論文 / 著書情報
Article / Book Information

論題(和文)	
Title(English)	Translations of First-Class Environments to Records
著者(和文)	西崎 真也
Authors(English)	Shin-ya NISHIZAKI, Yohji AKAMA
出典(和文)	, , , pp. 81-92
Citation(English)	Proceedings of First International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs, , , pp. 81-92
発行日 / Pub. date	1998, 7

Translation of first-class environments to records

Shin-ya Nishizaki * Yohji Akama †

Abstract

We have studied a calculus with first-class environments which originates from the $\lambda\sigma$-calculus. We give a translation of the calculus to a record calculus and show fundamental properties by using the translation. First we give a translation of simply typed calculus $\lambda^{\rightarrow}_{env}$ into simply-typed record calculus $\lambda^{\rightarrow}_{record}$, which is used for proving strong normalization of $\lambda^{\rightarrow}_{env}$. Second we introduce a translation of untyped calculus λ_{env} into untyped λ-calculus and investigate extensional conversion of λ_{env}-calculus with the translation.

1 Introduction

First-class environments

We treat various kinds of object in programming languages. Most of objects, e.g. integers or boolean values can be passed and returned between procedures. However, we cannot always use all of them in such a way. For example, procedures themselves are passed and returned between procedures in Lisp. In contrast, it is impossible to do so in BASIC. Objects which can be passed and returned between procedures, are said to be *first-class.*

Many implementations ([MIT], [Lau90]) of programming language Scheme enable us to utilize environments as first-class objects by the following two primitives[1]:

- `(the-environment)` which returns the current environment, and
- `(eval` ⟨ *list* ⟩ ⟨ *environment* ⟩`)` which returns the result of evaluation of the expression represented by ⟨ *list* ⟩ under ⟨ *environment* ⟩.

By these primitives, we can export an environment to anywhere independently of the textual structure of a program:

```
((lambda (env) (eval '(+ x 1) env))
 (let ((x 1)) (the-environment))).
```

The expression `(+ x 1)` is not evaluated under the environment where it appears, but under the one where `(the-environment)` appears. Therefore, the result is `2`.

This construct will support packaging of procedures (see [AS85]) or object-oriented programming [Jag94]

Moreover we expect that first-class environment mechanism gives us insight for integrating programming languages in other paradigms into a functional language, since the notion of environment is common among many programming languages. For example, since statements in imperative languages can be regarded as functions between environments, we therefore describe them in our calculus, which shows us a direction of integration of functional languages and imperative languages.

The purpose of our research is that we establish a theoretical framework for computing mechanism of first-class environment.

*Department of Mathematics and Informatics, Chiba University. (shinya@math.s.chiba-u.ac.jp)
†Department of Information Science, the University of Tokyo. (akama@is.s.u-tokyo.ac.jp)

[1] this facility is not defined in its standardization ([RC86])

Calculus with explicit substitutions. The $\lambda\sigma$-calculus[ACCL91] enables us to study λ-calculus more finely by manipulate substitutions explicitly. Environment mechanism is formalized as substitution manipulation in the $\lambda\sigma$-calculus. The syntactic class of terms and that of substitutions are separated in $\lambda\sigma$-calculus, and consequently environments are not treated as first-class environments in the calculus. For example, in the $\lambda\sigma$-calculus we cannot represent a function which takes a substitution as an argument. For example the following is not a valid term:

$$(\lambda s.a[s])((b/x)\cdot(c/y)\cdot id)$$

Hence if it is allowed to use substitutions as terms, in other words, we merge terms and substitutions, then we can obtain a calculus with first-class environments.

Reflection and reification between environments and records. In reflective programming, *reification* is a computation mechanism that transforms execution-time status into a data object, and *reflection* is its opposite direction counterpart. We therefore may say that `the-environment` allows reification of environment and `eval` reflection of environment. Data objects that environments are reified to are essentially represented as *records*, in other words, labelled products or association list. From viewpoint of reflective programming, we may say that first-class environment mechanism allows to reify and reflect environments. In many cases techniques in studies of record can applied also to those of first-class environment. In this paper we investigate the calculus with first-class environments by using a translation of calculus with first-class environment into the λ-calculus with records.

Sato-Burstall environment calculus $\lambda\epsilon$. In [SB], Sato and Burstall introduced $\lambda\epsilon$, a simply typed calculus with environments as first class values. The treatment of environment in their calculus derives from **let** declaration (SML, LISP). Their calculus lacks a mechanism for `the-environment`, while our λ_{env} has. They defined appropriate notion of free variables in their calculus with the help of their *environment types*. In their calculus, substitution can distribute over λ-binder, unlike our calculus. Their calculus satisfies not only confluence but also the conservativity over the simply typed λ-calculus.

Yoshida's weak λ-calculus λf with shared environments. In [Yos93], Yoshida formulated functional execution with shared environments. Although the calculus does not treat the environments as first-class objects, it incorporates the notion of name passing into the functional regime, treating variables in environments as channels of communication. This enables simple formulation of reduction rules under sequence of shared environments, resulting in a functional calculus whose weak reduction enjoys confluence and normalizability. And the leftmost reduction strategy of the calculus is shown to be optimal. However, the calculus is weak in a sense that no redex inside an abstraction is not allowed to be contracted.

2 Untyped environment calculus λ_{env}

We assume that a set of *variables* is given in advance of the definition of terms.

Definition 2.1 (terms) *Terms* of λ_{env} are defined inductively by the following grammar:

$$\begin{array}{rcll} M & ::= & x & \text{(variable)} \\ & | & \lambda x.M & (\lambda\text{-abstraction}) \\ & | & (MN) & \text{(function application)} \\ & | & id & \text{(identity environment)} \\ & | & (M/x)\cdot N & \text{(environment extension)} \\ & | & (M\circ N) & \text{(environment composition)} \end{array}$$

where x is a metavariable on variables and M, N on terms. The set of terms of λ_{env} is called Λ_{env} and the set of term of the usual λ-calculus is written Λ. We will regard Λ as a subset of Λ_{env} later. $(-/-)\cdot(-)$ has higher precedence than $(-)\circ(-)$: $L\circ(M/x)\cdot N$ should be read as $L\circ((M/x)\cdot N)$.

Similarly in the λ-calculus, the first three constructs, x, $\lambda x.M$ and (MN), are called a *variable,* a *lambda abstraction,* and a *function application* respectively. These intended meanings are same as the ones of the λ-calculus. The other three constructs are newly introduced in the environment calculus.

id is called the *identity environment*, which returns the current environment when it is evaluated. A construct $(M \circ N)$ is called an *environment composition*: if the term N is evaluated to an environment value, then $(M \circ N)$ evaluates to a value of the term M under the environment represented by N. The identity environment corresponds to Scheme's primitive `(the-environment)` and an environment composition $(M \circ N)$ to `(eval 'M N)`. A construct $((M/x) \cdot N)$ is a value of an environment value of N appended a binding from a variable x to a value of N. For example a term $(\lambda x.x/f) \cdot id$ is an environment where a variable f is bound to an identity function $\lambda x.x$.

Definition 2.2 (reduction) *Reduction* of λ_{env} is defined inductively by the following rules:
σ-rules:

Ass	$(L \circ M) \circ N \rightsquigarrow L \circ (M \circ N)$	VarRef	$x \circ ((M/x) \cdot N) \rightsquigarrow M$
IdL	$id \circ M \rightsquigarrow M$	VarSkip	$y \circ ((M/x) \cdot N) \rightsquigarrow y \circ N$ where $x \not\equiv y$
IdR	$M \circ id \rightsquigarrow M$	DApp	$(M_1 M_2) \circ N \rightsquigarrow (M_1 \circ N)(M_2 \circ N)$
DExtn	$((L/x) \cdot M) \circ N \rightsquigarrow ((L \circ N)/x) \cdot (M \circ N)$		

β-rules:

Beta1 $((\lambda x.M) \circ L)N \rightsquigarrow M \circ ((N/x) \cdot L)$ Beta2 $(\lambda x.M)N \rightsquigarrow M \circ ((N/x) \cdot id)$

The reduction relation determined by σ-rules is denoted by $\rightsquigarrow_\sigma$, and the reduction relation determined by β-rules is denoted by $\rightsquigarrow_\sigma$.

The reflexive-transitive closure of $\rightsquigarrow$ is written $\overset{*}{\rightsquigarrow}$.

We show how the above reduction works in the following examples:

Example (reduction) A term $\lambda x.\lambda y.id$ is a function which takes two arguments and returns the environment where x is bound to the first argument, y to the second, and the other variables are bound similarly to the environment when the term $\lambda x.\lambda y.id$ is evaluated.

$$(\lambda x.\lambda y.id)MN \rightsquigarrow ((\lambda y.id) \circ (M/x) \cdot id)N \rightsquigarrow id \circ (N/y) \cdot (M/x) \cdot id \rightsquigarrow (N/y) \cdot (M/x) \cdot id.$$

The next is more complicated example. A term $(\lambda e.x \circ e)$ represents a function that takes an environment value and returns an value of variable x under the environment. The term above is applied to it in the following example:

$$\begin{aligned}&(\lambda e.x \circ e)((\lambda x.\lambda y.id)MN) \rightsquigarrow (x \circ e) \circ ((((\lambda x.\lambda y.id)MN)/e) \cdot id)\\ &\rightsquigarrow x \circ (e \circ ((((\lambda x.\lambda y.id)MN)/e) \cdot id)) \rightsquigarrow x \circ ((\lambda x.\lambda y.id)MN)\\ &\overset{*}{\rightsquigarrow} x \circ ((N/y) \cdot (M/x) \cdot id) \overset{*}{\rightsquigarrow} M\end{aligned}$$

Proposition 2.1 (strong normalization of σ-reduction) The σ-reduction enjoys strong normalization and confluence.

Proof The strong normalization is clear since each reduction decreases the following measure:

$$\begin{array}{rclrcl} |x| &=& 1 & |id| &=& 1\\ |MN| &=& |M| + |N| + 1 & |M \circ N| &=& |M| \times (|N| + 1)\\ |\lambda x.M| &=& |M| \times 2 & |(M/x) \cdot N| &=& |M| + |N| + 1 \end{array}$$

The confluence follows by Newman's Lemman.

Theorem 2.1 $\rightsquigarrow$ is confluent.

Proof Our proof is adapted from the confluence proof of weak $\lambda\sigma$-calculus of [Cur86]. We use *interpretation method* of Curien-Hardin-Lévy [CHL96]. For it, we have proved the following:

1. If $M \rightsquigarrow_\beta M'$, then $\sigma(M)$ $(\rightsquigarrow_\beta \overset{!}{\rightsquigarrow}_\sigma)^*$ $\sigma(M')$.

   Here $\overset{!}{\rightsquigarrow}_\sigma$ is a σ-reduction to a normal form.

2. $\rightsquigarrow_\beta \overset{!}{\rightsquigarrow}_\sigma$ is confluent on a set of σ-normal terms.

   The proof is by a parallel reduction method.

3 Simply-typed calculus $\lambda^{\rightarrow}_{env}$

In this section we show a simply-typed calculus with first-class environments, written $\lambda^{\rightarrow}_{env}$.

We assume that a set *TypeVar* of *type variables*, which has countably infinite elements, are given. We next define a set of types of our calculus and its subset which called *environment types*.

Definition 3.1 (types and environment types) *Environment types*[2] and *types* are defined inductively by the following grammar:

$$\begin{array}{lcll} E & ::= & \{x_1 : A_1\} \cdots \{x_n : A_n\} & \text{(environment type)} \\ A & ::= & \alpha \mid A \to B \mid E & \text{(type)} , \end{array}$$

where E is a metavariable on environment types and A and B on types. We impose the following conditions on the environment types and types: **(i)** We do not distinguish environment types by order of $\{x_i : A_i\}$-bindings, in other words, an environment type is a pair of a finite set of $\{x_i : A_i\}$-bindings. **(ii)** The variables x_i in an environment type are distinct, i.e. $x_i \neq x_j$ for any $i \neq j$. *EnvType* is the set of all environment types and *Type* is the set of all types.

It is trivially true that *EnvType* is a subset of *Type*.

The condition **(i)** enables us to identify environments which have distinct orders of bindings: for example, we do not distinguish $\{x : \alpha\}\{y : \beta\}$ and $\{y : \beta\}\{x : \alpha\}$. Of course, we can define the calculus without the condition **(i)**. In that case, however, an environment cannot be regarded as a correspondence between variable identifiers and their values.

Typability of $\lambda^{\rightarrow}_{env}$ is defined by the following typing judgements:

$$\frac{}{\{x : A\}E \vdash x : A}\ \mathsf{Var}$$

$$\frac{\{x : A\}E \vdash M : B}{\{x : C\}E \vdash \lambda x.M : A \to B}\ \mathsf{Lam} \qquad \frac{E \vdash M : A \to B \quad E \vdash N : A}{E \vdash MN : B}\ \mathsf{App}$$

$$\frac{}{E \vdash id : E}\ \mathsf{Id}$$

$$\frac{E \vdash N : E' \quad E' \vdash M : A}{E \vdash M \circ N : A}\ \mathsf{Comp} \qquad \frac{E \vdash M : A \quad E \vdash N : \{x : C\}E'}{E \vdash (M/x) \cdot N : \{x : A\}E'}\ \mathsf{Extn}$$

Intuitively $E \vdash M : A$ means that if we evaluate M under an environment of type E then we get a value of type A. The six rules described above are easily understood except two rules Lam and Extn. The typing rule of λ-abstraction in the classical λ-calculus is

$$\frac{\{x : A\}E \vdash M : B}{E \vdash \lambda x.M : A \to B}\ \mathsf{Lam'}.$$

If we adopt this rule instead of the former one, then the typability given by Lam' becomes strictly weaker than that given by Lam: for example, consider typing of a term $\lambda x.\lambda x.x$. This term cannot be typed by Lam' but can be typed by Lam:

$$\dfrac{\dfrac{\dfrac{}{\{x : \beta\}\{x : \alpha\} \vdash x : \beta}\ \mathsf{Var}}{\{x : \alpha\} \vdash \lambda x.x : \beta \to \beta}\ \mathsf{Lam'}}{\vdash \lambda x.\lambda x.x : \alpha \to \beta \to \beta}\ \mathsf{Lam'}$$

Type derivation tree (1)

$$\dfrac{\dfrac{\dfrac{}{\{x : \beta\} \vdash x : \beta}\ \mathsf{Var}}{\{x : \alpha\} \vdash \lambda x.x : \beta \to \beta}\ \mathsf{Lam}}{\{x : \gamma\} \vdash \lambda x.\lambda x.x : \alpha \to \beta \to \beta}\ \mathsf{Lam}$$

Type derivation tree (2)

The tree (1) is not valid since it it is inevitable that an environment type not satisfied the conditions **(i)** and **(ii)** appears.

We finish this section with an example of typing:

Example The term $\lambda x.\lambda y.id$ is typed as

$$\vdash (\lambda x.\lambda y.id) : \alpha \to \beta \to \{y : \beta\}\{x : \alpha\}.$$

[2] Environment type given in the previous works [Nis95a] are extensible by using a type variable for environment types. We omit such notion for simplicity.

4 Translation of typed calculus

In this section we give a translation of the simply-typed calculus $\lambda^{\rightarrow}_{\bar{e}nv}$ into a simply-typed λ-calculus with records, and by using this translation, we can show strong normalization of $\lambda^{\rightarrow}_{\bar{e}nv}$. We introduce a translation of $\lambda^{\rightarrow}_{\bar{e}nv}$ in the first part of this section and show strong normalization in the second part.

4.1 Record calculus $\lambda^{\rightarrow}_{record}$

We here introduce a simply-typed λ-calculus $\lambda^{\rightarrow}_{record}$ with records. Its type system is given as *explicit typing*, i.e. Church's style.

We assume that two sets of identifiers are given: one is a set of *variables* and another a set of *labels*. We first define *types* of $\lambda^{\rightarrow}_{record}$ inductively as follows:

$$A ::= \alpha \mid A \rightarrow B \mid \{l_1 : A_1, \ldots, l_n : A_n\}$$

The *raw terms* of $\lambda^{\rightarrow}_{record}$ are defined by the following grammar:

$$M ::= x^A \mid \lambda x{:}A.M \mid (MN) \mid \langle\,\rangle \mid \langle l = M \mid N\rangle \mid M.l \mid M\backslash l$$

These seven kinds of terms are called a *variable*, a *λ-abstraction*, a *function application*, an *empty record*, a *record extension*, a *record reference*, and a *record restriction*, respectively.

A *type assignment* is a function whose domain is a finite set of variables and whose codomain is a (finite) set of types. A type assignment that maps $x_1, \ldots, x_n$ to $A_1, \ldots, A_n$ respectively, is written $x_1 : A_1 \cdots x_n : A_n$.

The next seven rules are typing rules of $\lambda^{\rightarrow}_{record}$:

$$\frac{1 \leq i \leq n}{x_0 : A_0 \cdots x_n : A_n \vdash {x_i}^{A_i} : A_i} \qquad \frac{x : A\ \Gamma \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \rightarrow B} \qquad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{}{\Gamma \vdash \langle\,\rangle : \{\,\}} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \{l_1 : A_1, \ldots, l_n : A_n\}}{\Gamma \vdash \langle l = M \mid N\rangle : \{l : A, l_1 : A_1, \ldots, l_n : A_n\}}$$

$$\frac{\Gamma \vdash M : \{l_1 : A_1, \ldots, l_n : A_n\}}{\Gamma \vdash M.l_i : A_i} \qquad \frac{\Gamma \vdash M : \{l_1 : A_1, \ldots, l_n : A_n\}}{\Gamma \vdash M\backslash l_i : \{l_1 : A_1, \ldots, l_{i-1} : A_{i-1}, l_{i+1} : A_{i+1}, \ldots, l_n : A_n\}}$$

A *typed term*, say M, is a raw term which satisfies typing judgement $\Gamma \vdash M : A$ for some Γ and A defined above.

Reduction of $\lambda^{\rightarrow}_{record}$ given by the following rules:

$$\begin{array}{llll}
\beta & (\lambda x{:}A.M)N \underset{\text{record}}{\rightarrow} [N\backslash x]M, & & \\
\mathsf{RefHit} & \langle l = M \mid N\rangle.l \underset{\text{record}}{\rightarrow} M, & \mathsf{RefSkip} & \langle l' = M \mid N\rangle.l \underset{\text{record}}{\rightarrow} N.l, \\
\mathsf{RestHit} & \langle l = M \mid N\rangle\backslash l \underset{\text{record}}{\rightarrow} N, & \mathsf{RestSkip} & \langle l' = M \mid N\rangle\backslash l \underset{\text{record}}{\rightarrow} \langle l' = M \mid N\backslash l\rangle
\end{array}$$

Theorem 4.1 (strong normalization of $\lambda^{\rightarrow}_{record}$) $\lambda^{\rightarrow}_{record}$ enjoys strong normalization.

This theorem is proved with reducibility method [GTL89]: the detailed proof can be found in the previous paper [Nis95a].

We next give a translation of the simply-typed calculus $\lambda^{\rightarrow}_{\bar{e}nv}$ in the record calculus $\lambda^{\rightarrow}_{record}$; we will define a translation of types and a translation of terms. For simplicity we will use a notation $[\![-]\!]$ for these translations, since we have no danger of confusion. We assume that a one-to-one mapping $\lceil - \rceil$ from variables (in $\lambda^{\rightarrow}_{\bar{e}nv}$) to labels (in $\lambda^{\rightarrow}_{record}$). We then define a translation mapping $[\![-]\!]$ from types of $\lambda^{\rightarrow}_{\bar{e}nv}$ to types of $\lambda^{\rightarrow}_{record}$:

$$\begin{array}{l}
[\![\{x_1 : A_1\} \cdots \{x_n : A_n\}]\!] = \{[\![x_1]\!] : [\![A_1]\!], \ldots, [\![x_n]\!] : [\![A_n]\!]\} \\
[\![\alpha]\!] = \alpha \\
[\![A \rightarrow B]\!] = [\![A]\!] \rightarrow [\![B]\!].
\end{array}$$

Actually the translation of terms is not given as a mapping from types but as a mapping from type derivation tree of a typed term. This will cause no problem since each typed term of $\lambda^{\rightarrow}_{record}$ has a type derivation tree *uniquely*.

The translation of terms is defined inductively as a function that maps a pair of a type derivation tree of $\lambda^{\rightarrow}_{\overrightarrow{env}}$ and a $\lambda^{\rightarrow}_{record}$-term to a $\lambda^{\rightarrow}_{record}$-term, as follows:

$$
\begin{array}{l}
[\![\{x:A\}E \vdash x:A]\!](L) = L.\lceil x \rceil \\
[\![\{x:C\}E \vdash \lambda x.M : A \to B]\!](L) = \lambda v{:}[\![A]\!].[\![\{x:A\}E \vdash M : B]\!](\langle \lceil x \rceil = v^{[\![A]\!]} \mid L\rangle) \\
[\![E \vdash MN : B]\!](L) = [\![E \vdash M : A \to B]\!](L)\ [\![E \vdash N : A]\!](L) \\
[\![E \vdash id : E]\!](L) = L \\
[\![E \vdash M \circ N : A]\!](L) = [\![H \vdash M : A]\!]([\![E \vdash N : H]\!](L)) \\
[\![E \vdash (M/x)\cdot N : \{x:A\}E]\!](L) = \langle \lceil x \rceil = [\![E \vdash M : A]\!](L) \mid [\![E \vdash N : \{x:C\}E]\!](L)\rangle
\end{array}
$$

where L is a term of $\lambda^{\rightarrow}_{record}$. When there is no danger of confusion, we will abbreviate $[\![E \vdash M : A]\!](L)$ as $[\![M]\!](L)$.

Typing of a term is preserved during the translation:

Proposition 4.1 (typing preservation of translation) The translation $[\![-]\!](-)$ preserves typing: for any $E \vdash M : A$ and $\Gamma \vdash L : [\![E]\!]$, then a term $[\![E \vdash M : A]\!](L)$ is well-typed as $\Gamma \vdash [\![E \vdash M : A]\!](L) : [\![A]\!]$.

The proposition is proved by induction on the structure of the translation, which is checked easily.

Lemma 4.1 Let L be a λ-term. If $M \rightsquigarrow N$ then either $[\![M]\!](L) \underset{\text{record}}{\rightarrow} [\![N]\!](L)$ or $[\![M]\!](L) \equiv [\![N]\!](L)$ holds. Especially if M is reduced to N with Beta1 or Beta2, then $[\![M]\!](L)$ is reduced to $[\![N]\!](L)$ with β-reduction in $\lambda^{\rightarrow}_{record}$.

We should notice that the above lemma says implicitly that the translation $[\![-]\!](-)$ never deletes any redexes of Beta1 or Beta2. The proof is in Appendix A.

Theorem 4.2 (strong normalization of $\lambda^{\rightarrow}_{\overrightarrow{env}}$) The simply-typed lambda calculus $\lambda^{\rightarrow}_{\overrightarrow{env}}$ with first-class environments enjoys strong normalization of the reduction $\rightsquigarrow$.

Proof Suppose that there exists an infinite reduction sequence:

$$M_1 \rightsquigarrow M_2 \rightsquigarrow M_3 \rightsquigarrow \cdots.$$

By Proposition 2.1, this sequence has to include infinite steps of Beta1 or Beta2. On the other hand, the sequence is translated as follows:

$$[\![M_1]\!]e \overset{*}{\rightsquigarrow} [\![M_2]\!]e \overset{*}{\rightsquigarrow} [\![M_2]\!]e \overset{*}{\rightsquigarrow} \cdots,$$

for a variable e. By Proposition 4.1, the translated reduction sequence has to include infinite steps of β, which contradicts to Proposition 4.1.

5 Translation of untyped calculus

In this section we show that the translation of environments in records can be also applied to untyped calculus in order to investigate properties of untyped calculus λ_{env}. We use the classical untyped λ-calculus instead of λ-calculus with records: record labels are represented by Church numerals and records by λ-terms.

It is assumed that each variable corresponds uniquely to a Church numeral, i.e. there exists a one-to-one correspondence $\lceil - \rceil$ from each variable to a Church numeral is given.

The translation given in this section uses classical λ-terms for representing records, instead of λ-calculus with records: a record extension $\langle \lceil x \rceil = M \mid N \rangle$ is represented by (**update** N $\lceil x \rceil$ M) and a record reference $M.\lceil x \rceil$ by (**lookup** M $\lceil x \rceil$), where **update** and **lookup** are λ-terms which appropriately given:

Proposition 5.1 (update and lookup) There exist closed λ-terms **update** and **lookup** satisfying the following equations:

$$
\begin{array}{lcl}
(\textbf{lookup}(\textbf{update}\ M\ \lceil x \rceil\ N)\ \lceil x \rceil) & \underset{\beta}{=} & N \\
(\textbf{lookup}(\textbf{update}\ M\ \lceil x \rceil\ N)\ \lceil y \rceil) & \underset{\beta}{=} & (\textbf{lookup}\ M\ \lceil y \rceil).
\end{array}
$$

where $x \not\equiv y$.

A translation of untyped terms of λ_{env} is defined similarly as follows:

Definition 5.1 (Translation $[\![-]\!](-)$ for untyped calculus) An *translation* $[\![-]\!](-)$ is a function which maps a λ_{env}-term M and a λ-term ρ to a λ-term $[\![M]\!](\rho)$ defined inductively as follows:

$$\begin{array}{rcl@{\qquad}rcl} [\![x]\!]\rho & \equiv & (\mathbf{lookup}\ \rho\ \ulcorner x \urcorner) & [\![id]\!]\rho & \equiv & \rho \\ [\![\lambda x.M]\!]\rho & \equiv & \lambda v.[\![M]\!](\mathbf{update}\ \rho\ \ulcorner x \urcorner\ v) & [\![(M/x)\cdot N]\!]\rho & \equiv & \mathbf{update}\ ([\![N]\!]\rho)\ \ulcorner x \urcorner\ ([\![M]\!]\rho) \\ [\![(MN)]\!]\rho & \equiv & ([\![M]\!]\rho)([\![N]\!]\rho) & [\![M \circ N]\!]\rho & \equiv & [\![M]\!]([\![N]\!]\rho) \end{array}$$

where the variable v does not appear in M and ρ.

We know that the combinatory logic CL_w with weak equality is strictly weaker than β-equality of λ-calculus, however, if we incorporate the extensionality axiom

$$\frac{\forall Z.\ (M\ Z) = (N\ Z)}{M = Z}\ ext$$

into the equivalence, then the equivalence recovers the same power as $\beta\eta$-equivalence. The fact arouses our interest in an equivalence that includes both $\rightsquigarrow$ and an extensionality axiom; such an equivalence includes the β-equivalence of the classical λ-calculus, even though we do not have any α-equivalence.

Definition 5.2 (extensional equivalence) An *extensional equivalence* $\underset{ext}{\approx}$ is the least equivalence relation which includes the reduction relation $\rightsquigarrow$ and satisfies the *extensional axiom*:

$$\frac{\forall Z \in \Lambda_{env}.\ (MZ) \underset{ext}{\approx} (NZ)}{M \underset{ext}{\approx} N}\ \mathsf{ext}$$

Another equivalence relation $\underset{\omega}{\approx}$ is defined similarly with the following axiom instead of the axiom ext:

$$\frac{\forall Z \in \Lambda^0.\ (MZ) \underset{\omega}{\approx} (NZ)}{M \underset{\omega}{\approx} N}\ \omega$$

where Λ^0 is the set of all closed λ-terms.

The following property is trivial by the definitions of these equivalences:

Proposition 5.2 Let M and N be λ_{env}-terms. If $M \underset{ext}{\approx} N$, then $M \underset{\omega}{\approx} N$.

Distribution of a substitution into a λ-abstraction without α-equivalence

We cannot derive equivalence between $(\lambda x.\lambda y.x)y$ and $\lambda y'.y$ under the equality generated by the reduction $\rightsquigarrow$, since we have no distributivity axiom of environment composition into λ-abstraction. However if we have the extensionality axiom, then the extensionality axiom makes such distribution possible; for example, it is derived from the extensionality axiom that two terms are equivalent:

$$\begin{array}{ll} & (\lambda x.\lambda y.x)y \underset{ext}{\approx} \lambda y'.y \\ \Leftarrow & \forall Z \in \Lambda_{env}.((\lambda x.\lambda y.x)y)Z \underset{ext}{\approx} (\lambda y'.y)Z \qquad \text{(by rule ext)} \\ \Leftarrow & \forall Z \in \Lambda_{env}.x \circ ((y/x)\cdot(Z/y)\cdot id) \underset{ext}{\approx} y \circ ((Z/y')\cdot id) \\ \Leftarrow & \forall Z \in \Lambda_{env}.y \underset{ext}{\approx} y. \end{array}$$

Since first-class environment mechanism makes α-conversion difficult, we developed our calculus without it. So we cannot introduce distribution axiom of environment composition into λ-abstraction without renaming, which makes the reduction weak one, consequently. Though not quite satisfactorily, we can use extensionality axiom as a substitute. This is another motivation of investigation about the extensional equivalence.

In the rest of this section we show consistency of extensional equivalence of λ_{env} and comparison of strength between the equivalence of λ_{env} and the equivalence of the classical λ-calculus, as another application of the translation of environments given previously.

Lemma 5.1 Let M and ρ be λ-terms such that $(\mathbf{lookup}\ \rho\ \lceil x \rceil) \underset{\beta}{=} x$ for any free variable x in the λ-term M. Then $[\![M]\!]\rho \underset{\beta}{=} M$. In particular, for any closed λ-term M, $[\![M]\!]\rho \underset{\beta}{=} M$.

The proof is given in Appendix B.

By using this lemma we obtain the following property:

Proposition 5.3 (Soundness of the extensional equivalence) Let M and N be λ_{env}-terms such that $M \underset{\omega}{\approx} N$. For any λ-term ρ, $[\![M]\!]\rho \underset{\beta\omega}{=} [\![N]\!]\rho$ holds. In particular, if $M \underset{ext}{\approx} N$. then $[\![M]\!]\rho \underset{\beta\omega}{=} [\![N]\!]\rho$ holds.

The following theorem is derived from the proposition proved above.

Theorem 5.1 (consistency of the extensional equivalence) There are two different terms under the extensional conversion, i.e.

$$\exists M, N \in \Lambda_{env}.\neg(M \underset{ext}{\approx} N).$$

Proof Suppose that the conclusion is false, i.e. for any λ_{env}-term M and N, $M \underset{ext}{\approx} N$.

Consider the case that $M \equiv \lambda f.\lambda x.x$ and $N \equiv \lambda f.\lambda x.fx$. By the assumption above, $\lambda f.\lambda x.x \underset{ext}{\approx} \lambda f.\lambda x.fx$. By Proposition 5.3, we then obtain that $\lambda f.\lambda x.x \underset{\beta\omega}{=} \lambda f.\lambda x.fx$. However this contradicts the consistency of $\beta\omega$-conversion.
qed.

In addition to the above theorem we also obtain the following proposition:

Proposition 5.4 For $M, N \in \Lambda$, if $M \underset{ext}{\approx} N$ then $M \underset{\beta\omega}{=} N$.

Proof If we take a λ-term ρ satisfying that

$$(\mathbf{lookup}\ \rho\ \lceil x \rceil) \underset{\beta}{=} x$$

for any variables occurring in either M or in N, this is derived from Proposition 5.3 by using Lemma 5.1.

The property in the opposite direction of the above proposition also holds:

Theorem 5.2 For λ-terms M and N, if $M \underset{\beta\omega}{=} N$ then $M \underset{\omega}{\approx} N$.

This theorem is proved by showing that each axiom of $\underset{\beta\omega}{=}$ is derived over $\underset{\omega}{\approx}$. The proof is in Appendix D.

6 Concluding remarks

We introduced an untyped calculus with first-class environments and its simply typed version and showed a method with translations of environments into records. The method is originally introduced in order to prove strong normalization of simply typed calculus with first-class environments in [Nis95a]. The type system there is different to the one in this paper (and in the paper [Nis95b]): we could not type a term with shadowing among bindings, e.g. $\lambda x.\lambda x.x$. Actually the method works well also in the type system in this paper. Moreover, we tried to use the translation in order to investigate relationship between the untyped calculus with first-class environments and the classical λ-calculus.

In the rest of this section we comment on several points in related topics and a future direction.

Translation for strong calculus

It is shown that the typed $\lambda\sigma$-calculus does not enjoy strong normalization as Melliès showed in [Mel95], and therefore, our method fails to be applied to strong calculus.

Although we presented only the weak calculus, we can extend it to strong one by introducing shift operator $\uparrow x$ with a name in the calculus, similarly to the $\lambda\sigma$-calculus:

$$\begin{array}{ll} \text{(term)} & M ::= \cdots \mid \uparrow_x \\ \text{(reduction rule for shift)} & \uparrow_x \circ (M/x) \cdot N \rightsquigarrow N \\ & \uparrow_x \circ (M/y) \cdot N \rightsquigarrow (M/y) \cdot \uparrow_x \circ N \\ \text{(translation for shift)} & [\![\uparrow_x]\!](L) = L \backslash \lceil x \rceil \end{array}$$

The distributivity axiom of environment composition into λ-abstraction is formulated as

$$(\lambda x.M) \circ N \rightsquigarrow \lambda x.(M \circ ((x/x) \cdot (N \circ \uparrow_x)))$$

If we translate these two terms, then we find that the result from the right-hand side is reduced to the one from the left-hand side; therefore we fails to prove strong normalization of the strong calculus.

Translation for the other weak calculi

The method with the translation from environments to records can be also applied to the weak calculus $\lambda\sigma_w$ with explicit substitutions [CHL96] and the weak calculus $W\beta$ of categorical combinators [Har87]. The translation of the former is obtained similarly, if we read a closure $a[s]$ as an environment composition $a \circ s$. The translation of the latter is defined, if we use a λ-calculus with pairs instead of the record calculus $\lambda^{\rightarrow}_{record}$. We can obtain strong normalization for these calculi by using such translations.

Future direction

Until now, we have not studied the nameless calculus with first-class environments. There seems be an essential gap between the calculus with names and the nameless calculus: de Bruijn-transformation from terms with names to nameless terms fails to be defined in the calculus with first-class environments. Nevertheless, the nameless calculus with first-class environments itself is interesting to be worth studying. We investigated extensional equivalence of λ_{env} in this paper; if the nameless calculus is developed, the works in the substitution calculi (e.g. [Kes96]) will be helpful for our study.

We are planing to prove the standardization theorem for λ_{env}. Then we will investigate efficient evaluation strategies of λ_{env}, in order to design abstract machines (e.g. SECD machines). Then, we will prove the usefulness of our calculus by developing the programming (transformation) techniques.

Acknowledgement

The authors thank Pierre-Louis Curien, Thérèse Hardin, Delia Kesner, Masami Hagiya, and Masahiko Sato for fruitful discussions.

References

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[CHL96] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.

[Cur86] Pierre-Louis Curien. Categorical combinators. *Information and Control*, 69:188–254, 1986.

[GTL89] Jean-Yves Girard, P. Taylor, and Yves Lafont. *Proofs and Types*, volume 7 of *Cambridge Tracts in Computer Science*. Cambridge University Press, 1989.

[Har87] Thérèse Hardin. *Résultats de Confluence pour les Règles Fortes de la Logique Combinatoire Catégorique et Liens avec les Lambda-Calculs.* PhD thesis, L'université Paris VII, October 1987.

[Jag94] Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems*, 16(3):456–492, 1994.

[Kes96] Delia Kesner. Confluence properties of extensional and non-extensional lambda calculi with explicit substitutions. In *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (Lecture Notes in Computer Science. vol. 1103)*, pages 184–199. Springer, 1996.

[Lau90] Oliver Laumann. *Reference Manual for the Elk Extension Language Interpreter*, 1990.

[Mel95] Paul-André Melliès. Typed λ-calculi with explicit substitutions may not terminate. In *Second International Conference on Typed Lambda Calculi and Applications, TLCA '95 (Lecture Notes in Computer Science. vol. 902)*. Springer-Verlag, 1995.

[MIT] MIT. *MIT Scheme Reference Manual.*

[Nis95a] Shin-ya Nishizaki. Simply typed lambda calculus with first-class environments. *Publication of Research Institute for Mathematical Sciences Kyoto University*, 30(6):1055–1121, 1995.

[Nis95b] Shin-ya Nishizaki. Type inference for simply-typed environment calculus with shadowing. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming*, 1995.

[RC86] J. Rees and W. Clinger. Revised^3 repord on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[SB] Masahito Sato and Rod Burstall. Explicit environments (extended abstract). 1998.

[Yos93] Nobuko Yoshida. Optimal reduction in weak-λ-calculus with shared environments. In *6th ACM Conference on Functional Programming Languages and Computer Architecture*, 1993.

A Proof of Lemma 4.1

Proposition A.1 Let $M[\ ]$ be a context (i.e. a λ_{env}-term with a hole), and ρ a λ-term. For a λ_{env}-term N, there exists a λ-term ρ' such that $[\![M[N]]\!]\rho$ has $[\![N]\!]\rho'$ as a subterm.

By this proposition, it is sufficient that we check the cases that a redex appears at the root.

If M is reduced to N with one of the rules Ass, IdL, IdR, DExtn, and Dapp, then $[\![M]\!]\rho \equiv [\![N]\!]\rho$.

If M is reduced to N with the rule VarRef, then $[\![M]\!]\rho$ is reduced to $[\![N]\!]\rho$ in $\lambda^{\rightarrow}_{record}$ with the rule RefHit. Similarly VarSkip corresponds to RefSkip.

If M is reduced to N with either Beta1 or Beta2, then $[\![M]\!]\rho$ is reduced to $[\![N]\!]\rho$ with β.

qed.

B Proof of Lemma 5.1

We need the following lemma in order to prove Lemma 5.1:

Lemma B.1 Let M an λ_{env}-term. For any λ-terms ρ and N and any variable x,

$$([\![M]\!]\rho)[x{:=}N] \equiv [\![M]\!](\rho[x{:=}N])$$

holds, where $[v{:=}N]$ is a substitution of N for variable v, which is defined in the λ-calculus.

Variables in $([\![M]\!]\rho)$ are come from ρ only; the variables in M are translated into terms like $(\mathbf{lookup}\rho\lceil x\rceil)$. The proof is done straight-forwardly by induction on the structure of the term M.

Proof The proof is by induction on M: the case $M \equiv x$: by the assumption itself.
The case $M \equiv \lambda x.M'$:

$$\begin{aligned}
&[\![\lambda x.M']\!]\rho \\
&\quad \equiv \quad \lambda v.([\![M']\!](\mathbf{update}\ \rho\ \lceil x \rceil\ v)) \\
&\quad \equiv \quad \lambda v.(([\![M']\!](\mathbf{update}\ \rho\ \lceil x \rceil\ v))[v{:=}x][x{:=}v]) \\
&\quad \equiv \quad \lambda v.(([\![M']\!](\mathbf{update}\ \rho\ \lceil x \rceil\ x))[x{:=}v]) \\
&\quad \equiv \quad \lambda v.(M'[x{:=}v]) \\
&\qquad\quad \text{by the induction hypothesis for } M', \\
&\quad \equiv \quad \lambda x.M' \\
&\quad \equiv \quad M
\end{aligned}$$

The case $M \equiv (M_1 M_2)$:

$$\begin{aligned}
&[\![M_1 M_2]\!]\rho \\
&\quad \equiv \quad [\![M_1]\!]\rho[\![M_2]\!]\rho \\
&\quad \equiv \quad (M_1 M_2) \\
&\qquad\quad \text{by the induction hypothesis for } M_1 \text{ and that for } M_2.
\end{aligned}$$

C Proof of Proposition 5.3

Proof The proof of the first part is by induction on the structure of $\underset{ext}{\approx}$.

The cases of the reflexivity axiom, of the symmetricity axiom, and of the transitivity axiom are trivial.

In the cases of Ass,IdL,IdR,DExtn,and DApp, It is easily shown that $M \rightsquigarrow N$ implies $[\![M]\!]\rho \equiv [\![N]\!]\rho$.

The cases of VarRef: let $M \rightsquigarrow N$ be an instance of VarRef, i.e. $M \equiv x \circ ((M_1/x) \cdot M_2)$ and $N \equiv M_1$. Then for arbitrary λ-term ρ,

$$\begin{aligned}
&[\![x \circ (M_1/x) \cdot M_2]\!]\rho \\
&\quad \equiv \quad [\![x]\!](\mathbf{update}\ [\![M_2]\!]\rho\ \lceil x \rceil\ [\![M_1]\!]\rho) \\
&\quad \equiv \quad (\mathbf{lookup}\ (\mathbf{update}\ [\![M_2]\!]\rho\ \lceil x \rceil\ [\![M_1]\!]\rho)\ \lceil x \rceil) \\
&\quad \underset{\beta}{=} \quad [\![M_1]\!]\rho.
\end{aligned}$$

The case of VarSkip: this case is proved similarly to the previous case.

The case of Beta1: let $M \rightsquigarrow N$ be an instance of Beta1, i.e.

$$\begin{aligned}
M &\equiv ((\lambda x.M_1) \circ M_2)M_3; \\
N &\equiv M_1 \circ ((M_3/x) \cdot M_2).
\end{aligned}$$

Then for arbitrary λ-term ρ,

$$\begin{aligned}
&[\![((\lambda x.M_1) \circ M_2)M_3]\!]\rho \\
&\quad \equiv \quad (\lambda v.[\![M_1]\!](\mathbf{update}\ [\![M_2]\!]\rho\ \lceil x \rceil\ v))\ [\![M_3]\!]\rho \\
&\quad \underset{\beta}{=} \quad [\![M_1]\!](\mathbf{update}\ [\![M_2]\!]\rho\ \lceil x \rceil\ v)[v{:=}[\![M_3]\!]\rho] \\
&\quad \equiv \quad [\![M_1]\!](\mathbf{update}\ [\![M_2]\!]\rho\ \lceil x \rceil\ [\![M_3]\!]\rho) \\
&\qquad\quad \text{(by Lemma B.1)} \\
&\quad \equiv \quad [\![M_1 \circ ((M_3/x) \cdot M_2)]\!]\rho
\end{aligned}$$

Notice that the variable v does not appear in λ-terms **update**, $[\![M_2]\!]\rho$, and $\lceil x \rceil$. The case of Beta2: similarly proved to the case of Beta1.

The case of the axiom ω: let M and N be λ_{env}-terms and ρ a λ-term such that

$$\forall Z \in \Lambda^0.\ \forall \rho \in \Lambda.\ [\![MZ]\!]\rho \underset{\beta\omega}{=} [\![NZ]\!]\rho.$$

Then

$$\begin{array}{ll} & \forall Z \in \Lambda^0.\ \forall \rho \in \Lambda.\ (\llbracket M \rrbracket \rho)(\llbracket Z \rrbracket \rho) \underset{\beta\omega}{=} (\llbracket N \rrbracket \rho)(\llbracket Z \rrbracket \rho) & \\ \Rightarrow & \forall Z \in \Lambda^0.\ \forall \rho \in \Lambda.\ (\llbracket M \rrbracket \rho) Z \underset{\beta\omega}{=} (\llbracket N \rrbracket \rho) Z & \text{(by Lemma 5.1)} \\ \overset{(1)}{\Rightarrow} & \forall \rho \in \Lambda.\ \llbracket M \rrbracket \rho \underset{\beta\omega}{=} \llbracket N \rrbracket \rho & \text{(by } \omega\text{-rule)} \end{array}$$

The second part is derived from the fact that if $M \underset{ext}{\approx} N$ then $M \underset{\beta\omega}{=} N$, which is trivial by the definitions of $\underset{ext}{\approx}$ and $\underset{\omega}{\approx}$. **qed.**

You may with to prove that

if $M \underset{ext}{\approx} N$ then $M \underset{\beta ext}{=} N$, i.e. $M \underset{\beta\eta}{=} N$.

However this statement cannot be proved by the above method: If we want to use the extensionality axiom like the step (1), $(\llbracket Z \rrbracket \rho)$ must take enough arbitrary value. ρ is not a function on variables, but a value in the semantic domain, which is a result from introducing first-class environments into the calculus. We know that $(\llbracket Z \rrbracket \rho)$ covers all over closed λ-terms if we take closed λ-terms for Z, by Lemma 5.1. At the moment we do not know the range of $(\llbracket Z \rrbracket \rho)$ for arbitrary λ_{env}-term Z; if it covers all over λ-terms, then the statement above-mentioned will hold.

D Proof of Theorem 5.2

We need the following properties as preparation for proving Theorem 5.2.

Proposition D.1 If $x_i \not\equiv x_{i+1}$,

$$\begin{array}{l} M \circ (N_1/x_1) \cdot \cdots (N_i/x_i) \cdot (N_{i+1}/x_{i+1}) \cdot \cdots (N_n/x_n) \cdot L \\ \underset{ext}{\approx} \quad M \circ (N_1/x_1) \cdot \cdots (N_{i+1}/x_{i+1}) \cdot (N_i/x_i) \cdot \cdots (N_n/x_n) \cdot L. \end{array}$$

If $x_j \equiv x_{j+1}$,

$$\begin{array}{l} M \circ (N_1/x_1) \cdot \cdots (N_j/x_j) \cdot (N_{j+1}/x_{j+1}) \cdot \cdots (N_n/x_n) \cdot L \\ \underset{ext}{\approx} \quad M \circ (N_1/x_1) \cdot \cdots (N_j/x_j) \cdot (N_{j+2}/x_{j+2}) \cdot \cdots (N_n/x_n) \cdot L. \end{array}$$

Proposition D.2 If x does not occur freely in a λ-term M, we know that

$$M \circ (N/x) \cdot id \underset{ext}{\approx} M.$$

Lemma D.1 Substitution operation in the λ-calculus can be simulated by the conversion $\underset{\omega}{\approx}$, i.e. for any λ-terms M and N and any variable x,

$$M \circ (N/x) \cdot id \underset{ext}{\approx} M[x{:=}N].$$

Proof (Proposition 5.2) The proof is done by induction on the structure of derivation tree of $M \underset{\beta\omega}{=} N$.

The key parts of the proof are in the case that $M \underset{\beta\eta}{=} N$ is derived from β-rule and the case that $M \underset{\beta\eta}{=} N$ is derived from η-rule.

The case of the β-rule: assume that M is $(\lambda x.M_1)M_2$ and N $M_1[x{:=}M_2]$. Then the following equations hold:

$$\begin{array}{lll} M & \underset{\omega}{\approx} & M_1 \circ (M_1/x) \cdot id \\ & \underset{\omega}{\approx} & N \qquad\qquad \text{by Lemma D.1} \end{array}$$

The case of the axiom ω: let M and N be λ-terms and ρ a λ-term such that $\forall Z \in \Lambda^0.\ MZ \underset{\beta\omega}{=} NZ$. By the induction hypothesis, we have $\forall Z \in \Lambda^0.\ MZ \underset{\omega}{\approx} NZ$. Then from the axiom ω of $\underset{\omega}{\approx}$, it is derived that $M \underset{\omega}{\approx} N$. **qed.**