A Constructive Formalization of the Catch and Throw Mechanism

Hiroshi Nakano
Faculty of Science and Technology
Ryukoku University
Seta, Otsu, 520-21, Japan
nakano@rins.ryukoku.ac.jp

Abstract

The catch/throw mechanism is a programming construct for non-local exit. In the practical programming, this mechanism plays an important role when programmers handle exceptional situations. In this paper we give a constructive formalization which captures the mechanism in the proofs-as-programs notion. We introduce a modified version of LJ equipped with inference rules corresponding to the operations of catch and throw. Then we show that we can actually extract programs which make use of the catch/throw mechanism from proofs under a certain realizability interpretation. Although the catch/throw mechanism provides only a restricted access to the current continuation, the formulation remains constructive in contrast to the works due to Griffin and Murthy on more powerful facilities such as call/cc (call-with-current-continuation) of Scheme.

1 Introduction

The catch/throw mechanism provides a facility for non-local exit. We can find examples of the mechanism in some practical programming languages such as C-language [4] and Common Lisp [9]. This mechanism plays an important role when programmers handle exceptional situations. Suppose, for example, we have to construct a program P with a specification represented by a sequent $\Gamma \rightarrow C \vee E$ of LJ combining three subprograms $Q : \Gamma \rightarrow A \vee E$, $R : \Gamma A \rightarrow B \vee E$ and $S : \Gamma B \rightarrow C \vee E$, where C is the normal output for the input Γ, and E is an error signal which denotes that there is something wrong in the input. The specifications of Q, R and S say that such errors may be detected in the execution of these subprograms. Under this situation, the construction of P in LJ would be as Figure 1, where applications of structural rules are omitted. The constructed program P would work as follows. The program P first calls the subprogram Q and gets its return value, then checks the value whether it denotes an error or not. If not, P calls the subprogram R with that value and gets its return value. P again checks the value and calls S if it does not denote an error. Eventually, P returns the value returned by S. If P detects an error in the values returned by Q or S, it immediately returns a value denoting an error. We can find an inefficiency that whenever P gets values from the subprograms it must check them whether they denote a error or not. This is often found in the practical programming without the catch/throw mechanism. If the mechanism is available, programmers can concentrate on the main stream of programming as if there must not happen any exceptional situation since error signals are passed through bypasses provided by the mechanism. Following the proofs-as-programs notion in the opposite direction, we can find that the problem comes from the restriction of LJ that only one formula is admissible to the right hand side of sequents. If the restriction is dropped like LK, we can construct P' of a specification $\Gamma \rightarrow C\,E$ from subprograms $Q' : \Gamma \rightarrow A\,E$, $R' : \Gamma A \rightarrow B\,E$ and $S' : \Gamma B \rightarrow C\,E$ as follows:

$$\dfrac{\overset{\vdots\, Q'}{\Gamma \rightarrow A\,E} \qquad \dfrac{\overset{\vdots\, R'}{\Gamma A \rightarrow B\,E} \quad \overset{\vdots\, S'}{\Gamma B \rightarrow C\,E}}{\Gamma A \rightarrow C\,E}\,(cut)}{\Gamma \rightarrow C\,E}\,(cut),$$

where structural rules are omitted again. The proof is much simpler than the previous one and easy to develop. The point is that exceptional conclusions are admitted beside the main conclusion and we can proceed the proof construction as if they do not exist. It reflects the programmer's reasoning behind the catch/throw mechanism. Of course, we must justify

$$\dfrac{\begin{matrix}\vdots\, Q\\ \Gamma\rightarrow A\vee E\end{matrix}\qquad \dfrac{\dfrac{\begin{matrix}\vdots\, R\\ \Gamma A\rightarrow B\vee E\end{matrix}\qquad \dfrac{\begin{matrix}\vdots\, S\\ \Gamma B\rightarrow C\vee E\end{matrix}\qquad \dfrac{\dfrac{}{E\rightarrow E}\,(init)}{E\rightarrow C\vee E}\,(\rightarrow\vee)}{\Gamma B\vee E\rightarrow C\vee E}\,(\vee\rightarrow)}{\Gamma A\rightarrow C\vee E}\,(cut)\qquad \dfrac{\dfrac{}{E\rightarrow E}\,(init)}{E\rightarrow C\vee E}\,(\rightarrow\vee)}{\Gamma A\vee E\rightarrow C\vee E}\,(\vee\rightarrow)}{\Gamma\rightarrow C\vee E}\,(cut),$$

Figure 1: Exception handling without catch/throw.

$$\dfrac{}{A\rightarrow A;}\,(init)\qquad \dfrac{\Gamma_1\rightarrow A;\Delta_1\quad \Gamma_2 A\rightarrow C;\Delta_2}{\Gamma_1\Gamma_2\rightarrow C;\Delta_1\Delta_2}\,(cut)\qquad \dfrac{\Gamma\rightarrow E;\Delta}{\Gamma\rightarrow A;E\Delta}\,(throw)\qquad \dfrac{\Gamma\rightarrow A;A\Delta}{\Gamma\rightarrow A;\Delta}\,(catch)$$

$$\dfrac{\Gamma\rightarrow A;\Delta_1 BC\Delta_2}{\Gamma\rightarrow A;\Delta_1 CB\Delta_2}\,(\rightarrow x)\qquad \dfrac{\Gamma\rightarrow A;EE\Delta}{\Gamma\rightarrow A;E\Delta}\,(\rightarrow c)\qquad \dfrac{\Gamma\rightarrow A;\Delta}{\Gamma\rightarrow A;E\Delta}\,(\rightarrow w)$$

$$\dfrac{\Gamma_1 AB\Gamma_2\rightarrow C;\Delta}{\Gamma_1 BA\Gamma_2\rightarrow C;\Delta}\,(x\rightarrow)\qquad \dfrac{\Gamma_1 AA\Gamma_2\rightarrow C;\Delta}{\Gamma_1 A\Gamma_2\rightarrow C;\Delta}\,(c\rightarrow)\qquad \dfrac{\Gamma\rightarrow C;\Delta}{\Gamma A\rightarrow C;\Delta}\,(w\rightarrow)$$

$$\dfrac{\Gamma_1\rightarrow A;\Delta_1\quad \Gamma_2\rightarrow B;\Delta_2}{\Gamma_1\Gamma_2\rightarrow A\wedge B;\Delta_1\Delta_2}\,(\rightarrow\wedge)\qquad \dfrac{\Gamma A\rightarrow C;\Delta}{\Gamma A\wedge B\rightarrow C;\Delta}\,(\wedge_1\rightarrow)\qquad \dfrac{\Gamma B\rightarrow C;\Delta}{\Gamma A\wedge B\rightarrow C;\Delta}\,(\wedge_2\rightarrow)$$

$$\dfrac{\Gamma\rightarrow A;\Delta}{\Gamma\rightarrow A\vee B;\Delta}\,(\rightarrow\vee_1)\qquad \dfrac{\Gamma\rightarrow B;\Delta}{\Gamma\rightarrow A\vee B;\Delta}\,(\rightarrow\vee_2)\qquad \dfrac{\Gamma_1 A\rightarrow C;\Delta_1\quad \Gamma_2 B\rightarrow C;\Delta_2}{\Gamma_1\Gamma_2 A\vee B\rightarrow C;\Delta_1\Delta_2}\,(\vee\rightarrow)$$

$$\dfrac{\Gamma A\rightarrow B;}{\Gamma\rightarrow A\supset B;}\,(\rightarrow\supset)\qquad \dfrac{\Gamma_1\rightarrow A;\Delta_1\quad \Gamma_2 B\rightarrow C;\Delta_2}{\Gamma_1\Gamma_2 A\supset B\rightarrow C;\Delta_1\Delta_2}\,(\supset\rightarrow)$$

$$\dfrac{\Gamma\rightarrow A;E\Delta}{\Gamma\rightarrow A\lhd E;\Delta}\,(\rightarrow\lhd)\qquad \dfrac{\Gamma_1 A\rightarrow B;\Delta_1\quad \Gamma_2 E\rightarrow C;\Delta_2}{\Gamma_1\Gamma_2 A\lhd E\rightarrow C;B\Delta_1\Delta_2}\,(\lhd\rightarrow)$$

Figure 2: Inference rules.

such a logic constructively so that correct programs can be extracted from the proofs. In the following sections, we present such an attempt to extract a logical structure from the programmer's reasoning concerning exception handling by the catch/throw mechanism.

2 The formal system

We introduce a sequent calculus to capture the catch/throw mechanism. But it is not important whether the formalization is done by the sequent calculus style or not. We adopt such a style here just for convenience of readers who are familiar with the correspondence between classical logic and control operators such as *call/cc*, for it becomes clear when we regard our formal system as an intermediate system between LJ and LK. We are also able to consider a natural deduction style formalization, and it would be more suitable for natural development of programs.

We restrict the logic to a minimal propositional logic in this paper since the restricted system is enough for us to observe the essence how the mechanism works. Note that it can be easily extended to the predicate logic.

Formulas Formulas of our system consist of atomic formulas, conjunctions ($A\wedge B$), disjunctions ($A\vee B$), implications ($A\supset B$) and exceptions ($A\lhd B$). The last one is introduced to handle the catch/throw mechanism and represents another kind of disjunction. We give a precise meaning to the connective $\lhd$ by a realizability interpretation later.

Sequents Sequents of the system are of the form:

$$A_1 \ldots A_m \rightarrow C ; E_1 \ldots E_n,$$

where m and n can be 0. They look like the ones of LK rather than LJ ignoring the semicolons ";" between C and $E_1 \ldots E_n$. Actually, their purely logical meaning is the same as LK. In this sense, the semicolons are negligible. But they play a certain role for the constructive meaning of the sequents. The formula C between the arrow and the semicolon represents the main conclusion and $E_1 \ldots E_n$ represent exceptional conclusions. The sequent $A_1 \ldots A_m \rightarrow C ; E_1 \ldots E_n$ represents a program such that when we execute the program supplying values that satisfies the specification $A_1 \ldots A_m$ for the corresponding free variables of the program, it normally returns a value which meets C, otherwise the program exits with a value which meets one of $E_1 \ldots E_n$. Note that if $n = 0$, i.e. $E_1 \ldots E_n$ is empty, the meaning of the sequent is just the same as the usual interpretation of the sequent of LJ.

Inference rules The inference rules of the formal system are listed in Figure 2. If we ignore semicolons in the sequents, they are almost the same as the ones of LK except for the logical rules concerning the new connective $\lhd$. The only essential difference is that there must be exactly one formula on the right hand side of the sequent when we apply the right-implication rule $(\rightarrow\supset)$. Roughly, our system can be regarded as the propositional fragment of LK with this restriction on the right-implication rule. But it should be also noted that every right logical rule introduces a logical connective into the formula between the arrow and the semicolon, i.e. the main conclusion.

3 A realizability interpretation

Before discussing the syntactical properties of the formal system, we give an example of realizability interpretation of the system to explain how the catch/throw mechanism can be captured in it. The interpretation is defined by an abstract machine which manipulates expressions equipped with the catch/throw mechanism. We start with the definition of the expression syntax.

Constants and variables We first assume the following disjoint sets of individual constants, individual variables, tag constants and tag variables are given, where the set of tag constants is a representation of the set of natural numbers. We use $\bar{n}$ to denote the tag constant which represents a natural number n, and $\hat{t}$ to denote the natural number represented by a tag constant t.

C_e A set of individual constants $c, d, \ldots$.

V_e A countably infinite set of individual variables $x, y, z, \ldots$.

C_t A set of tag constants $\bar{0}, \bar{1}, \bar{2}, \ldots$.

V_t A countably infinite set of tag variables $u, v, w, \ldots$.

Tags Tag constants and tag variables are called tags. We denote the set of tags by T. That is,

$$T \ ::= \ C_t \ | \ V_t.$$

We use $s, t, \ldots$ to denote tags.

Expressions Expressions E are defined as follows.

$$\begin{array}{rl} E ::= & C_e \ | \ V_e \ | \ \mathbf{let}\, V_e{=}E.E \ | \ EE \ | \ \lambda V_e.E \\ | & <E,E> \ | \ \mathbf{proj_1} E \ | \ \mathbf{proj_2} E \\ | & \mathbf{inj_1} E \ | \ \mathbf{inj_2} E \ | \ \mathbf{case} E\, V_e.E\, V_e.E \\ | & \mathbf{throw} T E \ | \ \mathbf{catch} V_t E \ | \ ET \ | \ \kappa V_t.E \end{array}$$

The expressions $\kappa V_t.E$ and ET are used for tag-abstraction and tag-instantiation, respectively. We use $e, f, \ldots$ to denote expressions. Free and bound occurrences of variables are defined in the standard manner. We regard a tag variable u as bound in $\mathbf{catch}\, u e$ and $\kappa u.e$. We use $e[M/x]$ to denote the expression obtained from an expression e by replacing each free occurrence of a variable x by an expression or a tag M.

An abstract stack machine We now define an abstract machine which evaluates expressions. The machine is designed only to illustrate how the catch/throw mechanism works. We leave other mechanisms required for the evaluation of expressions being abstract (cf. [5, 8]). It has a stack of expressions and a special symbol denoted by $*$ which may appear only at the top of the stack. The state of the machine is determined only by the state of this stack. We represent a state of the stack as follows.

$$\begin{array}{c} \text{bottom} \leftarrow \qquad \rightarrow \text{top} \\ [\, e_1,\, e_2,\, e_3,\, \ldots,\, e_n,\, M \,], \end{array}$$

where $e_1 \ldots e_n$ are expressions and M is an expression or the special symbol $*$.

Transition rules The abstract machine changes its state as follows:

$$
\begin{aligned}
[\vec{f}, \lambda x.e_2, e_1, *] &\Rightarrow [\vec{f}, e_2[e_1/x]] \\
[\vec{f}, \mathbf{let}\, x{=}e_1.e_2] &\Rightarrow [\vec{f}, \lambda x.e_2, e_1] \\
[f_1,\ldots,f_n,\ldots,\mathbf{throw}\,\bar{n}\,e] &\Rightarrow [f_1,\ldots,f_n,e] \\
[f_1,\ldots,f_n,\mathbf{catch}\,u\,e] &\Rightarrow [f_1,\ldots,f_n,e[\bar{n}/u]] \\
[\vec{f}, <e_1,e_2>] &\Rightarrow [\vec{f}, <e_1,e_2>, *] \\
[\vec{f}, \mathbf{proj_1} <e_1,e_2>] &\Rightarrow [\vec{f}, e_1] \\
[\vec{f}, \mathbf{proj_2} <e_1,e_2>] &\Rightarrow [\vec{f}, e_2] \\
[\vec{f}, \mathbf{inj_1}\, e] &\Rightarrow [\vec{f}, \mathbf{inj_1}\, e, *] \\
[\vec{f}, \mathbf{inj_2}\, e] &\Rightarrow [\vec{f}, \mathbf{inj_2}\, e, *] \\
[\vec{f}, \mathbf{case}(\mathbf{inj_1}\, e)x.e_1\, y.e_2] &\Rightarrow [\vec{f}, e_1[e/x]] \\
[\vec{f}, \mathbf{case}(\mathbf{inj_2}\, e)x.e_1\, y.e_2] &\Rightarrow [\vec{f}, e_2[e/y]] \\
[\vec{f}, \lambda x.e] &\Rightarrow [\vec{f}, \lambda x.e, *] \\
[\vec{f}, (\lambda x.e_1)e_2] &\Rightarrow [\vec{f}, e_1[e_2/x]] \\
[\vec{f}, \kappa u.e] &\Rightarrow [\vec{f}, \kappa u.e, *] \\
[\vec{f}, (\kappa u.e)t] &\Rightarrow [\vec{f}, e[t/u]] \\
[\vec{f}, c] &\Rightarrow [\vec{f}, c, *],
\end{aligned}
$$

where $\vec{f}$ denotes a sequence of expressions. The rule applicable for a state is uniquely determined by the object at the top of the stack. Let $\overset{*}{\Rightarrow}$ be the transitive reflexive closure of the relation $\Rightarrow$. If $[e_1] \overset{*}{\Rightarrow} [e_2,*]$, the expression e_2 is unique modulo alpha-conversion. This represents that e_2 is the head normal form of e_1. We denote it by $Eval(e_1) = e_2$. It should be noted that the machine has no evaluation strategy for the expressions except for let-expressions. For example, there is no rules to apply when an expression $((\lambda x.x)(\lambda x.x))c$ is the top object of the stack. To get c as the result of evaluation, the expression should be $\mathbf{let}\, y{=}(\lambda x.x)(\lambda x.x).\, yc$. The machine first evaluates e_1 of $\mathbf{let}\, x{=}e_1.e_2$ to get its head normal form, then proceed to the evaluation of e_2 with it. Only let-expressions determine the evaluation order. We must specify the evaluation order of expressions by let-expressions explicitly. Note also that let-expressions are the only expressions that push something to the stack. The expressions $f_1 \ldots f_n$ of the stack $[f_1,\ldots,f_n,e]$ must be of the form of lambda-abstraction, and the composition of $f_1 \ldots f_n$ represents the continuation after the evaluation of the expression e (cf. [2]). The catch/throw mechanism of the machine provides a restricted access to the continuation through tags. Although it does not provide a way to treat it as a first class citizen as in Scheme, it does not require any explicit copying of the control context.

Realizability interpretation of formulas We now define a realizability for formulas using the abstract machine. Let $\mathcal{A}$ be a mapping which assigns a subset of C_e to each atomic formula. Let e and A be an expression and a formula, respectively. We define the realizability relation denoted by $e\ \mathbf{r}\ A$ between expressions and formulas as follows.

1. $e\ \mathbf{r}\ A$ iff $e \in \mathcal{A}(A)$, if A is an atomic formula.
2. $e\ \mathbf{r}\ A_1 \wedge A_2$ iff $e = <e_1,e_2>$ for some e_1 and e_2 such that $e_1\ \mathbf{r}\ A_1$ and $e_2\ \mathbf{r}\ A_2$.
3. $e\ \mathbf{r}\ A_1 \vee A_2$ iff $e = \mathbf{inj}_i e'$ for some i ($i = 1,2$) and $e'\ \mathbf{r}\ A_i$.
4. $e\ \mathbf{r}\ A_1 \supset A_2$ iff $e = \lambda x.e'$ and $Eval(e'[f/x])\ \mathbf{r}\ A_2$ for any expression f such that $f\ \mathbf{r}\ A_1$.
5. $e\ \mathbf{r}\ A_1 \triangleleft A_2$ iff $e = \kappa u.e'$ and one of the following holds.
   (a) $Eval(e'[\bar{n}/u])\ \mathbf{r}\ A_1$ for any natural number n.
   (b) $[f_1, \ldots, f_n, \ldots, f_{n+m}, e'[\bar{n}/u]] \overset{*}{\Rightarrow} [f_1,\ldots,f_n,e'',*]$ and $e''\ \mathbf{r}\ A_2$ for any natural numbers n and m.

If the relation holds between an expression and a formula, we say that the expression realizes the formula, and the expression is a realizer of the formula.

Realizability interpretation of sequents We now consider a triple which consists of a sequence of individual variables of length m, an expression and a sequence of tag variables of length n for each sequent $A_1 \ldots A_m \rightarrow C; B_1 \ldots B_n$. We assume the free individual and tag variables of the expression are included in the two sequences. We define a realizability relation $\mathbf{r}$ between such triples and sequents as follows. The relation

$$< x_1 \ldots x_m, e, u_1 \ldots u_n > \ \mathbf{r}\ \ A_1 \ldots A_m \rightarrow C; B_1 \ldots B_n$$

holds if and only if one of the following two holds for any natural number l, expressions $f_1,\ldots,f_l,g_1,\ldots,g_m$ such that $g_1\ \mathbf{r}\ A_1$, ..., $g_m\ \mathbf{r}\ A_m$, and tag constants $t_1,\ldots,t_n$ such that $\hat{t_0} \leq l$, ..., $\hat{t_n} \leq l$.

1. $[f_1,\ldots,f_l,e[\vec{g}/\vec{x},\vec{t}/\vec{u}] \overset{*}{\Rightarrow} [f_1,\ldots,f_l,e',*]$ and $e'\ \mathbf{r}\ C$.
2. $[f_1,\ldots,f_l,e[\vec{g}/\vec{x},\vec{t}/\vec{u}] \overset{*}{\Rightarrow} [f_1,\ldots,f_{\hat{t_i}},e',*]$ and $e'\ \mathbf{r}\ B_i$ for some i ($0 \leq i \leq n$),

where the expression $e[\vec{g}/\vec{x}, \vec{t}/\vec{u}]$ stands for $e[g_1/x_1,\ldots,g_m/x_m,t_1/u_1,\ldots,t_n/u_n]$. If the relation holds between a triple and a sequent, we say that the triple realizes the sequent, and the triple is a realizer of the sequent. Note that this definition of realizability is essentially the same as the standard realizability definition of LJ in the case that the formula C does not include any occurrence of $\lhd$ and $B_1 \ldots B_m$ is empty. It should also be noted that the logical connective $\lhd$ corresponds to the semicolon of a sequent as so does $\supset$ to the arrow.

Soundness of the formal system The following soundness theorem assures us that we can regard the proofs of the formal system as programs which satisfy the specification defined by the realizability interpretation of the conclusion.

Theorem 1 *If a sequent is derivable in the formal system, we can construct a realizer of the sequent.*

Realizer construction The theorem is proved by induction on the structure of the derivation. For example, if $(catch)$ is the last inference rule of the derivation of a sequent $\Gamma \to A;\Delta$, we have a realizer of $\Gamma \to A;A\Delta$ by the induction hypothesis. Let $< \vec{x}, e, u\vec{v} >$ be the realizer. We can take $< \vec{x}, \mathbf{catch}\, ue, \vec{v} >$ as the realizer for the conclusion. We only summarize how to construct the realizer according to the derivation to Figure 3 here, and omit the details of the proof. Note that the let-expressions included in the constructed realizers direct the evaluation of the expressions in call-by-value fashion.

4 The formal system as a logic

In this section we discuss the basic property of the formal system considering it as a logic. First we consider the new logical connective $\lhd$.

Definition 2 *We use $\tilde{A}$ to denote the formula obtained from a formula A by replacing every occurrence of the logical connective $\lhd$ by $\vee$. If $\tilde{A} = \tilde{B}$, then we denote it by $A \simeq B$. If $\Gamma = A_0 \ldots A_n$, $\Delta = B_0 \ldots B_n$ and $A_i \simeq B_i$ for any i $(0 \le i \le n)$, then we denote it by $\Gamma \simeq \Delta$.*

Lemma 3 *If $A \simeq A'$, then $A \to A'$; is a drivable sequent of the system.*

Proof. Straightforward induction on the structure of the formula A. The basic idea comes from the following two derivations.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\ \text{ind. hyp.}\\ A\to A';\end{array}}{A\to A'\vee B';}(\to\vee_1)\quad \dfrac{\begin{array}{c}\vdots\ \text{ind. hyp.}\\ B\to B';\end{array}}{B\to A'\vee B';}(\to\vee_2)}{A\lhd B\to A'\vee B';A'\vee B'}(\lhd\to)}{A\lhd B\to A'\vee B';}(catch)$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\ \text{ind. hyp.}\\ A\to A';\end{array}\quad \dfrac{\begin{array}{c}\vdots\ \text{ind. hyp.}\\ B\to B';\end{array}}{B\to A';B'}(throw)}{A\vee B\to A';B'}(\vee\to)}{A\vee B\to A'\lhd B';}(\to\lhd)$$

Theorem 4 *If $\Gamma \to A;\Delta$ is a derivable sequent of the system, and if $\Gamma \simeq \Gamma'$, $A \simeq A'$ and $\Delta \simeq \Delta'$, then $\Gamma' \to A';\Delta'$ is also derivable.*

Proof. Induction on the structure of the derivation of the sequent $\Gamma \to A;\Delta$. Apply Lemma 3 in the case that the last rule is $(init)$. □

The theorem says that the logical meaning of $A\lhd B$ is essentially the same as $A\vee B$. The difference between them consists only in their implementation.

If we identify $A\lhd B$ with $A\vee B$, the formal system can be regarded as a variant of the propositional fragment of LJ.

Theorem 5 *A sequent $A_1 \ldots A_m \to C;$ is derivable in the system if and only if $\tilde{A}_1 \ldots \tilde{A}_m \to \tilde{C}$ is derivable in (the propositional fragment of) LJ.*

Proof. The *if* part is trivial because the propositional fragment of LJ can be regarded as a subsystem of ours. For the *only if* part, prove the following conjecture by induction on the structure of the derivation: *If $A_1 \ldots A_m \to C;E_1 \ldots E_n$ is derivable, then $\tilde{A}_1 \ldots \tilde{A}_m \to \tilde{C} \vee \tilde{E}_1 \vee \ldots \vee \tilde{E}_n$ is derivable in (the propositional fragment of) LJ.* The theorem is a corollary of the conjecture. □

As a corollary of the theorem, we get the disjunction property of the system.

Corollary 6 *If $\to A\vee B;$ is a derivable sequent, then we can derive $\to A;$ or $\to B;$.*

There must be no formulas on the right hand side of the semicolon when we apply the right-implication rule $(\to\supset)$. This restriction is the most significant difference from LK, and it keeps the system constructive. LK with this kind of restrictions is known as a variant of LJ, which is essentially equivalent to LJ (cf.[6, 10, 11]). The same restriction is also required for $(\to\forall)$-rule in the case of predicate calculus. If we dropped the restriction, the system would become a classical one.

The cut-elimination theorem holds for the formal system.

Theorem 7 *If a sequent is derivable, then we can derive it without (cut)-rule.*

$$\begin{array}{c} \dfrac{}{A\to A;}\ (init) \\ < x,x,> \end{array} \qquad \begin{array}{c} < \vec{x},e_1,\vec{u} > \quad < \vec{y}z,e_2,\vec{v} > \\ \dfrac{\Gamma_1\to A;\Delta_1 \quad \Gamma_2 A\to C;\Delta_2}{\Gamma_1\Gamma_2\to C;\Delta_1\Delta_2}\ (cut) \\ < \vec{x}\vec{y},\mathbf{let}\,z=e_1.e_2,\vec{u}\vec{v} > \end{array} \qquad \begin{array}{c} < \vec{x},e,\vec{v} > \\ \dfrac{\Gamma\to E;\Delta}{\Gamma\to A;E\Delta}\ (throw) \\ < \vec{x},\mathbf{let}\,y=e.\mathbf{throw}\,uy,u\vec{v} > \end{array} \qquad \begin{array}{c} < \vec{x},e,u\vec{v} > \\ \dfrac{\Gamma\to A;A\Delta}{\Gamma\to A;\Delta}\ (catch) \\ < \vec{x},\mathbf{catch}\,ue,\vec{v} > \end{array}$$

$$\begin{array}{c} < \vec{x},e,\vec{u}v_1v_2\vec{w} > \\ \dfrac{\Gamma\to A;\Delta_1 BC\Delta_2}{\Gamma\to A;\Delta_1 CB\Delta_2}\ (\to x) \\ < \vec{x},e,\vec{u}v_2v_1\vec{w} > \end{array} \qquad \begin{array}{c} < \vec{x},e,\vec{v} > \\ \dfrac{\Gamma\to A;\Delta}{\Gamma\to A;E\Delta}\ (\to w) \\ < \vec{x},e,u\vec{v} > \end{array} \qquad \begin{array}{c} < \vec{x},e,u_1u_2\vec{v} > \\ \dfrac{\Gamma\to A;EE\Delta}{\Gamma\to A;E\Delta}\ (\to c) \\ < \vec{x},e[u/u_1,u/u_2],u\vec{v} > \end{array}$$

$$\begin{array}{c} < \vec{x}y_1y_2\vec{z},e,\vec{u} > \\ \dfrac{\Gamma_1 AB\Gamma_2\to C;\Delta}{\Gamma_1 BA\Gamma_2\to C;\Delta}\ (x\to) \\ < \vec{x}y_2y_1\vec{z},e,\vec{u} > \end{array} \qquad \begin{array}{c} < \vec{x},e,\vec{u} > \\ \dfrac{\Gamma\to C;\Delta}{\Gamma A\to C;\Delta}\ (w\to) \\ < \vec{x}y,e,\vec{u} > \end{array} \qquad \begin{array}{c} < \vec{x}y_1y_2\vec{z},e,\vec{u} > \\ \dfrac{\Gamma_1 AA\Gamma_2\to C;\Delta}{\Gamma_1 A\Gamma_2\to C;\Delta}\ (c\to) \\ < \vec{x}y\vec{z},e[y/y_1,y/y_2],\vec{u} > \end{array}$$

$$\begin{array}{c} < \vec{x},e_1,\vec{u} > \quad < \vec{y},e_2,\vec{v} > \\ \dfrac{\Gamma_1\to A;\Delta_1 \quad \Gamma_2\to B;\Delta_2}{\Gamma_1\Gamma_2\to A\wedge B;\Delta_1\Delta_2}\ (\to\wedge) \\ < \vec{x}\vec{y},\mathbf{let}\,z_1=e_1.\mathbf{let}\,z_2=e_2.<z_1,z_2>,\vec{u}\vec{v} > \end{array} \qquad \begin{array}{c} < \vec{x}z,e,\vec{u} > \\ \dfrac{\Gamma A\to C;\Delta}{\Gamma A\wedge B\to C;\Delta}\ (\wedge_1\to) \\ < \vec{x}y,\mathbf{let}\,z=\mathbf{proj_1}\,y.e,\vec{u} > \end{array} \qquad \begin{array}{c} < \vec{x}z,e,\vec{u} > \\ \dfrac{\Gamma B\to C;\Delta}{\Gamma A\wedge B\to C;\Delta}\ (\wedge_2\to) \\ < \vec{x}y,\mathbf{let}\,z=\mathbf{proj_2}\,y.e,\vec{u} > \end{array}$$

$$\begin{array}{c} < \vec{x},e,\vec{u} > \\ \dfrac{\Gamma\to A;\Delta}{\Gamma\to A\vee B;\Delta}\ (\to\vee_1) \\ < \vec{x},\mathbf{let}\,y=e.\mathbf{inj_1}\,y,\vec{u} > \end{array} \qquad \begin{array}{c} < \vec{x},e,\vec{u} > \\ \dfrac{\Gamma\to B;\Delta}{\Gamma\to A\vee B;\Delta}\ (\to\vee_2) \\ < \vec{x},\mathbf{let}\,y=e.\mathbf{inj_2}\,y,\vec{u} > \end{array} \qquad \begin{array}{c} < \vec{x}z_1,e_1,\vec{u} > \quad < \vec{y}z_2,e_2,\vec{v} > \\ \dfrac{\Gamma_1 A\to C;\Delta_1 \quad \Gamma_2 B\to C;\Delta_2}{\Gamma_1\Gamma_2 A\vee B\to C;\Delta_1\Delta_2}\ (\vee\to) \\ < \vec{x}\vec{y}z,\mathbf{case}\,z\,z_1.e_1\,z_2.e_2,\vec{u}\vec{v} > \end{array}$$

$$\begin{array}{c} < \vec{x}y,e,> \\ \dfrac{\Gamma A\to B;}{\Gamma\to A\supset B;}\ (\to\supset) \\ < \vec{x},\lambda y.e,> \end{array} \qquad \begin{array}{c} < \vec{x},e_1,\vec{u} > \quad < \vec{y}z',e_2,\vec{v} > \\ \dfrac{\Gamma_1\to A;\Delta_1 \quad \Gamma_2 B\to C;\Delta_2}{\Gamma_1\Gamma_2 A\supset B\to C;\Delta_1\Delta_2}\ (\supset\to) \\ < \vec{x}\vec{y}z,\mathbf{let}\,z''=e_1.\mathbf{let}\,z'=zz''.e_2,\vec{u}\vec{v} > \end{array}$$

$$\begin{array}{c} < \vec{x},e,u\vec{v} > \\ \dfrac{\Gamma\to A;E\Delta}{\Gamma\to A\triangleleft E;\Delta}\ (\to\triangleleft) \\ < \vec{x},\kappa u.e,\vec{v} > \end{array} \qquad \begin{array}{c} < \vec{x}z_1,e_1,\vec{v} > \quad < \vec{y}z_2,e_2,\vec{w} > \\ \dfrac{\Gamma_1 A\to B;\Delta_1 \quad \Gamma_2 E\to C;\Delta_2}{\Gamma_1\Gamma_2 A\triangleleft E\to C;B\Delta_1\Delta_2}\ (\triangleleft\to) \\ < \vec{x}\vec{y}z,\mathbf{let}\,z_2=(\mathbf{catch}\,w'\,\mathbf{let}\,z_1=zw'.\mathbf{let}\,z'=e_1.\mathbf{throw}\,uz').e_2,u\vec{v}\vec{w} > \end{array}$$

Figure 3: Realizer construction.

We omit the proof for lack of space. The proof becomes more complicated than the case of LJ/LK because it required a special condition to apply $(\to\supset)$-rule and the new connective $\triangleleft$ has been introduced. Unfortunately, the computational behavior of the catch/throw mechanism is not captured by the cut-elimination process. Consider the following simple example.

$$\begin{array}{c} \dfrac{\dfrac{\dfrac{}{A\to A;}\ (init)}{A\to A;A}\ (throw)}{A\to A;}\ (catch) \\ < x,\mathbf{catch}\,u\,(\mathbf{let}\,y=x.\mathbf{throw}\,u\,y),> \end{array}$$

It is a cut-free derivation, but the realizer includes a catch-throw pair. Another kind of proof translation should be considered to explain the mechanism.

5 Some comments on the formalization

As mentioned above, sequents of our formal system can be regarded as the ones of LK ignoring semicolons. Let us compare the system with LK. First, there must be at least one formula on the right hand side of sequents. But this comes from that the system is a minimal logic and is not an important difference for us.

The three right-structural rules of LK are divided into five rules, and there is no right-exchange rules over the semicolon. The rules (*catch*) and $(\to c)$ correspond to the right-contraction rule of LK. The former introduces a catch-expression. The latter means a sharing of one tag variable by two throw-expressions, i.e. multiple throw-expressions would be caught by one catch-expression afterwards. The right-weakening rule of LK is divided into (*throw*) and $(\to w)$. The former corresponds to a throw-expression. The latter means an introduction of a redundant tag variable. We note that the rule $(\to w)$ is a derived rule of other rules.

$$< \vec{x}, e, \vec{v} >$$
$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\to A;\Delta}{\Gamma\to E;A\Delta}\,(throw)}{\Gamma\to A;EA\Delta}\,(throw)}{\Gamma\to A;AE\Delta}\,(\to x)}{\Gamma\to A;E\Delta}\,(catch)$$
$$< \vec{x}, \mathbf{catch}\, w\ (\mathbf{let}\, y{=}(\mathbf{let}\, z{=}e.\mathbf{throw}\, w\, z).\mathbf{throw}\, u\, y), u\vec{v} >$$

But we adopt $(\to w)$ as a primitive rule because the realizer given above introduces a redundant throw-expression, i.e. a throw-expression never visited.

We do not have a right-exchange rule over the semicolon, but it is also a derived rule as follows.

$$< \vec{x}, e, u\vec{w} >$$
$$\dfrac{\dfrac{\dfrac{\Gamma\to A;E\Delta}{\Gamma\to E;AE\Delta}\,(throw)}{\Gamma\to E;EA\Delta}\,(\to x)}{\Gamma\to E;A\Delta}\,(catch)$$
$$< \vec{x}, \mathbf{catch}\, u\, (\mathbf{let}\, y{=}e.\mathbf{throw}\, v\, y), v\vec{w} >$$

In contrast to $(\to w)$, we leave it as a derived rule because there is no primitive realizer construction corresponding to the rule.

We have a restriction on $(\to\supset)$-rule to keep the system constructive as mentioned before. If we dropped the restriction, the following anomaly would occur. Consider the following derivation of $A\vee(A\supset B)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A\to A;}\,(init)}{A\to A\vee(A\supset B);}\,(\to\vee_1)}{A\to B;A\vee(A\supset B)}\,(throw)}{\to A\supset B;A\vee(A\supset B)}\,(\to\supset)}{\to A\vee(A\supset B);A\vee(A\supset B)}\,(\to\vee_2)}{\to A\vee(A\supset B);}\,(catch)$$

The realizer would be $\mathbf{catch}\, u\ (\mathbf{let}\, y{=}\lambda x.(\mathbf{let}\, z{=}(\mathbf{let}\ x'{=}x.\mathbf{inj_1}\, x').\mathbf{throw}\, u\, z).\mathbf{inj_2}\, y)$. Removing redundant let-expressions, we get $\mathbf{catch}\, u\ \mathbf{inj_2}(\lambda x.\mathbf{throw}\, u\ (\mathbf{inj_1}\, x))$. The evaluation process of the expression would be as follows.

$$\begin{aligned}
&[\lambda x_1.e_1,\ldots,\lambda x_n.e_n, \mathbf{catch}\, u\, \mathbf{inj_2}(\lambda x.\mathbf{throw}\, u(\mathbf{inj_1}\, x))]\\
&\Rightarrow [\lambda x_1.e_1,\ldots,\lambda x_n.e_n, \mathbf{inj_2}(\lambda x.\mathbf{throw}\, \bar{n}(\mathbf{inj_1}\, x)), *]\\
&\Rightarrow [\lambda x_1.e_1,\ldots,\lambda x_{n-1}.e_{n-1},\\
&\qquad\qquad e_n[\mathbf{inj_2}(\lambda x.\mathbf{throw}\, \bar{n}(\mathbf{inj_1}\, x))/x_n]]
\end{aligned}$$

Note that the tag constant $\bar{n}$ in the last is meaningless because the the corresponding control context has been lost. From a computational point of view, this problem can be solved by introducing more powerful facilities for non-local exit such as *call/cc* of Scheme. But it affects the realizability interpretation of formulas. For example, although the realizers of disjunctions still have a certain constructive meaning yet, they do not always contain the information which formula of $A\vee B$ is realized by them (cf. [3, 7]). It should be noted that the system without the restriction becomes a classical one, and we do not have the disjunction property anymore.

The restriction on $(\to\supset)$-rule leads us to introduce the new connective $\triangleleft$. We can not construct any function that may throw something to the outside of the function without the new connective. Such a function is represented by a formula of the form $A\supset(B\triangleleft E)$, and is called with a value for A and a tag for E. Normally, the function returns a value of B, otherwise it throws a value of E to the given tag. Concerning the left logical rule of $\triangleleft$, there is another possibility of formulation as follows:

$$< \vec{x}z, e, \vec{v} >$$
$$\dfrac{\Gamma A\to C;\Delta}{\Gamma A\triangleleft E\to C;E\Delta}\,(\triangleleft\to)$$
$$< \vec{x}y, \mathbf{let}\, z{=}y\, u.e, u\vec{v} >$$

From the viewpoint of realizer construction, it corresponds to a more primitive construction than the original one listed in Figure 2, and is easy to understand for programmers. But it requires (*cut*)-rule to be equivalent to the original. So the cut-elimination theorem does not hold if we substitute it for the original. This is the only reason why we adopt the original formulation.

6 Conclusion

We have presented a formal system which captures the catch/throw mechanism in the proofs-as-programs notion. Although the system is just a variant of LJ, it admits extra conclusions beside the main one. So we can construct proofs which handle the exceptional situations efficiently. We showed that we can actually extract programs that make use of the catch/throw mechanism from such proofs. Our work can be regarded as a higher order extension of the work concerning *goto* statements in Hoare logic (cf. [1]), whose main idea is also the existence of extra post-conditions.

Although, from a computational point of view, the catch/throw mechanism provides only a restricted access to the current continuation, we can extract correct programs without any restriction required for the case of more powerful facilities such as *call/cc* (cf. [7]). And more important, there exists a characteristic way of programmer's reasoning concerning exception handling behind the mechanism. We wonder whether there also exists such a reasoning corresponding to the use of *call/cc* and its variants beyond the catch/throw.

Acknowledgements

The author wishes to thank Susumu Hayashi for a great number of helpful suggestions and invaluable encouragement regarding this work. He pointed out the connection between this work and the literature concerning variants of LJ and the treatment of *goto* in Hoare logic. Thanks are also due to the referees for helpful comments on the detailed abstract of this paper.

References

[1] S. Alagić and M. A. Arbib, *The Design of Well-Structured and Correct Programs*, Springer-Verlag, 1978.

[2] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba, A syntactic theory of sequential control, *Theoretical Computer Science*, Vol. 52(3), pp. 205-237, 1987.

[3] T. G. Griffin, A formulae-as-types notion of control, *Conf. Rec. ACM Symp. on Principles of Programming Languages*, pp. 47-58, 1990.

[4] B. W. Kernighan and D. M. Ritchie, *The C programming language (2nd ed.)*, Prentice Hall, 1989.

[5] P. J. Landin, The mechanical evaluation of expressions, *Computer Journal*, Vol. 6(4), 1964.

[6] S. Maehara, Eine Darstellung intuitionistic Logik und der Klassishen, *Nagoya Math. Journal*, Vol. 7, pp. 45-64, 1954.

[7] C. R. Murthy, An evaluation semantics for classical proofs, *Proc. IEEE Symp. on Logic in Computer Science*, pp. 96-107, 1991.

[8] G. D. Plotkin, Call-by-name, call-by-value and the λ-calculus, *Theoretical Computer Science*, Vol. 1, pp. 125-159, 1975.

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

[10] K. Schütte, *Vollständige Systeme Modaler und Intuitionistischer Logik*, Springer-Verlag, 1968.

[11] G. Takeuti, *Proof theory (2nd ed.)*, North Holland, 1987.