw}(u^A, x^A)$. The calculus $\mathsf{LK}_{c/t}$ has no side-condition on the λ-rule, so the figure above is a proof in $\mathsf{LK}_{c/t}$ (and $\mathsf{LK}^+_{c/t}$).

Let $a, b, c, \cdots$ be metavariables for terms. If $a : A$ is derived using these rules, we write $\Gamma \vdash a : A \ ; \Delta$ where Γ is the set of free individual variables in a, and Δ is the set of free tag variables in a. For example, for the term $a \equiv \mathtt{throw}(u^C, \mathtt{apply}(\mathtt{throw}(v^A, x^A), y^B))$, we have $\Gamma \vdash a : D \ ; \Delta$ if we put $\Gamma \equiv \{x^A, y^B\}$ and $\Delta \equiv \{u^C, v^A\}$. In the following, we shall consider typable terms by the type inference rules above, and not preterms in general.

The calculi $\mathsf{LK}_{c/t}$ and $\mathsf{LK}^+_{c/t}$ correspond to the classical propositional calculus. We assume readers are familiar with the Curry-Howard isomorphism; for instance, an implicational type in $\mathsf{LK}_{c/t}$ can be regarded as a formula in logic.

Theorem 3. *Let A be an implicational type in $\mathsf{LK}_{c/t}$ (or $\mathsf{LK}^+_{c/t}$). A is provable in the classical propositional calculus if and only if $\emptyset \vdash a : A \ ; \emptyset$ in $\mathsf{LK}_{c/t}$ for some term a.*

(Proof Sketch) It is easy to see that, $\mathsf{LK}_{c/t}$ can prove all the classically valid theorems since we already gave the proof of the law of the double-negation-elimination in $\mathsf{LK}_{c/t}$. The inverse direction can be shown by an interpretation similar to [13], but details are omitted. □

2.2 Reduction Rules of $\mathsf{LK}_{c/t}$

In order to give the reduction rules of $\mathsf{LK}_{c/t}$, we first state substitutions. We have three kinds of substitutions in this calculus: $a[b/x^B]$, $a[v^B/u^B]$, and $a[b/^* u^{B \to C}]$. The first two $a[b/x^B]$ and $a[v^B/u^B]$ are usual substitutions. The former substitutes a term for an individual variable and the latter substitutes a tag variable for a tag variable. Note that $a[b/x^B]$ is defined only when b has type B. Simultaneous substitutions $[b_1/x_1^{B_1}, \cdots, b_n/x_n^{B_n}]$ and $[v_1^{B_1}/u_1^{B_1}, \cdots, v_n^{B_n}/u_n^{B_n}]$ are

defined as usual. The third form $a[b/^* u^{B\to C}]$ is used for the reduction of the newly introduced constructor $\circ$, and it is defined in the next subsection.

The notion of 1-step reduction in $\mathsf{LK}_{c/t}$ is the same as those defined by Nakano, and is defined as the compatible closure of the reduction rules given in Table 2. Namely, for any term-context $C[\,]$, we have $C[a] \to_1 C[b]$ if and only if $a \to_1 b$.

$\mathtt{apply}(\lambda x.a, b) \to_1 a[b/x]$	
$a[\mathtt{throw}(u,b)/x] \to_1 \mathtt{throw}(u,b)$	$(\mathit{if}\ a \not\equiv x)$
$\mathtt{catch}(u,a) \to_1 a$	$(\mathit{if}\ u \notin FTV(a))$
$\mathtt{catch}(u,\mathtt{throw}(u,a)) \to_1 a$	$(\mathit{if}\ u \notin FTV(a))$
$(\kappa u.a) \bullet v \to_1 a[v/u]$	

Table 2: 1-Step Reduction Rules of $\mathsf{LK}_{c/t}$

For instance, we have the following reductions:

$$\mathtt{catch}(u, \mathtt{apply}(\mathtt{throw}(u,x), y)) \to_1 \mathtt{catch}(u, \mathtt{throw}(u,x)) \to_1 x$$

$$(\kappa v.\mathtt{apply}(\mathtt{throw}(v,a), b)) \bullet u \to_1 \mathtt{apply}(\mathtt{throw}(u,a), b) \to_1 \mathtt{throw}(u,a)$$

Instead of having a one-step reduction like $\mathtt{catch}(u, a[\mathtt{throw}(u,b)/x]) \to_1 b$, the catch/throw mechanism splits into two steps as follows:

$$\mathtt{catch}(u, a[\mathtt{throw}(u,b)/x]) \to_1 \mathtt{catch}(u, (\mathtt{throw}(u,b))) \to_1 b$$

Since we did not restrict any evaluation strategy, the reduction in $\mathsf{LK}_{c/t}$ is non-deterministic, moreover it is not Church-Rosser. For instance, the following term reduces to both x^A and y^A:

$$\mathtt{catch}(u^A, \mathtt{apply}(\mathtt{throw}(u^A, x^A), \mathtt{throw}(u^A, y^A)))$$

We do not think that this is a defect of $\mathsf{LK}_{c/t}$ because (1) as far as the strong normalizability is concerned, it is preferable to have as strong reduction rules as possible, 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. In fact, the second author showed in [14] that Murthy's example[7] can be expressed in a catch/throw calculus.

We may obtain a various confluent calculus as a subcalculus of $\mathsf{LK}_{c/t}$ by restricting reduction rules. Our results (Subject Reduction and Strong Normalization) hold for any properly formulated subcalculus of $\mathsf{LK}_{c/t}$.

2.3 Reduction Rules of $\mathsf{LK}^+_{c/t}$

Defining the notion of 1-step reduction in $\mathsf{LK}^+_{c/t}$ is relatively more difficult than in $\mathsf{LK}_{c/t}$. We first define the third form of substitution $a[b/^*u^{B\to C}]$ which was left undefined. This substitution is close to one in Parigot's $\lambda\mu$-calculus[12]. It is defined only when b has type B. Intuitively, $a[b^B/^*u^{B\to C}]$ replaces all the subterms in the form $\mathtt{throw}(u^{B\to C}, c)$ where $u^{B\to C}$ is free in a, by $\mathtt{throw}(u^C, \mathtt{apply}(c, b))$. For brevity, we use the same name u for the tag variable after the substitution even if its type is changed (note that $u^{B\to C}$ and u^C are different tag variables). The precise definition given below is more complex than this intuitive explanation because free tag variables may appear also in $c \bullet u$.

Definition 4 (Substitution for a Tag Variable). In the following, the type of u is $B \to C$ and the type of b is B.

$$
\begin{aligned}
a[b/^*u] &\triangleq a \quad (if\ u \notin FTV(a)) \\
(\lambda x.a)[b/^*u] &\triangleq \lambda x.a[b/^*u] \\
\mathtt{apply}(a,c)[b/^*u] &\triangleq \mathtt{apply}(a[b/^*u], c[b/^*u]) \\
\mathtt{catch}(v,a)[b/^*u] &\triangleq \mathtt{catch}(v, a[b/^*u]) \quad (if\ u \not\equiv v) \\
\mathtt{throw}(u,a)[b/^*u] &\triangleq \mathtt{throw}(u, \mathtt{apply}(a[b/^*u], b)) \\
\mathtt{throw}(v,a)[b/^*u] &\triangleq \mathtt{throw}(v, a[b/^*u]) \quad (if\ u \not\equiv v) \\
(\kappa v.a)[b/^*u] &\triangleq \kappa v.a[b/^*u] \quad (if\ u \not\equiv v) \\
(a \bullet u)[b/^*u] &\triangleq (a[b/^*u] \circ b) \bullet u \\
(a \bullet v)[b/^*u] &\triangleq a[b/^*u] \bullet v \quad (if\ u \not\equiv v) \\
(a \circ c)[b/^*u] &\triangleq (a[b/^*u]) \circ (c[b/^*u])
\end{aligned}
$$

Note that the cases for $\mathtt{catch}(u,a)[b/^*u]$ and $(\kappa u.a)[b/^*u]$ are included in the first clause (u is not free in the terms). As an example of this substitution, $((\kappa v.\mathtt{throw}(u,a)) \bullet u)[c/^*u]$ is $((\kappa v.\mathtt{throw}(u, \mathtt{apply}(a[c/^*u], c))) \circ c) \bullet u$. We can easily verify that, if $\Gamma_1 \vdash a : A \ ; \ \Delta_1 \cup \{u^{B\to C}\}$ and $u^{B\to C} \in FTV(a)$, and $\Gamma_2 \vdash c : B \ ; \ \Delta_2$, then $\Gamma_1 \cup \Gamma_2 \vdash a[c/^*u] : A \ ; \ \Delta_1 \cup \{u^C\} \cup \Delta_2$.

The notion of 1-step reduction in $\mathsf{LK}^+_{c/t}$ is defined by Table 2 above and Table 3 below.

$(\kappa u.a) \circ b \longrightarrow_1 \kappa u.\ a[b/^*u]$

Table 3: Added 1-Step Reduction Rule of $\mathsf{LK}^+_{c/t}$

This reduction rule reflects the intended meaning of the $\circ$-construct. Suppose $\kappa u^{B\to C}.a$ and b are of type $A \lhd (B \to C)$ and B, respectively. Then $(\kappa u^{B\to C}.a) \circ b$

is of type $A \lhd C$ and it reduces to $\kappa u^C.a[b/^*u]$ where b is applied to all the `throw`-expressions in a whose tag is u.

We use the following abbreviations:

$$\begin{array}{rcl} \mathtt{apply}(\ldots\mathtt{apply}(b,d_1)\ldots,d_n) & \text{as} & \mathtt{apply}(b,\overline{d_1,\ldots,d_n}) \\ (\ldots(b \circ d_1)\ldots) \circ d_n & \text{as} & b \circ \overline{d_1,\ldots,d_n} \end{array}$$

A successive substitution in the form $(\cdots(a[b_1/^*u])\cdots)[b_n/^*u]$ is abbreviated as $a[\overline{b_1,\cdots,b_n}/^*u]$. In the following we shall use this form of substitution only when b_i does not contain u free. We shall also use a mixed simultaneous substitution such as $[b_1/x_1,\cdots,\overline{c_1^1,\cdots,c_1^k}/^*u_1,\cdots]$ in the following.

We define $a \to b$ (zero or more step reduction), and $a \to_+ b$ (one or more step reduction) as usual. Then we have the subject reduction theorem for $\mathsf{LK}_{c/t}$ and $\mathsf{LK}^+_{c/t}$.

Theorem 5 (Subject Reduction). *In either $\mathsf{LK}_{c/t}$ or $\mathsf{LK}^+_{c/t}$, if $\Gamma \vdash a : A \;;\; \Delta$ and $a \to b$, then $\Gamma' \vdash b : A \;;\; \Delta'$ for some $\Gamma' \subset \Gamma$ and $\Delta' \subset \Delta$.*

Proof. It is an easy exercise by induction on the length of the reduction.

Here, we verify only the case $(\kappa u.a) \circ b \to_1 \kappa u.a[b/^*u]$. Suppose $(\kappa u.a) \circ b$ is a well-typed term. Then, we have $\Gamma_1 \vdash \kappa u.a : A \lhd (B \to C) \;;\; \Delta_1$ and $\Gamma_2 \vdash b : B \;;\; \Delta_2$ for some $\Gamma_1, \Gamma_2, \Delta_1, \Delta_2$. The first clause implies that $\Gamma_1 \vdash a : A \;;\; \Delta_1 \cup \{u^{B \to C}\}$. Then we have $\Gamma_1 \cup \Gamma_2 \vdash a[b/^*u] : A \;;\; \Delta_1 \cup \{u^C\} \cup \Delta_2$. It follows that $\Gamma_1 \cup \Gamma_2 \vdash \kappa u.a[b/^*u] : A \lhd C; \Delta_1 \cup \Delta_2$. □

3 Programming Examples in $\mathsf{LK}_{c/t}$

This section shows the expressiveness of $\mathsf{LK}_{c/t}$.

The first examples are Griffin's classical encoding of logical connectives such as conjunction and disjunction.

$$\begin{array}{rcl} A \wedge B & \equiv & \neg(A \to \neg B) \\ \mathtt{pair}(a^A, b^B) & \equiv & \lambda x^{A \to \neg B}.\mathtt{apply}(\mathtt{apply}(x,a),b) \\ \mathtt{car}(c^{A \wedge B}) & \equiv & \mathtt{catch}(u^A, \mathtt{abort}(\mathtt{apply}(c, \lambda x^A.\lambda y^B.\mathtt{throw}(u^A, x)))) \\ \mathtt{cdr}(c^{A \wedge B}) & \equiv & \mathtt{catch}(v^B, \mathtt{abort}(\mathtt{apply}(c, \lambda x^A.\lambda y^B.\mathtt{throw}(v^B, y)))) \\ A \vee B & \equiv & \neg A \to \neg\neg B \\ \mathtt{inl}(a^A) & \equiv & \lambda x^{\neg A}.\lambda y^{\neg B}.\mathtt{apply}(x,a) \\ \mathtt{inr}(b^B) & \equiv & \lambda x^{\neg A}.\lambda y^{\neg B}.\mathtt{apply}(y,b) \\ \mathtt{case}(a^{A \vee B}; x^A.b^C; y^B.c^C) & \equiv & \mathtt{catch}(u^C, \mathtt{abort}(\mathtt{apply}(a, \overline{d,e}))) \\ d & \equiv & \lambda x^A.\mathtt{throw}(u^C, b) \\ e & \equiv & \lambda y^B.\mathtt{throw}(u^C, c) \end{array}$$

As expected, we have $\mathtt{car}(\mathtt{pair}(a,b)) \to a$ and $\mathtt{cdr}(\mathtt{pair}(a,b)) \to b$. Similarly, we have $\mathtt{case}(\mathtt{inl}(a); x.b; y.c) \to b[a/x]$ and $\mathtt{case}(\mathtt{inr}(a); x.b; y.c) \to c[a/y]$.

The second example taken from the first author's previous work[5] uses the catch/throw mechanism in a higher-order function. The function `sqrt-sum` calculates, given a list of integers, the sum of square root of each element. If there is a negative number in the list, it immediately stops the computation and returns the number. The program is written in Common Lisp like this:

```
(defun sqrt-sum (x)
    (catch 'negative (sqrt-sum2 x)))

(defun sqrt-sum2 (x)
    (if (null x) 0
       (if (< (car x) 0) (throw 'negative (car x))
          (+ (sqrt (car x)) (sqrt-sum2 (cdr x))))))
```

This program is written in $\mathsf{LK}_{c/t}$, assuming that $\mathsf{LK}_{c/t}$ is extended to have integers, lists and so on.

$$\begin{aligned}
\texttt{sqrt-sum} &\stackrel{\triangle}{=} \lambda x.\mathtt{catch}(u, \mathtt{apply}(\texttt{sqrt-sum2}, x) \bullet u) \\
\texttt{sqrt-sum2} &\stackrel{\triangle}{=} \lambda x.\kappa u.\mathtt{Rec}(f, 0, x) \\
f &\stackrel{\triangle}{=} \lambda yzw.(\texttt{if (< } y \texttt{ 0) (throw } u\ y\texttt{) (+ (sqrt } y\texttt{) } w\texttt{))}
\end{aligned}$$

where `Rec` is the recursor on the list type which has the following reduction rules:

$$\begin{aligned}
\mathtt{Rec}(f, a, nil) &\to_1 a \\
\mathtt{Rec}(f, a, cons(b, c)) &\to_1 \mathtt{apply}(\mathtt{apply}(\mathtt{apply}(f, b), c), \mathtt{Rec}(f, a, c))
\end{aligned}$$

We need the catch/throw mechanism through the λ-abstraction. Again this example cannot be written in $\mathsf{L}_{c/t}$.

4 Strong Normalizability

In this section, we prove the strong normalizability of $\mathsf{LK}_{c/t}$ and $\mathsf{LK}^{+}_{c/t}$.

4.1 Tait-Girard's reducibility method

Tait-Girard's reducibility method[2] is a standard technique to prove the strong normalizability of typed lambda calculi. We first give an overview of the method.

1. Define the set of terms $\mathtt{Red}(A)$ for each type A. This is by induction on the type.

$$\begin{aligned}
a \in \mathtt{Red}(A) &\stackrel{\triangle}{=} a \textit{ is strongly normalizing (if } A \textit{ is atomic)}, \\
a \in \mathtt{Red}(A \to B) &\stackrel{\triangle}{=} \mathtt{apply}(a, b) \in \mathtt{Red}(B) \textit{ for any } b \in \mathtt{Red}(A).
\end{aligned}$$

   If $a \in \mathtt{Red}(A)$, the term a is called reducible.

2. Prove three conditions called (CR-1), (CR-2), and (CR-3).

   (CR-1) If $a \in \mathtt{Red}(A)$, then a is strongly normalizing.
   (CR-2) If $a \in \mathtt{Red}(A)$ and $a \to_1 b$, then $b \in \mathtt{Red}(A)$.
   (CR-3) If a is neutral, and for any b s.t. $a \to_1 b$, $b \in \mathtt{Red}(A)$ holds, then $a \in \mathtt{Red}(A)$.

   In this definition, a neutral term is either a variable or a term in the form $\mathtt{apply}(a, b)$.
3. Finally, prove that, for every term a and a substitution θ which substitutes reducible terms for variables, $a\theta$ is reducible. This is by induction on the term.

If we try to directly apply this method to $\mathsf{LK}_{c/t}$, in the final step above, we must prove that (roughly) $\mathtt{catch}(u, a)$ is reducible whenever a is reducible. Suppose a is $\mathtt{throw}(u, b)$. Then we must show that, if b is strongly normalizing, then it is reducible. But it is not possible in general.

Another difficulty is the definition of $\mathtt{Red}(A \lhd B)$. We are inclined to define that $a \in \mathtt{Red}(A \lhd B)$ if and only if $a \bullet u^B \in \mathtt{Red}(A)$ for any tag variable u^B. However this condition does not work. Lillibridge constructed a non-terminating expression using the exception mechanism in Standard ML where handlers have dynamic extents[6]. An intensive study on this example led us to realize that the definition on $\mathtt{Red}(A \lhd B)$ must rely on the type B to some extent.

The conclusion of this analysis is that (i) we must have a stronger induction hypothesis in the final step above, and (ii) we must have another kind of elimination rule which breaks the type $A \lhd B$ into a combination of A and a subtype of B. In order to solve (i), we shall use a generalized substitution $[b/^* u]$ in the final step. This idea is similar to Parigot's proof of the strong normalization of his $\lambda\mu$-calculus. For (ii), we have (already) introduced the construct $a \circ b$ which converts a term a of type $A \lhd (B \to C)$ to a term of type $A \lhd C$. Using these two improvements, our proof proceeds in a similar way as the standard proof.

4.2 Proof of the Strong Normalizability of $\mathsf{LK}_{c/t}$

Our target is the strong normalizability of $\mathsf{LK}^+_{c/t}$.

For each type A, the reducibility set $\mathtt{Red}(A)$ is defined as a subset of terms of type A.

Definition 6 (Reducibility).

$$
\begin{aligned}
a \in \mathtt{Red}(A) &\stackrel{\triangle}{=} a \textit{ is strongly normalizing} \quad \textit{(if } A \textit{ is atomic)} \\
a \in \mathtt{Red}(A \to B) &\stackrel{\triangle}{=} \mathtt{apply}(a, b) \in \mathtt{Red}(B) \ \textit{ for any } b \in \mathtt{Red}(A) \\
a \in \mathtt{Red}(A \lhd B) &\stackrel{\triangle}{=} a \bullet u \in \mathtt{Red}(A) \textit{ for any } u^B \quad \textit{(if } B \textit{ is atomic)} \\
a \in \mathtt{Red}(A \lhd (B \to C)) &\stackrel{\triangle}{=} a \bullet u \in \mathtt{Red}(A) \ \textit{ for any } u^{B \to C}, \\
&\qquad \textit{and } a \circ b \in \mathtt{Red}(A \lhd C) \ \textit{ for any } b \in \mathtt{Red}(B)
\end{aligned}
$$

Note that $\mathtt{Red}(A)$ is defined by induction on the type A. We also note that, if $a \in \mathtt{Red}(A)$, then $a[v/u] \in \mathtt{Red}(A)$.

We say a is *reducible* if $a \in \mathtt{Red}(A)$ and the type A is apparent from the context.

Definition 7 (Neutral Term). A term is neutral if it is one of the forms x, $\mathtt{apply}(a, b)$, $a \bullet u$, or $a \circ b$.

Lemma 8. *Suppose A is an implicational type. For any tag variable u^A and any strongly normalizing term a of type A, we have $\mathtt{throw}(u^A, a) \in \mathtt{Red}(B)$ for any type B.*

Similarly, for any strongly normalizing term a of type $\bot$, we have $\mathtt{abort}(a) \in \mathtt{Red}(B)$ for any type B.

Proof. By induction on the type B. □

In the following we can safely ignore the term $\mathtt{abort}(a)$, since it can be regarded as $\mathtt{throw}(u^{\bot}, a)$ if we restrict our attention to the reduction sequences.

Lemma 9. *Let a be a term of type A. Then the following conditions hold.*
(CR-1) If $a \in \mathtt{Red}(A)$, then a is strongly normalizing.
(CR-2) If $a \in \mathtt{Red}(A)$ and $a \to_1 b$, then $b \in \mathtt{Red}(A)$.
(CR-3) If a is neutral, and for any b s.t. $a \to_1 b$, $b \in \mathtt{Red}(A)$ holds, then $a \in \mathtt{Red}(A)$.

Proof. This lemma is proved by induction on the type A.

We shall prove the case for $A \equiv B \lhd (C \to D)$ only.

(CR-1) Suppose $a \in \mathtt{Red}(A)$. Take a tag variable $u^{C \to D}$. Then $a \bullet u \in \mathtt{Red}(B)$ by definition. By Induction Hypothesis, we have $a \bullet u$ is strongly normalizing, and so is a.

(CR-2) Suppose $a \in \mathtt{Red}(A)$ and $a \to_1 b$. Take any tag variable $u^{C \to D}$. Then $a \bullet u \to_1 b \bullet u$, so $b \bullet u \in \mathtt{Red}(B)$ by Induction Hypothesis.

Take any $c \in \mathtt{Red}(C)$. Then $a \circ c \to_1 b \circ c$, so $b \circ c \in \mathtt{Red}(A \lhd D)$ by Induction Hypothesis.

Hence $b \in \mathtt{Red}(A)$.

(CR-3) Suppose a is neutral, and for any b s.t. $a \to_1 b$, $b \in \mathtt{Red}(B \lhd (C \to D))$ holds.

Take any tag variable $u^{C \to D}$. Since a is neutral, any 1-step reduct of $a \bullet u$ is in the form (i) $b \bullet u$ (when $a \to_1 b$) or (ii) $\mathtt{throw}(v, d)$ (when $a \to_1 \mathtt{throw}(v, d)$). For the case (i), we have $b \bullet u \in \mathtt{Red}(B)$ by the assumption. For the case (ii), we have $\mathtt{throw}(v, d) \in \mathtt{Red}(B)$ by Lemma 8. Hence, $a \bullet u \in \mathtt{Red}(B)$ by Induction Hypothesis.

Take any $c \in \mathtt{Red}(C)$. Since a is neutral, any 1-step reduct of $a \circ c$ is in the form (i) $b \circ c$ (when $a \to_1 b$) or (ii) $\mathtt{throw}(v, d)$ (when $a \to_1 \mathtt{throw}(v, d)$). For the case (i), we have $b \circ c \in \mathtt{Red}(B \lhd D)$ by the assumption. For the case (ii), we have $\mathtt{throw}(v, d) \in \mathtt{Red}(B \lhd D)$ by Lemma 8. Hence, $a \circ c \in \mathtt{Red}(B \lhd D)$ by Induction Hypothesis.

Consequently, we have $a \in \mathtt{Red}(A)$. □

Lemma 10. *Suppose a is of type $B_1 \to \cdots B_n \to C$ where C is an atomic type (B_i is an arbitrary type).*

The term a is reducible if and only if, for any $b_1 \in \mathtt{Red}(B_1), \cdots, b_n \in \mathtt{Red}(B_n)$, the term $\mathtt{apply}(a, \overline{b_1, \cdots, b_n})$ is strongly normalizing.

Proof. Since the "only-if" part is immediate from the definition, we shall prove the "if"-part by the induction on n.

The base case ($n = 0$) is immediate. For the induction step, suppose a is of type $B_1 \to B_2 \to \cdots B_{n+1} \to C$, and for any $b_1 \in \mathtt{Red}(B_1), \cdots, b_{n+1} \in \mathtt{Red}(B_{n+1})$, the term $\mathtt{apply}(a, \overline{b_1, \cdots, b_{n+1}})$ is strongly normalizing. The term $\mathtt{apply}(a, b_1)$ is of type $B_2 \to \cdots B_{n+1} \to C$ whose length is n, hence by applying Induction Hypothesis, we have $\mathtt{apply}(a, b_1)$ is reducible. Since b_1 is an arbitrary reducible term of type B_1, we have a is reducible. □

A substitution in the following form called a reducible substitution if b_i and ${c_i}^j$ are reducible for any i and j:

$$[b_1/x_1, \cdots, b_n/x_n, \overline{c_1^1, \cdots, c_1^{k_1}}/^* u_1, \cdots, \overline{c_m^1, \cdots, c_m^{k_m}}/^* u_m]$$

We use θ as a metavariable of a reducible substitution.

Lemma 11. *Suppose a is a term of type A and $a \to_1 b$. If $a\theta$ is reducible for any reducible substitution θ, then $b\theta$ is reducible for any reducible substitution θ.*

Proof. We shall only prove the case $a \equiv (\kappa u.c) \bullet v$ and $b \equiv c[v/u]$. Let θ be an arbitrary reducible substitution in the form $[\cdots, \overline{b_1, \cdots, b_n}/^* v, \cdots]$. (If θ does not contain v free, the proof is easier.) We may assume θ does not contain u free. Then

$$a\theta \to c\theta[\overline{b_1, \cdots, b_n}/^* u][v/u] \equiv b\theta$$

Hence, by (CR-2), we have $b\theta$ is reducible for any reducible θ. □

Theorem 12. *If $\Gamma \vdash a : A \ ; \ \Delta$ in $\mathsf{LK}^+_{c/t}$, then $a\theta$ is reducible for any reducible substitution θ.*

Proof. By induction on the derivation of $\Gamma \vdash a : A \ ; \ \Delta$.

In the following, $\nu(a)$ is the maximum length of reduction sequences starting from a (it is defined only when a is strongly normalizing). Also θ is any reducible substitution. Lemmas 9 and 11 will be used without being explicitly mentioned.

(Cases: a is a variable or $\mathtt{apply}(b,c)$) Straightforward.

(Case: a is $\lambda x^B.c$) We shall prove $\mathtt{apply}(\lambda x^B.c\theta, b) \in \mathtt{Red}(C)$ for any $b \in \mathtt{Red}(B)$. We can prove it by the induction on $\nu(c\theta) + \nu(b)$, using the fact $c\theta[b/x] \in \mathtt{Red}(C)$ by Induction Hypothesis.

(Case: a is $\mathtt{catch}(u,b)$) Since $\mathtt{catch}(u, b\theta)$ is neutral, we shall prove by the induction on $\nu(b\theta)$, that all the 1-step reducts of $\mathtt{catch}(u, b\theta)$ are reducible. The term $\mathtt{catch}(u, b\theta)$ 1-step-reduces to (i) $\mathtt{catch}(u, c)$ (if $b\theta \to_1 c$), (ii) $\mathtt{throw}(v, c)$ (if $b\theta \to_1 \mathtt{throw}(v, c)$), (iii) $b\theta$ (if $u \notin FTV(b\theta)$), or (iv) c (if $b\theta \equiv \mathtt{throw}(u, c)$ and $u \notin FTV(c)$).

(i) is reducible by Induction Hypothesis of this (inner) induction.
(ii) and (iii) are reducible by Induction Hypothesis of the main induction.
We shall prove (iv) is reducible. By Induction Hypothesis, we have that $b\theta[\overline{d_1, \cdots, d_n}/^* u]$ is reducible for any reducible $d_1, \cdots, d_n$. Then we have $\mathtt{apply}(c, \overline{d_1, \cdots, d_n})$ is strongly normalizing for any reducible $d_1, \cdots, d_n$. Since the type of c is implicational, we can apply Lemma 10 and conclude that c is reducible.

(Case: a is $\mathtt{throw}(u, b)$) By Lemma 8, we only have to prove that, for any reducible $d_1, \cdots, d_n$, the term $\mathtt{apply}(b\theta, \overline{d_1, \cdots, d_n})$ is strongly normalizing but it follows from that $b\theta$ is reducible.

(Case: a is $\kappa u.b$) We first prove that $(\kappa u.b\theta) \bullet v$ is reducible for any θ and v. It is proved by the induction on $\nu(b\theta)$ using that $b\theta[v/u]$ is reducible by Induction Hypothesis.
We then prove that $(\kappa u.b\theta) \circ c$ is reducible for any θ and any reducible c. It is proved by the induction on $\nu(b\theta) + \nu(c)$ using that $b\theta[c/^* u]$ is reducible by Induction Hypothesis.

(Case: a is $b \bullet u$) Our goal is to show $(b \bullet u)\theta$ is reducible for any θ. It can be rewritten as $(b\theta' \circ \overline{d_1, \cdots, d_n}) \bullet u$ is reducible for any θ' and any reducible $d_1, \cdots, d_n$. By Induction Hypothesis $b\theta'$ is reducible, hence so is $(b\theta' \circ \overline{d_1, \cdots, d_n}) \bullet u$.

(Case: a is $b \circ c$) This case is straightforward from the definition of $\mathtt{Red}(A \lhd B)$.

Hence we have the goal. □

From the theorem, we easily have that all typable terms in $\mathsf{LK}^+_{c/t}$ are reducible, hence by (CR-1), strongly normalizing. Since $\mathsf{LK}_{c/t}$ is a subcalculus of $\mathsf{LK}^+_{c/t}$, we have the following result.

Corollary 13. *The calculi $\mathsf{LK}^+_{c/t}$ and $\mathsf{LK}_{c/t}$ are strongly normalizing.*

5 Concluding Remarks

We have examined the calculus $\mathsf{LK}_{c/t}$, the classicalized version of Nakano's $\mathsf{L}_{c/t}$, and proved several properties such as subject reduction and strong normalizability. We explained why a direct application of Tait-Girard's reducibility method does not work, and then showed how to overcome the difficulty by introducing a missing elimination rule. Our proof is purely syntactic and simple, thus extensible if we add data types such as integers and trees.

Independently to Nakano and us, de Groote[1] proposed a calculus for the exception mechanism in Standard ML. Since one can abstract over exception types in Standard ML, his calculus also contains the equivalent notion of tag-abstraction/application. However, he chooses a call-by-value evaluation strategy, so his result on normalizability is weaker than ours.

Recently, there has been intensive research on program extraction from classical proofs. Most researchers use some formulation of the first-class continuation (the `call/cc` mechanism in Scheme or Standard ML), or Parigot's $\lambda\mu$-calculus.

The second author[13, 14] and de Groote[1] pointed out the classical catch/throw (or exception) calculi can be candidates for program extraction from classical proofs. We briefly examined this idea in this paper, yet we need to work out much more examples.

Acknowledgement

We would like to express our heartful thanks to Hiroshi Nakano, Makoto Tatsuta and Izumi Takeuti for helpful comments on earlier works. The first author is supported in part by Grant-in-Aid for Scientific Research from the Ministry of Education, Science and Culture of Japan, No. 09780266.

References

1. de Groote, P.: "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**, pp. 201–215. Springer, 1995.
2. Girard. J.-Y., Y. Lafont, P. Taylor: *Proofs and Types*, Cambridge University Press, 1989.
3. Griffin, T.: "A Formulae-as-Types Notion of Control", Proc. 17th ACM Symposium on Principles of Programming Languages, pp. 47-58, 1990.
4. Kameyama, Y. and M. Sato: "The Strong Normalizability of an Intuitionistic Natural Deduction System with the Catch and the Throw Rules", draft.
5. 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, pp. 106-122, 1997.
6. Lillibridge, M.: "Exceptions Are Strictly More Powerful Than Call/CC", CMU-CS-95-178, Carnegie Mellon University, 1995.
7. Murthy, C.: "An Evaluation Semantics for Classical Proofs", *Proc. IEEE Symposium on Logic in Computer Science*, pp.96-107, 1991.
8. Nakano, H.: "A Constructive Formalization of the Catch and Throw Mechanism", *Proc. IEEE Symposium on Logic in Computer Science*, pp. 82-89, 1992.
9. Nakano, H.: "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**, pp. 61-72, 1994.
10. Nakano, H.: "A constructive logic behind the catch and throw mechanism", *Annals of Pure and Applied Logic*, Vol. 69, pp. 269-301, 1994.
11. Nakano, H.: "Logical Structures of the Catch and Throw Mechanism", *Ph. D Dissertation*, University of Tokyo, 1995.
12. Parigot, M.: "Strong Normalization for Second Order Classical Natural Deduction", *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, pp. 39-46, 1993.
13. Sato, M.: "Intuitionistic and Classical Natural Deduction Systems with the Catch and the Throw Rules", *Theoretical Computer Science* 175 (1), pp. 75-92, 1997.
14. Sato, M.: "Classical Brouwer-Heyting-Kolmogorov Interpretation", Algorithmic Learning Theory, *Lecture Notes in Artificial Intelligence* **1316**, pp. 176–196, 1997.
15. Steele Jr., G. L.: *Common Lisp: The Language*, Digital Press, 1984.