Science of Computer Programming 21 (1993) 165–190
Elsevier

Programs with continuations and linear logic

Shin-ya Nishizaki
Research Institute for Mathematical Sciences, Kyoto University, Kyoto 606-01, Japan

Communicated by M. Hagiya
Revised October 1992

Abstract

Nishizaki, S., Programs with continuations and linear logic, Science of Computer Programming 21 (1993) 165–190.

The purpose of this paper is to investigate the programs with continuations in the framework of classical linear logic which can also be regarded as improvement of traditional classical logic. First, simply typed λ-calculus $\lambda_c^{\rightarrow}$ with continuation primitives is introduced. Second, a translation from $\lambda_c^{\rightarrow}$ to linear logic is given. Last, a correspondence between them is presented.

1. Introduction

The notion of *continuation*, which means the *rest of computation*, was originally discovered by van Wijngaarden [25], Mazurkiewicz [17], and Morris [18], independently. Strachey and Wadsworth used this notion for the denotational semantics of jump-statements [24].

Recently, this notion has become popular not only among theoreticians but also among working programmers in Lisp's dialect *Scheme* [21]. In Scheme, one can flexibly use continuations like integers and lists, i.e. *continuation as first-class object*. Primitive *call-with-current-continuation* (abbreviated *call/cc*) enables one to use this facility. A simple example of *call/cc* is as follows:

Correspondence to: S. Nishizaki, Department of Information Science, Faculty of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, 113, Japan. E-mail: sin@is.s.u-tokyo.ac.jp.

```
(call/cc
  (lambda (exit)
    (for-each
      (lambda (x) (if (negative? x) (exit x)))
      '(54 0 37 -3 245 19))
    #t))
```

$\xrightarrow{\text{The result}}$ -3.

(This example is cited from the *revised*^{3} *report* [21].)

Primitive *call/cc* supports advanced control structures such as coroutine [14] and backtrack [13] in addition to global exit such as the above example.

The *continuation-passing-style transformation* (abbreviated *CPS-transformation*) is one of the main topics in the research area on continuation. Roughly speaking, this transformation added one more argument to each function and the additional argument explicitly simulates passing of a continuation in calling of functions. Originally, the aim of this translation is to obtain a program that is independent of evaluation strategies from a program that is dependent on a certain evaluation strategy. This translation may be understood from various aspects: embedding from the λ-calculus with continuation primitives to the usual λ-calculus [5], program transformation [26] and compilation technique [2,7,22].

Recently, Filinski [4] studied the duality between value and continuation in the framework of category theory. Under this duality, we can identify

a function from *values* of type A to *values* of type B

with

a function from *continuations* of type B to *continuations* of type A.

Intuitively, we may more symmetrically say that

value = *result* of computation,

continuation = *rest* of computation.

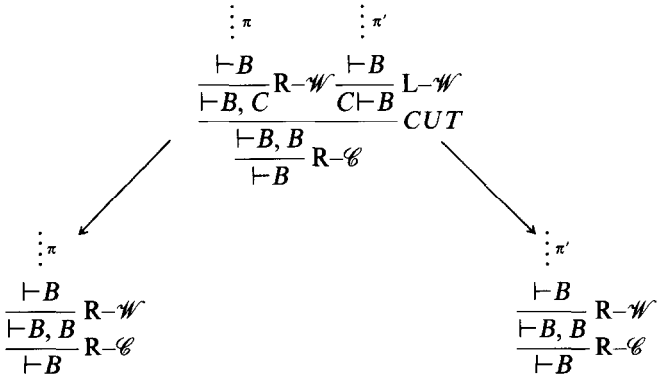
Program languages with continuation primitives are very powerful in the aspect of control, however, they have an unpleasant feature: their computation is *not confluent* unlike usual λ-calculi. In other words, results of computation depend on the evaluation strategy, e.g. call-by-value, call-by-name, etc. The following is a typical example:

Nondeterminism of computation

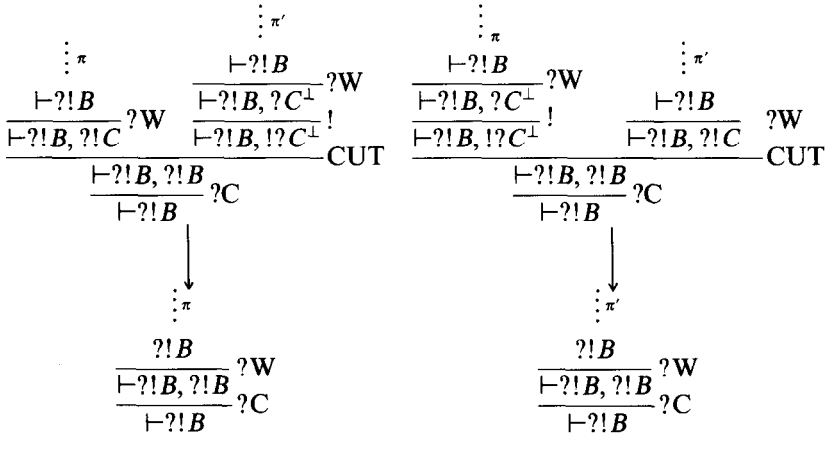
```
                (call/cc(lambda (exit)
                  ((lambda(x) 1) (exit 2))))
                      ↙                  ↘
                    1                      2
(call-by-name)                         (call-by-value)
```

We leave programming and turn to logic. We can find the similar phenomenon in traditional classical logic, which is "nondeterminism" of classical logic [10, p. 151].

Nondeterminism of proof normalization



It is pointed out in [10] that such a nondeterminism of classical logic is caused by the *unrestricted* use of the weakening rule and the contraction rule to both sides of the sequents. In intuitionistic logic, this problem is avoided by *restricting* the application of the right weakening rule and right contraction rule. In Girard's (classical) linear logic [8,9], this problem is solved by the more careful use of these structural rules. The "deterministic" proof normalization is one of the advantages of linear logic. The following are proofs in classical linear logic, corresponding to the above one:



We find that some information about determinism is added to the proof during the translation from traditional classical logic to classical linear logic.

The two systems discussed above, λ-calculus and logic, are connected by a Curry–Howard isomorphism. This isomorphism had been restricted to one between intuitionistic logic and simply typed λ-calculus until Griffin [12] extended this to the

correspondence between *classical* logic and simply typed λ-calculus with *continuation* primitives.

$$\begin{array}{ccc} \lambda^{\rightarrow}_{\mathscr{C},\mathscr{A}} & \xrightarrow{\text{CPS}} & \lambda^{\rightarrow} \\ \text{extended Curry–Howard isomorphism} \Big\| & & \Big\| \text{traditional Curry–Howard isomorphism} \\ \text{classical logic} & \xrightarrow{\neg\neg} & \text{intuitionistic logic} \end{array}$$

where $\lambda^{\rightarrow}_{\mathscr{C},\mathscr{A}}$ is the simple typed λ-calculus with the continuation primitives $\mathscr{C}$ and $\mathscr{A}$ [5] and $\lambda^{\rightarrow}$ the simple typed λ-calculus. $\mathscr{C}$ and $\mathscr{A}$ correspond to the $\neg\neg$-elimination rule and the $\bot$-elimination rule, respectively.

To be exact, we should say that

CPS-transformation
$= \neg\neg$-translation + Friedman's A-translation [11,16],

which is shown by Murthy [19, 20]. He applied this correspondence to the program extraction from classical proofs.

The following is an example of the classical proof corresponding to the programs which implements *call/cc* using primitives $\mathscr{A}$ and $\mathscr{C}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x:A]\ [k:\neg A]}{\dfrac{B}{\bot}\bot\mathscr{I}}\rightarrow\mathscr{E}}{A\rightarrow B}\rightarrow\mathscr{I}(x) \qquad [f:(A\rightarrow B)\rightarrow A]}{A}\rightarrow\mathscr{E}}{\dfrac{\bot}{\neg\neg A}\neg\neg\mathscr{E}}\rightarrow\mathscr{I}(k)}{((A\rightarrow B)\rightarrow A)\rightarrow A}\rightarrow\mathscr{I}(f)$$

The conclusion $((A \rightarrow B) \rightarrow A) \rightarrow A$ is famous as *Peirce's law* which cannot be proved in intuitionistic logic, but made only by implication.

We finish this section with the motivation of this paper.

Motivation

As we have seen, it is possible to avoid the nondeterminism from classical logic in the following ways:

- *intuitionistic logic*: the right weakening rule and right contraction rule are *inhibited*;
- *linear logic*: the weakening rule and contraction rule are treated *more carefully*.

The first solution has been selected in previous approaches: $\neg\neg$-translation (= CPS-transformation) eliminates the nondeterminism by embedding from classical logic to intuitionistic logic. We here try another solution, i.e. *linear logic*.

2. Programming language $\lambda_c^{\rightarrow}$

First of all, we have to introduce the programming language we will investigate. We define simply typed λ-calculus $\lambda_c^{\rightarrow}$ with primitives for continuation handling. First, we define types. Second, terms and, last, a computation rule in this section.

Definition 1 (*Types*). *AtomicType* is a countable set of given *atomic types*. *Types* of $\lambda_c^{\rightarrow}$ are inductively defined as follows:

$$A ::= \alpha \mid A \rightarrow B,$$

where $\alpha, \beta, \ldots$ are metavariables over atomic types and $A, B, \ldots$ over types. *Type* is a set of types defined as above.

Note that types are similar to the usual simply typed λ-calculi: $\lambda_c^{\rightarrow}$ does not have the bottom type $\perp$ which designates the top-level type (cf. IS_t [12]).

Definition 2 (*Terms*). *Var* is a countable set of given *variables*. *Terms* of $\lambda_c^{\rightarrow}$ are defined inductively as follows:

$$M ::= x \mid (\lambda x{:}A.M) \mid (M\ N) \mid \mathscr{K}_{A,B} \mid \mathscr{A}_{A,B}\{M\},$$

where $x, y, \ldots$ are metavariables over variables and $M, N, \ldots$ over terms. *Term* is a set of terms defined as above.

$(\lambda x{:}A.N)$, $(M\ N)$, $\mathscr{K}_{A,B}$, and $\mathscr{A}_{A,B}\{M\}$ are called a *λ-abstraction*, a *function application*, a *call/cc-constant* and an *abort-term*, respectively. We used Felleisen's notation of continuation primitives, where $\mathscr{K}_{A,B}$ equals *call/cc* in the programming language Scheme.

We notice that "$\mathscr{A}_{A,B}\{\ldots\}$" is not defined as a function constant like $\mathscr{K}_{A,B}$, but as a *special form* in the terminology of Common Lisp [23] because of the evaluation order defined later. We use braces { } in abort terms in order to emphasize this point.

We introduce several notions in the following before defining of typing of $\lambda_c^{\rightarrow}$.

Definition 3 (*Type environment*). A *type environment* is a function whose domain is a finite set of variables $\{x_1, \ldots, x_n\}$ and whose codomain is the set of types *Type*. $\Gamma, \ldots, \Gamma_1, \Gamma_2, \ldots$ are used as metavariables over type environments. *TypeEnv* is the set of type environments. A type environment $\{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}$ is abbreviated to $\{[x_1{:}\alpha_1], \ldots, [x_n{:}\alpha_n]\}$.

Definition 4 (*Restriction of a type environment*). Suppose that Γ is a type environment over $\{x_1, \ldots, x_n\}$, A a type, x an arbitrary variable which satisfies that Γ maps x to A if $x \in \{x_1, \ldots, x_n\}$. The *restriction* of Γ by $[x{:}A]$, written $\Gamma\backslash[x{:}A]$, is a function

whose domain is $\{x_1, \ldots, x_n\} \backslash \{x\}$ and which maps variables to the same types as Γ does except for the variable x, i.e.

$$\Gamma \backslash [x:A] : Domain(\Gamma) \backslash \{x\} \to Var,$$
$$y \mapsto \Gamma(y) \quad (\text{where } y \neq x).$$

For example $\{[x:A][y:B][f:A \to B]\} \backslash [x:A] = \{[y:B][f:A \to B]\}$.

Definition 5 (*Merge of type environments*). Suppose that Γ_1 and Γ_2 are type environments with their domains $\{x_1, \ldots, x_m\}$ and $\{y_1, \ldots, y_n\}$, respectively, which satisfy $\Gamma_1(x) = \Gamma_2(x)$ for every variable x such that $x \in \{x_1, \ldots, x_m\} \cap \{y_1, \ldots, y_n\}$.

The *merge* of type environments Γ_1 and Γ_2, written $\Gamma_1 \cup \Gamma_2$, is a type environment whose domain is the union of the domains of Γ_1 and Γ_2 and which maps variables to the same types as Γ_1 and Γ_2, i.e.

$$\Gamma_1 \cup \Gamma_2 : \{x_1, \ldots, x_m\} \cup \{y_1, \ldots, y_n\} \to Var,$$
$$x(\in \{x_1, \ldots, x_m\}) \mapsto \Gamma_1(x),$$
$$y(\in \{y_1, \ldots, y_n\}) \mapsto \Gamma_2(y).$$

For example $\{[x:A][y:B]\} \cup \{[x:A][f:A \to B]\} = \{[x:A][y:B][f:A \to B]\}$ and the merge of $\{[x:A]\}$ and $\{[x:B]\}$ (where $A \neq B$) is impossible.

Definition 6 (*Type judgement, type inference rules and type derivation tree*). A *type judgement* $\Gamma \vdash M:A$ (where Γ is a type environment, M a term and A a type) is a relation among type environments, terms and types defined inductively by the following *type inference rules*. A tree with a type judgement as a root and inference rules as nodes and leaves is called a *type derivation tree*. The type inference rules of $\lambda_c^{\to}$ are as follows:

(1) $$\frac{}{\{[x:A]\} \vdash x:A}\ Var$$

(2) $$\frac{}{\vdash call/cc_{A,B}:((A \to B) \to A) \to A}\ Con_{\mathscr{A}}$$

(3) $$\frac{\Gamma \vdash M:B}{\Gamma \backslash [x:A] \vdash (\lambda x:A.M):A \to B}\ Lam$$

(4) $$\frac{\Gamma_1 \vdash M:A \to B \quad \Gamma_2 \vdash N:A}{\Gamma_1 \cup \Gamma_2 \vdash (M\ N):B}\ App$$

(5) $$\frac{\Gamma \vdash M:A}{\Gamma \vdash \mathscr{A}_{A,B}\{M\}:B}\ Special_{\mathscr{A}}$$

We have the following property on these notions.

Proposition 7 (Type environment and free variables). *Let Γ be a type environment, M a term and A a type. If $\Gamma \vdash M:A$ is satisfied, then Domain(Γ) equals the set of free variables occurring in term M*

(Domain(Γ) denotes the domain of Γ when we regard Γ as a function.)

The above result leads to the uniqueness of a type derivation tree as follows:

Proposition 8 (Uniqueness of a type derivation tree). *Suppose that Γ is a type environment, M a term and A a type. If there exists a type derivation tree whose root is $\Gamma \vdash M:A$, it is unique.*

The uniqueness of a type derivation tree is desirable because we define the translation from terms of $\lambda_c^{\rightarrow}$ to proofs of linear logic indirectly: the domain of the translation is *not* terms but type derivation trees.

Some people may think that the definition of typing is already completed. This is true in cases of the usual typed λ-calculi, but *not* true in this case: typings of the usual typed λ-calculi are defined inductively on the structure of terms, in other words, a type derivation tree is built up from *local* to *global*. On the other hand, typing of $\lambda_c^{\rightarrow}$ cannot be defined only in such a manner. The following additional condition, which has a global nature, is required:

Definition 9 (*Abort-typing condition*). Suppose that M is a term, A a type and Γ a type environment such that $\Gamma \vdash M:A$ is satisfied.

It is said that M satisfies the *abort-typing condition* if and only if every abort-term occurring in M has the body of type A. In other words, every abort-term in M occurs in the form of $\mathscr{A}_{A,\ldots}\{\ldots\}$.

Definition 10 (*Typing of $\lambda_c^{\rightarrow}$*). It is said that M is a term of type A in a type environment Γ if and only if M satisfies $\Gamma \vdash M:A$ and the abort-typing condition.

Next, the *call-by-value* computation of $\lambda_c^{\rightarrow}$ is presented in the style of Felleisen [5], i.e., given as rewriting rules. We first prepare a few notions and then introduce the computation.

Definition 11 (*Value*). A value of $\lambda_c^{\rightarrow}$, where $V, W, \ldots,$ are used as metavariables, is defined inductively as follows:

$$V ::= x \mid (\lambda x{:}A.M) \mid \mathscr{K}_{A,B}.$$

Informally, we can explain that a value is a term which cannot be computed any more. The notion of *evaluation context* defined next is the device for pointing the place which should be computed each time (the call-by-value computation is intended here).

Definition 12 (*Evaluation context*). An *evaluation context* is defined inductively as follows:

$$E[\ ] ::= [\ ] \mid (E[\ ]N). \mid (V\,E[\ ])$$

We use $E[\ ], \ldots$ as metavariables over evaluation contexts. $[\ ]$ is called a *hole.*

If $E[\ ]$ is an evaluation context, then $E[M]$ denotes the term that results from placing M in the hole of $E[\ ]$.

Definition 13 (*Computation of* $\lambda_c^{\rightarrow}$). The computation rule of $\lambda_c^{\rightarrow}$ is a relation between *Term* and *Term*, written $\rightarrow_{comp}$, such that

$$\rightarrow_{comp} = \rightarrow_{\beta_v} \cup \rightarrow_{\mathscr{K}} \cup \rightarrow_{\mathscr{A}}.$$

Each subrelation is defined as follows:

$$E[((\lambda x{:}A.M)\,V)] \rightarrow_{\beta_v} E[M[x := V]],$$

$$E[(\mathscr{K}_{A,B}\,V)] \rightarrow_{\mathscr{K}} E[(V(\lambda x{:}A.\mathscr{A}_{A',B}\{E[x]\}))]$$

where A' is a type of $E[\ldots]$,

$$E[\mathscr{A}_{A,B}\{M\}] \rightarrow_{\mathscr{A}} M.$$

At the end of this section, we return to the abort-typing condition. Some people may doubt its necessity. A term M which satisfies type judgement $\Gamma \vdash M{:}A$ and not the abort-typing condition causes a *dynamic type error*, i.e. a type error in the execution time.

For example, suppose that $\mathscr{A}_{A,B}\{M\}$ satisfies $\Gamma \vdash \mathscr{A}_{A,B}\{M\}{:}B$, but not the abort-typing condition. Then $\rightarrow_{\mathscr{A}}$ can be applied to this term.

$$\underbrace{\mathscr{A}_{A,B}\{M\}}_{\text{Type } A} \rightarrow_{\mathscr{A}} \underbrace{M}_{\text{Type } B}.$$

Such terms are excluded thanks to the abort-typing condition.

Proposition 14 (Preservation of the abort-typing-condition). *If M is a term of type A and N a term such that $M \rightarrow_{comp} N$, then N is also a term of type A. Therefore, the abort-typing condition is preserved during the computation.*

3. Translation from $\lambda_c^{\rightarrow}$ to linear logic

In this section, we introduce a translation from $\lambda_c^{\rightarrow}$ to linear logic, which consists of two translations: one is from types to propositions of linear logic, called τ, and the other is from terms to *proof nets* (a proof notation of linear logic), called T.

Translations from λ-calculi to linear logic have been introduced by several researchers. The simplest is the translation from Lafont's *linear* λ-calculus [15] to linear logic. We remind the definition of the linear λ-calculus before the discussion on the translation.

Definition 15 (*Lafont's linear λ-calculus (simply typed version)*). The difference with the usual simply typed λ-calculus is in the λ-abstraction.

$$\begin{aligned} M ::= \; & x^A \\ | \; & \lambda x^A.M \qquad \text{where } x \textit{ must} \text{ occur } \textit{once and only once} \text{ in } M \\ | \; & (M^{A\to B} N^A) \end{aligned}$$

For example, $\lambda f^{A\to B}.\lambda x^A.fx$ is correct, however, $\lambda x^A.\lambda y^B.x$ is *not* correct (because variable y used in λ-abstraction λy^B does not occur in its body x). The restriction above on the λ-abstraction may be intuitively explained by the fact that an input value is used *once and only once*, i.e. the duplication and/or deletion of an input value is prohibited. Such a restricted λ-abstraction will be called a *linear* λ-abstraction and such a function a *linear* function.

The translation from types of linear λ-calculus to propositions of linear logic is defined as follows:

Definition 16 (*Type-translation of linear λ-calculus*: τ_{LLC}).

$$\tau_{LLC}(A) = A \text{ (where } A \text{ is an atomic type)},$$

$$\tau_{LLC}(A \to B) = \tau_{LLC}(A) \multimap \tau_{LLC}(B).$$

The linear implication $\multimap$ is regarded as the type of linear functions.

Let us observe from another viewpoint. It holds that

$$\begin{aligned} A \multimap B &= A^\perp \parr B \\ &= A^\perp \parr B^{\perp\perp} \\ &= B^{\perp\perp} \parr A^\perp \\ &= B^\perp \multimap A^\perp. \end{aligned}$$

Therefore, we may say that a function of type $A \multimap B$ is one which uses an input value of type $B^\perp$ once and only once and then returns an output value of type $A^\perp$. The meaning of the linear negation $A^\perp$ in the programming language can be explained by Filinski's duality between values and continuations [6]: a function from values of type A to values of type B is regarded as one from *continuations* of type B to *continuations* of type A. Thus, the following holds:

value of type $A^\perp \equiv$ continuation of type A.

Girard gave the translation from polymorphic lambda calculus *system F* to linear logic. The translation of type is as follows:

Definition 17 (*Translation* $\tau_F(\cdot)$). Translation $\tau_F(\cdot)$ from types of system F to propositions of linear logic is defined inductively as follows:

$$\tau_F(\alpha) = \alpha \quad \text{(where } \alpha \text{ is atomic)},$$
$$\tau_F(A \to B) = !\tau_F(A) \multimap \tau_F(B),$$
$$\tau_F(\forall \alpha.A) = \wedge \alpha.\tau_F(A).$$

There is no restriction on λ-abstraction in this calculus. The function of type $!A \multimap B$ is one which uses an input value of type A *an arbitrary number of times* and returns an output value of type B. Let us observe this function type from the viewpoint of continuation in the same way as above.

$$\begin{aligned} !A \multimap B &= (!A)^{\perp} \parr B \\ &= (!A)^{\perp} \parr B^{\perp\perp} \\ &= B^{\perp\perp} \parr (!A)^{\perp} \\ &= B^{\perp} \multimap (!A)^{\perp}. \end{aligned}$$

The function of type $!A \multimap B$ is, therefore, one which uses a continuation of type B *once and only once* and returns a continuation of type $!A$.

We find that deletion and/or duplication of the given continuation is impossible. We try to add the modality ! to the underlined part of $\underline{B^{\perp}} \multimap (!A)^{\perp}$ for possibility of deletion and duplication. Then, we obtain

$$!B^{\perp} \multimap (!A)^{\perp}, \quad \text{i.e.} \quad !A \multimap ?B$$

Under this translation, the type of value returned by the function of type $!A \multimap ?B$ is $?B$. However, we cannot obtain any value (of type $?C$) from this value of type $?B$ and another function of type $!B \multimap ?C$: because $?B \vdash !B$ is not generally provable, especially when B is atomic. For this reason, we use $!A \multimap !B$ instead and add ! to the underlined part of $\underline{(!B)^{\perp}} \multimap (!A)^{\perp}$ ($= !A \multimap !B$) as before, finally obtaining:

$$!(!B)^{\perp} \multimap (!A)^{\perp}, \quad \text{i.e.} \quad !A \multimap ?!B.$$

In contrast to the former case, a function of type $!A \multimap ?!B$ can be applied to a value of type $?!A$ in linear logic.

The meaning of modalities occurring in $!A \multimap ?!B$ should be summarized here:

$\underline{!}A \multimap \underline{?}!B$

$\underline{!} \cdots$: the possibility of duplication and/or deletion of an input *value*;

$\cdots \underline{?} \cdots$: the possibility of duplication and/or deletion of an input *continuation*.

The formal definition of the transformation τ from types of $\lambda_c^{\rightarrow}$ to propositions of linear logic is as follows:

Definition 18 (*Translation* τ). The translation τ from types of $\lambda_c^{\rightarrow}$ to propositions of linear logic is defined as follows:

$$\tau(\alpha) = \alpha,$$

$$\tau(A \to B) = !\tau(A) \multimap ?!\tau(B)$$

$$= ?\tau(A)^{\perp} \parr ?!\tau(B).$$

We finish with the definition of two abbreviations as preparation for the next section.

Notation 19 ($?\tau(\Gamma)^{\perp}$, $C?s$). Suppose that Γ is type environment $\{[x_1 : A_1], \ldots, [x_n : A_n]\}$. Then $?\tau(\Gamma)^{\perp}$ is an abbreviation of the following sequence:

$$?\tau(A_1)^{\perp}, \ldots, ?\tau(A_n)^{\perp}.$$

The abbreviation

$$\frac{?\tau(\Gamma_1)^{\perp} \quad ?\tau(\Gamma_2)^{\perp}}{?\tau(\Gamma_1 \cup \Gamma_2)^{\perp}} C?s$$

denotes the several $C?$-links which connect the corresponding entries between Γ_1 and Γ_2. If there is no corresponding entry, there is no $C?$-link. For example, if $\Gamma_1 = \{[x : A], [y : B]\}$ and $\Gamma_2 = \{[y : B], [f : A \to B]\}$, then the above abbreviation means

$$?\tau(A)^{\perp} \quad \frac{?\tau(B)^{\perp} \quad ?\tau(B)^{\perp}}{?\tau(B)^{\perp}} C?s \quad ?\tau(A \to B)^{\perp}$$

We give an example in the case that two type environments have no corresponding entry. If $\Gamma_1 = \{[x : A], [y : B]\}$ and $\Gamma_2 = \{[z : B], [f : A \to B]\}$, then the above abbreviation merely means

$$?\tau(A)^{\perp} \; ?\tau(B)^{\perp} \; ?\tau(B)^{\perp} \; ?\tau(A \to B)^{\perp}.$$

A translation from terms of $\lambda_c^{\rightarrow}$ to proofs of linear logic is introduced. We do not define the translation on the terms themselves, but indirectly on the type derivation trees. A λ-term is *not* a tree in a strict sense because it has variable-bindings of λ-abstraction as implicit edges. A type derivation tree, on the other hand, is a tree since informations on variable occurrences are attached to each node. For this reason, the translation is defined more simply on type derivation trees than on terms. In the

following, translation T is defined with T_{value} and T_{term} as subroutines. First, these two subroutines are presented.

Definition 20 (*Translations T_{value} and T_{term}*). We define two translations T_{value} and T_{term} from type derivation trees to proof nets by mutual induction on the structure of the type derivation tree. These translations are defined on type derivation trees satisfying a weaker condition than the abort-typing condition, i.e. types of bodies of abort-terms are equal to each other but may not be equal to the type of the term derived by the type derivation tree: these translations are defined inductively, therefore, type derivation trees passed as arguments in each induction step are subtrees of some type derivation tree satisfying the abort-typing condition. We call this type which is equal to every abort-term's body, the *top-level type.*

Translation T_{value} is defined satisfying the following condition:

Suppose that Π is a type derivation tree with root judgement $\Gamma \vdash V : A$.

If A is atomic, then the result $T_{\text{value}}(\Pi)$ is a proof net with $?\tau(\Gamma)^{\perp}\ \tau(A)$ as its conclusion. Otherwise, i.e. if A is a function type $B \to C$ and V does not include any abort-term, then $T_{\text{value}}(\Pi)$ has as conclusion

$$?\tau(\Gamma)^{\perp}\ ?\tau(B)^{\perp}\ ?!\tau(C),$$

if V includes some abort-term, $T_{\text{value}}(\Pi)$ has as conclusion

$$?\tau(\Gamma)^{\perp}\ ?\tau(B)^{\perp}\ ?!\tau(C)\ ?!\tau(O),$$

where O is a top-level type.

Similarly, translation T_{term} satisfies a condition of the same kind as above.

Suppose that Π is a type derivation tree with root judgement $\Gamma \vdash M : A$. The proof net $T_{\text{term}}(\Pi)$ has as conclusion $?\tau(\Gamma)^{\perp}\ ?!\tau(A)$ if M does not include any abort-term, else $?\tau(\Gamma)^{\perp}\ ?!\tau(A)\ ?!\tau(O)$, where O is the top-level type.

We call the part $?\tau(\Gamma)^{\perp}$ a *variable door*, $?!\tau(A)$ an *output door* and $?!\tau(O)$ an *unbinding door.*

These properties are straightforwardly checked in the following definition.

Each case of T_{value} is as follows:

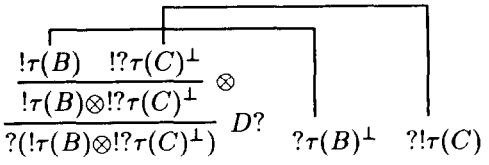
- *Variables.* If a type derivation tree Π is

$$\begin{array}{l} \vdots\,\Pi \\ [x : A] \vdash x : A \end{array}$$

and A atomic, then $T_{\text{value}}(\Pi)$ is

$$\dfrac{\overline{\tau(A)^{\perp}}}{?\tau(A)^{\perp}}\,D?\quad \tau(A)$$

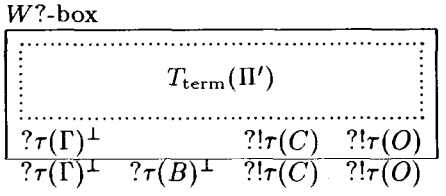
otherwise, i.e. if A is a function type $B \to C$, then $T_{\text{value}}(\Pi)$ is



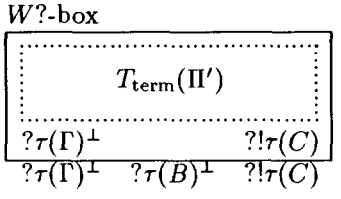
- *λ-abstraction.* If a type derivation tree Π is

$$\dfrac{\begin{array}{c}\vdots\Pi' \\ \Gamma \vdash M : C\end{array}}{\Gamma \vdash (\lambda x : B.M) : B \to C}\, Lam,$$

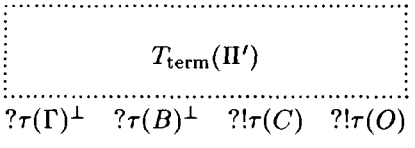
where x does not freely occur in M (i.e. Γ is not defined on x), then $T_{\text{value}}(\Pi)$ is



If $T_{\text{value}}(\Pi')$ has no unbinding door $?!\tau(O)$, then $T_{\text{value}}(\Pi')$ has no unbinding door either



If x occurs freely in M, then $T_{\text{value}}(\Pi)$ is $T_{\text{value}}(\Pi')$ itself

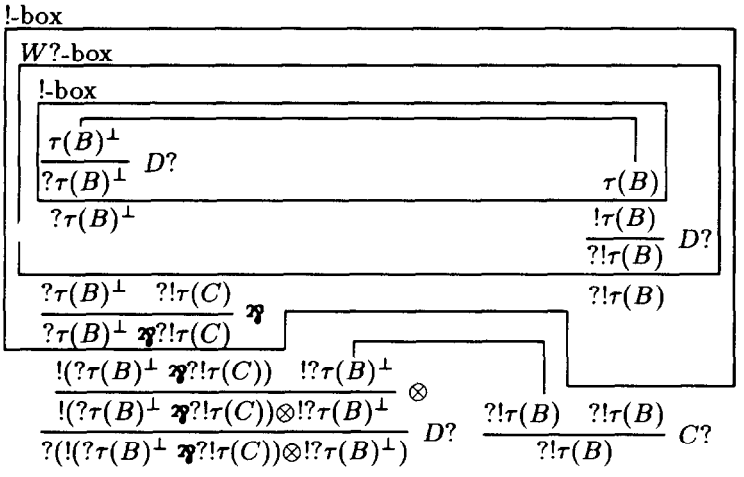


If $T_{\text{value}}(\Pi')$ has no unbinding door $?!\tau(O)$, then $T_{\text{value}}(\Pi)$ has no unbinding door, in the same way as the above case.

- *Constant call/cc.* If a type derivation tree Π is

$$\dfrac{}{\vdash \mathscr{K}_{B,C} : ((B \to C) \to B) \to B}\, Con_{\mathscr{K}},$$

then $T_{\text{value}}(\Pi)$ is

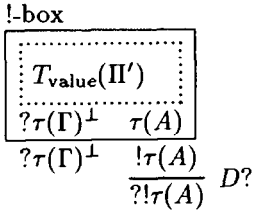


Each case of T_{term} is as follows:

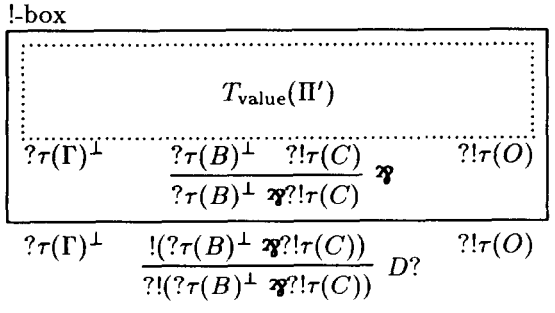
- *Value of atomic type.* If a type derivation tree Π is

$$\begin{array}{l}\vdots\Pi'\\ \Gamma \vdash V : A,\end{array}$$

V value, and A atomic, then $T_{\text{term}}(\Pi)$ is



- *Value of function type.* If a type derivation tree Π is the same as above but A is a function type, then $T_{\text{term}}(\Pi)$ is

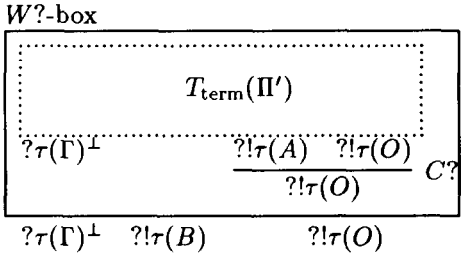


If $T_{\text{value}}(\Pi')$ has no unbinding door $?!\tau(O)$, then $T_{\text{term}}(\Pi)$ has no unbinding door, in the same way as the other cases, i.e. λ-abstraction.

- *Abort-term.* If a type derivation tree Π is

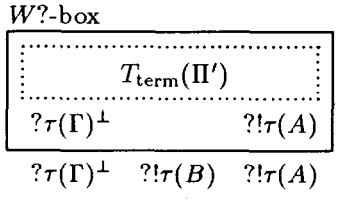
$$\dfrac{\begin{array}{c}\vdots\,\Pi' \\ \Gamma \vdash M : A\end{array}}{\Gamma \vdash \mathscr{A}_{A,B}\{M\} : B}\ Special_{\mathscr{A}}$$

and if $T_{\text{term}}(\Pi)$ has an unbinding door, then $T_{\text{term}}(\Pi)$ is



Note that type A is equivalent to the top-level type O because of the input condition of translation T_{term}.

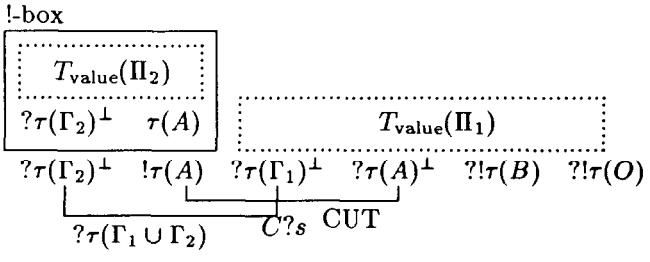
If $T_{\text{term}}(\Pi)$ has no unbinding door, then $T_{\text{term}}(\Pi)$ is



- *Application.* If a type derivation tree Π is

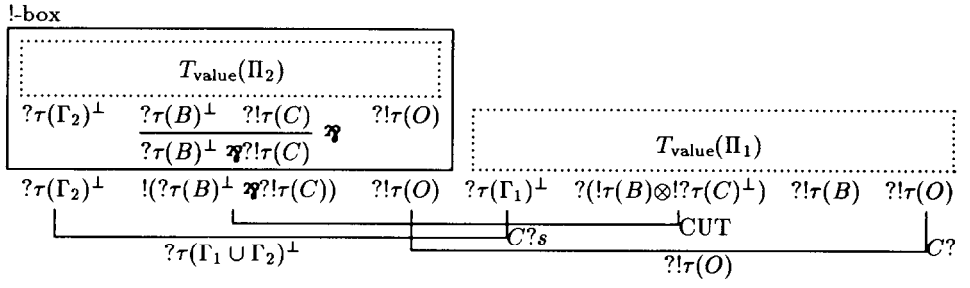
$$\dfrac{\begin{array}{c}\vdots\,\Pi_1 \\ \Gamma_1 \vdash M : A \to B\end{array}\quad\begin{array}{c}\vdots\,\Pi_2 \\ \Gamma_2 \vdash N : A\end{array}}{\Gamma_1 \cup \Gamma_2 \vdash (M\ N) : B}\ App,$$

if both M and N are values and A is atomic, then $T_{\text{term}}(\Pi)$ is



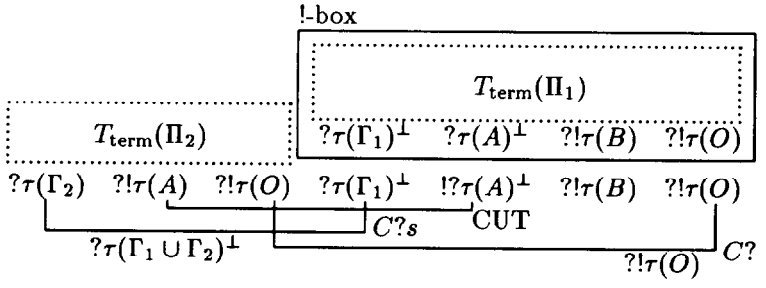
Note that $T_{\text{value}}(\Pi_2)$ has no unbinding door because a value of atomic type must be a variable.

If both M and N are values but A is a function type $B \to C$, then $T_{\text{term}}(\Pi)$ is



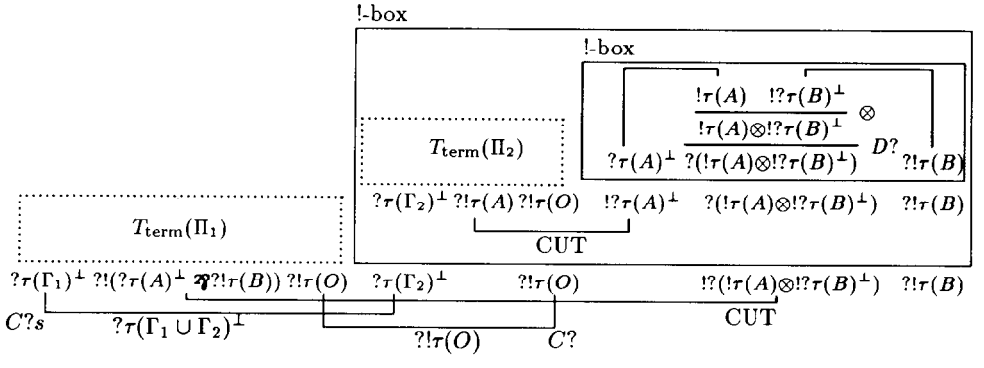
where the above proof net does not have a $C?$-link between $?!\tau(O)$ if $T_{\text{value}}(\Pi_2)$ has no unbinding door $?!\tau(O)$.

If M is a value but N is not a value, then $T_{\text{term}}(\Pi)$ is



where the above proof net does not have $C?$-link between $?!\tau(O)$ if $T_{\text{value}}(\Pi_2)$ has no unbinding door $?!\tau(O)$.

Otherwise, i.e. if neither M nor N is a value, then $T_{\text{term}}(\Pi)$ is



where the above proof net does not have $C?$-link between $?!\tau(O)$ if $T_{\text{value}}(\Pi_2)$ has no unbinding door $?!\tau(O)$.

T is defined as follows:

Definition 21 (*Translation* T). Suppose that Π is a type derivation tree with a root judgement $\Gamma \vdash M : O$.

If $T_{\text{term}}(\Pi)$ has no unbinding door

$$\dfrac{\vdots\ T_{\text{term}}(\Pi)\ \vdots}{?\tau(\Gamma)^{\perp} \quad ?!\tau(O)}$$

then $T(\Pi)$ is $T_{\text{term}}(\Pi)$ itself, else, i.e. $T_{\text{term}}(\Pi)$ has an unbinding door

$$\dfrac{\vdots\ T_{\text{term}}(\Pi)\ \vdots}{?\tau(\Gamma)^{\perp} \quad ?!\tau(O) \quad ?!\tau(O)}$$

then $T(\Pi)$ is

$$\dfrac{\dfrac{\vdots\ T_{\text{term}}(\Pi)\ \vdots}{?\tau(\Gamma)^{\perp} \quad ?!\tau(O) \quad ?!\tau(O)}}{?!\tau(O)}\ C?$$

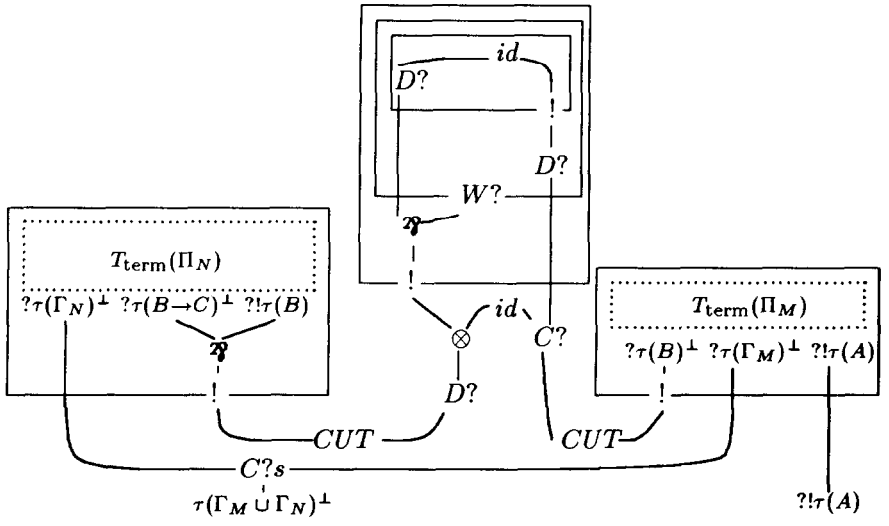
Note that it is due to the abort-typing condition that the type of the unbinding door is equal to that of the term $?!\tau(O)$.

We finish this section by presenting an example of translation T.

Example 22. Let A, B, and C be types, x and f variables, M and N terms, Γ_M and Γ_N type environments, and Π_M and Π_N type derivation trees such that

$$\vdots\ \Pi_M \qquad\qquad \vdots\ \Pi_N$$
$$\Gamma_M \cup \{[x:B]\} \vdash M : A, \qquad \Gamma_N \cup \{[f: B \to C]\} \vdash N : B.$$

And suppose that M and N include no abort-term. Then a term $(\lambda x : B.M)$ $(\mathscr{K}_{B,C}(\lambda f : B \to C.N))$ is translated as follows:



where we omit formulas labeled to each node of the proof net for simplicity.

The middle part of the above proof net corresponds to $(\mathscr{K}_{B,C}(\lambda f{:}B \to C.N))$. The middle $C?$-rule duplicates the continuation of $(\mathscr{K}_{B,C}(\lambda f{:}B \to C.N))$ which is translated to the left !-box. The $W?$-box deletes a continuation when f is called in N. The inner $D?$-rule is explained as "reading a *value*" similarly to the translation from system F to proof net [8] and the outer $D?$-rule as "reading a *continuation*". These points are the main differences of this translation from the one for system F.

4. Relation between $\lambda_c^{\to}$ and linear logic

We can find the correspondence between $\lambda_c^{\to}$ and linear logic: evaluation contexts are mapped to !-boxes, call-by-value redexes to the outmost CUT-links, and computation to proof normalization.

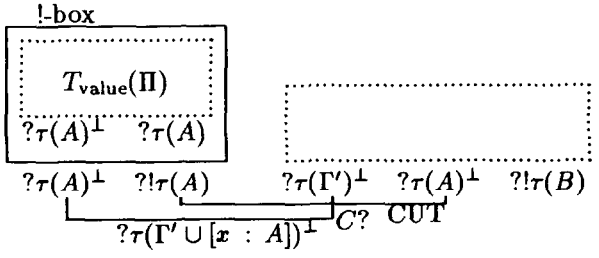
4.1. Evaluation context in the proof net

An evaluation context is mapped to the !-box of the proof net by the translation T.

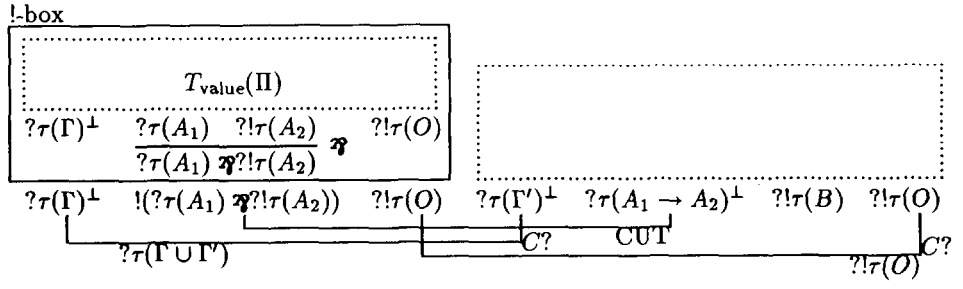
Proposition 23 (Evaluation context as !-box). *Let M be a term of type A under a type environment Γ, $E[\ ]$ an evaluation context such that $E[M]$ is a term of type C under $\Gamma \cup \Gamma'$, Π a type derivation tree of M, and Π' a type derivation tree of $E[M]$*

$$\overset{\vdots\,\Pi}{\Gamma \vdash M : A}, \qquad \overset{\vdots\,\Pi'}{\Gamma \cup \Gamma' \vdash E[M] : C}.$$

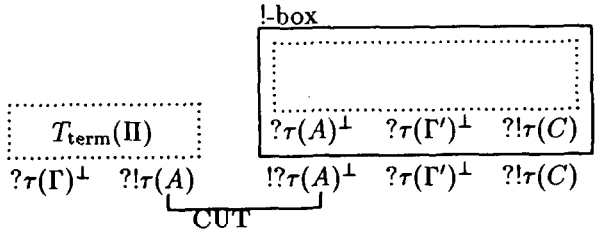
If M is a value and its type A atomic (actually M is a variable of atomic type A), then $T(\Pi')$ *is*



If M is a value and its type A a function type $A_1 \to A_2$*, then* $T(\Pi')$ *is*



If M is not a value, then $T(\Pi')$ *is*



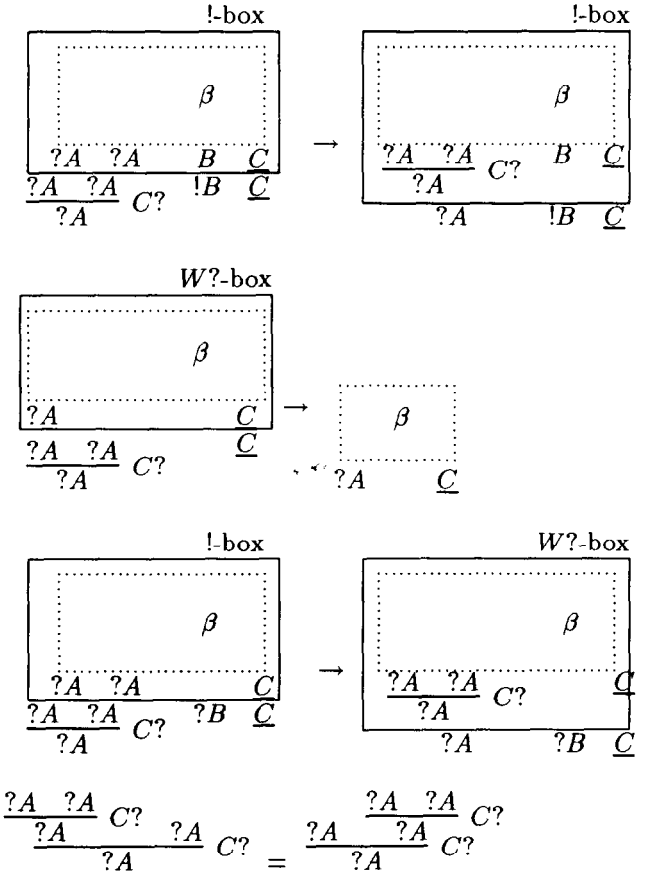
We here find a feature that a subterm pointed by an evaluation context appears at the outmost, in other words, a call-by-value redex is transformed to an outmost CUT-link. This feature is common to CPS-transformation: call-by-value CPS-transformation maps a call-by-value redex to a redex of the weak head reduction.

4.2. Computation and proof normalization

The correspondence between computation in $\lambda_c^{\to}$ and proof normalization in linear logic is presented in the following. Roughly speaking, each step of the computation can be simulated by the cut elimination of a proof net. The proof normalization is

a little weak in the aspect of handling $C?$-links (contraction) and the $W?$-box (weakening).

Definition 24 (*Extension of cut elimination of proof nets*). The cut elimination of proof nets is here extended by the addition of the following contractions and an equality:



Under the proof normalization defined in the above, the following holds:

Theorem 25 (Computation as proof normalization). *The computation rule* $\rightarrow_{comp}$ *of* $\lambda_c^{\rightarrow}$ *corresponds to several steps of elimination of CUT-links: if* M, N *are terms such that* $M \rightarrow_{comp} N$, *then the* $T(M)$ *are rewritten to* $T(N)$ *by the proof normalization procedure extended as above and elimination of useless outmost* $W?$*-boxes.*

Here, we comment on "elimination of useless outmost $W?$-boxes": The formulas $\tau(\Gamma)$ in the auxiliary door of $T(M)$ exactly correspond to free variables $FreeVar(M)$, on the other hand, the number of free variables $FreeVar(M)$ occurring in M often decreases. The decreased free variables correspond to the eliminated outmost $W?$-boxes.

Proof (Outline). The most subtle point is the handling of $C?$-links and $W?$-boxes, which corresponds to the handling of variables in λ-terms. A detailed discussion about this would obsure the outline of our argument. For this reason, we do not go into these details.

First, we have to reconstruct the substitution operation: the proof normalization of a proof net does not simulate the usual substitution operation. So, we change the substitution as follows:

Definition 26 (*Substitution simulated by proof normalization*). A substitution, written $M[x:=N]$, where we substitute a term N for a free variable x in a term M, is defined in the following, using a "subroutine" $M\{x:=N\}$ defined only in the case that a variable x strictly occurs in a term M.

$$M[x:=N] \to M \qquad \sigma_{Start0}$$
$$\text{if } x \notin FreeVar(M);$$
$$\to M\{x:=N\} \qquad \sigma_{Start1}$$
$$\text{if } x \in FreeVar(M);$$

$$(M'\,M'')\{x:=N\} \to (M'\{x:=N\}\; M''\{x:=N\}) \qquad \sigma_{App11}$$
$$\text{if } x \in FreeVar(M'),\ FreeVar(M'');$$

$$(M'\,M'')\{x:=N\} \to (M'\{x:=N\}\; M'') \qquad \sigma_{App10}$$
$$\text{if } x \in FreeVar(M'),\ x \notin FreeVar(M'');$$

$$(M'\,M'')\{x:=N\} \to (M'\; M''\{x:=N\}) \qquad \sigma_{App01}$$
$$\text{if } x \notin FreeVar(M'),\ x \in FreeVar(M'');$$

$$(\lambda y{:}A.M)\{x := N\} \to (\lambda y{:}A.M\{x:=N\}); \qquad \sigma_{Lam}$$
$$\text{where } x \neq y$$

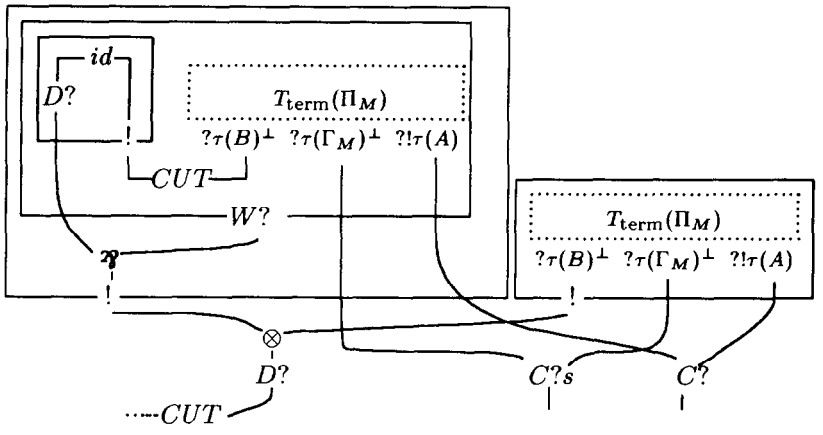
$$\mathscr{A}_{A,B}\{M\}\{x:=N\} \to \mathscr{A}_{A,B}\{M\{x:=N\}\} \qquad \sigma_{abort}$$

We can easily check that the substitution $M[x := N]$ defined above is equivalent to the usual one.

Then, if we notice the following points, the proof is straightforward.

- the shape of evaluation contexts after the translation, explained above;
- the correspondence between each substitution step defined above and each proof normalization step;
- outmost CUT-links correspond to steps of the computation (except the substitution operation);
- the correspondence between free variables of a subterm and auxiliary doors of the proof box (which originates from the subterm) through the translation T. □

Example 27. Last, we present an example. We observe a normalization for the proof net $T((\lambda x{:}\ B.M)\ (\mathscr{K}_{B,C}(\lambda f{:}\ B \to C.N)))$ in the example at Section 3. If we contract the outmost right CUT-link, it becomes



(this figure shows only the left part changed by the normalization: the lower CUT-link is connected to the !-box of $T_{\mathrm{term}}(\Pi_N)$.)

Remind that

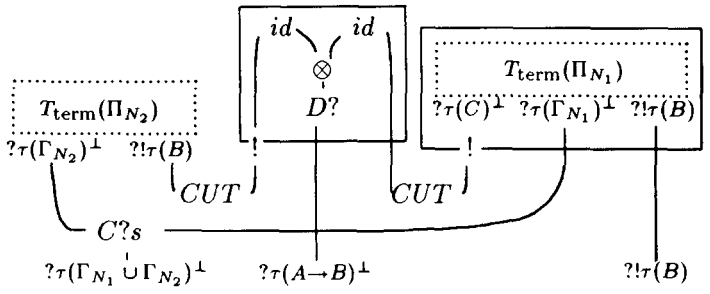
$$(\lambda x{:}\ B.M)(\mathscr{K}_{B,C}((\lambda f{:}\ B \to C.N)))$$
$$\to_{comp}(\lambda x{:}\ B.M)((\lambda f{:}\ B \to C.N)(\lambda z{:}\ B.\mathscr{A}_{A,C}\{(\lambda x{:}\ B.M)z\})).$$

In the above proof net, the inside of the $W?$-box corresponds to $(\lambda x{:}\ B.M)z$ and the one of the left !-box to $(\lambda z{:}\ B.\mathscr{A}_{A,C}\{\lambda x{:}\ B.M)z\})$. These normalization steps correspond to this computation transition and we find that the !/C?-SC step (see [8]) at the right CUT-link duplicates the continuation and that the $W?$-box in the proof net translated by $\mathscr{K}_{B,C}$ is an early form which grows up into an abort-term.

For a more detailed observation, suppose that N is $(\lambda y{:}\ C.N_1)\ (fN_2)$ such that

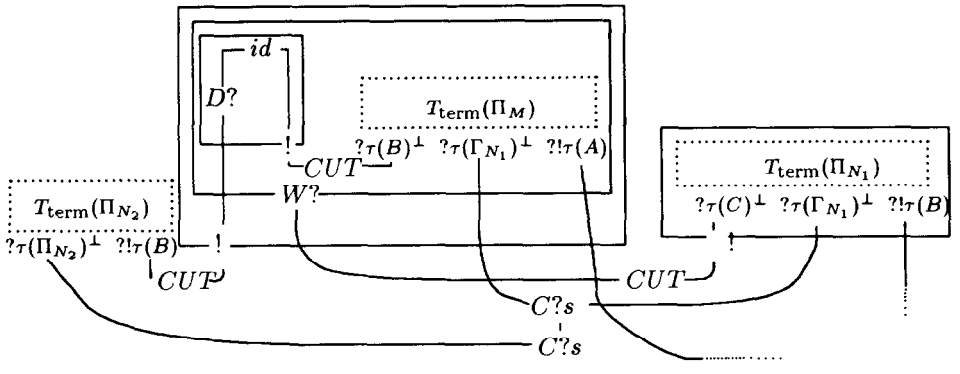
$$\vdots\,\Pi_{N_1} \qquad \vdots\,\Pi_{N_2}$$
$$\Gamma_{N_1} \cup \{[y{:}\ C]\} \vdash N_1{:}\ B, \qquad \Gamma_{N_2} \vdash N_2{:}\ B.$$

Then $T_{\text{term}}(N)$ is



The middle part of the !-box has the faculty of connecting the given function to an argument port and an output port.

The normalization proceeds further under this supposition. The outmost right CUT-link can now be contracted:



(This figure shows only the changed part: the $?!\tau(A)$ is connected to the !-box of $T_{\text{term}}(\Pi_M)$ by a $C?s$-link and $?!\tau(B)$ it by a CUT-link.)

This part corresponds to $(\lambda z{:}B.\mathscr{A}_{A,C}\{(\lambda x{:}B.M)z\})N_2$. If the normalization of $T_{\text{term}}(\Pi_{N_2})$ proceeds and the middle !-box which includes the *W?-box*, has disappeared as a result of normalization, then the *W?*-box deletes the right !-box which is a continuation. It is important that the deletion of the continuation by the *W?*-box does not depend on term N_1: even if variable f does not occur in N_1, the continuation is deleted, which is one of the features of *call-by-value* evaluation.

5. Conclusion

We first introduced the programming language $\lambda_c^{\rightarrow}$ with continuation primitives. Secondly we introduced the translation from $\lambda_c^{\rightarrow}$ to linear logic, which consists of two translations: translation τ from types to propositions and translation T from terms

and proof nets. Lastly, the relationship between $\lambda_c^{\rightarrow}$ and linear logic was investigated. The following is the main result of this paper:

> In the translated type $!\tau(A) \multimap ?!\tau(B)$ of a function type $A \to B$, the first modality ! shows us the possibility of duplication and/or deletion of an input *value* (of type A) and the second modality ? the possibility of duplication and/or deletion of an input *continuation*. A characteristic of a programming language with continuation primitives is explicitly presented by modality of linear logic.

6. Related works and future works

Curry–Howard isomorphism between programming language with continuation primitives and classical logic have been studied previously [12,19,20]. There correspond various notions of the programming language to ones of classical logic: continuation primitives $\mathscr{C}$, $\mathscr{A}$ and $\mathscr{K}$ correspond to inference rules of classical logic, the $\neg\neg$-elimination rule, the $\bot$-elimination rule and Peirce's law, respectively, and continuation-passing-style transformation corresponds to $\neg\neg$-translation. If we translate negation $\neg A$ ($= A \to \bot$) as $(!A) \multimap \bot$ with multiplicative absurdity $\bot$ rather than additive absurdity 0 (like [8, p. 91]: this replacement goes well since the $\bot$-elimination rule does not occur in proofs translated by $\neg\neg$-translation. The composition of $\neg\neg$-translation and the above translation also gives a translation from classical logic to linear logic, which is equivalent to the one given in this paper in the sense of $\multimap\multimap$ because $\neg\neg A \multimap\multimap ?!A$ is provable in classical linear logic. So, the contributions of this paper in comparison with previous works are the following:

- expressing the possibility of deletion and duplication of continuations by modality ?, !;
- investigating the mechanism of continuations in a proof net.

There are several directions for future work. One is the translation from the programming language to linear logic. We have used many !-boxes in the translation and these boxes ensure the determinism. The abuse of boxes does harm to another strong point of proof nets, i.e. *parallelism*, which is the point to be improved in our work.

Another direction is improvement of $\lambda_c^{\rightarrow}$: this direction is not related to the continuation itself. The treatment of free variables in the definition of the translation is quite complicated. In usual λ-calculi, the substitution has been thought to be a "cheap" operation. However, it is not "cheap" and cannot be disregarded in linear logic. Such a gap causes the complicated treatment at the part corresponding to the free variables when we define the translation. Therefore, we should bring λ-calculus closer to linear logic. $\lambda\sigma$-calculus [1] is recently proposed by Abadi et al. Substitutions are manipulated explicitly, hence, the substitution is not treated as a cheap operation,

which is fit for our work. The reconstruction of this work by this calculus will fill the gap between λ-calculus and linear logic and the situation will be improved.

We have studied the simply typed language. Danos proposed the untyped version of proof net, called *pure net* [3]. We may extend our work to the untyped version with his work.

In this paper, we investigated only two primitives $\mathscr{K}$ and $\mathscr{A}$. Other primitives have been proposed, for example *shift, reset* by Danvy and Filinski [4]. The study of new primitives is also a future work.

Acknowledgements

I am deeply indebted to Masami Hagiya for his help in preparing this paper and wish to thank P.-L. Curien, J.-Y. Girard, C.A. Gunter, R. Hasegawa, S. Hayashi, T. Ito, A.R. Meyer, C. Murthy, H. Nakano, H. Tsuiki, and the anonymous referees. I owe the discussion in Section 6 to C. Murthy and one of the anonymous referees.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, Explicit substitutions, in: *Proceedings 17th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Francisco, CA (1990).

[2] A.W. Appel and T. Jim, Continuation-passing, closure-passing style, in: *Proceedings 16th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Austin, TX (1989).

[3] V. Danos, Dynamic graphs, an alternative way to compute λ-terms, in: *Proceedings 3rd Italian Conference on Theoretical Computer Science*, Mantova (1989).

[4] O. Danvy and A. Filinski, Abstracting control, in: *Proceedings ACM Conference on Lisp and Functional Programming* (1990).

[5] M. Felleisen, D.P. Friedman, E. Kohlbecker and B. Duba, A syntactic theory of sequential control, *Theoret. Comput. Sci.* **52** (1987) 205–237.

[6] A. Filinski, Declarative continuations and categorical duality, Master's Thesis, DIKU Report No. 89/11, University of Copenhagen (1989).

[7] P. Fradet and D. Le Métayer, Compilation of functional languages by program transformation, *ACM Trans. Programming Languages and Systems* **13** (1) (1991) 21–51.

[8] J.-Y. Girard, Linear logic, *Theoret. Comput. Sci.* **50** (1987) 1–102.

[9] J.-Y. Girard, Towards a geometry of interaction, in: *Categories in Computer Science and Logic*, Contemporary Mathematics **92** (1989) 69–108.

[10] J.-Y. Girard, P. Taylor and Y. Lafont, *Proofs and Types*, Cambridge Tracts in Computer Science **7** (Cambridge University Press, Cambridge, 1989).

[11] H. Friedman, Classically and intuitionistically provably recursive functions, in: D.S. Scott and G.H. Muller, eds., *Higher Set Theory*, Lecture Notes in Mathematics **699** (Springer, Berlin, 1978) 21–28.

[12] T.G. Griffin, A formulae-as-types notion of control, in: *Proceedings 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA (1990).

[13] C.T. Haynes, Logic continuations, in: *Proceedings 3rd International Conference on Logic Programming* (Springer, Berlin, 1986) 671–685.

[14] C.T. Haynes, D.P. Friedman and M. Wand, Continuations and coroutines, in: *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (1984) 293–298.

[15] Y. Lafont, The linear abstract machine, *Theoret. Comput. Sci.* **59** (1988) 157–180.
[16] D. Leivant, Syntactic translations and provably recursive functions, *J. Symbolic Logic* **50** (1985) 682–688.
[17] A.W. Mazurkiewicz, Proving algorithms by tail functions, *Inform. and Control* **18** (1971) 220–226.
[18] J.H. Morris Jr, A bonus from van Wijngaarden's device, *Comm. ACM* **15** (1972) 773.
[19] C.R. Murthy, An evaluation semantics for classical proofs, in: *Proceedings 6th Annual IEEE Symposium on Logic in Computer Science* (1991).
[20] C.R. Murthy and R.L. Constable, Finding computational content in classical proofs (1990).
[21] J. Rees and W. Clinger, Revised^3 report on the algorithmic language Scheme, *SIGPLAN Notices* **21** (12) (1986) 37–79.
[22] G.L. Steele Jr, Rabbit, a compiler for Scheme, Ai-tr-474, MIT AI Lab., Cambridge, MA (1978).
[23] G.L. Steele Jr, S.E. Fahlman, R.P. Gabriel, D.A. Moon and D.L. Weinreb, Common Lisp: the language (1984).
[24] C. Strachey and C.P. Wadsworth, Continuations: a mathematical semantics for handling full jumps, Technical Monograph prg-11, Oxford University Computing Laboratory, Programming Research Group, Oxford (1974).
[25] A. van Wijngaarden, *Recursive Definition of Syntax and Semantics* (North-Holland, Amsterdam, 1966) 13–24.
[26] M. Wand, Continuation-based program transformation strategies, *J. ACM* **27** (1) (1978) 174–180.