Let-Binding in a Linear Lambda Calculus with First-Class Continuations

Shinya Nishizaki
Department of Computer Science Tokyo Institute of Technology Tokyo, Japan
nisizaki@cs.titech.ac.jp

Abstract. In our previous work, we proposed a linear lambda calculus with first-class continuations. In the usual lambda calculus, an argument value can be duplicated and deleted, but the continuation cannot. In our calculus, a value is allowed to be neither duplicated nor deleted, but a continuation is allowed, so it can be considered to be dual with the usual lambda calculus. In this paper, we extend a linear lambda calculus with first-class continuations by adding let-binding. We define a syntax, a call-by-value reduction, and a type system of the calculus. Then, we discuss the let-polymorphism in our calculus.

1. Introduction
In this section, we discuss several background ideas.

1.1. Linear Logic and Linear Lambda Calculus
Girard et al. [1] [2] proposed using linear logic to reconstruct traditional constructive logic. The intuitive negation is separated into two operations: one is linear negation, which gives the pure negation, and the other is a reaffirmation modality.

In the sequent calculus proposed by Gentzen [2], there are several rules that are closely related to classical logic:

\[
\begin{align*}
\frac{\Gamma, A, \Delta \vdash \Delta}{\Gamma, A \vdash \Delta} & \quad \text{LC} \\
\frac{\Gamma \vdash A, \Delta}{\Gamma, A \vdash \Delta} & \quad \text{LW} \\
\frac{\Gamma \vdash A, A, \Delta}{\Gamma, A \vdash \Delta} & \quad \text{RC} \\
\frac{\Gamma \vdash A, A, \Delta}{\Gamma, A \vdash \Delta} & \quad \text{RW}
\end{align*}
\]

In the classical fragment \(LK\) of sequent calculus, one uses a sequent such that

\[A_1, \ldots, A_m \vdash B_1, \ldots, B_n.\]

The intuitionistic fragment \(LJ\) is obtained by restricting the right-hand side structural rules \(RC\) and \(RW\). This is done by imposing a syntactical restriction on sequents such that the number of formulas on the right-hand side is, at most, one—that is,

\[A_1, \ldots, A_m \vdash B.\]

In linear logic, the structural rules are controlled by the modal operators! And?
Note that the left-hand rules $LC$ and $LW$ are identified by the right-hand rules $RC$ and $RW$ through the following linear negations:

$$(!A)^\perp = (?A)^\perp$$

And

$$A_1, \ldots, A_m \vdash B_1, \ldots, B_n \iff \vdash A_1^{\perp}, \ldots, A_m^{\perp}, B_1, \ldots, B_n.$$

Linear lambda calculus is a typed lambda calculus corresponding to linear logic under the Curry-Howard isomorphism, where the type assignments, such as $\Gamma, x : A$, are sequences of pairs each consisting of a variable and a type, rather than a partial mapping of variables to types. In the rule above, the variable $x$ can be used only once; $x$ is not allowed to occur in $\Gamma$.

It is known that classical logic corresponds to the mechanism of first-class continuation under the Curry-Howard isomorphism [3, 4, 5, 6, and 7]. As mentioned in [6], in a lambda calculus with first-class continuations, the function type $B \to A$ of call-by-value calculus is represented as $!A \to ?!B$ in linear logic. The of-course modality for the domain type, $!A$, means that an argument can be deleted and duplicated, and the why-not modality of $?!B$ means that a continuation can be deleted and duplicated.

In [8] and [9], we studied linear lambda calculus with first-class continuations; that is, a continuation can be both deleted and duplicated, but an argument can be neither deleted nor duplicated. In this type of calculus, the function type $A \to B$ is represented as $A \to ?!B$.

Actually, the lambda abstraction and the function application in this calculus are more restrictive than in linear lambda calculus, in which a function of type $A \to B$ is represented as $A \to ?!B$ because of the why-not modality attached to the codomain type $B$.

### 1.2. Let-Binding

In various types of functional programming [10] [11] [12], let-binding is provided in order to bind a local variable to a value. For example,

```plaintext
let twice(f,x)=f(f(x))
and square(x) = x * x
in
  twice(square, 10)
end
```

In untyped lambda calculus, a let expression (let $x = M$ in $N$) is equivalent to a beta redex $(\lambda x, M)N$ under operational semantics. Also, the let-binding is not an essential extension and is considered to be syntax sugar.

An ML polymorphism in an ML-type system [13] is also called a let-polymorphism, since the second-order polymorphism is allowed only for the let-bound variables. For example, in the second-order polymorphic lambda calculus System F (in the Curry style),
Wells [14] has shown that the type inference for System F is undecidable; therefore, the type system is not realistic for practical, functional programming languages.

The ML polymorphic-type system was invented as a realistic subsystem of System F [13][15] and is now incorporated into various functional programming languages such as Standard ML [12], OCaml [11], Haskell [16], and others.

For example, an expression

\[
\lambda f \forall \alpha. (\alpha \rightarrow \alpha) \cdot (f (\beta \rightarrow \beta \rightarrow \beta) \ f \beta \rightarrow \beta).
\]

Can be typed in ML-polymorphic lambda calculus. In the type system, polymorphism is allowed for the let-bound variables. In the example above, the let-bound variable \( f \) is of the type \( \forall \alpha. \alpha \rightarrow \alpha \), and the first and second occurrences of \( f \) in the body are of types \( (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \) and \( (\beta \rightarrow \beta) \), respectively. If one identifies a let-binding \( \text{let}\ x = M\ \text{in}\ N \) as a substitution \( x[\lambda x.M] \), the substitution derives the polymorphism. Consider the aforementioned example; the corresponding substitution is

\[
(f f) [(\lambda x. x)/x];
\]

That is,

\[
(\lambda x.x)(\lambda x.x).
\]

The first and second appearances of the term \( (\lambda x.x) \) are typed differently, as \( (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \) and \( (\beta \rightarrow \beta) \), respectively. Hence, the let-polymorphism is not extended much further from the simple type theory.

1.3. Purpose of the Research

We extend a linear lambda calculus with first-class continuations, proposed in [8] and [9], by attaching let-binding. Using let-binding, a term

\[
\text{let}\ f = \lambda x. x\ \text{in}\ (f f)
\]

Can be typed in which, for example, the bound variable \( f \) can appear several times.

Here, we propose a linear lambda calculus with first-class continuations extended with let-binding (LCL). We define its syntax and operational semantics and establish the type system of the calculus.

2. Linear Lambda Calculus with First-Class Continuations and Nonlinear Let-Bindings

In this section, we propose an LCL, extending a linear lambda calculus with the first-class continuations proposed in [8] and [9].

2.1. Syntax of LCL

Definition 1 (Type): Suppose that the primitive types \( \alpha, \beta, \gamma, ... \) are given in advance. The types of LCL are defined recursively by the following grammar:

\[
A ::= \alpha | (A \rightarrow B).
\]

In this definition, we put together type variables and atomic data types as the primitive types. The type \( (A \rightarrow B) \) is called a function type; its domain type is \( A \) and codomain type is \( B \).

Definition 2 (Term): A countable set of term variables (or simply, variables) \( x, y, z, ... \) is assumed to be given.

The terms of LCL are defined recursively by the following grammar:
\[ M ::= x \mid (\lambda x. M) \mid (M N) \mid \text{callcc} \mid \text{abort}(M) \mid (\text{let } x = V \text{ in } M), \]

Where \( V \) is a value defined as

\[ M ::= x \mid (\lambda x. M) \mid \text{callcc}. \]

The term \((\lambda x. M)\) is called a lambda abstraction, \((M N)\) is called a function application, and \((\text{let } x = V \text{ in } M)\) is called a let-expression. The term \text{callcc} is a constant. In some cases, we introduce the constants \( c, c', ... \) and the function symbols \( f, g, ... \). For example, we can formalize natural numbers using a constant \( \text{zero} \) and a successor function \( \text{succ}(\cdot) \).

If constants and function symbols are incorporated into the terms, then terms that consist of these symbols are also values. If one has a constant \( \text{zero} \) and a unary function symbol \( \text{succ}(\cdot) \), then \( \text{succ}(\text{succ}(\text{succ}(\text{zero})) \) is a value.

2.2. Call-by-Value Reduction

Definition 3 (Evaluation Context): The call-by-value evaluation context is defined by the following grammar:

\[
E[\cdot] ::= [\cdot] \mid (E[\cdot] M) \mid (V E[\cdot]) \mid (\text{let } x = V \text{ in } E[\cdot]).
\]

Definition 4 (Call-by-Value Reduction): The call-by-value reduction of LCL is a binary relationship between terms and is defined by the following rules:

\[
E[(\lambda x. M)V] \rightarrow E[M[V/x] \]
\[
E[(\text{callcc} M)] \rightarrow E[(M (\lambda x. \text{abort}(E[x])))]
\]
\[
E[\text{abort}(M)] \rightarrow M
\]
\[
E[(\text{let } x = V \text{ in } M)] \rightarrow E[M[V/x]].
\]

2.3. Type System

We give a simple type system to the LCL, which is based on the type system introduced in [8] and [9].

Definition 5 (Type Assignment): A type assignment \( x_1 : A_1, ..., x_n : A_n \) is a multiset of pairs consisting of a variable and a type, and the variables \( x_1, ..., x_n \) are distinct from one another. If one writes \( \Gamma, x : A \), \( x \) is assumed to be none of the variables occurring in \( \Gamma \).

Definition 6 (Typing): The typing judgement \( \Gamma \vdash M : A \) is a ternary relationship among a type assignment \( \Gamma \), a term \( M \), and a type \( A \) defined inductively by the following typing rules:
3. Let-Polymorphism

If one extends the type system introduced above, one can introduce let-polymorphism into the LCL, similarly to the ML type system.

First, one should introduce polytypes in the definition of types.

**Definition 7 (Polytype):** Polytypes, sometimes called type schema, $\sigma, \tau, \ldots$ are defined as

$$\sigma \equiv \forall \alpha_1 \cdots \forall \alpha_n. A,$$

Where $n \geq 0$.

The types defined in the previous section are called monotypes, for the purpose of identification. If $n = 0$, $\sigma$ is considered to be a monotype.

The type assignments are changed as a multiset of pairs consisting of a variable and a polytype, such as

$$x_1: \sigma_1, \ldots, x_n: \sigma_n.$$

The typing rule for a variable should be changed as

$$\frac{x : A \vdash x : A}{\text{Var}}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x. M) : (A \rightarrow B)}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : A}$$

$$\frac{\Gamma \vdash \lambda x. V : (A \rightarrow B)}{\Gamma \vdash (\lambda x. V W) : B}$$

$$\frac{\Gamma \vdash M : (A \rightarrow B) \quad \Gamma \vdash V : A}{\Gamma \vdash (M V) : B}$$

$$\frac{\Gamma \vdash N : A}{\Gamma \vdash (V N) : B}$$

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \text{callc} : ((A \rightarrow B) \rightarrow C) \rightarrow C}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma, x_1 : A, \ldots, x_n : A \vdash M : B}{\Gamma \vdash \text{let } x = V \text{ in } M[x/x_1, \ldots, x/x_n] : B}$$

**Let**
For example, a term
\[
\texttt{let } f = \lambda x. x \texttt{ in } (f f)
\]
Is typed as follows.

\[
\begin{array}{l}
\vdots \\
\texttt{let } f = \lambda x. x \texttt{ in } (f f) : \beta \rightarrow \beta \\
\end{array}
\]

The right-hand side subtree is as follows:

\[
\begin{array}{l}
f_1 : \forall \alpha. \alpha \rightarrow \alpha \quad f_1 : (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \\
f_2 : \forall \alpha. \alpha \rightarrow \alpha \quad f_2 : \forall \alpha. \alpha \rightarrow \alpha \rightarrow (f_1 f_2) \\
\end{array}
\]

4. Concluding Remarks
In this paper, we extend a linear lambda calculus with first-class continuations by adding let-binding expressions. We introduce a syntax, a reduction, and a type system of the extended LCL.
In future work, we will address the following issues:
- A subject-reduction theorem for LCL
- A reduction relation that does not depend on the evaluation strategies of the calculus and its confluence property
- Duality between LCL and the usual lambda calculus
- An abstract machine based on our calculus [8, 9]

Acknowledgments
This work was supported by Grants-in-Aid of Scientific Research (C) (24500009).

References
[1] J-Y. Girard, “Linear Logic,” *Theoretical Computer Science*, vol. 50, pp. 1–101, 1987.
[2] J-Y. Girard and Y. Lafont, *Proofs and types*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989, vol. 7.
[3] T. G. Griffin, “A formulae-as-types notion of control,” in Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, 1990.
[4] C. Murthy, “An evaluation semantics for classical proofs,” in Proc. 5th IEEE Annual Symposium on Logic in Computer Science. IEEE Computer Society Press, 1991.
[5] A. Filinski, “Declarative continuations and categorical duality,” 1989.
[6] S. Nishizaki, “Programs with continuations and linear logic,” *Science of Computer Programming*, vol. 21, no. 2, pp. 95–116, 1994.
[7] P. Wadler, “Call-by-Value Is Dual to Call-by-Name — Reloaded,” ser. Lecture Notes in Computer Science, vol. 3467. Springer Berlin Heidelberg, 2005, pp. 185–203.
[8] K. Narita, S. Nishizaki, and T. Mizuno, “A simple abstract machine for functional first-class continuations,” 2010 International Symposium on Communications and Information Technology, IEEE, pp. 111—114, 2010, doi: 10.1109 / ISCIT. 2010. 5664903.
[9] K. Narita and S. Nishizaki, “A parallel abstract machine for the RPC calculus,” International Conference on Informatics and Information Science, pp. 320—332, Springer, 2011, doi: 10.1007 / 978-3-642-25462-8_28.