[13] Sato, M. Intuitionistic and Classical Natural Deduction Systems with the Catch and the Throw Rules. *Theoretical Computer Science*, to appear.

[14] Steele Jr., G. L. *Common Lisp: The Language*. Digital Press, 1984.

[15] Takahashi, M. Parallel Reductions in $\lambda$-Calculus. *J. Symbolic Computation*, 7:113–123, 1989.

In other words, the catch/throw mechanism should be explained in an intuitionistic framework.

**Acknowledgements**

The author would like to express his heartful thanks to Professor Masahiko Sato for invaluable supports. He also thanks to Prof. Makoto Tatsuta, Dr. Hiroshi Nakano and anonymous referees for helpful comments. This work was supported in part by Grant-in-Aid for Scientific Research from the Ministry of Education, Science and Culture of Japan, No. 08780232.

## References

[1] Clinger, W. and J. Rees, editors. *Revised$^4$ Report on the Algorithmic Language Scheme*, 1991.

[2] de Groote, Ph. A Simple Calculus of Exception Handling. In Dezani-Ciancaglini, M. and G. Plotkin, editors, *Typed Lambda Calculi and Applications, Lecture Notes in Computer Science* **902**, pages 201–215. Springer, 1995.

[3] Girard, J.-Y., Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[4] Griffin, T. A Formulae-as-Types Notion of Control. *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 47–58, 1990.

[5] Kameyama, Y. and M. Sato. The Strong Normalizability of an Intuitionistic Natural Deduction System with the Catch and the Throw Rules. manuscript, 1996.

[6] Murthy, C. An Evaluation Semantics for Classical Proofs. *Proc. 6th Annual IEEE Symposium on Logic in Computer Science*, pages 96–107, 1991.

[7] Nakano, H. A Constructive Formalization of the Catch and Throw Mechanism. *Proc. 7th Annual IEEE Symposium on Logic in Computer Science*, pages 82–89, 1992.

[8] Nakano, H. A Constructive Logic behind the Catch and Throw Mechanism. *Annals of Pure and Applied Logic*, 69:269–301, 1994.

[9] Nakano, H. The Non-deterministic Catch and Throw Mechanism and Its Subject Reduction Property. In N. D. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language and Computation, Festschrift in Honor of Satoru Takasu, Lecture Notes in Computer Science* **792**, pages 61–72. Springer, 1994.

[10] Nakano, H. *Logical Structures of the Catch and Throw Mechanism*. Ph. D Dissertation, University of Tokyo, 1995.

[11] Parigot, M. $\lambda\mu$-calculus: An Algorithmic Interpretation of Classical Natural Deduction. In A. Voronkov, editor, *Proc. International Conference on Logic Programming and Automated Reasoning, Lecture Notes in Artificial Intelligence* **624**, pages 190–201. Springer, 1992.

[12] Rehof, N. J. and M. H. S$\phi$rensen. The $\lambda_\Delta$-calculus. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software, Lecture Notes in Computer Science* **789**, pages 516–542, 1994.

We do not have the normal form theorem nor the disjunction property for the presented subcalculus. For instance, $\mathtt{?}u.\mathtt{pair}(\mathtt{!}u.\mathtt{0}, \mathtt{0})$ is a closed normal term of type $(B \times \mathtt{Int}) + \mathtt{Int}$. This is because the `throw`-mechanism in the subcalculus is not strong enough to capture the actual catch/throw mechanism.

In the same way as $\mathsf{T}_{c/t}$, a similar subcalculus of $\mathsf{T}^{-}_{c/t}$ can be defined, though it is nothing interesting, since the disjunction property no longer holds.

## 6. Concluding Remarks

We addressed the expressiveness problem of existing catch/throw calculi such as Nakano's $\mathsf{L}_{c/t}$ and Sato's $\mathsf{NJ}_{c/t}$, and proposed a new formulation method of catch/throw calculi. As concrete examples, we define two calculi $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$ as extensions of Gödel's system $T$. The reduction rules are fully compatible in the sense that, for any context $C[\ ]$, if $a \rightsquigarrow_1 b$, then $C[a] \rightsquigarrow_1 C[b]$. Also the calculi are theoretically well-founded; namely, it enjoys the subject reduction property, the strong normalization property, and corresponds to the intuitionistic propositional calculus. We showed that meaningful examples which handles run-time errors properly can be written in our calculi.

The expressive powers of $\mathsf{T}_{c/t}$ and $\mathsf{NJ}_{c/t}$ (or $\mathsf{L}_{c/t}$) are incomparable, since we cannot throw a function (for instance, $\lambda x.x$) in $\mathsf{T}_{c/t}$ but we can in $\mathsf{NJ}_{c/t}$. However, it is straightforward to extend our system to include $\mathsf{NJ}_{c/t}$ (and $\mathsf{L}_{c/t}$ essentially). The strong normalization property of the resulting system can be proved using the method in [5].

In the calculi $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$, only a few data types are allowed. However, it should be straightforward to add arbitrary inductive data types which does not have functional types. Also, we only have structured recursion in our calculus. We believe that adding general recursion to our calculus and proving the strong normalization are not so difficult, but it is left for further research.

### Comparison with Other Works

Ph. de Groote proposed a calculus which has an ML-like exception mechanism[2]. Under almost the same motivation with Nakano, he modified the semantics (and the syntax) of the exception mechanism so that the name of exception should have the static scope. Therefore, his system is quite similar to (a classical version of) $\mathsf{L}_{c/t}$. A major difference between de Groote's system and Nakano's style is that the former is classical (no restriction on the implication introduction rule), while the latter is intuitionistic. Since Sato's $\mathsf{NK}_{c/t}$ is a classical version of $\mathsf{NJ}_{c/t}$, de Groote's calculus and $\mathsf{NK}_{c/t}$ are very close calculi. It seems that each system can interpret the other.

Recently there have been much interest in extracting computational meaning from classical proofs[4, 6, 11, 12]. All these studies are based on the double-negation-elimination rule and the corresponding proofs are written using `call/cc` or other devices of program constructs, which are apparently much stronger than the catch/throw mechanism.

If we drop the side-condition in $\mathsf{T}_{c/t}$, we can also have a classical system, and the law of the excluded-middle has a proof $p$ as stated before. Using this $p$, we can give a proof of the double-negation-elimination rule, namely, $\lambda z^{\neg\neg T}.\mathtt{case}(p, x.x, y.\mathtt{abort}(\mathtt{apply}(z, y)))$ is a proof of $\neg\neg T \supset T$. But we believe this term does not have any computational meaning, since it is a closed normal term (recall that the outer-most symbol of $p$ is `?`). As a conclusion, we believe the catch/throw mechanism is too weak to extract computational meaning from classical logic.

*Proof.* Suppose $\vdash a : T + S$. By the previous theorem, $a$ is in one of the forms $\mathtt{inl}(b)$, $\mathtt{inr}(b)$ or $!u.b$ for some $b$ and $u$. Since $a$ does not have free tag variables, it is not $!u.b$. Hence, we have $\vdash b : T$ or $\vdash b : S$ for some $b$. □

Since the disjunction property contradicts to the law of the excluded middle (provided the system is consistent), it shows $\mathsf{T}^{-}_{c/t}$ is a truly constructive system. It is said that the `call/cc`-calculi can extract computational meaning from classical calculi, thus classical calculi can be basis for constructive programming. However, those calculi lack the disjunction property and the meaning of a term is not easy to understand. On the contrary, the meaning of terms and types in $\mathsf{T}^{-}_{c/t}$ can be clearly explained using the above theorems.

## 5. Confluent Subcalculus

As we mentioned, the calculus $\mathsf{T}_{c/t}$ was designed so that they allow all possible reductions non-deterministically, and thus serves as a basis for the catch/throw calculi. The fact $\mathsf{T}_{c/t}$ is not confluent makes it difficult to develop a logical theory which can reason about properties of programs in $\mathsf{T}_{c/t}$. However, this point can be recovered since there is at least one confluent subcalculus of $\mathsf{T}_{c/t}$ which inherits the good properties from $\mathsf{T}_{c/t}$. In this section we consider a confluent subcalculus of $\mathsf{T}_{c/t}$.

The source of the non-confluency is the reduction rule $a[!u.b/x] \leadsto_1 !u.b$. It is a natural rule, but is too strong. To keep confluency, we have to regard, for instance, the term $\mathtt{pair}(!u.x, !u.y)$ as a normal term while we may reduce $\mathtt{car}(!u.x)$ to $!u.x$. Based on this idea, we eliminate the rule $a[!u.b/x] \leadsto_1 !u.b$ and add a set of reduction rules shown in Table 5-1.

| |
|---|
| $\mathtt{car}(!u.b) \leadsto_1 !u.b$ |
| $\mathtt{cdr}(!u.b) \leadsto_1 !u.b$ |
| $\mathtt{case}(!u.b, x.c, y.d) \leadsto_1 !u.b$ |
| $\mathtt{apply}(!u.b, c) \leadsto_1 !u.b$ |
| $\mathtt{abort}(!u.b) \leadsto_1 !u.b$ |
| $\mathtt{if}(!u.b, c, d) \leadsto_1 !u.b$ |
| $\mathtt{Rec}(b, c, !u.d) \leadsto_1 !u.d$ |
| $\mathtt{ListRec}(b, c, !u.d) \leadsto_1 !u.d$ |
| $\mathtt{tapply}(!u.b, v) \leadsto_1 !u.b$ |

Table 5-1: One-step Reduction Rules in Confluent Subcalculus

By these rules, $!u.b$ can go through a destructor, but cannot go through a constructor. For instance $\mathtt{inl}(!u.x) \leadsto_1 !u.x$ is not allowed. As a result, there are much more normal forms in this subcalculus than the full version of $\mathsf{T}_{c/t}$. Nevertheless, this subcalculus is expressive in that the examples given in this paper can be written in this subcalculus, too[e].

This subcalculus is close to Rehof and Sørensen's catch/throw subcalculus of the $\lambda_\Delta$ calculus except that our calculus has many more constructs such as disjunction and the `tapply` construct.

**Theorem 5.1 (Properties of the Subcalculus)** *The properties of Subject Reduction, Strong Normalization, and Confluency hold for the subcalculus of* $\mathsf{T}_{c/t}$ *defined above.*

*Proof.* The first two are immediate consequences of those for $\mathsf{T}_{c/t}$. The confluency is proved by a standard method; say, Takahashi's method of parallel reduction[15]. □

[e]This is because the return values are integers or list of integers in all the examples. We cannot write such a function that it returns a pair of integers, and it uses the catch/throw mechanism.

Among the one-step reducts of $\mathtt{outl}(?u.b')$, the key case is $\mathtt{outl}(\mathtt{inl}(b'))$ where $u \notin FTV(b')$. Since $b'$ is reducible, so is $\mathtt{outl}(\mathtt{inl}(b'))$.

Among the one-step reducts of $\mathtt{outr}(?u.b')$, the key case is $\mathtt{outr}(\mathtt{inr}(c))$ where $b' \equiv\ !u.c$. Since $b'$ is strongly normalizing, so is $c$. Since the type of $c$ is a data type, $c$ is reducible. (This is the keypoint why we restricted the type of thrown terms to data types.) Hence $\mathtt{outr}(\mathtt{inr}(c))$ is reducible.

Suppose $?u.b' \rightsquigarrow ?v.c$. Then we have $u \equiv v$ and $b' \rightsquigarrow c$. Since $b'$ is reducible, so is $c$.

Hence $?u.b'$ is reducible.

**(Case: $a$ is $!u.b$)** Let $\theta$ be a reducible substitution. Let $b'$ be $b\theta$.

Since $b'$ is reducible, it is strongly normalizing. It is easy to prove $!u.b'$ is reducible (for any type).

**(Case: $a$ is $\mathtt{tapply}(b,u)$)** Let $\theta$ be a reducible substitution. Let $b'$ be $b\theta$.

Our goal is to show $\mathtt{tapply}(b',u)$ is reducible. It is proved by induction on $\mu(b')$. The key cases are that (1) $b' \rightsquigarrow ?v.c$, and $\mathtt{tapply}(b',u) \rightsquigarrow c[u/v]$, (2) $b' \rightsquigarrow \mathtt{inl}(c)$, and $\mathtt{tapply}(b',u) \rightsquigarrow c$, and (3) $b' \rightsquigarrow \mathtt{inr}(c)$, and $\mathtt{tapply}(b',u) \rightsquigarrow\ !u.c$. For the case (1), we have $c$ is reducible since $b'$ is a reducible term of a sum type. Renaming of free tag variables does not affect the reducibility of the term, hence $c[u/v]$ is reducible. For the cases (2) and (3), we immediately have $c$ and $!u.c$ are reducible since $b'$ is reducible.

Hence we have the goal. □

Theorem 4.1 is an immediate consequence of this lemma.

### *4.2. Normal Form Property and Disjunction Property of* $\mathsf{T}^-_{c/t}$

In addition to the two properties stated in the previous subsection, the calculus $\mathsf{T}^-_{c/t}$ enjoys two more properties.

**Theorem 4.2 (Normal Form Property for $\mathsf{T}^-_{c/t}$)** *In $\mathsf{T}^-_{c/t}$, if $\vdash a : T\ ;\ \Delta$ is provable and $a$ is normal, then $a$ is a constructor term or in the form $!u.b$ for some $u$ and $b$.*

We first note that this theorem is not trivial; it states that a term in the form $?u.a$ with no free individual variables cannot be normal.

*Proof.* We prove the theorem by induction on the term $a$.

If $a$ is a constructor term or in the form $!u.b$, then we have the conclusion. If $a$ is a destructor term, then applying the induction hypothesis, we have that $a$ is not normal. Contradiction.

Suppose that $a$ is in the form of $?u.b$. Then we can prove $FTV(b) = \emptyset$ by induction on $b$. (The crucial point is that, in $\mathsf{T}^-_{c/t}$ a functional type is not allowed for the type $b$.) Then $?u.b \rightsquigarrow_1 \mathtt{inl}(b)$, contradicting to the assumption that $a$ is normal. □

The normal form property does not hold for $\mathsf{T}_{c/t}$; for example, $?u.(\lambda x.(!u.x))$ is a well-typed closed normal term.

**Corollary 4.1 (Disjunction Property for $\mathsf{T}^-_{c/t}$)** *In $\mathsf{T}^-_{c/t}$, if $\vdash a : T + S$ is provable, then either $\vdash b : T$ or $\vdash b : S$ is provable for some term $b$.*

**Definition 4.1 (Reducibility)**

$$
\begin{array}{rcl}
a \in Red(T) & \triangleq & a \textit{ is strongly normalizing } (\textit{if } T \textit{ is an atomic type or a data type}) \\
a \in Red(T \times S) & \triangleq & \mathtt{car}(a) \in Red(T) \textit{ and } \mathtt{cdr}(a) \in Red(S) \\
a \in Red(T + S) & \triangleq & \mathtt{outl}(a) \in Red(T) \textit{ and } \mathtt{outr}(a) \in Red(S) \\
 & & \textit{and for any } u \textit{ and } b, \textit{ if } a \rightsquigarrow \mathtt{?}u.b, \textit{ then } b \in Red(T) \\
a \in Red(T \to S) & \triangleq & \textit{for any } b \in Red(T), \mathtt{apply}(a, b) \in Red(S)
\end{array}
$$

There is ambiguity in this definition; for a datatype $D$, more than one clause can be applied. However, we can easily prove that the two clauses are equivalent for datatypes, so it does not matter.

The definition of reducibility is standard except for the type $T + S$. The additional clause is necessary to verify the case for $\mathtt{tapply}(a, u)$ in Lemma 4.3. See the proof below.

We say a term $a$ of type $T$ is reducible if $a \in Red(T)$.

**Definition 4.2 (Neutral Term)** *A term is neutral if it is one of the forms* $x$, $\mathtt{car}(a)$, $\mathtt{cdr}(a)$, $\mathtt{case}(a, x.b, y.c)$, $\mathtt{apply}(a, b)$, $\mathtt{tapply}(a, u)$, $\mathtt{if}(a, b, c)$, $\mathtt{Rec}(a, b, c)$, *and* $\mathtt{ListRec}(a, b, c)$

As in the standard proof, the following lemma holds for $\mathsf{T}_{c/t}$.

**Lemma 4.2** *Let $a$ be a term of type $T$. Then the following conditions hold.*
*(CR-1) If $a \in Red(T)$, then $a$ is strongly normalizing.*
*(CR-2) If $a \in Red(T)$ and $a \rightsquigarrow_1 b$, then $b \in Red(T)$.*
*(CR-3) If $a$ is neutral and $b \in Red(T)$ holds for any $b$ such that $a \rightsquigarrow_1 b$, then $a \in Red(T)$.*

*Proof.* This lemma is proved by induction on the type $T$. □

A substitution in the form $[b_1/x_1, \cdots, b_n/x_n]$ is *reducible* if $b_1, \cdots, b_n$ are reducible.

**Lemma 4.3** *If $\Gamma \vdash a : T \ ; \ \Delta$ in $\mathsf{IF}_{c/t}$, then we have $a\theta \in Red(T)$ for any reducible substitution $\theta$.*

*Proof.* By induction on the derivation of $\Gamma \vdash a : T \ ; \ \Delta$ (or equivalently induction on the term $a$). For a strongly normalizing term $a$, we write the length of the longest reduction sequence of $a$ as $\mu(a)$.

Since all the cases except $\mathtt{abort}(b)$, $\mathtt{?}u.b$, $\mathtt{!}.b$ and $\mathtt{tapply}(b, u)$ can be proved in the same way as the standard proof, we shall treat only these cases.

**(Case: $a$ is $\mathtt{abort}(b)$)** Let $\theta$ be a reducible substitution. By the induction hypothesis $b\theta$ is reducible, hence strongly normalizing. We can easily show that $\mathtt{abort}(b) \in Red(T)$ for any strongly normalizing term $b$ and an arbitrary type $T$ by induction on $T$.

**(Case: $a$ is $\mathtt{?}u.b$)** Let $\theta$ be a reducible substitution. We may assume $(\mathtt{?}u.b)\theta \equiv \mathtt{?}u.b\theta$. Let $b'$ be $b\theta$.

Our goal is to show $\mathtt{?}u.b'$ is reducible, namely, $\mathtt{outl}(\mathtt{?}u.b')$ and $\mathtt{outr}(\mathtt{?}u.b')$ are reducible and if $\mathtt{?}u.b' \rightsquigarrow \mathtt{?}v.c$, then $c$ is reducible.

The first two are proved by induction on $\mu(b')$ as follows:

$$\begin{aligned}
\times &\triangleq \lambda xy.\,\mathtt{Rec}(\mathtt{0},\ \lambda ab.\ +(a,y),\ x)\\
\mathtt{getbody} &\triangleq \lambda a.\mathtt{case}(a, x.x, y.y)\\
\mathtt{multiply} &\triangleq \lambda x.\,\mathtt{getbody}(?u.\mathtt{ListRec}(\mathtt{suc}(\mathtt{0}),\ \lambda abc.\,\mathtt{if}(\mathtt{zerop}(b),\ !u.\mathtt{0},\ \times(a,b)),\ x))
\end{aligned}$$

By adopting the call-by-name evaluation strategy, the above program implements the multiplication slightly more efficiently than the usual one. This example can be written in $\mathsf{L}_{c/t}$ or $\mathsf{NJ}_{c/t}$, since the thrown term is a constant.

The next function `firstelem` receives a list of lists of integers, and returns a list of the first elements of each element list. For example, $\mathtt{firstelem}(\mathtt{cons}(\mathtt{cons}(1,\mathtt{nil}),\mathtt{cons}(\mathtt{cons}(2,\mathtt{nil}),\mathtt{nil})))$ is $\mathtt{inl}(\mathtt{cons}(1,\mathtt{cons}(2,\mathtt{nil})))$. If some list is the empty list `nil`, then it is an error, so it returns the rest of the input list to indicate where the error occurs.

$$\begin{aligned}
\mathtt{null} &\triangleq \lambda x.\,\mathtt{ListRec}(\mathtt{true},\ \lambda abc.\,\mathtt{false},\ x)\\
\mathtt{first} &\triangleq \lambda x.\,\mathtt{ListRec}(\mathtt{0},\ \lambda abc.\ b,\ x)\\
\mathtt{firstelem} &\triangleq \lambda x.\,?u.(\mathtt{ListRec}(\mathtt{nil},\ \lambda abc.\,\mathtt{if}(\mathtt{null}(b),\ !u.c,\ \mathtt{cons}(\mathtt{first}(b),a)),\ x))
\end{aligned}$$

The function `firstelem` is not a well-typed term in $\mathsf{L}_{c/t}$ nor $\mathsf{NJ}_{c/t}$ (even if we add data types to them).

## 4. Properties of the calculi

In this section, we show that several theoretically interesting properties hold for $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$, namely, the subject reduction property and the strong normalization property. Moreover, $\mathsf{T}^{-}_{c/t}$ satisfies the normal form theorem which implies the disjunction property.

### *4.1. Subject Reduction and Strong Normalization*

**Lemma 4.1 (Subject Reduction Property)** *For both of $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$, if $\Gamma \vdash a : T \ ; \ \Delta$ and $a \rightsquigarrow a'$, then $\Gamma \vdash a' : T \ ; \ \Delta$.*

*Proof.* By induction on the length of the reduction. □

The proofs of the subject reduction property for $\mathsf{L}_{c/t}$ and $\mathsf{NJ}_{c/t}$ are not straightforward, since we must show the side-condition is kept through the reduction. On the contrary, to show the subject reduction property for $\mathsf{T}_{c/t}$ is quite easy, since we have no complicated side-condition.

We then prove the strong normalization for $\mathsf{T}_{c/t}$ using Tait-Girard's reducibility method.

**Theorem 4.1 (Strong Normalization)** *The calculi $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$ are strongly normalizing.*

In the rest of this subsection, we shall prove this theorem.

First we take a fresh variable $k^T$ for each type $T$, and fix it. We define two auxiliary functions as follows:

$$\begin{aligned}
\mathtt{outl}(c) &\triangleq \mathtt{case}(c, x.x, y.k^T)\ (if\ c\ has\ type\ S+T)\\
\mathtt{outr}(c) &\triangleq \mathtt{case}(c, x.k^S, y.y)\ (if\ c\ has\ type\ S+T)
\end{aligned}$$

Since we consider well-typed terms only, these functions are well-defined.

Next, we define reducibility.

Table 2-2 gives the notion of one-step reduction which corresponds to the data types. They are all standard.

$$
\begin{array}{c}
a[!u.b/x] \leadsto_1 !u.b \ \ (where\ x \in FV(a)\ and\ a \not\equiv x) \\
?u.a \leadsto_1 \mathtt{inl}(a) \ \ (where\ u \notin FTV(a)) \\
?u.(!u.a) \leadsto_1 \mathtt{inr}(a) \ \ (where\ u \notin FTV(a)) \\
\mathtt{tapply}(?u.a, v) \leadsto_1 a[v/u] \\
\mathtt{tapply}(\mathtt{inl}(a), u) \leadsto_1 a \\
\mathtt{tapply}(\mathtt{inr}(a), u) \leadsto_1 !u.a
\end{array}
$$

Table 2-3: One-step Reduction for Catch/Throw

Table 2-3 gives the notion of one-step reduction related to the catch/throw mechanism. The intuitive meaning of $?u.a$ is that, if something is thrown by the `!`-construct, then the result is in the form of $\mathtt{inr}(b)$ where $b$ is the thrown term, and if nothing is thrown during the computation of $a$, then the result is in the form of $\mathtt{inl}(a)$. Instead of having a one-step reduction like $?u.a[!u.b/x] \leadsto_1 \mathtt{inr}(b)$, we split the throw mechanism into two steps following $\mathsf{L}_{c/t}$:

$$?u.a[!u.b/x] \leadsto_1 ?u.(!u.b) \leadsto_1 \mathtt{inr}(b)$$

Note that, we used the notation for substitution in the lefthand-side of the first rule in Table 2-3. By using substitution, reductions like $\lambda x.(!u.x) \leadsto_1 !u.x$ are prohibited.

The last three reduction rules are for the `tapply`-construct. The first one $\mathtt{tapply}(?u.a, v) \leadsto_1 a[v/u]$ is easily understood since `?` is the tag-abstraction, and `tapply` is the tag-application. The rest two are for calculation of other cases. The following is an example of reductions using these rules.

$$\mathtt{tapply}(?v.\mathtt{car}(!v.a), u) \leadsto_1 \mathtt{car}(!u.a) \leadsto_1 !u.a$$

The reduction $a[!u.b/x] \leadsto_1 !u.b$ is in a sense quite strong one. While it represents the intuitive operational semantics of the `throw`-constructs, it makes the whole calculus not Church-Rosser. For instance, $?u.\mathtt{pair}(!u.x, !u.y)$ reduces to $\mathtt{inr}(x)$ and $\mathtt{inr}(y)$. Nevertheless, we are interested in this style, since we can define a confluent catch/throw calculus in many ways as a subcalculus of our non-deterministic calculus. In addition, for several properties such as strong normalization, it is sufficient to prove one for the full calculus. We shall show in Section 5 that our calculus contains at least one meaningful confluent subcalculus.

We say $a$ is a normal term if no reduction rule can be applied to the term $a$. We define $a \leadsto b$ (zero or more step reduction), and $a \leadsto_+ b$ (one or more step reduction) as usual.

## 3. Examples in $\mathsf{T}_{c/t}$

We give a few examples which exhibits the expressiveness of $\mathsf{T}_{c/t}$.

The first example is an implementation of `multiply` given before. The function `multiply` receives a list of integers, and returns the multiplication of all elements. If it finds `0`, then it immediately exits.

$$
\begin{array}{rcl}
\mathtt{zerop} & \triangleq & \lambda x.\, \mathtt{Rec}(\mathtt{true},\ \lambda ab.\mathtt{false},\ x) \\
+ & \triangleq & \lambda xy.\, \mathtt{Rec}(y,\ \lambda ab\ .\mathtt{suc}(a),\ x)
\end{array}
$$

**Theorem 2.1** *Let $T$ be a type in $\mathsf{T}_{c/t}$. Then $\Gamma \vdash a : T$ is provable for some term $a$ in $\mathsf{T}_{c/t}$ if and only if $\phi(\Gamma) \vdash \phi(T)$ is provable in* IPC.

*Proof.* The if-part of the theorem is trivial.

The only-if-part is shown by induction on the size of the proof $\Gamma \vdash a : T$.

For the base case, the last inference rule is the variable rule. Then $a$ is a variable and $a : T \in \Gamma$, so we have $\phi(\Gamma) \vdash \phi(T)$.

For the induction step, we first note that the last inference rule is not the `!`-rule, since the last sequent of the proof is $\Gamma \vdash a : T \ ; \ \{\}$. Suppose that the last inference rule is the `?`-rule as follows.

$$\frac{\begin{array}{c}\vdots \\ \Gamma \vdash a : T \ ; \ \{u : D\}\end{array}}{\Gamma \vdash \mathtt{?}u.a : T + D}$$

where $D$ is a data type. Since $\phi(D)$ is a provable formula, we have the following proof in $M$:

$$\frac{\overline{\phi(D)}}{\phi(T) \vee \phi(D)}$$

Suppose that the last inference rule is not the `?`-rule, then it is straightforward to translate the proof of $\Gamma \vdash a : T$ to a proof of $\phi(\Gamma) \vdash \phi(T)$ using the induction hypothesis. □

Next, we define reduction rules of $\mathsf{T}_{c/t}$. The reduction rules of $\mathsf{T}^{-}_{c/t}$ are the same as those of $\mathsf{T}_{c/t}$. In the following, $C[\ ]$ represents a context in a usual sense.

| |
|---|
| $C[a] \leadsto_1 C[b]$ (if $a \leadsto_1 b$) |
| $\mathtt{car}(\mathtt{pair}(a, b)) \leadsto_1 a$ |
| $\mathtt{cdr}(\mathtt{pair}(a, b)) \leadsto_1 b$ |
| $\mathtt{case}(\mathtt{inl}(a), x.b, y.c) \leadsto_1 b[a/x]$ |
| $\mathtt{case}(\mathtt{inr}(a), x.b, y.c) \leadsto_1 c[a/y]$ |
| $\mathtt{apply}(\lambda x.a, b) \leadsto_1 a[b/x]$ |

Table 2-1: One-step Reduction for Logic

Table 2-1 gives the notion of one-step reduction which corresponds to the standard normalization procedure in the propositional logic (except the commuting conversions). Note that, in the `abort`-rule, the lefthand side is written using substitution so that no free variable in $b$ may be captured in $a$. For instance, $\lambda y.\mathtt{abort}(y) \leadsto_1 \mathtt{abort}(y)$ is disallowed (as expected).

By the first line, our reduction rules are compatible with *any* context. This is contrary to the `call/cc` calculi and de Groote's `exception` calculi where restricted contexts are allowed for reducing `call/cc` or `exception`. The motivation of Rehof and Sørensen's $\lambda_\Delta$-calculus is similar to us, but their calculus is only call-by-name in the catch/throw mechanism.

| |
|---|
| $\mathtt{if}(\mathtt{true}, a, b) \leadsto_1 a$ |
| $\mathtt{if}(\mathtt{false}, a, b) \leadsto_1 b$ |
| $\mathtt{Rec}(a, b, \mathtt{0}) \leadsto_1 a$ |
| $\mathtt{Rec}(a, b, \mathtt{suc}(c)) \leadsto_1 \mathtt{apply}(\mathtt{apply}(b, \mathtt{Rec}(a, b, c)), c)$ |
| $\mathtt{ListRec}(a, b, \mathtt{nil}) \leadsto_1 a$ |
| $\mathtt{ListRec}(a, b, \mathtt{cons}(c, d)) \leadsto_1 \mathtt{apply}(\mathtt{apply}(\mathtt{apply}(b, \mathtt{ListRec}(a, b, d)), c), d)$ |

Table 2-2: One-step Reduction for Data Type

$$\frac{\Gamma \vdash b : D \ ; \ \Delta}{\Gamma \vdash \ !u.b : T \ ; \ \Delta \cup \{u : D\}}$$

$$\frac{\Gamma \vdash a : S \ ; \ \Delta \cup \{u : D\}}{\Gamma \vdash \ ?u.a : S + D \ ; \ \Delta} \qquad \frac{\Gamma \vdash c : T + D \ ; \ \Delta}{\Gamma \vdash \texttt{tapply}(c, u) : T \ ; \ \Delta \cup \{u : D\}}$$

In these three rules, the type $D$ must be a data type.
In addition, the type $S$ must be a data type for the calculus $\mathsf{T}^{-}_{c/t}$.

Table 1-3: Inference Rules for Catch/Throw

Table 1-3 gives inference rules for the catch/throw mechanism. Note that, the type a thrown term ($b$ in the introduction rule of $!u.b$) must be a data type as well as the type of $c$ in $\texttt{tapply}(c, u)$ must be $T + D$ with $D$ a data type. These conditions are crucial to our proof of the strong normalization property and also to keep our system intuitionistic. If we would remove this condition, we would obtain a classical calculus as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\{x : T\} \vdash x : T}}{\{x : T\} \vdash \ !u.x : \bot \ ; \ \{u : T\}}}{\vdash \lambda x.(!u.x) : T \to \bot \ ; \ \{u : T\}}}{\vdash \ ?u.(\lambda x.(!u.x)) : (T \to \bot) + T}$$

The type $(T \to \bot) + T$ corresponds to the law of the excluded middle $\neg T \vee T$, thus the system becomes a classical logic. By restricting the type of the thrown term, $(T \to \bot) + T$ is provable only when $T$ is a data type. Since a data type is always inhabited[d], $T$ is logically equivalent to a provable formula, therefore the formula $(T \supset \bot) \vee T$ is provable in the intuitionistic logic.

For the calculus $\mathsf{T}^{-}_{c/t}$, an additional side-condition that the type $S$ must be a data type in the introduction rule of $?u.a$. For instance, $?u.(\lambda x.x)$ is a term in $\mathsf{T}_{c/t}$, but is not a term in $\mathsf{T}^{-}_{c/t}$.

If $\Gamma \vdash a : T \ ; \ \Delta$ holds for some $\Gamma, T$, and $\Delta$, we say $a$ is a typable term (or simply a term) of type $T$. Note that a term may have free individual/tag variables. Let $a, b, c, \cdots$ be metavariables for terms.

We write $\Gamma \vdash a : T$ if $\Delta \equiv \emptyset$, and $\vdash a : T$ if $\Gamma \equiv \Delta \equiv \emptyset$.

The following figure is an example of type inference which shows the term $\lambda x.(?u.\texttt{pair}(!u.x, x))$ has the type $\texttt{Int} \to ((T \times \texttt{Int}) + \texttt{Int})$ for an arbitrary type $T$.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\{x : \texttt{Int}\} \vdash x : \texttt{Int}}}{\{x : \texttt{Int}\} \vdash \ !u.x : T \ ; \ \{u : \texttt{Int}\}} \quad \cfrac{}{\{x : \texttt{Int}\} \vdash x : \texttt{Int} \ ; \ \{u : \texttt{Int}\}}}{\{x : \texttt{Int}\} \vdash \texttt{pair}(!u.x, x) : T \times \texttt{Int} \ ; \ \{u : \texttt{Int}\}}}{\{x : \texttt{Int}\} \vdash \ ?u.\texttt{pair}(!u.x, x) : (T \times \texttt{Int}) + \texttt{Int}}}{\vdash \lambda x.(?u.\texttt{pair}(!u.x, x)) : \texttt{Int} \to ((T \times \texttt{Int}) + \texttt{Int})}$$

We now make explicit the claim that $\mathsf{T}_{c/t}$ (and $\mathsf{T}^{-}_{c/t}$) is intuitionistic. Let IPC be the intuitionistic propositional calculus. Let $\phi$ be a mapping from a type in $\mathsf{T}_{c/t}$ to a formula in IPC so that $\phi(\bot) \equiv \bot$, $\phi(D) \equiv \bot \supset \bot$ ($D$ is a data type), $\phi(A) \equiv A$ ($A$ is an atomic type other than $\bot$, Int and Bool), $\phi(T_1 \times T_2) \equiv \phi(T_1) \wedge \phi(T_2)$, $\phi(T_1 + T_2) \equiv \phi(T_1) \vee \phi(T_2)$, and $\phi(T_1 \to T_2) \equiv \phi(T_1) \supset \phi(T_2)$. The mapping $\phi$ naturally extends to sets of types, namely, $\phi(\{T_1, \cdots, T_n\}) \equiv \{\phi(T_1), \cdots, \phi(T_n)\}$. Then we have the following theorem which shows the relationship between $\mathsf{T}_{c/t}$ and IPC.

[d] This is why we excluded $\bot$ from the definition of the data type.

- Inference Rules for `Bool`, `Int`, and `List`.
- Inference Rules for the Catch/Throw mechanism

The first two classes correspond to Gödel's system (with a slight extension), and standard. The last class is the most important for our calculus. We give these classes in order.

$$\frac{}{\Gamma \cup \{x : T\} \vdash x : T \ ; \ \Delta}$$

$$\frac{\Gamma \vdash a : \bot \ ; \ \Delta}{\Gamma \vdash \mathtt{abort}(a) : T \ ; \ \Delta}$$

$$\frac{\Gamma \vdash a : T \ ; \ \Delta \quad \Gamma \vdash b : S \ ; \ \Delta}{\Gamma \vdash \mathtt{pair}(a,b) : T \times S \ ; \ \Delta} \qquad \frac{\Gamma \vdash c : T \times S \ ; \ \Delta}{\Gamma \vdash \mathtt{car}(c) : T \ ; \ \Delta} \qquad \frac{\Gamma \vdash c : T \times S \ ; \ \Delta}{\Gamma \vdash \mathtt{cdr}(c) : S \ ; \ \Delta}$$

$$\frac{\Gamma \vdash a : T \ ; \ \Delta}{\Gamma \vdash \mathtt{inl}(a) : T + S \ ; \ \Delta} \qquad \frac{\Gamma \vdash b : S \ ; \ \Delta}{\Gamma \vdash \mathtt{inr}(b) : T + S \ ; \ \Delta}$$

$$\frac{\Gamma \vdash c : T + S \ ; \ \Delta \quad \Gamma \cup \{x : T\} \vdash a : C \ ; \ \Delta \quad \Gamma \cup \{y : S\} \vdash b : C \ ; \ \Delta}{\Gamma \vdash \mathtt{case}(c, x.a, y.b) : C \ ; \ \Delta}$$

$$\frac{\Gamma \cup \{x : T\} \vdash b : S \ ; \ \Delta}{\Gamma \vdash \lambda x.b : T \to S \ ; \ \Delta} \qquad \frac{\Gamma \vdash c : T \to S \ ; \ \Delta \quad \Gamma \vdash a : T \ ; \ \Delta}{\Gamma \vdash \mathtt{apply}(c,a) : S \ ; \ \Delta}$$

Table 1-1: Inference Rules for Logical Connectives

Table 1-1 gives standard inference rules for propositional logical connectives. Note that, the introduction rule of $\lambda x.b$ has no side-condition.

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{Bool} \ ; \ \Delta} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{Bool} \ ; \ \Delta}$$

$$\frac{\Gamma \vdash a : \mathtt{Bool} \ ; \ \Delta \quad \Gamma \vdash b : T \ ; \ \Delta \quad \Gamma \vdash c : T \ ; \ \Delta}{\Gamma \vdash \mathtt{if}(a,b,c) : T \ ; \ \Delta}$$

$$\frac{}{\Gamma \vdash \mathtt{0} : \mathtt{Int} \ ; \ \Delta} \qquad \frac{\Gamma \vdash a : \mathtt{Int} \ ; \ \Delta}{\Gamma \vdash \mathtt{suc}(a) : \mathtt{Int} \ ; \ \Delta}$$

$$\frac{\Gamma \vdash a : T \ ; \ \Delta \quad \Gamma \vdash b : T \to \mathtt{Int} \to T \ ; \ \Delta \quad \Gamma \vdash c : \mathtt{Int} \ ; \ \Delta}{\Gamma \vdash \mathtt{Rec}(a,b,c) : T \ ; \ \Delta}$$

$$\frac{}{\Gamma \vdash \mathtt{nil} : \mathtt{List}(T) \ ; \ \Delta} \qquad \frac{\Gamma \vdash a : T \ ; \ \Delta \quad \Gamma \vdash b : \mathtt{List}(T) \ ; \ \Delta}{\Gamma \vdash \mathtt{cons}(a,b) : \mathtt{List}(T) \ ; \ \Delta}$$

$$\frac{\Gamma \vdash a : S \ ; \ \Delta \quad \Gamma \vdash b : S \to T \to \mathtt{List}(T) \to S \ ; \ \Delta \quad \Gamma \vdash c : \mathtt{List}(T) \ ; \ \Delta}{\Gamma \vdash \mathtt{ListRec}(a,b,c) : S \ ; \ \Delta}$$

Table 1-2: Inference Rules for `Bool`, `Int`, and `List`

Table 1-2 gives standard inference rules for data types.

**Definition 2.2 (Preterm)**

$$
\begin{array}{rcl}
t & ::= & x \\
& | & \texttt{pair}(t,t) \mid \texttt{car}(t) \mid \texttt{cdr}(t) \\
& | & \texttt{inl}(t) \mid \texttt{inr}(t) \mid \texttt{case}(t, x.t, x.t) \\
& | & \lambda x.t \mid \texttt{apply}(t,t) \\
& | & \texttt{true} \mid \texttt{false} \mid \texttt{if}(t,t,t) \\
& | & \texttt{0} \mid \texttt{suc}(t) \mid \texttt{Rec}(t,t,t) \\
& | & \texttt{nil} \mid \texttt{cons}(t,t) \mid \texttt{ListRec}(t,t,t) \\
& | & \texttt{abort}(t) \\
& | & \texttt{?}u.t \mid \texttt{!}u.t \mid \texttt{tapply}(t,u)
\end{array}
$$

We use metavariables $s, t$ for preterms. Most of the above definition is standard; the second, third, and fourth lines define preterms related to $\times, +$ and $\rightarrow$ types. The fifth, sixth, and seventh lines define preterms related to `Bool`, `Int` and `List` types. The second last line corresponds to the $\bot$-elimination rule. The last line defines three constructs `?`, `!`, and `tapply` which represent the catch/throw mechanism as in Sato's style.

The terms $\texttt{?}u.t$ and $\texttt{!}u.t$ roughly correspond to the programs `(catch 'u t)` and `(throw 'u t)` in Common Lisp. For instance, `(catch 'u (car (throw 'u (lambda (x) x))))` in Common Lisp can be roughly represented as $\texttt{?}u.\texttt{car}(\texttt{!}u.(\lambda x.x))$. The `?` and `!` primitives have slightly different meaning than the catch/throw primitives in Common Lisp. For the term $\texttt{?}u.\texttt{car}(\texttt{!}u.(\lambda x.x))$, the type $S$ of the thrown term $\lambda x.x$ and the type $T$ of the body of the `?`-expression $\texttt{car}(\texttt{!}u.(\lambda x.x))$ can be different. The type of the whole term $\texttt{?}u.\texttt{car}(\texttt{!}u.(\lambda x.x))$ is $T + S$.

Among these preterms, the preterm $\texttt{tapply}(t,u)$ is the most difficult to understand. It was introduced by Sato as an inverse operation of the `catch` or `?` constructs, namely, it makes the tag variable $u$ free.

Preterms in the form $\texttt{abort}(t)$, $\texttt{pair}(t,t)$, $\texttt{inl}(t)$, $\texttt{inr}(t)$, $\lambda x.t$, `true`, `false`, `0`, $\texttt{suc}(t)$, `nil` and $\texttt{cons}(t,t)$ are *constructor* preterms. Preterms in the form $\texttt{car}(t)$, $\texttt{cdr}(t)$, $\texttt{case}(t,x.t,x.t)$, $\texttt{apply}(t,t)$, $\texttt{if}(t,t,t)$, $\texttt{Rec}(t,t,t)$, $\texttt{ListRec}(t,t,t)$, and $\texttt{tapply}(t,u)$ are *destructor* preterms. Note that we excluded $\texttt{?}u.t$ and $\texttt{!}u.t$ from these definitions.

An individual variable is bound by the $\lambda$- and `case`-constructs; for instance, in the preterm $\texttt{case}(s,x.t,y,t')$, $x$ is bound in $t$ and $y$ is bound in $t'$. A tag variable is bound by the `?`-construct. Other occurrences of individual variables and tag variables are free. 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 substitutions $t[s/x]$ and $t[v/u]$ are defined as usual.

A judgement is in the form $\Gamma \vdash t : T \ ; \ \Delta$ where $t$ is a preterm, $T$ is a type, and $\Gamma$ and $\Delta$ are finite sets of type assignment of individual variables and that of tag variables respectively. For example,

$$\{x : \texttt{Int}\} \vdash \texttt{pair}(\texttt{!}u.x, x) : \texttt{Bool} \times \texttt{Int} \ ; \ \{u : \texttt{Int}\}$$

is a judgement. In the following, we assume that $\Gamma$ and $\Delta$ in a judgement are *compatible* sets; namely, if $x : T \in \Gamma$ and $x : S \in \Gamma$ then $T \equiv S$, and if $u : T \in \Delta$ and $u : S \in \Delta$ then $T \equiv S$.

To derive a judgement, we have the following three classes of inference rules.

- Inference Rules for Logical Connectives

to intuitionistic calculi, since there are computationally meaningless terms in $\mathsf{NK}_{c/t}$.

### *2.2. Gödel's system* $\mathsf{T}$

Gödel's system $\mathsf{T}$[3] is a kind of strictly typed lambda calculi. It has atomic types including `Bool` (Booleans) and `Int` (integers), and the type constructors $\times$ (product) and $\to$ (function space). Well-typed terms are introduced in a standard way. For example, the term $\mathtt{pair}(a, b)$ is of type $T \times S$ if $a$ is of type $T$ and $b$ is of type $S$. The terms `true` and `false` are of type `Bool`, and `0` and $\mathtt{suc}(n)$ are of type `Int` (if $n$ is of type `Int`).

Corresponding to the elimination rules for the types `Bool` and `Int`, function symbols `if` and `Rec` are introduced with the following reductions[c]:

$$\begin{aligned}
&\mathtt{if}(\mathtt{true}, a, b) \rightsquigarrow_1 a\\
&\mathtt{if}(\mathtt{false}, a, b) \rightsquigarrow_1 b\\
&\mathtt{Rec}(a, f, \mathtt{0}) \rightsquigarrow_1 a\\
&\mathtt{Rec}(a, f, \mathtt{suc}(n)) \rightsquigarrow_1 \mathtt{apply}(\mathtt{apply}(f, (\mathtt{Rec}(a, f, n))), n)
\end{aligned}$$

where $\rightsquigarrow_1$ is the notion of one-step reduction. The function symbol `Rec` is called a *recursor*, and realizes primitive recursion over the type `Int`.

Note that the second argument of `Rec` ($f$ in the above reduction) has a function type, therefore it is usually in the form $\lambda m.a$. For example, the addition of two integers can be implemented by the term

$$\lambda x.\lambda y.\ \mathtt{Rec}(y,\ \lambda m.\lambda n.\mathtt{suc}(m),\ x)$$

and the second argument of `Rec` has the type `Int` $\to$ `Int` $\to$ `Int`. Therefore, if we take Nakano's side-condition, the second argument of `Rec` cannot contain a subterm like $!u.C[m]$ and we cannot escape from a nested recursive call with a value which depends on $m$.

### *2.3. Definition of* $\mathsf{T}_{c/t}$ *and* $\mathsf{T}^{-}_{c/t}$

Our two calculi $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$ are extensions of Gödel system $\mathsf{T}$. Since the `catch`-construct is interpreted by the sum type in Sato's style, we introduce the type constructor $+$ for the sum type as well as term-constructs for the catch/throw mechanism. We also introduce the type constructor `List`(_) for lists to describe useful examples. We use the notation of $\mathsf{NJ}_{c/t}$ for the catch/throw mechanism, but side-conditions are different than $\mathsf{L}_{c/t}$ and $\mathsf{NJ}_{c/t}$.

We first define data types and types. We assume there are finitely many atomic types including $\bot$, `Bool` and `Int`. We use the metavariable $A$ for atomic types other than `Bool`, and `Int`.

**Definition 2.1 (Data Type $D$ and Type $T$)**

$$\begin{aligned}
D &::= \mathtt{Bool} \mid \mathtt{Int} \mid D \times D \mid D + D \mid \mathtt{List}(D)\\
T &::= A \mid D \mid T \times T \mid T + T \mid T \to T
\end{aligned}$$

For example, $\mathtt{Int} + \mathtt{Bool} \times \mathtt{List}(\mathtt{Int})$ is a data type (and a type), while $\mathtt{Int} \to \mathtt{Int}$ is a type but not a data type. We excluded atomic types other than `Bool` and `Int` from the definition of data types, in particular, $\bot$ is not a data type.

We use $S, T$, and $D, E$ for metavariables for types and data types, respectively. We assume there are infinitely many individual variables $x, y, z$ and tag variables $u, v$. We then define preterms as follows:

[c] We use slightly different notations than the original system.

The rest of this paper is organized as follows: Section 2 briefly describes Nakano's $\mathsf{L}_{c/t}$, Sato's $\mathsf{NJ}_{c/t}$, and Gödel's $\mathsf{T}$, and then defines our calculi $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$. Section 3 shows the expressiveness of $\mathsf{T}_{c/t}$ by examples. Section 4 shows several interesting properties for $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$ including the strong normalization property. Section 5 gives a confluent subcalculus of our system. Finally, concluding remarks are given in Section 6.

## 2. Definition of the Calculi $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$

In this section, a brief overview of the calculi $\mathsf{L}_{c/t}$, $\mathsf{NJ}_{c/t}$ and $\mathsf{T}$ is given, and then $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$ are defined.

### *2.1. Existing Catch/Throw Calculi*

Nakano's $\mathsf{L}_{c/t}$ gives a logical interpretation of the catch/throw mechanism. Under the Curry-Howard isomorphism, the resuling system can be viewed as a type system. He noticed that, in order to formulate a logically consistent theory, the tag variables should have static (or lexical) extent, while tags have dynamic extent in real programming languages, such as Common Lisp. Consider the following program in Common Lisp:

```
(defun infinite (x)
  (apply (catch 'tag (apply x (list nil)))
         (list x)))

(infinite '(lambda (y) (throw 'tag 'infinite)))
```

The execution of the last line does not terminate, though this program uses no recursion nor looping constructs. Apparently, the problem is in the dynamic nature of the scope of the tag; at the beginning of the execution, the tag `tag` was not in the scope of the `catch`-construct in the function `infinite`, but it was later captured by `catch` as the evaluation proceeds.

This is parallel to the situation that, for the binding mechanism of individual variables, static binding is much more suitable than dynamic binding to be interpreted in a logically well-founded theory. De Groote made a similar analysis to the exception mechanism in the programming languages ML, and proposed a calculus where the name of exceptions has the static scoping rule[2].

Nakano's system $\mathsf{L}_{c/t}$ satisifes the properties of subject reduction and strong normalization. It also has an elegant correspondence to the the intuitionistic propositional logic with multiple consequences.

Recently, Sato re-organized Nakano's calculus so that he unified the two logical connectives $\lhd$ and $\vee$ into one, and two abstraction mechanisms of tag variables `catch` and $\kappa$ into one `?`, thus simplified the system. Sato also considered a classical system which has no side-condition on the $\lambda$-rule. His two systems are called $\mathsf{NJ}_{c/t}$ (the intuitionistic version) and $\mathsf{NK}_{c/t}$ (the classical version). Since $\mathsf{NJ}_{c/t}$ is simpler than $\mathsf{L}_{c/t}$, and it can interpret $\mathsf{L}_{c/t}$, we take $\mathsf{NJ}_{c/t}$ as the starting point of our study.

As we mentioned in the introduction, both of $\mathsf{L}_{c/t}$ and $\mathsf{NJ}_{c/t}$ have an expressiveness problem. The classical calculus $\mathsf{NK}_{c/t}$ does not have the problem, however, we want to confine ourselves

side-condition in $\mathsf{L}_{c/t}$. However, this solution has a severe demerit. Each time the function `new-sqrt-sum2` is called, we must apply not only a list but also a tag variable to it, which is apparently not natural and inefficient.

The same problem occurs if we use the catch/throw mechanism with the map function:

```
(catch 'err
    (mapcar '(lambda (x)
                (if (minusp x) (throw 'err x) (sqrt x)))
            '(1 -1 3)))
```

Again we cannot type this example in $\mathsf{L}_{c/t}$ without inserting abstraction/application of tag variables.

Since higher-order'ness is the heart of functional programming, we believe that this defect is serious. We must therefore seek another formulation of calculi where higher-order functions with the catch/throw mechanism can be written naturally and efficiently. Note that, if we simply drop the side-condition, the program above can be well-typed, but we obtain a classical calculus[7, 13]. In such a system, the type corresponding to the law of the excluded-middle is inhabited by a closed normal term, but we can get no computational information from the term unlike the `call/cc`-based classical calculus. From the viewpoint of program extraction from proofs, we must exclude such a term. We shall touch upon this point in Section 6.

In summary, we want to look for an intuitionistic calculus for the catch/throw mechanism, which is natural, and expressive enough to handle run-time errors such as the above example.

Sato[13] proposed another calculus $\mathsf{NJ}_{c/t}$. The main motivation of $\mathsf{NJ}_{c/t}$ is, to eliminate the extra constructs in $\mathsf{L}_{c/t}$, namely, the tag abstraction (`tag-abst` in the above example) and the logical connective $\lhd$. Note that the logical connective $\lhd$ is the type of the tag abstraction, and it it logically equivalent to $\vee$ in $\mathsf{L}_{c/t}$. By unifying $\lhd$ and $\vee$, and also `catch` and `tag-abst`, Sato obtained a simpler, but equally expressive system as $\mathsf{L}_{c/t}$. The side-condition of the implication introduction rule remains essentially the same[b] as $\mathsf{L}_{c/t}$. Since Sato's formulation is easier to treat, we use Sato's constructs `?` and `!` in this paper.

In this paper, we propose a new formulation of the catch/throw mechanism and propose two calculi $\mathsf{T}_{c/t}$ and $\mathsf{T}^{-}_{c/t}$ as extensions of Gödel's system $\mathsf{T}$. Since the system $\mathsf{T}$ has the integer type and the primitive recursion operator over it, we can write a recursive function and use the catch/throw mechanism in a meaningful way. Our calculi have no side-conditions on the implication introduction rule, and we can use the catch/throw mechanism in a recursive program which may exit from a nested recursive call at a time. Instead of having side-conditions, we restrict the type of a thrown term to a non-functional data type such as `Bool` and `Int` $\times$ `Int`.

We shall show an example in $\mathsf{T}_{c/t}$ which handles run-time errors in a meaningful way. We also show that, $\mathsf{T}_{c/t}$ has nice theoretical properties such as the subject reduction property and the strong normalization property. Compared with the proofs of these properties for $\mathsf{L}_{c/t}$ and $\mathsf{NJ}_{c/t}$, the proof for $\mathsf{T}_{c/t}$ is much simpler. The calculus $\mathsf{T}^{-}_{c/t}$ is a slightly weaker calculus than $\mathsf{T}_{c/t}$ in that the body of the `catch`-expression also has a non-functional data type. $\mathsf{T}^{-}_{c/t}$ enjoys two more desirable properties, namely, the normal form property and the disjunction property, the latter of which demonstrates the truly constructive nature of our calculus. We believe that among the existing calculi and our calculi, $\mathsf{T}^{-}_{c/t}$ best models the actual computation with the catch/throw mechanism.

[b] In [13], the side-condition in $\mathsf{NJ}_{c/t}$ is the same as the Nakano's first calculus; there should be no free tag variable in the body of the $\lambda$-term. However, the side-condition can be relaxed in exactly the same way as $\mathsf{L}_{c/t}$. The author and Sato proved in [5] the strong normalization property for $\mathsf{NJ}_{c/t}$ with the relaxed condition.

subject reduction and strong normalization[10]. If we add a primitive recursion operator on the list type to $\mathsf{L}_{c/t}$, we can type the following program in Common Lisp:

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

(defun multiply2 (xl)
    (if (null xl) 1
       (if (zerop (car xl)) (throw 'zero 0)
          (* (car xl) (multiply2 (cdr xl))))))
```

This program calculates, given a list of integers, the multiplication of all the elements. If the list contains `0`, then it immediately exits from the recursive call, so it is slightly more efficient than a usual implementation without the catch/throw mechanism.

However, the side-condition prohibits us to use the catch/throw mechanism in a more realistic way, for instance, handling run-time errors. Let us consider the following example in Common Lisp:

```
(defun sqrt-sum (xl)
    (catch 'err (sqrt-sum2 xl)))

(defun sqrt-sum2 (xl)
    (if (null xl) 0
       (if (minusp (car xl)) (throw 'err (car xl))
          (+ (sqrt (car xl)) (sqrt-sum2 (cdr xl))))))
```

This program calculates, given a list of integers, the sum of square roots of its elements. It uses the catch/throw mechanism to detect a run-time error when the list contains a negative integer. In this case, we usually want to know not only an error, but also the integer which caused the error. Even if we extend $\mathsf{L}_{c/t}$ with a recursion operator, the above program cannot be well-typed since the body of the `throw`-expression depends on the variable `xl` which is bound by the $\lambda$-construct. In order to overcome this problem, Nakano invented the mechanism of abstraction/application of tag variables. The above program can be rewritten using the mechanism[a] as follows:

```
(defun new-sqrt-sum (xl)
    (catch 'err (tag-apply (new-sqrt-sum2 xl) 'err))

(defun new-sqrt-sum2 (xl)
  (tag-abst u
    (if (null xl) 0
       (if (minusp (car xl)) (throw u (car xl))
          (+ (sqrt (car xl))
             (tag-apply (new-sqrt-sum2 (cdr xl)) u))))))
```

In this example, `tag-apply` and `tag-abst` represent the application and the abstraction of a tag variable so that the tag variable `u` is no longer free in `new-sqrt-sum2`. Since the function `new-sqrt-sum2` has no free tag variables, it is well-typed with the existence of the

[a] We use different notations than $\mathsf{L}_{c/t}$ in this paper.

# A New Formulation of the Catch/Throw Mechanism

Yukiyoshi Kameyama
*Department of Information Science, Kyoto University*
*Kyoto, 606-01, JAPAN*
E-mail: kameyama@kuis.kyoto-u.ac.jp

ABSTRACT

The catch/throw mechanism in Common Lisp gives a simple control structure for non-local exits. Nakano[7, 9] and Sato[13] proposed intuitionistic calculi with inference rules which give logical interpretations of the catch/throw-constructs. Although the calculi are theoretically well-founded, we cannot use the catch/throw mechanism for handling run-time errors in a meaningful way, because of the side-condition of the implication-introduction rule (the formulation rule of the $\lambda$-abstract). This deficiency is critical if we use higher-order functions with the catch/throw mechanism. In this paper, we propose a new formulation of catch/throw calculi, which has no side-condition on the implication-introduction rule. By restricting the types of thrown terms to data types (non-functional types) instead, we obtain a strongly normalizing calculus for the catch/throw mechanism where we can write higher-order functions which handles run-time errors.

## 1. Introduction

Recently, control structures such as continuation have been intensively studied in the community of functional programming. The catch/throw mechanism in Common Lisp[14] gives a simple control structure which implements non-local exits. The following example shows how to use the catch/throw mechanism in Common Lisp:

```
(catch 'u (+ 1 3 -3 5)) ⇝ 6
(catch 'u (+ 1 3 (throw 'u -3) 5)) ⇝ -3
```

This mechanism is quite useful if we want to escape from nested function calls when an exceptional situation occurs (such as a run-time error).

Nakano[7] proposed a constructive logical system with inference rules which give logical interpretations of the catch/throw mechanism. He regarded the tag $u$ as a kind of variables (called a tag variable), and the `catch`-expression (`catch` $u$ $a$) as a binding mechanism of this tag-variable $u$. A program (closed term) must not have free variables, thus every `throw`-construct must be in the scope of some `catch`-construct which has the same tag. He noticed that, if there is no side-condition on the implication introduction rule (the formulation rule of the $\lambda$-abstract), the system becomes classical, while putting an appropriate side-condition on the rule results in an intuitionistic calculus. Later he relaxed the condition so that, the body of $\lambda$-terms may contain free tag variables, but the thrown terms must not depend on variables which are bound by some $\lambda$-construct. For instance, the term $\lambda x.$(`throw` $u$ `pair`$(y, z)$) has a free tag variable $u$, but it is well-typed since the thrown term `pair`$(y, z)$ does not depend on the variable $x$ which is bound by the $\lambda$-construct. On the other hand, the term $\lambda x.$(`throw` $u$ `pair`$(x, z)$) is not well-typed, since the thrown term `pair`$(x, z)$ depends on the variable $x$. The dependency on variables is defined in a slightly complex manner due to the existence of the `case`-construct. For the precise definition of the condition, refer to [10] or [5].

Nakano's system $\mathsf{L}_{c/t}$ is constructed with this side-condition, and has a nice correspondence to the intuitionistic propositional calculus. Also it enjoys several desirable properties such as