Strong Normalizability of the Simply-Typed Lambda Calculus with Environment Extraction from Function Closures

Kosuke KANESHITA[1] and Shinya NISHIZAKI[2]

[1] Institute of Science Tokyo (formerly Tokyo Institute of Technology), Ookayama, Meguro, Tokyo 152-8552, JAPAN kosuke.kaneshita@lambda.cs.titech.ac.jp
[2] Institute of Science Tokyo (formerly Tokyo Institute of Technology), Ookayama, Meguro, Tokyo 152-8552, JAPAN nisizaki@cs.titech.ac.jp

Abstract. A function closure is a construct that encapsulates the body of a function, along with the bindings of free variables to their corresponding values in the scope of the function. Environment extraction refers to the process of retrieving these bindings from a function closure. A lambda calculus with environment extraction is a computational framework designed to facilitate this process.
The simple type system, a basic type theory, is based atomic types and function types. The lambda calculus equipped with a simple type system, commonly known as the simply typed lambda calculus, exhibits the property of strong normalization. Strong normalization ensures that, regardless of the order of reductions, the reduction sequence always terminates in a finite number of steps, yielding an irreducible normal form. In this study, we extend the simply typed lambda calculus by integrating the mechanism of environment extraction. We establish that the strong normalization property is preserved in this extended framework. The proof proceeds as follows: first, we define the λX calculus, which incorporates environment extraction. Then, we introduce the λrp calculus, which uses records and pairs. We define translation rules between the λX calculus and the λrp calculus, and prove their soundness. We then assign types to both calculi and verify type preservation. In addtion, we formalize the notion of term length in the λX calculus. Finally, we prove the strong normalization of the λrp calculus and use this result to establish the strong normalization of the λX calculus.

Keywords: typed lambda calculus, function closure, environment, strong normalizability

1 Introduction

1.1 Function Closure

A function closure is a construct that combines the body of a function with the lexical environment in which the function was defined. This environment

J. Caro et al. (eds.), Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2024) , Atlantis Highlights in Computer Sciences 23,
https:// doi.org/10.2991/978-94-6463-684-0_3

includes any local variables that were in scope at the time the closure was created. Closures are particularly useful for implementing functions that require a persistent state.

From an academic perspective, function closures are important because they enable higher-order functions-functions that can take other functions as arguments or return them as results. This capability is a cornerstone of functional programming, supporting the development of abstract, concise, and expressive code.

Function closures are extensively employed in different programming contexts, such as:

Data Encapsulation: They allow the creation of private variables that remain inaccessible from external scopes.

Callback Functions: In asynchronous programming, they are frequently used to preserve state throughout asynchronous operations.

Functional Programming: They facilitate the creation of higher-order functions like map, filter, and reduce, which are fundamental in functional programming.

To summarize, closures constitute a potent feature in many programming languages, providing a mechanism for state maintenance and enabling the creation of more modular and maintainable code. Their ability to capture and retain access to their lexical environment makes them essential in both theoretical and practical aspects of programming.

1.2 Manipulating Function Closures in Emacs Lisp

The concept of function closures is available in programming languages that feature first-class functions with lexical scope. This allows functions to be passed as arguments to or returned from other functions.

However, in the majority of programming language implementations, it is not possible to inspect or explicitly modify the internal state of function closures. An exception to this is Emacs Lisp[1], which has supported lexical scope and function closures since version 26.1. In Emacs Lisp, function closures are represented as S-expressions rather than inaccessible data containing the interpreter's internal state, thus allowing user access.

```
>>  (setq lexical-binding t)
    t
>>  (setq increment
          (let ((counter 0))
      (lambda (n)
        (setq counter (+ counter n))
        counter)
      ))
    (closure ((counter . 0) t) (n)
        (setq counter (+ counter n)) counter)
```

In this example, a function closure that take an argument n and increments the counter by n is created. The counter is stored in the lexical environment of the function closure. The function closure is then assigned to the variable `increment`. The last two lines of the output show the S-expression that represents the function closure. The first component `((counter . 0) t)` displays the lexical environment, which contains the variable `counter` with an initial value of `0`. The second component `(n)` displays the formal parameter, and the third component displays the function body, which increments the counter by parameter `n` and returns the new value.

If you give the actual parameter, say `42`, to the function closure using `funcall`, then the counter will be incremented by `42` and the new value will be returned. The bound value in the lexical environment is updated accordingly.

```
>> (funcall increment 42)
    42
>> increment
    (closure ((counter . 42) t) (n)
        (setq counter (+ counter n)) counter)
```

In Emacs Lisp, an alternative method for creating a function closure is to construct an S-expression that represents the function closure based on an existing function closure. The following example illustrates the creation of a new function closure through the replacement of the lexical environment of an existing environment, where the value of the counter is assigned as "42."

```
>> (setq increment-42 (cons (car increment)
                           (cons '((counter . -42) t)
                              (cddr increment))))
(closure ((counter . -42) t)
         (n)
          (setq counter (+ counter n)) counter)
>> (funcall increment-42 10)
-32
```

In EmacsLisp, you can evaluate an S-expression using an environment extracted from the function closure.

```
>> (eval '(+ counter 10) (cadr increment-42))
-22
```

In the given example, the S-expression `(+ counter 10)` is evaluated within the environment extracted from the function closure `increment-42`, where `-32` is bound to the variable `counter`. Consequently, the result of this evaluation is `-22`.

2 Research Purpose

In [2], we proposed a lambda calculus framework that incorporates an environment extraction feature, as demonstrated by the function closures in the

examples herein. This framework was developed to facilitate the study of environment extraction mechanisms within lambda calculus, based on the first-class environment[3][4][5] and the explicit substitution[6]. In aper[2], we introduced both untyped and simply-typed systems, provided the subject reduction theorem, and demonstrated the translation to conventional lambda calculus along with the soundness of this translation.

In this paper, we further investigate the property of strong normalization, which is a fundamental characteristic of typed lambda calculus, and we provide a proof for the strong normalization theorem.

3 Related Works

This study is based on the previous research conducted by the authors. The following papers are related to the current study:

Lambda Calculi with First-class Environments We studied the untyped and simply-typed lambda calculi with first-class environments in [3]. In [4], we studied the ML-polymorphic lambda calculus with first-class environments and its type inference algorithm..

Lambda Calculi with Environment Extraction In [2], we introduced the lambda calculus with environment extraction, which is an extension of the lambda calculus with first-class environments. We defined the syntax, reduction rules, and type system of the lambda calculus with environment extraction and provided the translation rules to the lambda calculus with records and pairs. In the paper, we investigated fundamental properties such as the soundness of the translation rules and the subject reduction theorem.

Lambda Calculus with Environment Transplanting In [7], we propose the concept of environment transplanting. Environment transplanting involves overwriting the environment of one function closure with that of another function closure, effectively transplanting it. Unlike environment extraction, which requires extracting the environment from a function closure as a first-class environment, environment transplanting does not necessitate extracting the environment as a first-class entity. While first-class environments are powerful programming constructs, they often come with undesirable theoretical properties. By avoiding first-class environments, environment transplanting is expected to enhance the theoretical properties of the system.

In addition to the research conducted by the authors, the following papers are relevant to the present study.

Sato's explicit environments In [8], the authors introduce $\lambda\epsilon$, a simply typed calculus with first-class environments, generalizing explicit substitutions and records. Unlike our work, their approach handles variable scopes within environments differently.

Implementations of programming language Scheme First-class environments are not supported in the current Scheme standard [9] . Although environments can be utilized using eval, the capability to capture the current

environment as a first-class entity is not provided. However, some Scheme language implementations, such as Elk[10], do offer mechanisms for first-class environments.

4 Untyped Calculi

In this paper, we first introduce the untyped lambda calculus with environment extraction, denoted as λx, and the untyped lambda calculus with records and pairs, denoted as λrp. We define the syntax and reduction rules for these two calculi. Subsequently, we provide the translation from λx to λrp.

Additionally, the typed calculi in this paper are defined in the Curry style, meaning that the type system is defined independently of the terms and reduction rules. The type system is elaborated in a later section.

4.1 Untyped Lambda Calculus with Environment Extraction

We introduce the untyped lambda calculus with environment extraction, denoted as λx. The syntax and reduction rules for λx are defined as follows, similar to Paper [2]

Definition 1 (Term of λx). The *terms* $M, N, \ldots$ of the lambda calculus λx with environment extraction are defined recursively according to the following grammar.

$$M ::= c \mid x \mid (\lambda x.M) \mid (M\ N) \mid (M/x) \cdot N \mid (M \circ N) \mid \mathrm{X}(M)$$

In the aforementioned grammar, c represents a constant, x a variable, $(\lambda x.M)$ a lambda abstraction, $(M\ N)$ a function applicaiton, similar to the usual lambda calculus. The term $(M/x) \cdot N$ denotes the environment N with the binding of the variable x to the term M added to it. The term $(M \circ N)$, referred to as environment composition, denotes the value of M within the environment N. The term $\mathrm{X}(M)$, called an environment extraction, represents the environment attached to the function closure that constitutes the value of M.

Example 1. Suppose that 5 and 3 are constants. The following example demonstrates the use of environment extraction in the lambda calculus.

$$\mathrm{X}(((\lambda z.\lambda y.\lambda x.(f\ x\ y\ z))5)3)$$

The variable f is a function that takes three arguments. The term $(f\ x\ y\ z)$ represents the application of the function f to the arguments x, y, and z. The term $(\lambda z.\lambda y.\lambda x.(f\ x\ y\ z))$ is a function that takes three arguments and returns the application of f to these arguments. The term 5 is the argument z, and the term 3 is the argument y. The term X extracts the environment from the function closure and returns the environment value. In this example, the environment has bindings $z \mapsto 5$ and $y \mapsto 3$.

Definition 2 (Reduction Rules of λx). The redution $M \to_{\mathrm{x}} N$ of the lambda calculus with environment extraction λx is a binary relation between terms of λx, inductively defined by the following rules:

$$
\begin{aligned}
((\lambda x.M)\ N) &\to_{\mathrm{x}} M \circ ((N/x) \cdot id) && \textbf{Beta}_{\mathrm{x}}\\
(((\lambda x.M) \circ L)\ N) &\to_{\mathrm{x}} M \circ ((N/x) \cdot L) && \textbf{BetaClos}_{\mathrm{x}}\\
\mathrm{X}((\lambda x.M) \circ L) &\to_{\mathrm{x}} L && \textbf{Extract}_{\mathrm{x}}\\
\mathrm{X}(\lambda x.M) &\to_{\mathrm{x}} id && \textbf{ExtractId}_{\mathrm{x}}\\
c \circ L &\to_{\mathrm{x}} c && \textbf{Const}_{\mathrm{x}}\\
(x \circ ((M/x) \cdot N)) &\to_{\mathrm{x}} M && \textbf{VarRef}_{\mathrm{x}}\\
(y \circ ((M/x) \cdot N)) &\to_{\mathrm{x}} (y \circ M) && \textbf{VarSkip}_{\mathrm{x}}\\
&\text{where } x \not\equiv y. &&\\
((M\ N) \circ L) &\to_{\mathrm{x}} ((M \circ L)\ (N \circ L)) && \textbf{AppComp}_{\mathrm{x}}\\
(id \circ L) &\to_{\mathrm{x}} L && \textbf{IdL}_{\mathrm{x}}\\
(M \circ id) &\to_{\mathrm{x}} M && \textbf{IdR}_{\mathrm{x}}\\
(((M/x) \cdot N \circ L &\to_{\mathrm{x}} ((M \circ L/x) \cdot (N \circ L) && \textbf{ExtnComp}_{\mathrm{x}}\\
((M \circ N) \circ L) &\to (M \circ (N \circ L)) && \textbf{CompAssoc}_{\mathrm{x}}
\end{aligned}
$$

$$
\frac{M \to_{\mathrm{x}} M'}{\mathrm{X}(M) \to_{\mathrm{x}} \mathrm{X}(M')}\ \textbf{Extract}_{\mathrm{x}}
\qquad
\frac{M \to_{\mathrm{x}} M'}{(\lambda x.M) \to_{\mathrm{x}} (\lambda x.M')}\ \textbf{Lambda}_{\mathrm{x}}
$$

$$
\frac{M \to_{\mathrm{x}} M'}{(M\ N) \to_{\mathrm{x}} (M'\ N)}\ \textbf{AppL}_{\mathrm{x}}
\qquad
\frac{N \to_{\mathrm{x}} N'}{(M\ N) \to_{\mathrm{x}} (M\ N')}\ \textbf{AppR}_{\mathrm{x}}
$$

$$
\frac{M \to_{\mathrm{x}} M'}{(M/x) \cdot N \to_{\mathrm{x}} (M'/x) \cdot N}\ \textbf{ExtnL}_{\mathrm{x}}
\qquad
\frac{N \to_{\mathrm{x}} N'}{(M/x) \cdot N \to_{\mathrm{x}} (M/x) \cdot N'}\ \textbf{ExtnR}_{\mathrm{x}}
$$

$$
\frac{M \to_{\mathrm{x}} M'}{(M \circ N) \to_{\mathrm{x}} (M' \circ N)}\ \textbf{CompL}_{\mathrm{x}}
\qquad
\frac{N \to_{\mathrm{x}} N'}{(M \circ N) \to_{\mathrm{x}} (M \circ N')}\ \textbf{CompR}_{\mathrm{x}}
$$

Example 2. The following example shows parameter passing in λx, which returns a function closure which takes one argument with an environment that binds y to 4 and z to 2.

$$
\begin{aligned}
&(((\lambda z.\lambda y.\lambda x.(f\ x\ y\ z))4)2)\\
&\to_{\mathrm{x}} ((\lambda y.\lambda x.(f\ x\ y\ z))4) \circ ((2/z) \cdot id)\\
&\to_{\mathrm{x}} (\lambda x.(f\ x\ y\ z)) \circ ((4/y) \cdot (2/z) \cdot id)
\end{aligned}
$$

The next example demonstrates the environment extraction of the function closure.

$$\begin{aligned}
&\mathrm{X}(((\lambda z.\lambda y.\lambda x.(f\ x\ y\ z))4)2)\\
\xrightarrow{*}_{\mathrm{x}}\ &\mathrm{X}\big((\lambda x.(f\ x\ y\ z)) \circ ((4/y)\cdot(2/z)\cdot id)\big)\\
\to_{\mathrm{x}}\ &(4/y)\cdot(2/z)\cdot id.
\end{aligned}$$

5 Untyped Lamba Calculus with Records and Pairs

Symbols $l, l_1, l_2, \ldots$ are called *labels*. A record is a collection of fields, each of which is a pair of a label and a value. A pair is a collection of two values. The first and second components of a pair are called the first and second projections, respectively. We define λrp as an extension of λx that includes records and pairs. You should note that the labels are not constants in λrp.

Definition 3 (Terms of λrp). The *terms* $M, N, \ldots$ of the lambda calculus λrp with records and pairs are defined recursively according to the following grammar.

$$\begin{aligned}
M ::= c \ \mid\ x \ \mid\ (\lambda x.M) \ \mid\ (M\ N)\\
\mid\ \mathrm{update}(l, M, N) \ \mid\ \mathrm{lookup}(M, l)\\
\mid\ (M, N) \ \mid\ \mathrm{Fst}(M) \ \mid\ \mathrm{Snd}(M)
\end{aligned}$$

In the aforementioned grammar, c represents a constant, x a variable, $(\lambda x.M)$ a lambda abstraction, $(M\ N)$ a function applicaiton, similar to the usual lambda calculus and λx

The term $lookup(M, l)$ denotes the result of referencing record M by label l. The term (M, N) represents a pair cosisting of M and N.

The terms $\mathrm{Fst}(M)$ and $\mathrm{Snd}(M)$ denote the first and second components of the pair M, respectively, where Fst and Snd represent the first and second projection functions.

Definition 4 (Reductin Rules of λrp). The reduction $M \to N$ in λrp is a binary relation between terms of λrp, inductively defined by the following rules:

$$\begin{aligned}
((\lambda x.M)N) &\to M[x := N] && \textbf{Beta}\\
\mathrm{lookup}(\mathrm{update}(l, M, N),\ l) &\to M && \textbf{Record Hit}\\
\mathrm{lookup}(\mathrm{update}(l, M, N),\ l') &\to \mathrm{lookup}(N, l') && \textbf{Record Skip}\\
\mathrm{Fst}((M, N)) &\to M && \textbf{First Proj}\\
\mathrm{Snd}((M, N)) &\to N && \textbf{Second Proj}
\end{aligned}$$

where $l \neq l'$.

$$\frac{M \to M'}{(\lambda x.M) \to (\lambda x.M')}\ \textbf{Lam}$$

$$\frac{M \to M'}{(M\ N) \to (M'\ N)}\ \textbf{AppL} \qquad \frac{N \to N'}{(M\ N) \to (M\ N')}\ \textbf{AppR}$$

$$\frac{M \to M'}{\mathrm{update}(l, M, N) \to \mathrm{update}(l, M', N)}\ \textbf{UpdateL}$$

$$\frac{N \to N'}{\mathrm{update}(l, M, N) \to \mathrm{update}(l, M, N')}\ \textbf{UpdateR}$$

$$\frac{M \to M'}{\mathrm{lookup}(M, l) \to \mathrm{lookup}(M', l)}\ \textbf{Lookup}$$

$$\frac{M \to M'}{(M, N) \to (M', N)}\ \textbf{PairL} \qquad \frac{N \to N'}{(M, N) \to (M, N')}\ \textbf{PairR}$$

$$\frac{M \to M'}{\mathrm{Fst}(M) \to \mathrm{Fst}(M')}\ \textbf{Fst} \qquad \frac{M \to M'}{\mathrm{Snd}(M) \to \mathrm{Snd}(M')}\ \textbf{Snd}$$

6 Translation of λx to λrp

We assume an encoding function $\lceil x \rceil$ of a variable of λx to a label of λrp, thereby encoding variable symbols of λx as labels in λrp.

Definition 5 (Translation of λx to λrp). The translation funciton $[\![M]\!](R)$ from a term M of λx and a term R λrp of λrp to a term of λrp is defined inductively by the following equations:

$$\begin{aligned}
[\![c]\!](R) &= c \\
[\![x]\!](R) &= \mathrm{lookup}(R, \lceil x \rceil) \\
[\![(\lambda x.M)]\!](R) &= \big(\lambda u.[\![M]\!](\mathrm{update}(\lceil x \rceil, u, R)),\ R\big) \\
[\![(M\ N)]\!](R) &= \big(\mathrm{Fst}([\![M]\!](R))\big)([\![N]\!](R)) \\
[\![id]\!](R) &= R \\
[\![(M/x) \cdot N]\!](R) &= \mathrm{update}(\lceil x \rceil, [\![M]\!](R), [\![N]\!](R)) \\
[\![(M \circ N)]\!](R) &= [\![M]\!]([\![N]\!](R)) \\
[\![\mathrm{X}(M)]\!](R) &= \mathrm{Snd}([\![M]\!](R))
\end{aligned}$$

The transformation $[\![-]\!](-)$ presented in this paper differs from that introduced in [3]. Specifically, it represents the transformation of function closures, $[\![(\lambda x.M)]\!](R)$ as a pair of terms: one term $\lambda u.[\![M]\!](\mathrm{update}(\lceil x \rceil, u, R))$ interprets the function closure transformation using λrp, while the other term R represents the environment value associated with the function closure. The environment extraction $\mathrm{X}(M)$ is translated to the second component of the pair $[\![M]\!](R)$.

Example 1. Consider the case $[\![\lambda y.(f\ x\ y)]\!](R')$, where R' is a term of λrp,

$$R' = \mathrm{update}(\lceil y \rceil, 1, \mathrm{update}(\lceil x \rceil, 2, id)).$$

$$
\begin{aligned}
&[\![\lambda y.(f\ x\ y)]\!](R') \\
&= \lambda u.[\![(f\ x\ y)]\!](\mathrm{update}(\lceil y \rceil, u, R')) \\
&= \lambda u.\ \big(\mathrm{Fst}([\![(f\ x)]\!](\mathrm{update}(\lceil y \rceil, u, R')))\big)([\![y]\!](\mathrm{update}(\lceil y \rceil, u, R'))) \\
&= \lambda u.\ \big(\mathrm{Fst}([\![(f\ x)]\!](\mathrm{update}(\lceil y \rceil, u, R')))\big)(\mathrm{lookup}(\lceil y \rceil, R'))) \\
&= \lambda u.\ \big(\mathrm{Fst}(\mathrm{Fst}(\mathrm{lookup}(\lceil f \rceil, (\mathrm{update}(\lceil y \rceil, u, R'))))) \\
&\qquad\qquad (\mathrm{lookup}(\lceil x \rceil, (\mathrm{update}(\lceil y \rceil, u, R'))))))\big)(\mathrm{lookup}(\lceil y \rceil, R')))
\end{aligned}
$$

The following Theorem demonstrates the soundness of the translation rules, which ensures that the translation of a term in λx to a term in λrp preserves the reduction behavior of the original term. The theorem was proved in [2].

Theorem 1 (Soundness Theorem). If $M \to_{\mathrm{x}} N$ in λx, then $[\![M]\!](R) \xrightarrow{*} [\![N]\!](R)$ in λrp for any term R in λrp.

7 Simple Type Systems

In this section, we introduce the simple type systems of λx and λrp. We define the types and typing rules of λx and λrp. The type system for λx is similar to the one proposed in the previous paper [2] and the type system for λrp is the same as the one proposed in [3], though the concrete syntax of the terms is slightly different.

7.1 Simple Type System of λx

We assume that type variables are $\alpha, \beta, \ldots$ and type constants are $\iota, \kappa, \ldots$.

Definition 6 (Types and Enviornment Types of λx). The *types* $A, B, \ldots$ of the lambda calculus λx and the *environment types* $E, H, \ldots$ are defined mutual-recursively according to the following grammar.

$$
\begin{aligned}
A &::= \alpha \mid \iota \mid (E \triangleright A \to B) \mid E \\
E &::= \{x_1 : A_1\} \cdots \{x_n : A_n\}
\end{aligned}
$$

The type $(E \triangleright A \to B)$ represents the type of a function closure, where E is the type of an environment of a function closure, A is the argument type of the function body, and B is the return type.

Definition 7 (Typing Rules of λx). The typing relation $E \vdash M : A$ of the lambda calculus λx is a ternary relation between a typing environment E, a term M, and a type A, inductively defined by the following rules:

$$
\frac{}{E \vdash c : A_c}\ \mathbf{Const}_{\mathrm{x}} \qquad \frac{}{\{x : A\}E \vdash x : A}\ \mathbf{Var}_{\mathrm{x}}
$$

where A_c is the type of the constant c.

$$\frac{\{x:A\}E \vdash M : B}{\{x:C\}E \vdash (\lambda x.M) : (\{x:C\}E \triangleright A \to B)}\ \mathbf{Lam}_{\mathrm{x}}$$

$$\frac{E \vdash M : (H \triangleright A \to B) \quad E \vdash N : A}{E \vdash (M\ N) : B}\ \mathbf{App}_{\mathrm{x}}$$

$$\frac{}{E \vdash id : E}\ \mathbf{Id}_{\mathrm{x}} \quad \frac{E \vdash M : A \quad E \vdash N : \{x:C\}H}{E \vdash (M/x) \cdot N : \{x:A\}H}\ \mathbf{Extn}_{\mathrm{x}}$$

$$\frac{E \vdash N : H \quad H \vdash M : A}{E \vdash (M \circ N) : A}\ \mathbf{Comp}_{\mathrm{x}} \quad \frac{E \vdash M : (H \triangleright A \to B)}{E \vdash \mathrm{X}(M) : H}\ \mathbf{Extract}_{\mathrm{x}}$$

In [2], we have shown that the reduction preserves the typing of terms in λx.

Theorem 2 (Subject Reduction of λx). If $E \vdash M : A$ and $M \to_{\mathrm{x}} M'$, then $E \vdash M' : A$.

7.2 Simple Type System of λrp

We assume that type variables are $\alpha, \beta, \ldots$ and type constants are $\iota, \kappa, \ldots$, which are the same as those in λx. The symbols $l_1, l_2, \ldots$ represent labels of records.

Definition 8 (Types of λrp). The *types* $A, B, \ldots$ of the lambda calculus λrp are defined according to the following grammar:

$$A ::= \alpha \mid \iota \mid (A \times B) \mid (A \to B) \mid E$$
$$E ::= \{l_1 : A_1\} \cdots \{l_n : A_n\}$$

The type $A \to B$ is a function type from A to B, and $A \times B$ is a product type of A and B. The type E is an record type. The type $\{l_1 : A_1\} \cdots \{l_n : A_n\}$ represents a record type with fields l_1 of type A_1, ..., and l_n of type A_n.

A type assignment is a set of type assignments of variables to types. The type assignment $\{x_1 : A_1\} \cdots \{x_n : A_n\}$ represents the type assignment of x_1 to A_1, ..., and x_n to A_n.

Definition 9 (Typing Rules of λrp). The typing relation $\Gamma \vdash M : A$ of the lambda calculus λrp is a ternary relation between a type assignment Γ, a term M and a type A, inductively defined by the following rules:

$$\frac{}{\Gamma \vdash c : A_c}\ \mathbf{Const}_{\mathrm{rp}} \quad \frac{}{\{x:A\}\Gamma \vdash x : A}\ \mathbf{Var}_{\mathrm{rp}}$$

$$\frac{\{x:A\}\Gamma \vdash M : B}{\{x:A\}\Gamma \vdash (\lambda x.M) : (A \to B)}\ \mathbf{Lam}_{\mathrm{rp}} \quad \frac{\Gamma \vdash M : (A \to B) \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B}\ \mathbf{App}_{\mathrm{rp}}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \{l:C\}E}{\Gamma \vdash \mathrm{update}(l, M, N) : \{l:A\}E}\ \mathbf{Update}_{\mathrm{rp}} \quad \frac{\Gamma \vdash M : \{l:A\}E}{\Gamma \vdash \mathrm{lookup}(M, l) : E}\ \mathbf{Lookup}_{\mathrm{rp}}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : (A \times B)}\ \mathbf{Pair}_{\mathrm{rp}}$$

$$\frac{\Gamma \vdash M : (A \times B)}{\Gamma \vdash \mathrm{Fst}(M) : A}\ \mathbf{Fst}_{\mathrm{rp}} \qquad \frac{\Gamma \vdash M : (A \times B)}{\Gamma \vdash \mathrm{Snd}(M) : B}\ \mathbf{Snd}_{\mathrm{rp}}$$

The following theorem demonstrates that the reduction preserves the typing of terms in λrp, which was proved in [3].

Theorem 3 (Subject Reduction of λrp). If $\Gamma \vdash M : A$ and $M \to N$, then $\Gamma \vdash N : A$.

8 Strong Normalizability of λx

Definition 10 (Translation of Type). The translation $[\![A]\!]$ of a type A of λx to a type of λrp is defined inductively by the following equations:

$$\begin{aligned}
[\![\alpha]\!] &= \alpha \\
[\![\iota]\!] &= \iota \\
[\![(E \triangleright A \to B)]\!] &= ([\![A]\!] \to [\![B]\!]) \times [\![E]\!] \\
[\![\{x_1 : A_1\} \cdots \{x_n : A_n\}]\!] &= \{\lceil x_1 \rceil : [\![A]\!]\} \cdots \{\lceil x_n \rceil : [\![E]\!]\}
\end{aligned}$$

Note that the left hand side of the last equation, $\{x_1 : A_1\} \cdots \{x_n : A_n\}$, is an environment type of λx but the right hand side $\{\lceil x_1 \rceil : [\![A]\!]\} \cdots \{\lceil x_n \rceil : [\![E]\!]\}$ is a type of λrp, thoght the same notation is used.

Theorem 4 (Type preservation of $[\![-]\!](-)$). If $E \vdash M : A$ in λx and $\Gamma \vdash R : [\![E]\!]$ in λrp, then $\Gamma \vdash [\![M]\!](R) : [\![A]\!]$ in λrp.

Proof. We prove the theorem by induction on the derivation of $E \vdash M : A$.

Case $\mathbf{Const}_{\mathrm{x}}$: suppose that $E \vdash c : A_c$ and $[\![c]\!](R) = c$. Then $\Gamma \vdash c : A_c$, that is, $\Gamma \vdash c : [\![A_c]\!]$.

Case $\mathbf{Var}_{\mathrm{x}}$: suppose that $\{x : A\}E \vdash x : A$ and $\Gamma \vdash R : [\![\{x : A\}E]\!]$. Since $[\![\{x : A\}E]\!] = \{\lceil x \rceil : [\![A]\!]\}[\![E]\!]$, $\Gamma \vdash R : \{\lceil x \rceil : [\![A]\!]\}[\![E]\!]$ Since $[\![x]\!](R) = \mathrm{lookup}(R, \lceil x \rceil)$, $\Gamma \vdash \mathrm{lookup}(R, \lceil x \rceil) : A$, that is, $\Gamma \vdash [\![x]\!](R) : [\![A]\!]$.

Case $\mathbf{Lam}_{\mathrm{x}}$: suppose that

$$\frac{\begin{array}{c}\vdots \\ \{x : A\}E \vdash M : B\end{array}}{\{x : C\}E \vdash (\lambda x.M) : (\{x : C\}E \triangleright A \to B)}\ \mathbf{Lam}_{\mathrm{x}}$$

and $\Gamma \vdash R : [\![\{x : C\}E]\!]$. We know that $\{u{:}A\}\Gamma \vdash \mathrm{update}(\lceil x \rceil, u, R) : [\![\{x : A\}E]\!]$. By the induction hypothesis, $\Gamma \vdash [\![M]\!](\mathrm{update}(\lceil x \rceil, u, R)) : [\![B]\!]$, therefore, $\Gamma \vdash \lambda u.[\![M]\!](\mathrm{update}(\lceil x \rceil, u, R)) : [\![A]\!] \to [\![B]\!]$. We have

$$\Gamma \vdash \big(\lambda u.[\![M]\!](\mathrm{update}(\lceil x \rceil, u, R)),\ R\big) : ([\![A]\!] \to [\![B]\!]) \times [\![\{x : C\}E]\!]$$

that is,

$$\Gamma \vdash [\![(\lambda x.M)]\!](R) : [\![(\{x : C\}E \triangleright A \to B)]\!]$$

Case $\mathbf{App}_{\mathrm{x}}$: Suppose that $\Gamma \vdash R : [\![E]\!]$ and

$$\frac{\begin{array}{cc}\vdots & \vdots \\ E \vdash M : (H \triangleright A \to B) & E \vdash N : A\end{array}}{E \vdash (M\ N) : B}\ \mathbf{App}_{\mathrm{x}}$$

By the induction hypothesis, $\Gamma \vdash [\![M]\!](R) : [\![(H \triangleright A \to B)]\!]$ and then $\Gamma \vdash \mathrm{Fst}([\![M]\!](R)) : [\![A]\!] \to [\![B]\!]$

By the induction hypothesis, $\Gamma \vdash [\![N]\!](R) : [\![A]\!]$. Therefore, $\Gamma \vdash (\mathrm{Fst}([\![M]\!](R)))([\![N]\!](R)) : [\![B]\!]$. Consequently, $\Gamma \vdash [\![(M\ N)]\!](R) : [\![B]\!]$.

Case $\mathbf{Id}_{\mathrm{x}}$: Since $R = [\![id]\!](R)$, $\Gamma \vdash [\![id]\!](R) : [\![E]\!]$.

Case $\mathbf{Extn}_{\mathrm{x}}$: Suppose that $\Gamma \vdash R : [\![E]\!]$ and

$$\frac{\begin{array}{cc}\vdots & \vdots \\ E \vdash M : A & E \vdash N : \{x : C\}H\end{array}}{E \vdash (M/x) \cdot N : \{x : A\}H}\ \mathbf{Extn}_{\mathrm{x}}$$

By the induction hypothesis, $\Gamma \vdash [\![M]\!](R) : [\![A]\!]$ and $\Gamma \vdash [\![N]\!](R) : [\![\{x : C\}H]\!]$, which implies $\Gamma \vdash [\![N]\!](R) : \{\lceil x \rceil{:}[\![C]\!]\}[\![H]\!]$. Then, $\Gamma \vdash \mathrm{update}(\lceil x \rceil, [\![M]\!](R), [\![N]\!](R)) : \{\lceil x \rceil : [\![C]\!]\}[\![H]\!]$. Consequently, $\Gamma \vdash [\![(M/x) \cdot N]\!](R) : [\![\{x : A\}H]\!]$.

Case $\mathbf{Comp}_{\mathrm{x}}$: Suppose that $\Gamma \vdash R : [\![E]\!]$ and

$$\frac{\begin{array}{cc}\vdots & \vdots \\ E \vdash N : H & H \vdash M : A\end{array}}{E \vdash (M \circ N) : A}\ \mathbf{Comp}_{\mathrm{x}}$$

By the induction hypothesis, $\Gamma \vdash [\![N]\!](R) : [\![H]\!]$. Hence, by the induction hyposesis, $\Gamma \vdash [\![M]\!]([\![N]\!](R)) : [\![A]\!]$. Consequently, $\Gamma \vdash [\![(M \circ N)]\!](R) : [\![A]\!]$.

Case $\mathbf{Extract}_{\mathrm{x}}$: Suppose that $\Gamma \vdash R : [\![E]\!]$ and

$$\frac{\begin{array}{c}\vdots \\ E \vdash M : (H \triangleright A \to B)\end{array}}{E \vdash \mathrm{X}(M) : H}\ \mathbf{Extract}_{\mathrm{x}}$$

By the induction hypothesis, $\Gamma \vdash [\![M]\!](R) : [\![(H \triangleright A \to B)]\!]$. Then, $\Gamma \vdash [\![M]\!](R) : ([\![A]\!] \to [\![B]\!]) \times [\![H]\!]$. Therefore, $\Gamma \vdash \mathrm{Snd}([\![M]\!](R)) : [\![H]\!]$, that is, $\Gamma \vdash [\![\mathrm{X}(M)]\!](R) : [\![H]\!]$.

□

Definition 11 (σ-reduction). The σ-reduction is a reduction defined the rules other than $\mathbf{Beta}_{\mathrm{x}}$ and $\mathbf{BetaClos}_{\mathrm{x}}$ in $\lambda\mathrm{x}$, and it constitutes a subreduction of $\to_{\mathrm{x}}$.

The σ-reduction satisfies the strong normalizability and it can be proved using the following measurement function.

Definition 12 (Length of terms of λx). The length $length(M)$ of a term M of λx is defined inductively by the following equations:

$$\begin{aligned}
length(c) &= 1 \\
length(x) &= 1 \\
length(\lambda x.M) &= length(M) \times 2 \\
length((M\ N)) &= length(M) + length(N) + 1 \\
length(id) &= 1 \\
length((M/x) \cdot N) &= length(M) + length(N) + 1 \\
length((M \circ N)) &= length(M) \times (length(N) + 1) \\
length(\mathrm{X}(M)) &= length(M) + 1
\end{aligned}$$

Lemma 1 (Decrease of Length). If $M \to_{\mathrm{x}} N$ in λx, then $length(M) > length(N)$.

If the lemma is proved, the strong normalizability of σ-reduction can be proved by the well-foundedness of the natural numbers.

Theorem 5 (Strong Normalization of σ-reduction). There is no infinite reduction sequence by σ-reduction in λx.

The proof of Lemma is proved by checking decrease for each reduction rule of λx.

Proof. This is also proved in [3], except the case of $\mathbf{Extract}_{\mathrm{x}}$; we will show only the case here.

Case $\mathbf{Extract}_{\mathrm{x}}$: suppose that $\mathrm{X}((\lambda x.M) \circ L) \to_{\mathrm{x}} L$.

$$\begin{aligned}
length(\mathrm{X}((\lambda x.M) \circ L)) &= length((\lambda x.M) \circ L) + 1 \\
&= length(\lambda x.M) \times (length(L) + 1) + 1 \\
&> length(L)
\end{aligned}$$

□

To prove the Strong Normalization Theorem, we will establish the following lemma, which provides a detailed formulation of the Soundness Theorem 1. The following lemma demonstrates that a single reduction step by rule $\mathbf{Beta}_{\mathrm{x}}$ and $\mathbf{BetaClos}_{\mathrm{x}}$ in λx corresponds to a reduction sequence of one or more steps involving a $\mathbf{Beta}_{\mathrm{rp}}$ step in λrp.

Lemma 2. If $M \to_{\mathrm{x}} N$ by either Rule $\mathbf{Beta}_{\mathrm{x}}$ or $\mathbf{BetaClos}_{\mathrm{x}}$ in λx, then $[\![M]\!](R) \xrightarrow{+} [\![N]\!](R)$ in λrp for any term R in λrp.

Specifically, Both Rule $\mathbf{Beta}_{\mathrm{x}}$ and $\mathbf{BetaClos}_{\mathrm{x}}$ correspond to the reduction sequence of Rule $\mathbf{First\ Proj}_{\mathrm{rp}}$ and $\mathbf{Beta}_{\mathrm{rp}}$.

Proof. Case $\mathbf{Beta}_{\mathrm{x}}$: Suppose that $((\lambda x.M)\ N) \to_{\mathrm{x}} M \circ ((N/x) \cdot id)$. We have

$$\begin{aligned}
&[\![((\lambda x.M)\ N)]\!](R) \\
&= \mathrm{Fst}([\![(\lambda x.M)]\!](R))([\![N]\!](R)) \\
&= \mathrm{Fst}((\lambda u.[\![M]\!](\mathrm{update}(\ulcorner x \urcorner, u, R)),\ R))([\![N]\!](R)) \\
&\to_{\mathrm{x}} (\lambda u.[\![M]\!](\mathrm{update}(\ulcorner x \urcorner, u, R)))([\![N]\!](R)) && \mathbf{First\ Proj}_{\mathrm{rp}} \\
&\to_{\mathrm{x}} [\![M]\!](\mathrm{update}(\ulcorner x \urcorner, [\![N]\!](R), R)) && \mathbf{Beta}_{\mathrm{rp}}
\end{aligned}$$

On the other hand,

$$\begin{aligned}
[\![M \circ ((N/x) \cdot id)]\!](R) &= [\![M]\!]([\![((N/x) \cdot id)]\!](R)) \\
&= [\![M]\!](\mathrm{update}(\ulcorner x \urcorner, [\![N]\!](R), R))
\end{aligned}$$

Consequently, $[\![((\lambda x.M)\ N)]\!](R) \to\!\to [\![M \circ ((N/x) \cdot id)]\!](R)$

Case $\mathbf{BetaClos}_{\mathrm{x}}$: Suppose that $(((\lambda x.M) \circ L)\ N) \to_{\mathrm{x}} M \circ ((N/x) \cdot L)$. We have

$$\begin{aligned}
&[\![(((\lambda x.M) \circ L)\ N)]\!](R) \\
&= \mathrm{Fst}([\![(\lambda x.M) \circ L]\!](R))([\![N]\!](R)) \\
&= \mathrm{Fst}([\![(\lambda x.M)]\!]([\![L]\!](R)))([\![N]\!](R)) \\
&= \mathrm{Fst}((\lambda u.[\![M]\!](\mathrm{update}(\ulcorner x \urcorner, u, [\![L]\!](R)),\ [\![L]\!](R)))([\![N]\!](R)) \\
&\to_{\mathrm{x}} (\lambda u.[\![M]\!](\mathrm{update}(\ulcorner x \urcorner, u, [\![L]\!](R)))([\![N]\!](R)) && \mathbf{First\ Proj}_{\mathrm{rp}} \\
&\to_{\mathrm{x}} [\![M]\!](\mathrm{update}(\ulcorner x \urcorner, [\![N]\!](R), [\![L]\!](R)) && \mathbf{Beta}_{\mathrm{rp}}
\end{aligned}$$

On the other hand,

$$\begin{aligned}
[\![M \circ ((N/x) \cdot L)]\!](R) &= [\![M]\!]([\![((N/x) \cdot L)]\!](R)) \\
&= [\![M]\!](\mathrm{update}(\ulcorner x \urcorner, [\![N]\!](R), [\![L]\!](R))
\end{aligned}$$

Consequently, $[\![(((\lambda x.M) \circ L)\ N)]\!](R) \to\!\to [\![M \circ ((N/x) \cdot L)]\!](R)$

□

We prove the main result of this paper, the Strong Normalization Theorem of λx as follows:

Theorem 6 (Strong Normalization of λx). There is no infinite reduction sequence by $\to_{\mathrm{x}}$ in λx.

Proof. Suppose that there is an infinite reduction sequence $M_0 \to_{\mathrm{x}} M_1 \to_{\mathrm{x}} M_2 \to_{\mathrm{x}} \cdots$. By Theorem 5, the infinite sequence includes infinite number of reduction step by Rule $\mathbf{Beta}_{\mathrm{x}}$ or $\mathbf{BetaClos}_{\mathrm{x}}$.

If you translate the infinite reduction sequence to λrp by $[\![-]\!](R)$ for a term R, it makes an infinite reduction sequence which includes infinite number of reduction step by $\mathbf{Beta}_{\mathrm{rp}}$ in λrp.

However, the infinite reduction sequence is impossible by the Strong Normalization Theorem of λrp. Therefore, the infinite reduction sequence by $\to_{\mathrm{x}}$ is impossible.

□

9 Conclusion and Future Work

In conclusion, this study extends the simply typed lambda calculus by integrating the mechanism of environment extraction, resulting in the calculus denoted as λx. The primary contribution lies in proving that strong normalization holds for this extended calculus. To achieve this, we defined the λx calculus and introduced the λrp calculus, which incorporates records and pairs. We developed a rigorous framework for translating between these calculi, verified the soundness of the translation rules, and ensured type preservation. Additionally, we formalized the notion of term length in the λx calculus to support our proofs. The strong normalization of the λrp calculus, as demonstrated in [3], served as a foundation for establishing the strong normalization of the λx calculus. This work not only strengthens the theoretical basis of lambda calculi but also provides new insights into environment extraction in typed functional programming languages.

For future work, we plan to extend the λx calculus to include polymorphic types and investigate the strong normalization of this extended system. In [7], we explored the exchange of environments between function closures. Building on this, we aim to examine the strong normalization of a calculus that incorporates the transplantation of environments. Additionally, it would be intriguing to study environment extraction in the context of first-class environments, such as those investigated by Sato [11].

The mechanism of environment extraction represents a powerful concept in functional programming languages. It is inspired by the principle of selectively restricting the modification of S-expressions that encapsulate the environment.

10 Conclusion and Future Work

In conclusion, this study extends the simply typed lambda calculus by integrating the mechanism of environment extraction, resulting in the calculus denoted as λx. The primary contribution lies in proving that strong normalization holds for this extended calculus. To achieve this, we defined the λx calculus and introduced the λrp calculus, which incorporates records and pairs. We developed a rigorous framework for translating between these calculi, verified the soundness of the translation rules, and ensured type preservation. Additionally, we formalized the notion of term length in the λx calculus to support our proofs. The strong normalization of the λrp calculus, as demonstrated in [3], served as a foundation for establishing the strong normalization of the λx calculus. This work not only strengthens the theoretical basis of lambda calculi but also provides new insights into environment extraction in typed functional programming languages.

For future work, we plan to extend the λx calculus to include polymorphic types and investigate the strong normalization of this extended system. In [7], we explored the exchange of environments between function closures. Building on this, we aim to examine the strong normalization of a calculus that incorporates the transplantation of environments. Additionally, it would be intriguing to study environment extraction in the context of first-class environments, such as those investigated by Sato [11].

The mechanism of environment extraction represents a powerful concept in functional programming languages. It is inspired by the principle of selectively restricting the modification of S-expressions that encapsulate the environment.

Acknowledgements

We would like to express our deepest condolences on the passing of Prof. Dr. Takafumi Sakurai, who held an exceptional understanding and profound insights in this field. His contributions continue to inspire our work, and we honor his memory with great respect.

This work was supported by JSPS KAKENHI Grant Number JP20K11743.

References

1. Lewis, B., LaLiberte, D., Stallman, R.: GNU Emacs Lisp Reference Manual For Emacs Version 27.2 (2021), https://www.gnu.org/software/emacs/manual/elisp.html
2. Nishizaki, S., Aoyagi, Y.: Extracting environments from function closures. In: ICSCA '22: Proceedings of the 2022 11th International Conference on Software and Computer Applications. pp. 61–68. Association for Computing Machinery (2022)
3. Nishizaki, S.: Simply typed lambda calculus with first-class environments. Publications of Research Institute for Mathematical Sciences Kyoto University 30(6), 1055–1121 (1995)
4. Nishizaki, S.: A polymorphic environment calculus and its type inference algorithm. Higher-Order and Symbolic Computation 13, 239–278 (2000)
5. Nishizaki, S.: Evaluation strategy and translation of environment calculus. In: Information Computing and Applications. pp. 232–242. Springer Berlin Heidelberg (2013), https://doi.org/10.1007/978-3-642-53932-9_23
6. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. Journal of Functional Programming 1(4), 375–416 (October 1991)
7. Nishizaki, S.: Transplanting of environments between closures in the lambda calculus. In: ICSCA '243 Proceedings of the 2023 12th International Conference on Software and Computer Applications. pp. 122–130. Association for Computing Machinery (2023)
8. Sato, M., Sakurai, T., Burstall, R.: Explicit environments. Fundamenta Informaticae 45(1–2), 79–115 (2001)
9. Shinn, A., Cowan, J., Gleckler, A.A., et al.: Revised^7 report on the algorithmic language scheme (2013), available on: https://www.r7rs.org/
10. Laumann, O.: Elk – The Extension Language Kit Scheme Reference (1995), oliver Laumann
11. Sato, M., Sakurai, T., Kameyama, Y.: A simply typed context calculus with first-class environments. The Journal of Functional and Logic Programming 2002(3), 1–41 (2002)