A Meta Lambda Calculus with Cross-Level Computation

Kazunori Tobisawa
Graduate School of Mathematical Sciences, The University of Tokyo, Japan
tobisawa@ms.u-tokyo.ac.jp

Abstract

We propose meta lambda calculus λ^* as a basic model of textual substitution via metavariables. The most important feature of the calculus is that every β-redex can be reduced regardless of whether the β-redex contains meta-level variables or not. Such a meta lambda calculus has never been achieved before due to difficulty to manage binding structure consistently with α-renaming in the presence of meta-level variables. We overcome the difficulty by introducing a new mechanism to deal with substitution and binding structure in a systematic way without the notion of free variables and α-renaming.

Calculus λ^* enables us to investigate cross-level terms that include a certain type of level mismatch. Cross-level terms have been regarded as meaningless terms and left out of consideration thus far. We find that some cross-level terms behave as quotes and 'eval' command in programming languages. With these terms, we show a procedural language as an application of the calculus, which sheds new light on the notions of stores and recursion via meta-level variables.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic – lambda calculus and related systems

Keywords lambda calculus; metavariables; textual substitution; dynamic binding

1. Introduction

1.1 Metavariables and Meta Lambda Calculi

When discussing a formal language, we use metavariables. A metavariable is a symbol that stands for some syntactic object in a discussed language. For example, we use the following expression when defining β-reduction of the lambda calculus:

"$(\lambda x.M)N \to_\beta M\{N/x\}$ for variables x and terms M, N."

In this case, 'x', 'M' and 'N' are metavariables. If we instantiate the metavariables 'x', 'M', 'N' with specific syntactic objects in the lambda calculus, for example, $\mathtt{x}$, $\mathtt{xy}$, $\mathtt{z}$ respectively, then we get '$(\lambda\mathtt{x}.\mathtt{xy})\mathtt{z} \to_\beta (\mathtt{xy})\{\mathtt{z}/\mathtt{x}\}$' as an instance of β-reduction. Note that the expression '$(\mathtt{xy})\{\mathtt{z}/\mathtt{x}\}$' signifies the syntactic object $\mathtt{zy}$ in the lambda calculus, since we can calculate as $(\mathtt{xy})\{\mathtt{z}/\mathtt{x}\} = (\mathtt{x}\{\mathtt{z}/\mathtt{x}\})(\mathtt{y}\{\mathtt{z}/\mathtt{x}\}) = \mathtt{zy}$ after the instantiation of the metavariables.

Recently, several formal systems are proposed as extensions of the lambda calculus to internalize the notion of metavariables [9–11, 17, 18]. In this paper, such formal systems are collectively called **meta lambda calculi**. The most important feature of meta lambda calculi is that they include textual substitution to model instantiation of metavariables. The ordinary substitution in the lambda calculus is performed with α-renaming to avoid incidental variable binding, whereas textual substitution is performed without α-renaming by replacing each occurrence of a variable simply with a term. Thus textual substitution generates new bindings *dynamically*.

We illustrate a sketch of meta lambda calculi. Consider an extension of the syntax of the lambda calculus defined by the following BNF presented in [12]:

$$\text{Terms } M ::= x \mid \lambda x.M \mid M\ M \mid X \mid \delta X.M \mid M \odot M$$

where x ranges over the set of object-level variables and X ranges over the set of meta-level variables. The first three in the above BNF represent the object-level constructs, namely, the constructs in the lambda calculus. The last three represent meta-level constructs as counterparts of the object-level constructs. In the syntax, we have the following reduction sequence corresponding to the example discussed before:

$$\begin{aligned}&(\delta\mathtt{M}.((\delta\mathtt{N}.(\lambda\mathtt{x}.\mathtt{M})\mathtt{N})\odot\mathtt{z}))\odot\mathtt{xy}\\ &\Rightarrow_\beta (\delta\mathtt{M}.(\lambda\mathtt{x}.\mathtt{M})\mathtt{z})\odot\mathtt{xy} \Rightarrow_\beta (\lambda\mathtt{x}.\mathtt{xy})\mathtt{z} \to_\beta \mathtt{zy},\end{aligned}$$

where upper-case letters $\mathtt{M}$ and $\mathtt{N}$ are meta-level variables, and lower-case letters $\mathtt{x}$, $\mathtt{y}$ and $\mathtt{z}$ are object-level variables. In this paper, a meta-level application containing a meta-level abstraction as the left part is called a meta-level β-redex, and an object-level counterpart is called an object-level β-redex. For instance, $(\delta\mathtt{N}.(\lambda\mathtt{x}.\mathtt{M})\mathtt{N})\odot\mathtt{z}$ is a meta-level β-redex, and $(\lambda\mathtt{x}.\mathtt{M})\mathtt{N}$ is an object-level β-redex. Meta-level β-reduction, denoted by $\Rightarrow_\beta$ in Section 1 and Section 2, is performed on a meta-level β-redex in a similar way to the ordinary β-reduction but by textual substitution. In the above example, meta-level variable $\mathtt{M}$ is instantiated with the term $\mathtt{xy}$ by meta-level β-reduction, and the object-level variable $\mathtt{x}$ is dynamically bound by the binder $\lambda\mathtt{x}$.

1.2 Problem in Designing a Meta Lambda Calculus

When trying to define a meta lambda calculus formally, we face a subtle problem resulting from coexistence of the ordinary substitution with textual substitution [12]. Consider the following cases to examine object-level β-redexes containing meta-level variables.

(1) Consider a term $(\lambda\mathtt{x}.\mathtt{M})\mathtt{z}$. We cannot reduce the term simply by the ordinary substitution as $(\lambda\mathtt{x}.\mathtt{M})\mathtt{z} \to_\beta \mathtt{M}\{\mathtt{z}/\mathtt{x}\} = \mathtt{M}$, since such reduction gives us the following two reduction sequences

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676976

that are not confluent:

$$(\delta\mathtt{M}.(\lambda\mathtt{x}.\mathtt{M})\mathtt{z})\odot\mathtt{xy} \to_\beta (\delta\mathtt{M}.\mathtt{M})\odot\mathtt{xy} \Rightarrow_\beta \mathtt{xy},$$
$$(\delta\mathtt{M}.(\lambda\mathtt{x}.\mathtt{M})\mathtt{z})\odot\mathtt{xy} \Rightarrow_\beta (\lambda\mathtt{x}.\mathtt{xy})\mathtt{z} \to_\beta \mathtt{zy}.$$

The term zy in the second line is the intended result of the term $(\delta\mathtt{M}.(\lambda\mathtt{x}.\mathtt{M})\mathtt{z})\odot\mathtt{xy}$ as shown before. It is obviously a mistake to consider that $\mathtt{M}\{\mathtt{z}/\mathtt{x}\} = \mathtt{M}$, since M is a meta-level variable to be instantiated later with a term that may contain the object-level variable x.

(2) Consider a term $(\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x})\mathtt{M}$. We cannot reduce the term simply as $(\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x})\mathtt{M} \to_\beta (\lambda\mathtt{y}.\mathtt{x})\{\mathtt{M}/\mathtt{x}\} = \lambda\mathtt{y}.(\mathtt{x}\{\mathtt{M}/\mathtt{x}\}) = \lambda\mathtt{y}.\mathtt{M}$, since such reduction gives us the following two reduction sequences that are not confluent:

$$(\delta\mathtt{M}.(\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x})\mathtt{M})\odot\mathtt{y} \to_\beta (\delta\mathtt{M}.\lambda\mathtt{y}.\mathtt{M})\odot\mathtt{y} \Rightarrow_\beta \lambda\mathtt{y}.\mathtt{y},$$
$$(\delta\mathtt{M}.(\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x})\mathtt{M})\odot\mathtt{y} \Rightarrow_\beta (\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x})\mathtt{y} \to_\alpha (\lambda\mathtt{x}.\lambda\mathtt{z}.\mathtt{x})\mathtt{y} \to_\beta \lambda\mathtt{z}.\mathtt{y}.$$

The second reduction sequence gives us the intended result. Note that we have the following expression with a side condition in the definition of substitution in the lambda calculus:

$$\text{“}(\lambda y.N)\{M/x\} = \lambda y.(N\{M/x\}) \quad \text{if } y \notin \mathrm{FV}(M, x).\text{”}$$

Hence we cannot actually have $(\lambda\mathtt{y}.\mathtt{x})\{\mathtt{M}/\mathtt{x}\} = \lambda\mathtt{y}.(\mathtt{x}\{\mathtt{M}/\mathtt{x}\})$ without satisfying a side condition such as the one above. However, we do not have such side condition in the first place, since we have no information about the free variables occurring in the term for which M stands, namely, the term with which the meta-level variable M is instantiated later.

The above cases indicate that we cannot define easily object-level substitution for terms containing meta-level variables. The heart of the problem is that a meta-level variable is just a placeholder standing for some object-level term *unknown yet*.

1.3 Approaches in Previous Works

Approaches in previous works related to modeling metavariables and the problem discussed above are classified broadly into two categories. Here we explain briefly the essential points of the two categories respectively. Features of individual works are mentioned in Section 5.

(1) Meta-Level Variables Assigned with Interfaces

One approach is to assign each meta-level variable X with a finite list of object-level variables to be bound dynamically, which is called the **interface** of meta-level variable X in this paper. With assigning an *appropriate* interface to each meta-level variable, and with *embedding* such additional information into terms explicitly, instantiation of meta-level variables can be emulated by computation in calculi dealing with only static binding, in particular, the ordinary lambda calculus [12].

For example, consider the following meta-level β-reduction:

$$(\delta\mathtt{M}.(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x})(\mathtt{My}))\odot(\mathtt{yx})$$
$$\Rightarrow_\beta (\lambda\mathtt{x}.(\lambda\mathtt{y}.(\mathtt{yx})\mathtt{x})\mathtt{x})((\mathtt{yx})\mathtt{y}).$$

The meta-level β-reduction can be emulated by the following lambda term, which is obtained from the above term by assigning meta-level variable M with the interface (x, y):

$$(\lambda\mathtt{m}.(\lambda\mathtt{x}.(\lambda\mathtt{y}.(\underline{\mathtt{mxy}})\mathtt{x})\mathtt{x})((\underline{\mathtt{mxy}})\mathtt{y}))(\underline{\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{yx}})$$
$$\to_\beta^* (\lambda\mathtt{x}.(\lambda\mathtt{y}.(\mathtt{yx})\mathtt{x})\mathtt{x})((\mathtt{yx})\mathtt{y}).$$

To take a familiar example, the above fact corresponds to the fact that the C program with macros in Figure 1 can be emulated by the C program without macros in Figure 2. A meta-level variable assigned with an interface, for example, the list of x and y, is roughly equivalent to an ordinary function of x and y.

Figure 1. WithMacros.c

```
1: int main() {
2:   int x = 1;
3:   int y = x + 1;
4:   #define M y + x
5:   x = M + y;
6:   y = x;
7:   return M + x;
8: }
```

Figure 2. ViaInterface.c

```
1: int main() {
2:   int x = 1;
3:   int y = x + 1;
4:   int m(int x, int y)
5:     { return  y + x; }
6:   x = m(x, y) + y;
7:   y = x;
8:   return m(x, y) + x;
9: }
```

Calculus λm in [18] and the calculi in [4, 12, 14–16] adopt similar approaches to the one discussed above so as to consider metavariables to be assigned with interfaces. These calculi are inherently designed to deal with only static binding as the ordinary lambda calculus, and hence in these calculi, the notion of textual substitution is modeled by a mechanism like the ordinary substitution as illustrated above. As a result, the problem in question disappears from these calculi. In this paper, these calculi are called **lambda calculi with interfaces** distinctively from meta lambda calculi, since there is a difference between the concept modeled in meta lambda calculi and the concept modeled in lambda calculi with interfaces.

In order to clarify the challenge tackled in meta lambda calculi, we explain the essence of the difference between meta lambda calculi and lambda calculi with interfaces. In meta lambda calculi, meta-level variable M must satisfy the following principle for any object-level variables x and y:

$$\lambda x.\mathtt{M} \cong \lambda y.\mathtt{M} \quad \text{if and only if} \quad x = y,$$

where $\cong$ denotes the equivalence induced from rewriting rules. This principle is a consequence of the requirement that the term $\lambda x.\mathtt{M}$ is the counterpart of the expression '$\lambda x.M$' in the meta-language, and the term $\delta\mathtt{M}.\lambda x.\mathtt{M}$ is the counterpart of the function '$M \mapsto \lambda x.M$' on the set of *all* lambda terms. The principle corresponds to the fact that two functions '$M \mapsto \lambda x.M$' and '$M \mapsto \lambda y.M$' are equal if and only if $x = y$. Note that if the principle did not hold and we had some distinct object-level variables x and y such that $\lambda x.\mathtt{M} \cong \lambda y.\mathtt{M}$, then we would have nonsense equivalence such as $\lambda x.y \cong (\delta\mathtt{M}.\lambda x.\mathtt{M})\odot y \cong (\delta\mathtt{M}.\lambda y.\mathtt{M})\odot y \cong \lambda y.y$.

The goal of meta lambda calculi is to add new syntactic objects called meta-level variables satisfying the above principle into the ordinary lambda calculus. In contrast, lambda calculi with interfaces as well as the ordinary lambda calculus do not contain any term that corresponds to a meta-level variable. Namely, they do not contain a term N satisfying the following property for all variables x and y:

$$\lambda x.N \cong \lambda y.N \quad \text{if and only if} \quad x = y.$$

In the example shown before, the term mxy can be used in the ordinary lambda calculus to emulate the behavior of meta-level variable M. However, the term mxy itself cannot be an alternative to meta-level variable M generally. For instance, we have distinct variables a and b satisfying $\lambda\mathtt{a}.\mathtt{mxy} =_\alpha \lambda\mathtt{b}.\mathtt{mxy}$ in the ordinary lambda calculus, whereas $\lambda\mathtt{a}.\mathtt{M} \ncong \lambda\mathtt{b}.\mathtt{M}$ in meta lambda calculi.

(2) Level-Controlled Reduction

The other approach to resolve the problem in Section 1.2 is to restrict reduction rules by side conditions taking levels into account. To cite a case of calculus $\lambda\mathcal{M}$ in [18], object-level β-reduction $(\lambda x.M)N \to_\beta M\{N/x\}$ is defined with the following side condition:

$$\text{“}M \text{ and } N \text{ contain no meta-level constructs.”}$$

The other previous meta lambda calculi [9–11] also adopt similar approaches, although there are various technical differences. A reduction defined with such side conditions is called a **level-controlled reduction** in this paper. Level-controlled reductions bring confluence in meta lambda calculi, since wrong reduction sequences mentioned in Section 1.2 are eliminated by such side conditions. However, level-controlled reductions make some β-redexes stuck. For instance, the following object-level β-redex containing meta-level variables cannot be reduced in the previous meta lambda calculi due to such side conditions shown above:

$$(\lambda \mathtt{y}.\lambda \mathtt{z}.\mathtt{y}\mathtt{M}\mathtt{N})(\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{z}).$$

In other words, we cannot reduce an object-level β-redex until we instantiate meta-level variables in the β-redex with some object-level terms.

1.4 Our Purpose

In this paper, we propose meta lambda calculus λ^* with a new mechanism to reduce object-level β-redexes containing meta-level variables. Calculus λ^* enables us to advance computation in the presence of meta-level variables, and hence opens up new possibilities for reduction strategies that have been restricted by level-controlled reductions in previous meta lambda calculi. For example, we are able to introduce the notion of lazy evaluation in the presence of meta-level variables.

Another purpose of calculus λ^* is to examine those terms that include a certain type of level mismatch, such as an object-level abstraction applied to a meta-level abstraction like $(\lambda \mathtt{x}.(\mathtt{x} \odot \mathtt{M}))(\delta \mathtt{M}.\mathtt{M})$. Such terms are called cross-level terms in this paper. Cross-level terms have been regarded as meaningless terms, and left out of consideration by type systems or by level-controlled reductions in previous meta lambda calculi. One of distinct features of calculus λ^* is that the calculus makes it possible to compute cross-level terms. In Section 4, we show that some cross-level terms behave as quotes and 'eval' command in programming languages through an example. These terms provide us new insight into the notions of stores and recursion via meta-level variables.

2. Overview of Our Approach

2.1 An Analysis of Calculation in the Metalanguage

In the metalanguage, we can always:

- rewrite a β-redex '$(\lambda x.M)N$' to the expression '$M\{N/x\}$',
- perform α-renaming with some fresh variable, and
- pass down substitution '$\{N/x\}$' on a term to its subterms.

In this way, we can calculate a β-redex to perform substitution of variables even if the β-redex contains metavariables. For example, we can have the following calculation in the metalanguage:

$$\begin{aligned}
&(\lambda \mathtt{x}.(\lambda \mathtt{y}.M\mathtt{x})\mathtt{x})(M\mathtt{y}) \\
&\to_\alpha (\lambda \mathtt{x}.(\lambda v.M\{v/\mathtt{y}\}\mathtt{x})\mathtt{x})(M\mathtt{y}) \quad \text{where } v \text{ is fresh,} \\
&\to_\beta (\lambda v.M\{v/\mathtt{y}\}\{M\mathtt{y}/\mathtt{x}\}(M\mathtt{y}))(M\mathtt{y}) \\
&= (\lambda v.M\{v/\mathtt{y}, M\mathtt{y}/\mathtt{x}\}(M\mathtt{y}))(M\mathtt{y}) \\
&\to_\beta M\{v/\mathtt{y}, M\mathtt{y}/\mathtt{x}\}\{M\mathtt{y}/v\}(M\{M\mathtt{y}/v\}\mathtt{y}) \\
&= M\{M\mathtt{y}/\mathtt{y}, M\{M\mathtt{y}/v\}\mathtt{y}/\mathtt{x}, M\mathtt{y}/v\}(M\{M\mathtt{y}/v\}\mathtt{y}) \\
&= M\{M\mathtt{y}/\mathtt{y}, M\mathtt{y}/\mathtt{x}\}(M\mathtt{y}) \quad \text{since } v \text{ is fresh.}
\end{aligned}$$

Note that $\mathtt{x}$ and $\mathtt{y}$ are variables as syntactic objects in the lambda calculus, whereas 'v' is a metavariable that stands for some variable in the lambda calculus. If we instantiate metavariables 'M' in the last line with term $\mathtt{yx}$, we get the following result:

$$\begin{aligned}
&(\mathtt{yx})\{(\mathtt{yx})\mathtt{y}/\mathtt{y}, (\mathtt{yx})\mathtt{y}/\mathtt{x}\}((\mathtt{yx})\mathtt{y}) \\
&= (\mathtt{yxy})(\mathtt{yxy})(\mathtt{yxy}).
\end{aligned}$$

The key factors that enable us to advance calculation in the presence of metavariables are:

- substitution operators, and
- freshness conditions accompanied with α-renaming.

Note that these two factors interact with each other. In the above example, the freshness condition 'v is fresh' first arises in association with α-renaming of the binder $\lambda\mathtt{y}$ to λv. In the process of the calculation, the freshness condition deletes substitution operators '$\{M\mathtt{y}/v\}$' following metavariables 'M'. Furthermore, the last β-reduction that yields substitution operator '$\{M\mathtt{y}/v\}$' actually eliminates the freshness condition 'v is fresh' at the end of the calculation. Consequently, we have

$$(\lambda \mathtt{x}.(\lambda \mathtt{y}.M\mathtt{x})\mathtt{x})(M\mathtt{y}) \to_\beta^* M\{M\mathtt{y}/\mathtt{y}, M\mathtt{y}/\mathtt{x}\}(M\mathtt{y})$$

for any term M.

We attempt to incorporate the two notions of substitution operators and freshness conditions into a meta lambda calculus to achieve our purpose. It is not difficult to embed substitution operators as syntactic objects. A major obstacle is how to deal with freshness conditions and the interaction with substitution operators in a meta lambda calculus. As explained in the subsequent discussion, we overcome the obstacle by adopting indexed variables and skip operators as alternatives of freshness conditions.

2.2 Indexed Variables

Consider the extended syntax of the lambda calculus defined by the following BNF:

$$\text{Terms } M ::= v^d \mid \lambda v.M \mid M\ M$$

where v ranges over a set of names and d ranges over the set of nonnegative integers. In this syntax, a variable consists of a name v and a nonnegative integer d called the index of the variable. In a term, variable v^d skips d binders of v and hence is bound by the $(d+1)$-th binder of v. For instance, in term $\lambda \mathtt{x}.\lambda \mathtt{y}.\lambda \mathtt{x}.\mathtt{y}^0\mathtt{x}^1$, the variable $\mathtt{y}^0$ is bound by the binder $\lambda\mathtt{y}$ as in the ordinary lambda calculus, whereas the variable $\mathtt{x}^1$ skips the rightmost binder $\lambda\mathtt{x}$ and is bound by the leftmost binder $\lambda\mathtt{x}$. In a word, we consider an extension of the lambda calculus accompanied with de Bruijn indices [5, 16].

The extended syntax enables us to perform β-reduction without α-renaming, although we can still have the notion of α-renaming. For example, we can reduce a term $(\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}^0\mathtt{x}^0)(\lambda \mathtt{x}.\mathtt{y}^0)$ in the following two ways:

$$(\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}^0\mathtt{x}^0)(\lambda \mathtt{x}.\mathtt{y}^0) \to_\beta \lambda \mathtt{y}.\mathtt{y}^0(\lambda \mathtt{x}.\mathtt{y}^1) \to_\alpha \lambda \mathtt{z}.\mathtt{z}^0(\lambda \mathtt{x}.\mathtt{y}^0),$$

$$(\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}^0\mathtt{x}^0)(\lambda \mathtt{x}.\mathtt{y}^0) \to_\alpha (\lambda \mathtt{x}.\lambda \mathtt{z}.\mathtt{z}^0\mathtt{x}^0)(\lambda \mathtt{x}.\mathtt{y}^0) \to_\beta \lambda \mathtt{z}.\mathtt{z}^0(\lambda \mathtt{x}.\mathtt{y}^0).$$

The reduction sequence in the second line is regarded as the one in the ordinary lambda calculus. Note that if we equate α-equivalent terms as we usually do in the ordinary lambda calculus, then the above two reduction sequences are identical. As a result, the extended syntax provides us a more expressive notation virtually without changing the theory of the lambda calculus.

2.3 Skip Operators as Alternatives of Freshness Conditions

In the extended syntax, we no longer need the notion of freshness of variables. For example, we can calculate with metavariables as follows:

$$(\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}^0\mathtt{x}^0)M \to_\beta (\lambda \mathtt{y}.\mathtt{y}^0\mathtt{x}^0)\{M/\mathtt{x}\} = \lambda \mathtt{y}.\mathtt{y}^0(M * \uparrow_{\mathtt{y}}),$$

where $\uparrow_{\mathtt{y}}$ is a skip operator acting on the term M to increment indices of $\mathtt{y}$ in term M appropriately so as to skip the binder $\lambda\mathtt{y}$. For instance, $(\lambda\mathtt{x}.\mathtt{y}^0) * \uparrow_{\mathtt{y}}$ denotes $\lambda\mathtt{x}.\mathtt{y}^1$. The above calculation corresponds to the following calculation with a freshness condition in the ordinary lambda calculus:

$$(\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{yx})M \to_\alpha (\lambda\mathtt{x}.\lambda v.v\mathtt{x})M \to_\beta \lambda v.vM \quad \text{where } v \text{ is fresh.}$$

Skip operators free us from the need to perform α-renaming and the need to keep freshness conditions outside of terms. Consequently, skip operators can be more easily embedded into a meta lambda calculus than freshness conditions.

2.4 A Sketch of Meta Lambda Calculus λ^*

We construct meta lambda calculus λ^* in Section 3 by adopting indexed variables and by embedding both substitution operators and skip operators as syntactic objects in the calculus. Here we show a sketch to give an intuition about the calculus.

The syntax defined by the following BNF is a part of the syntax of calculus λ^*:

$$\begin{aligned} \text{Terms } M &::= v^d \mid \lambda v.M \mid M\ M \mid X[\sigma] \mid \delta X.M \mid M \odot M \\ \text{Substitutions } \sigma &::= \mathrm{id} \mid {}^{v}\!\downarrow(M) \cdot \sigma \mid \uparrow_v \cdot \sigma \end{aligned}$$

where v ranges over a set of names, d ranges over the set of non-negative integers, X ranges over the set of meta-level variables, and id signifies the empty sequence. The elements of form ${}^{v}\!\downarrow(M)$, called **push-elements**, are counterparts of substitution operators '$\{M/v\}$', and the elements of form $\uparrow_v$, called **pop-elements**, are counterparts of skip operators. In calculus λ^*, a substitution σ is dealt with as a finite sequence of push-elements and pop-elements in order to calculate the two kinds of elements in an integrated way to model the interaction between substitution operators and freshness conditions in the metalanguage. Note that a substitution σ can occur only with a meta-level variable X in the form $X[\sigma]$. The σ represents a substitution that is suspended to wait for instantiation of the meta-level variable X. For instance, term $\mathtt{M}[{}^{\mathtt{x}}\!\downarrow(\mathtt{z}^0)]$ corresponds to the expression '$M\{\mathtt{z}/\mathtt{x}\}$' that cannot be calculated any more in the metalanguage.

Action of a substitution σ on a term M, denoted by $M * \sigma$, is designed to reflect the behaviors of substitution operators and skip operators as alternatives of freshness conditions. For instance, the expression $(\mathtt{x}^0\mathtt{M}) * {}^{\mathtt{x}}\!\downarrow(\mathtt{z}^0)$ denotes the term $\mathtt{z}^0\mathtt{M}[{}^{\mathtt{x}}\!\downarrow(\mathtt{z}^0)]$, as the expression '$(\mathtt{x}M)\{\mathtt{z}/\mathtt{x}\}$' is reduced to '$\mathtt{z}(M\{\mathtt{z}/\mathtt{x}\})$' in the ordinary lambda calculus. Note that the substitution ${}^{\mathtt{x}}\!\downarrow(\mathtt{z}^0)$ is suspended on the meta-level variable $\mathtt{M}$, since the substitution cannot act any more until $\mathtt{M}$ is instantiated later. Also, the expression $(\lambda\mathtt{y}.\mathtt{y}^0\mathtt{x}^0) * {}^{\mathtt{x}}\!\downarrow(\mathtt{M})$ denotes $\lambda\mathtt{y}.\mathtt{y}^0\mathtt{M}[\uparrow_{\mathtt{y}}]$, which corresponds to the expression '$\lambda\mathtt{y}.\mathtt{y}^0(M * \uparrow_{\mathtt{y}})$' shown in Section 2.3.

The syntax enables us to advance computation in the presence of meta-level variables in the same way as the metalanguage. For example, the calculation shown in Section 2.1 is represented by the following reduction sequence in calculus λ^*:

$$\begin{aligned} &(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x})(\mathtt{My}) \\ &\to_\beta ((\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x}) * {}^{\mathtt{x}}\!\downarrow(\mathtt{My}) \\ &= (\lambda\mathtt{y}.\mathtt{M}[{}^{\mathtt{y}}\!\downarrow(\mathtt{y}) \cdot {}^{\mathtt{x}}\!\downarrow(\mathtt{M}[\uparrow_{\mathtt{y}}]\mathtt{y}^1) \cdot \uparrow_{\mathtt{y}}](\mathtt{M}[\uparrow_{\mathtt{y}}]\mathtt{y}^1))(\mathtt{My}) \\ &\to_\beta (\mathtt{M}[{}^{\mathtt{y}}\!\downarrow(\mathtt{y}) \cdot {}^{\mathtt{x}}\!\downarrow(\mathtt{M}[\uparrow_{\mathtt{y}}]\mathtt{y}^1) \cdot \uparrow_{\mathtt{y}}](\mathtt{M}[\uparrow_{\mathtt{y}}]\mathtt{y}^1)) * {}^{\mathtt{y}}\!\downarrow(\mathtt{My}) \\ &= \mathtt{M}[{}^{\mathtt{y}}\!\downarrow(\mathtt{My}) \cdot {}^{\mathtt{x}}\!\downarrow(\mathtt{M}[\uparrow_{\mathtt{y}} \cdot {}^{\mathtt{y}}\!\downarrow(\mathtt{My})]\mathtt{y}) \cdot \uparrow_{\mathtt{y}} \cdot {}^{\mathtt{y}}\!\downarrow(\mathtt{My})](\mathtt{M}[\uparrow_{\mathtt{y}} \cdot {}^{\mathtt{y}}\!\downarrow(\mathtt{My})]\mathtt{y}) \\ &\to_\varepsilon^* \mathtt{M}[{}^{\mathtt{y}}\!\downarrow(\mathtt{My}) \cdot {}^{\mathtt{x}}\!\downarrow(\mathtt{My})](\mathtt{My}), \end{aligned}$$

where indices equal to 0 are omitted. In the last line, ε-reduction eliminates substitutions of form $\uparrow_v \cdot {}^{v}\!\downarrow(N)$ to simulate the interaction between substitution operators and freshness conditions. If we instantiate the meta-level variables $\mathtt{M}$ in the last line with term $\mathtt{yx}$, we get the following intended result:

$$\begin{aligned} &(\mathtt{M}[{}^{\mathtt{y}}\!\downarrow(\mathtt{My}) \cdot {}^{\mathtt{x}}\!\downarrow(\mathtt{My})](\mathtt{My})) * {}^{\mathtt{M}}\!\downarrow(\mathtt{yx}) \\ &= ((\mathtt{yx}) * ({}^{\mathtt{y}}\!\downarrow((\mathtt{yx})\mathtt{y}) \cdot {}^{\mathtt{x}}\!\downarrow((\mathtt{yx})\mathtt{y})))((\mathtt{yx})\mathtt{y}) \\ &= (\mathtt{yxy})(\mathtt{yxy})(\mathtt{yxy}), \end{aligned}$$

where the meta-level push-element ${}^{\mathtt{M}}\!\downarrow(\mathtt{yx})$ represents the instantiation of $\mathtt{M}$. Note that the suspended substitution for object-level variables $\mathtt{y}$ and $\mathtt{x}$ following the meta-level variable $\mathtt{M}$ in the first line is resumed to obtain the result after the instantiation of $\mathtt{M}$.

In the discussion thus far, we illustrate a sketch of an extension of the lambda calculus added meta-level constructs. In the same way, we can easily add meta-meta-level constructs, meta-meta-meta-level constructs and so on. As a result, we obtain meta lambda calculus λ^* that includes infinitely hierarchical levels as meta lambda calculi in [9, 10, 18].

2.5 Blocks to Variable Binding

Lastly, we make a remark about a connection between object-level variable binding and meta-level application. In calculus λ^*, we have the following equality:

$$(M_1 \odot M_2) * {}^{v}\!\downarrow(N) = (M_1 * {}^{v}\!\downarrow(N)) \odot M_2.$$

This equality states that object-level variables v^d occurring in the right part M_2 of meta-level application $M_1 \odot M_2$ are prevented from being bound by outer binders of v [13]. As an example, we take a term $(\lambda\mathtt{y}.((\delta\mathtt{M}.(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x})(\mathtt{My}))\odot(\mathtt{yx})))(\mathtt{x1})$ that roughly corresponds to a part of the C program with macros in Figure 1. We should consider that the variable $\mathtt{y}$ in the right part '$\mathtt{yx}$' of the meta-level application is not bound by the leftmost binder $\lambda\mathtt{y}$. Otherwise, we could perform substitution for the $\mathtt{y}$ as follows:

$$\begin{aligned} &(\lambda\mathtt{y}.((\delta\mathtt{M}.(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x})(\mathtt{My}))\odot(\mathtt{yx})))(\underline{\mathtt{x1}}) \\ &\to_\beta ((\delta\mathtt{M}.(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x})(\mathtt{My}))\odot(\mathtt{yx})) * {}^{\mathtt{y}}\!\downarrow(\underline{\mathtt{x1}}) \\ &= ((\delta\mathtt{M}.(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x})(\mathtt{My})) * {}^{\mathtt{y}}\!\downarrow(\underline{\mathtt{x1}}))\odot((\mathtt{yx}) * {}^{\mathtt{y}}\!\downarrow(\underline{\mathtt{x1}})) \\ &= ((\delta\mathtt{M}.(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x})(\mathtt{My})) * {}^{\mathtt{y}}\!\downarrow(\underline{\mathtt{x1}}))\odot((\underline{\mathtt{x1}})\mathtt{x}). \end{aligned}$$

Such a reduction prevents a calculus from satisfying confluence, since we also have the following reduction sequence that generates new bindings dynamically:

$$\begin{aligned} &(\lambda\mathtt{y}.((\delta\mathtt{M}.(\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x})(\mathtt{My}))\odot(\underline{\mathtt{yx}})))(\mathtt{x1}) \\ &\Rightarrow_\beta (\lambda\mathtt{y}.(((\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{Mx})\mathtt{x})(\mathtt{My})) * {}^{\mathtt{M}}\!\downarrow(\underline{\mathtt{yx}})))(\mathtt{x1}) \\ &= (\lambda\mathtt{y}.(\lambda\mathtt{x}.(\lambda\mathtt{y}.(\underline{\mathtt{yx}})\mathtt{x})\mathtt{x})((\underline{\mathtt{yx}})\mathtt{y}))(\mathtt{x1}). \end{aligned}$$

Note that in the term *after* the meta-level β-reduction, the variables $\mathtt{y}$ in the two copies '$\mathtt{yx}$' are *newly bound* by one of the two binders $\lambda\mathtt{y}$ respectively. This fact indicates that in the term *before* the meta-level β-reduction, we should consider that the variable $\mathtt{y}$ in the right part '$\mathtt{yx}$' of the meta-level application copied by instantiation is *not bound yet* by any binder. Stated differently, in the C program with macros in Figure 1, the variable $\mathtt{y}$ in the fifth line signifies the *value* of $\mathtt{y}$ determined by assignment in the third line, whereas the variable $\mathtt{y}$ on the right of `#define` command in the fourth line signifies the *text* '$\mathtt{y}$' itself, which does not refer any value yet.

3. Meta Lambda Calculus λ^*

We formally define meta lambda calculus λ^* and show that the calculus satisfies confluence.

3.1 Terms and Substitutions

We consider the set $\mathcal{L}$ of **levels** as the set of positive integers. The object-level is 1, the meta-level is 2, the meta-meta-level is 3, and so

on. We assume that we are given the set $\mathcal{N}$ of **names** and function Lv of $\mathcal{N}$ into $\mathcal{L}$ that assigns a level to each name.

Definition 3.1. We define inductively the set Ter of **terms** and the auxiliary set Sub_ℓ for each level ℓ as follows:

$$\begin{aligned} v^d[\sigma] &\in Ter \quad \text{if } v \in \mathcal{N},\ d \in \mathbb{N},\ \sigma \in Sub_{\mathrm{Lv}(v)},\\ \lambda v.M &\in Ter \quad \text{if } v \in \mathcal{N},\ M \in Ter,\\ M \mathbin{@_\ell} N &\in Ter \quad \text{if } \ell \in \mathcal{L},\ M, N \in Ter,\\ \mathrm{id} &\in Sub_\ell,\\ {}^v{\downarrow}(M) \cdot \sigma &\in Sub_\ell \quad \text{if } v \in \mathcal{N}_{<\ell},\ M \in Ter,\ \sigma \in Sub_\ell,\\ \uparrow_v \cdot \sigma &\in Sub_\ell \quad \text{if } v \in \mathcal{N}_{<\ell},\ \sigma \in Sub_\ell, \end{aligned}$$

where $\mathcal{N}_{<\ell} = \{v \in \mathcal{N} \mid \mathrm{Lv}(v) < \ell\}$, $\mathbb{N}$ is the set of nonnegative integers, and id signifies the empty sequence. We define the set Sub of **substitutions** as the union $\bigcup_{\ell\in\mathcal{L}} Sub_\ell$. Note that a substitution is a finite sequence of push-elements ${}^v{\downarrow}(M)$ and pop-elements $\uparrow_v$ for names v and terms M. Push-elements and pop-elements are collectively called push-pop-elements.

Notation. As defined above, dots ' $\cdot$ ' represent punctuation marks for substitutions. We use the same dot to represent concatenation of substitutions. For instance, if substitution σ equals ${}^v{\downarrow}(M) \cdot \uparrow_v$ for name v and term M, then $\sigma \cdot \sigma$ denotes ${}^v{\downarrow}(M) \cdot \uparrow_v \cdot {}^v{\downarrow}(M) \cdot \uparrow_v$. In particular, $\mathrm{id} \cdot \sigma = \sigma \cdot \mathrm{id} = \sigma$ holds. We also use exponential notation, namely, $\sigma^0 = \mathrm{id}$ and $\sigma^{n+1} = \sigma \cdot \sigma^n$ for substitutions σ and nonnegative integers n. We omit superscripts '0' following names and brackets '[id]' around the empty sequence, when no confusion may occur.

Example 3.2. The following are terms, where $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ are names of level 1, and $\mathtt{M}$, $\mathtt{N}$ are names of level 2:

$$\begin{aligned} &(\lambda\mathtt{x}.\mathtt{y}^0[\mathrm{id}]) \mathbin{@_1} \mathtt{z}^0[\mathrm{id}],\\ &\mathtt{M}^0[{}^{\mathtt{x}}{\downarrow}(\mathtt{N}^1[\mathrm{id}]) \cdot \uparrow_{\mathtt{x}} \cdot {}^{\mathtt{y}}{\downarrow}(\lambda\mathtt{x}.\mathtt{M}^2[\uparrow_{\mathtt{y}} \cdot \uparrow_{\mathtt{z}}])]. \end{aligned}$$

We usually represent the term in the first line simply as $(\lambda\mathtt{x}.\mathtt{y}) \mathbin{@_1} \mathtt{z}$.

3.2 Action of Substitutions on Terms

Definition 3.3. Let v be a name and d a nonnegative integer. The $\langle v, d\rangle$**-component** $\sigma\langle v, d\rangle$ of a substitution σ is the term defined inductively as follows:

$$\begin{aligned} \mathrm{id}\langle v, d\rangle &\coloneqq v^d[\mathrm{id}],\\ ({}^w{\downarrow}(M) \cdot \sigma)\langle v, d\rangle &\coloneqq \begin{cases} M & \text{if } v = w \text{ and } d = 0,\\ \sigma\langle v, d - \delta_{vw}\rangle & \text{otherwise,} \end{cases}\\ (\uparrow_w \cdot \sigma)\langle v, d\rangle &\coloneqq \sigma\langle v, d + \delta_{vw}\rangle, \end{aligned}$$

where δ_{vw} is the integer defined by

$$\delta_{vw} \coloneqq \begin{cases} 1 & \text{if } v = w,\\ 0 & \text{otherwise.} \end{cases}$$

We give an intuitive explanation about $\langle v, d\rangle$-components of a substitution σ. Consider an $\mathcal{N}$-indexed family of infinite stacks of terms. Such a family of stacks is called an environment here. Push-elements ${}^v{\downarrow}(M)$ and pop-elements $\uparrow_v$ represent operations on the stack indexed by name v. Hence substitution σ as a composition of push-pop-elements represents an operation on environments. In what follows, the stack indexed by name v in an environment is simply called the v-stack. The environment whose v-stack consists of the infinite sequence $v^0, v^1, v^2, \ldots$ for each name v, is called the identity environment. The $\langle v, d\rangle$-component $\sigma\langle v, d\rangle$ signifies the element at depth d in the v-stack in the environment obtained by operating σ on the identity environment.

For instance, consider a substitution $\sigma = {}^v{\downarrow}(M) \cdot \uparrow_v \cdot \uparrow_v$. Substitution σ means the operation "pop from the v-stack, and then pop from the v-stack, and then push the term M into the v-stack." By operating σ on the identity environment, we obtain an environment whose v-stack consists of the infinite sequence M, v^2, v^3, $\ldots$, and thus we have $\sigma\langle v, 0\rangle = M$, $\sigma\langle v, 1\rangle = v^2$, $\sigma\langle v, 2\rangle = v^3$, and so on.

Definition 3.4. Let S be a subset of the set $\mathcal{N}$ of names. The S**-restriction** $\sigma{\restriction}S$ of a substitution σ is the substitution defined inductively as follows:

$$\begin{aligned} \mathrm{id}{\restriction}S &\coloneqq \mathrm{id},\\ ({}^v{\downarrow}(M) \cdot \sigma){\restriction}S &\coloneqq \begin{cases} {}^v{\downarrow}(M) \cdot (\sigma{\restriction}S) & \text{if } v \in S,\\ \sigma{\restriction}S & \text{otherwise,} \end{cases}\\ (\uparrow_v \cdot \sigma){\restriction}S &\coloneqq \begin{cases} \uparrow_v \cdot (\sigma{\restriction}S) & \text{if } v \in S,\\ \sigma{\restriction}S & \text{otherwise.} \end{cases} \end{aligned}$$

For levels ℓ and substitutions σ, we define $\sigma_{<\ell}$ and $\sigma_{\geq\ell}$ by

$$\begin{aligned} \sigma_{<\ell} &\coloneqq \sigma{\restriction}\{v \in \mathcal{N} \mid \mathrm{Lv}(v) < \ell\},\\ \sigma_{\geq\ell} &\coloneqq \sigma{\restriction}\{v \in \mathcal{N} \mid \mathrm{Lv}(v) \geq \ell\}. \end{aligned}$$

We abbreviate $\sigma_{<\mathrm{Lv}(v)}$ and $\sigma_{\geq\mathrm{Lv}(v)}$ to $\sigma_{<v}$ and $\sigma_{\geq v}$ respectively for names v.

Definition 3.5. We define inductively the term $M{*}\sigma$ resulting from **action** of a substitution σ on a term M, and the substitution $\tau \circ \sigma$ resulting from **composition** of substitutions σ and τ, as follows:

$$\begin{aligned} v^d[\tau] * \sigma &\coloneqq \begin{cases} v^d[\sigma] & \text{if } \sigma_{\geq v} = \tau = \mathrm{id},\\ \sigma\langle v, d\rangle * (\tau \circ \sigma)_{<v} & \text{otherwise,} \end{cases}\\ (\lambda v.M) * \sigma &\coloneqq \lambda v.(M * \Uparrow_v(\sigma)),\\ (M \mathbin{@_\ell} N) * \sigma &\coloneqq (M * \sigma) \mathbin{@_\ell} (N * \sigma_{\geq\ell}),\\ \mathrm{id} \circ \sigma &\coloneqq \sigma,\\ ({}^v{\downarrow}(M) \cdot \tau) \circ \sigma &\coloneqq {}^v{\downarrow}(M * \sigma_{\geq v}) \cdot (\tau \circ \sigma),\\ (\uparrow_v \cdot \tau) \circ \sigma &\coloneqq \uparrow_v \cdot (\tau \circ \sigma), \end{aligned}$$

where $\Uparrow_v(\sigma)$ denotes ${}^v{\downarrow}(v^0[\mathrm{id}]) \cdot (\sigma \circ \uparrow_v)$.

The first branch in the first line for action on a variable of name v states that substitution σ does nothing and stays as the suspended substitution on the variable, if the σ does not contain any push-pop-elements of level $\ell \geq \mathrm{Lv}(v)$. The second branch states that substitution σ replaces the variable $v^d[\tau]$ with the corresponding term $\sigma\langle v, d\rangle$, and the suspended substitution τ composed with σ further acts on the term. The restriction represented by '$<$' indicates that all push-pop-elements of level $\ell \geq \mathrm{Lv}(v)$ in σ are consumed by substitution for the variable $v^d[\tau]$. The expression $\Uparrow_v(\sigma)$ in the second line represents a substitution obtained by adjusting σ to skip the binder 'λv' appropriately. The restriction represented by '$\geq$' in the third line and the fifth line is due to blocks to variable binding discussed in Section 2.5. Note that $v^d[\tau] * \sigma = \sigma\langle v, d\rangle * (\tau \circ \sigma)_{<v}$ always holds by definition.

We show that $M * \sigma$ and $\sigma \circ \tau$ are well-defined for all terms M and all substitutions σ and τ by induction on height of terms and level of substitutions defined below.

Definition 3.6. The height $\mathrm{Ht}(M)$ of a term M and the height $\mathrm{Ht}(\sigma)$ of a substitution σ are the nonnegative integers defined inductively as follows:

$$\begin{aligned} \mathrm{Ht}(v^d[\sigma]) &\coloneqq \mathrm{Ht}(\sigma),\\ \mathrm{Ht}(\lambda v.M) &\coloneqq \mathrm{Ht}(M) + 1,\\ \mathrm{Ht}(M \mathbin{@_\ell} N) &\coloneqq \max\{\mathrm{Ht}(M), \mathrm{Ht}(N)\} + 1,\\ \mathrm{Ht}(\mathrm{id}) &\coloneqq 0,\\ \mathrm{Ht}({}^v{\downarrow}(M) \cdot \sigma) &\coloneqq \max\{\mathrm{Ht}(M) + 1, \mathrm{Ht}(\sigma)\},\\ \mathrm{Ht}(\uparrow_v \cdot \sigma) &\coloneqq \mathrm{Ht}(\sigma). \end{aligned}$$

Definition 3.7. The level $\mathrm{Lv}(\sigma)$ of a substitution σ is the nonnegative integer defined inductively as follows:

$$\mathrm{Lv}(\mathrm{id}) \coloneqq 0,$$
$$\mathrm{Lv}({}^{v}\!\downarrow(M)\cdot\sigma) \coloneqq \begin{cases} \mathrm{Lv}(\sigma) & \text{if } M = v^d[\mathrm{id}] \text{ for some } d \in \mathbb{N}, \\ \max\{\mathrm{Lv}(v), \mathrm{Lv}(\sigma)\} & \text{otherwise,} \end{cases}$$
$$\mathrm{Lv}(\uparrow_v \cdot \sigma) \coloneqq \mathrm{Lv}(\sigma).$$

Proposition 3.8. *Let M be a term and σ be a substitution. Then $M * \sigma$ is well-defined.*

Proof. We give a detailed proof for this proposition to clarify the mechanism of the calculus. Many other propositions in this section are proved in a similar way.

The above proposition is proved by induction on lexicographic ordering of pairs $\langle \mathrm{Lv}(\sigma), \mathrm{Ht}(M)\rangle$.

(1) Consider the case of $M = v^d[\tau]$.

First, we show that $\tau \circ \sigma$ is well-defined by induction on length of τ. The case of $\tau = \mathrm{id}$ and the case of $\tau = \uparrow_w \cdot \tau_1$ are trivial. Consider the case of $\tau = {}^{w}\!\downarrow(N)\cdot\tau_1$. Then $\tau\circ\sigma = {}^{w}\!\downarrow(N * \sigma_{\geq w})\cdot(\tau_1\circ\sigma)$. Note that $N*\sigma_{\geq w}$ is well-defined by induction hypothesis since $\mathrm{Lv}(\sigma_{\geq w}) \leq \mathrm{Lv}(\sigma)$ and $\mathrm{Ht}(N) < \mathrm{Ht}(M)$. Thus $\tau \circ \sigma$ is well-defined.

Next, we show that $\sigma\langle v, d\rangle * (\tau \circ \sigma)_{<v}$ is well-defined. If $\mathrm{Lv}(\sigma) \geq \mathrm{Lv}(v)$, then by induction hypothesis $\sigma\langle v, d\rangle * (\tau \circ \sigma)_{<v}$ is well-defined since $\mathrm{Lv}((\tau \circ \sigma)_{<v}) < \mathrm{Lv}(\sigma)$. If $\mathrm{Lv}(\sigma) < \mathrm{Lv}(v)$, then $\sigma\langle v, d\rangle = v^e$ for some $e \in \mathbb{N}$, and hence $\sigma\langle v, d\rangle * (\tau\circ\sigma)_{<v} = v^e[(\tau \circ \sigma)_{<v}]$ by definition.

(2) Consider the case of $M = \lambda v.M_1$.

We show that $\sigma \circ \uparrow_v$ is well-defined and $\mathrm{Lv}(\sigma \circ \uparrow_v) = \mathrm{Lv}(\sigma)$ by induction on length of σ. The case of $\sigma = \mathrm{id}$ and the case of $\sigma = \uparrow_w \cdot \sigma_1$ are trivial. Consider the case of $\sigma = {}^{w}\!\downarrow(N)\cdot\sigma_1$. Then $\sigma \circ \uparrow_v = {}^{w}\!\downarrow(N * (\uparrow_v)_{\geq w})\cdot(\sigma_1 \circ \uparrow_v)$. We prove below that $N * (\uparrow_v)_{\geq w}$ is well-defined.

If $\mathrm{Lv}(\sigma) > 0$, then $N * (\uparrow_v)_{\geq w}$ is well-defined by induction hypothesis since $\mathrm{Lv}((\uparrow_v)_{\geq w}) = 0$. If $\mathrm{Lv}(\sigma) = 0$, then $N = w^e$ for some $e \in \mathbb{N}$, and thus $N * (\uparrow_v)_{\geq w} = w^{e+\delta_{vw}}$.

Note that $N = w^e$ for some $e \in \mathbb{N}$ if and only if $N * (\uparrow_v)_{\geq w} = w^c$ for some $c \in \mathbb{N}$. Thus we have $\mathrm{Lv}(\sigma \circ \uparrow_v) = \mathrm{Lv}(\sigma)$.

Consequently, $M * \sigma = \lambda v.(M_1 * \Uparrow_v(\sigma))$ is well-defined by induction hypothesis since $\Uparrow_v(\sigma) = {}^{v}\!\downarrow(v)\cdot(\sigma\circ\uparrow_v)$ is well-defined and we have $\mathrm{Lv}(\Uparrow_v(\sigma)) = \mathrm{Lv}(\sigma)$ and $\mathrm{Ht}(M_1) < \mathrm{Ht}(M)$.

(3) The case of $M = M_1 \mathbin{@_\ell} M_2$ is trivial. □

Corollary 3.9. *Let σ and τ be substitutions. Then $\sigma \circ \tau$ is well-defined.*

3.3 Reductions on Terms

Definition 3.10. Let $\rightarrowtail$ be a binary relation on the set Ter of terms. We say that $\rightarrowtail$ is a reduction relation if $\rightarrowtail$ satisfies the following conditions:

$$v^d[\sigma \cdot {}^{w}\!\downarrow(M)\cdot\tau] \rightarrowtail v^d[\sigma \cdot {}^{w}\!\downarrow(M')\cdot\tau] \quad \text{if } M \rightarrowtail M',$$
$$\lambda v.M \rightarrowtail \lambda v.M' \quad \text{if } M \rightarrowtail M',$$
$$M \mathbin{@_\ell} N \rightarrowtail M' \mathbin{@_\ell} N \quad \text{if } M \rightarrowtail M',$$
$$N \mathbin{@_\ell} M \rightarrowtail N \mathbin{@_\ell} M' \quad \text{if } M \rightarrowtail M'.$$

We define **β-reduction** $\rightarrow_\beta$ as the least reduction relation satisfying the following expression:

$$(\lambda v.M) \mathbin{@_\ell} N \rightarrow_\beta M * {}^{v}\!\downarrow(N) \quad \text{if } \mathrm{Lv}(v) = \ell.$$

We also define **α-renaming** $\rightarrow_\alpha$ and **η-reduction** $\rightarrow_\eta$ as the least reduction relations satisfying the following expressions:

$$\lambda v.M \rightarrow_\alpha \lambda w.(M * ({}^{v}\!\downarrow(w)\cdot\uparrow_w)) \quad \text{if } \mathrm{Lv}(v) = \mathrm{Lv}(w),$$

and

$$\lambda v.((M * \uparrow_v) \mathbin{@_\ell} v) \rightarrow_\eta M \quad \text{if } \mathrm{Lv}(v) = \ell,$$

respectively.

Notation. Let $\rightarrowtail$ and $\rightarrowtail'$ be binary relations on a set. The composition of binary relations $\rightarrowtail$ and $\rightarrowtail'$ is denoted by $\rightarrowtail \cdot \rightarrowtail'$. The transitive closure of $\rightarrowtail$ is denoted by $\rightarrowtail^*$.

Remark 3.11. A term is said to be **annotation-free** if all indices in the term equal 0 and all substitutions in the term equal id. Consider the set Λ_1 of annotation-free terms consisting only of constructs of level 1. Then the set Λ_1 is just the set of terms in the ordinary lambda calculus under the assumption that we have infinitely many names of level 1 in the set $\mathcal{N}$ of names. Furthermore, β-reduction, α-renaming and η-reduction on the set Λ_1 coincide with the counterparts in the ordinary lambda calculus. As a result, the formalization of calculus λ^* provides us another definition of the ordinary lambda calculus in which β-reduction is defined without the notion of free variables and α-renaming.

3.4 Equivalence on Terms

The β-reduction defined in Definition 3.10 is not strictly confluent. Consider a term $M = (\lambda\mathtt{x}.((\lambda\mathtt{y}.\mathtt{N}) \mathbin{@_1} \mathtt{z})) \mathbin{@_1} \mathtt{y}$ with names $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ of level 1 and name $\mathtt{N}$ of level 2. We can obtain two distinct β-normal forms from M as follows:

$$M \rightarrow_\beta^* \mathtt{N}[{}^{\mathtt{y}}\!\downarrow(\mathtt{z})\cdot{}^{\mathtt{x}}\!\downarrow(\mathtt{y})], \text{ and}$$
$$M \rightarrow_\beta^* \mathtt{N}[{}^{\mathtt{y}}\!\downarrow(\mathtt{z})\cdot{}^{\mathtt{x}}\!\downarrow(\mathtt{y})\cdot\uparrow_{\mathtt{y}}\cdot{}^{\mathtt{y}}\!\downarrow(\mathtt{z})].$$

The two substitutions $\sigma = {}^{\mathtt{y}}\!\downarrow(\mathtt{z})\cdot{}^{\mathtt{x}}\!\downarrow(\mathtt{y})$ and $\tau = {}^{\mathtt{y}}\!\downarrow(\mathtt{z})\cdot{}^{\mathtt{x}}\!\downarrow(\mathtt{y})\cdot\uparrow_{\mathtt{y}}\cdot{}^{\mathtt{y}}\!\downarrow(\mathtt{z})$ in the above are obviously distinct sequences of push-pop-elements. However, σ and τ have the same $\langle v, d\rangle$-component for all names v and nonnegative integers d. We introduce an equivalence relation to equate such two substitutions. The equivalence relation leads the β-reduction to be confluent.

Definition 3.12. We define equivalence relations $\simeq_{\mathrm{t}}$ on the set Ter of terms and $\simeq_{\mathrm{s}}$ on the set Sub of substitutions inductively as follows:

$$\mathrm{id} \simeq_{\mathrm{s}} \mathrm{id},$$
$$\sigma \simeq_{\mathrm{s}} \tau \quad \text{if } \sigma\langle v, d\rangle \simeq_{\mathrm{t}} \tau\langle v, d\rangle \text{ for any } v \in \mathcal{N}, d \in \mathbb{N},$$
$$v^d[\sigma] \simeq_{\mathrm{t}} v^d[\tau] \quad \text{if } \sigma \simeq_{\mathrm{s}} \tau,$$
$$\lambda v.M \simeq_{\mathrm{t}} \lambda v.N \quad \text{if } M \simeq_{\mathrm{t}} N,$$
$$M_1 \mathbin{@_\ell} M_2 \simeq_{\mathrm{t}} N_1 \mathbin{@_\ell} N_2 \quad \text{if } M_1 \simeq_{\mathrm{t}} N_1 \text{ and } M_2 \simeq_{\mathrm{t}} N_2.$$

The symbols $\simeq_{\mathrm{t}}$ and $\simeq_{\mathrm{s}}$ are written simply as $\simeq$ by omitting the subscripts, when no confusion may occur.

We give another possible formalization of the equivalence on terms to illustrate the connection between equivalent terms more concretely by providing canonical form of substitutions.

Definition 3.13. We define **ε-reduction** on terms as the least reduction relation satisfying the following three expressions:

$$v^d[\sigma\cdot\uparrow_w\cdot{}^{w}\!\downarrow(M)\cdot\tau] \rightarrow_\varepsilon v^d[\sigma\cdot\tau],$$
$$v^d[\sigma\cdot{}^{w}\!\downarrow(w^e)\cdot\uparrow_w{}^{e+1}] \rightarrow_\varepsilon v^d[\sigma\cdot\uparrow_w{}^{e}],$$

and

$$v^d[\sigma\cdot\rho_1\cdot\rho_2\cdot\tau] \rightarrow_\varepsilon v^d[\sigma\cdot\rho_2\cdot\rho_1\cdot\tau]$$

for any $\rho_1 \in \{{}^{w}\!\downarrow(M), \uparrow_w\}$ and $\rho_2 \in \{{}^{u}\!\downarrow(N), \uparrow_u\}$ with $w \neq u$.

Definition 3.14. For each term M, we define $\#M$ as the number of push-pop-elements in M, namely, the number of up-arrows and down-arrows occurring in M. A term M is said to be **canonical** if $\#M \leq \#M'$ holds for any term M' satisfying $M \rightarrow_\varepsilon^* M'$. A

substitution σ is also said to be canonical if $v^d[\sigma]$ is a canonical term for some name v and nonnegative integer d.

Remark 3.15. A substitution σ is canonical, if and only if for each name v we have

$$\sigma\restriction\{v\} = {}^{v}\!\downarrow(M_0^v) \cdot {}^{v}\!\downarrow(M_1^v) \cdot \cdots \cdot {}^{v}\!\downarrow(M_{p_v-1}^v) \cdot \uparrow_v^{q_v}$$

for some nonnegative integers p_v, q_v and canonical terms $M_0^v, M_1^v, \ldots, M_{p_v-1}^v$, such that $M_{p_v-1}^v \neq v^{q_v-1}$ if $p_v > 0$ and $q_v > 0$. Suppose that the above equality hold. Then we have

$$\sigma\langle v, d\rangle = \begin{cases} M_d^v & \text{if } 0 \leq d < p_v, \\ v^{d-p_v+q_v} & \text{otherwise,} \end{cases}$$

for each nonnegative integer d.

Proposition 3.16. *Let M and N be terms. Then $M \simeq N$ holds if and only if $M \to_\varepsilon^* \cdot \leftarrow_\varepsilon^* N$ holds.*

Proof. It is straightforward that $M \to_\varepsilon N$ implies $M \simeq N$. We show that $M \simeq N$ implies $M \to_\varepsilon^* \cdot \leftarrow_\varepsilon^* N$.

We have canonical terms M_c and N_c such that $M \to_\varepsilon^* M_\mathrm{c}$ and $N \to_\varepsilon^* N_\mathrm{c}$. Note that $M \simeq N$ implies $M_\mathrm{c} \simeq N_\mathrm{c}$. We can prove that if $M_\mathrm{c} \simeq N_\mathrm{c}$ then $M_\mathrm{c} \to_\varepsilon^* N_\mathrm{c}$ by straightforward induction on $\mathrm{Ht}(M_\mathrm{c}) + \mathrm{Ht}(N_\mathrm{c})$. □

Theorem 3.17. *The following properties hold:*

$$\begin{aligned} M * \sigma &\simeq M' * \sigma' \quad \text{if } M \simeq M' \text{ and } \sigma \simeq \sigma', \\ \sigma \circ \tau &\simeq \sigma' \circ \tau' \quad \text{if } \sigma \simeq \sigma' \text{ and } \tau \simeq \tau', \\ \mathrm{id} \circ \sigma &\simeq \sigma \simeq \sigma \circ \mathrm{id}, \\ (\rho \circ \sigma) \circ \tau &\simeq \rho \circ (\sigma \circ \tau), \\ M * \mathrm{id} &\simeq M, \\ (M * \sigma) * \tau &\simeq M * (\sigma \circ \tau), \end{aligned}$$

for terms M, M' and substitutions ρ, σ, σ', τ.

In short, action of substitutions on terms and composition of substitutions preserve equivalence, and the set Sub of substitutions amounts to a monoid acting on the set Ter of terms up to equivalence. We prove the above properties in what follows.

Proposition 3.18. *Let M be a term. Then $M * \mathrm{id} \simeq M$ holds.*

Proof. We prove the stronger proposition that $M * \sigma \simeq M$ holds for any term M and any substitution σ satisfying the following condition: for any name v there exists a nonnegative integer p such that

$$\sigma\restriction\{v\} = {}^{v}\!\downarrow(v^0) \cdot {}^{v}\!\downarrow(v^1) \cdot \cdots \cdot {}^{v}\!\downarrow(v^{p-1}) \cdot \uparrow_v^p.$$

The stronger proposition is proved by straightforward induction on $\mathrm{Ht}(M)$. □

Lemma 3.19. *Let σ and τ be substitutions, v a name, d a nonnegative integer. Then $(\tau \circ \sigma)\langle v, d\rangle \simeq \tau\langle v, d\rangle * \sigma_{\geq v}$ holds.*

Proof. By straightforward induction on length of τ. □

Proposition 3.20. *Let M and M' be terms, σ and σ' substitutions. If $M \simeq M'$ and $\sigma \simeq \sigma'$, then $M * \sigma \simeq M' * \sigma'$ holds.*

Proof. The proposition is proved by induction on the lexicographic ordering of pairs $\langle \max\{\mathrm{Lv}(\sigma), \mathrm{Lv}(\sigma')\}, \mathrm{Ht}(M) + \mathrm{Ht}(M')\rangle$ as in the proof of Proposition 3.8. □

Corollary 3.21. *Let σ, σ', τ and τ' be substitutions such that $\sigma \simeq \sigma'$ and $\tau \simeq \tau'$. Then $\tau \circ \sigma \simeq \tau' \circ \sigma'$ holds.*

Proposition 3.22. *Let M be a term, σ and τ substitutions. Then $(M * \sigma) * \tau \simeq M * (\sigma \circ \tau)$ holds.*

Proof. The proposition is proved by induction on lexicographic ordering of pairs $\langle \mathrm{Lv}(\sigma) + \mathrm{Lv}(\tau), \mathrm{Ht}(M)\rangle$.

(1) Consider the case of $M = v^d[\rho]$.

We can prove that $(\rho\circ\sigma)\circ\tau \simeq \rho\circ(\sigma\circ\tau)$ by induction hypothesis and by the fact that $(\sigma \circ \tau)_{\geq \ell} = \sigma_{\geq \ell} \circ \tau_{\geq \ell}$ for each level ℓ. We show below that $(v^d[\rho] * \sigma) * \tau \simeq v^d[\rho] * (\sigma \circ \tau)$.

Suppose that $\mathrm{Lv}(\sigma) \geq \mathrm{Lv}(v)$. Then $(v^d[\rho]*\sigma)*\tau = (\sigma\langle v, d\rangle * (\rho\circ\sigma)_{<v}) * \tau \simeq \sigma\langle v, d\rangle * ((\rho\circ\sigma)_{<v} \circ \tau) \simeq \sigma\langle v, d\rangle * (\tau_{\geq v} \circ ((\rho\circ\sigma)\circ\tau)_{<v}) \simeq (\sigma\langle v, d\rangle * \tau_{\geq v}) * ((\rho\circ\sigma)\circ\tau)_{<v} \simeq (\sigma\circ\tau)\langle v, d\rangle * (\rho \circ (\sigma\circ\tau))_{<v} = v^d[\rho] * (\sigma\circ\tau)$ by induction hypothesis and by the fact that $(\rho\circ\sigma)_{<v} \circ \tau \simeq \tau_{\geq v} \circ ((\rho\circ\sigma)\circ\tau)_{<v}$.

Suppose that $\mathrm{Lv}(\sigma) < \mathrm{Lv}(v)$. Then $\sigma\langle v, d\rangle = v^e$ holds for some $e \in \mathbb{N}$. We have $(v^d[\rho] * \sigma) * \tau = v^e[(\rho\circ\sigma)_{<v}] * \tau = \tau\langle v, e\rangle * ((\rho\circ\sigma)_{<v} \circ \tau)_{<v} = \tau\langle v, e\rangle * ((\rho\circ\sigma)\circ\tau)_{<v} \simeq (\sigma\circ\tau)\langle v, d\rangle * (\rho \circ (\sigma\circ\tau))_{<v} = v^d[\rho] * (\sigma\circ\tau)$.

(2) Consider the case of $M = \lambda v.M_1$.

We show that $(\sigma \circ \uparrow_v) \circ \Uparrow_v(\tau) \simeq (\sigma\circ\tau) \circ \uparrow_v$.

Suppose that $\mathrm{Lv}(\sigma) > 0$. Then $((\sigma \circ \uparrow_v) \circ \Uparrow_v(\tau))\langle w, d\rangle \simeq (\sigma\langle w, d\rangle * (\uparrow_v)_{\geq w}) * \Uparrow_v(\tau)_{\geq w} \simeq \sigma\langle w, d\rangle * (\uparrow_v \circ \Uparrow_v(\tau))_{\geq w} \simeq \sigma\langle w, d\rangle * (\tau \circ \uparrow_v)_{\geq w} \simeq (\sigma\langle w, d\rangle * \tau_{\geq w}) * (\uparrow_v)_{\geq w} \simeq ((\sigma\circ\tau) \circ \uparrow_v)\langle w, d\rangle$ by induction hypothesis.

Suppose that $\mathrm{Lv}(\sigma) = 0$. Then $\sigma\langle w, d\rangle = w^e$ holds for some $e \in \mathbb{N}$. We have $((\sigma \circ \uparrow_v) \circ \Uparrow_v(\tau))\langle w, d\rangle \simeq (\sigma\langle w, d\rangle * (\uparrow_v)_{\geq w}) * \Uparrow_v(\tau)_{\geq w} = (w^e * (\uparrow_v)_{\geq w}) * \Uparrow_v(\tau)_{\geq w} = w^{e+\delta_{vw}} * \Uparrow_v(\tau)_{\geq w} \simeq (\tau \circ \uparrow_v)\langle w, e\rangle \simeq \tau\langle w, e\rangle * (\uparrow_v)_{\geq w} \simeq (w^e * \tau_{\geq w}) * (\uparrow_v)_{\geq w} = (\sigma\langle w, d\rangle * \tau_{\geq w}) * (\uparrow_v)_{\geq w} \simeq ((\sigma\circ\tau) \circ \uparrow_v)\langle w, d\rangle$.

Therefore we have $(M*\sigma)*\tau = \lambda v.((M_1 * \Uparrow_v(\sigma)) * \Uparrow_v(\tau)) \simeq \lambda v.(M_1 * (\Uparrow_v(\sigma) \circ \Uparrow_v(\tau))) = \lambda v.(M_1 * ({}^{v}\!\downarrow(v * \Uparrow_v(\tau)_{\geq v}) \cdot ((\sigma \circ \uparrow_v) \circ \Uparrow_v(\tau)))) \simeq \lambda v.(M_1 * ({}^{v}\!\downarrow(v) \cdot ((\sigma \circ \tau) \circ \uparrow_v))) = \lambda v.(M_1 * \Uparrow_v(\sigma\circ\tau)) = M * (\sigma \circ \tau)$ by induction hypothesis.

(3) The case of $M = M_1 \mathbin{@_\ell} M_2$ is trivial. □

Corollary 3.23. *Let ρ, σ and τ be substitutions. Then we have $(\rho \circ \sigma) \circ \tau \simeq \rho \circ (\sigma \circ \tau)$.*

3.5 Confluence of Reductions

We prove that β-reduction is confluent up to equivalence by the technique of parallel reduction [19]. In other words, we prove confluence of βε-reduction.

Definition 3.24. We define auxiliary binary relations $\Rrightarrow_\mathrm{t}$ on the set Ter of terms and $\Rrightarrow_\mathrm{s}$ on the set Sub of substitutions inductively as follows:

$$\begin{aligned} (\lambda v.M) \mathbin{@_{\mathrm{Lv}(v)}} N &\Rrightarrow_\mathrm{t} M' * {}^{v}\!\downarrow(N') \quad \text{if } M \Rrightarrow_\mathrm{t} M',\ N \Rrightarrow_\mathrm{t} N', \\ v^d[\sigma] &\Rrightarrow_\mathrm{t} v^d[\sigma'] \quad \text{if } \sigma \Rrightarrow_\mathrm{s} \sigma', \\ \lambda v.M &\Rrightarrow_\mathrm{t} \lambda v.M' \quad \text{if } M \Rrightarrow_\mathrm{t} M', \\ M_1 \mathbin{@_\ell} M_2 &\Rrightarrow_\mathrm{t} M_1' \mathbin{@_\ell} M_2' \quad \text{if } M_1 \Rrightarrow_\mathrm{t} M_1',\ M_2 \Rrightarrow_\mathrm{t} M_2', \\ \mathrm{id} &\Rrightarrow_\mathrm{s} \mathrm{id}, \\ {}^{v}\!\downarrow(M) \cdot \sigma &\Rrightarrow_\mathrm{s} {}^{v}\!\downarrow(M') \cdot \sigma' \quad \text{if } M \Rrightarrow_\mathrm{t} M',\ \sigma \Rrightarrow_\mathrm{s} \sigma', \\ \uparrow_v \cdot \sigma &\Rrightarrow_\mathrm{s} \uparrow_v \cdot \sigma' \quad \text{if } \sigma \Rrightarrow_\mathrm{s} \sigma'. \end{aligned}$$

Note that we have $\to_\beta^* = \Rrightarrow_\mathrm{t}^*$. The symbols $\Rrightarrow_\mathrm{t}$ and $\Rrightarrow_\mathrm{s}$ are written simply as $\Rrightarrow$ by omitting the subscripts, when no confusion may occur.

Lemma 3.25. *Let M and M' be terms, σ and σ' substitutions. If $M \Rrightarrow \cdot \simeq M'$ and $\sigma \Rrightarrow \cdot \simeq \sigma'$, then $M * \sigma \Rrightarrow \cdot \simeq M' * \sigma'$ holds.*

Proof. By straightforward induction on the lexicographic ordering of pairs $\langle \mathrm{Lv}(\sigma), \mathrm{Ht}(M)\rangle$. □

Lemma 3.26. *Let M and N be terms. If $M \simeq \cdot \Rrightarrow N$ holds, then we have $M \Rrightarrow \cdot \simeq N$.*

Proof. We can prove the following proposition by straightforward induction on $\mathrm{Ht}(L)$: if $M \simeq L \Rrightarrow N$ for some term L, then there exists a term L' such that $M \Rrightarrow L' \simeq N$. □

Definition 3.27. For each term M and each substitution σ, we define term $M^\wedge$ and substitution $\sigma^\wedge$ inductively as follows:

$$
\begin{aligned}
(v^d[\sigma])^\wedge &\coloneqq v^d[\sigma^\wedge],\\
(\lambda v.M)^\wedge &\coloneqq \lambda v.(M^\wedge),\\
(M \mathbin{@_\ell} N)^\wedge &\coloneqq \begin{cases} L^\wedge * {}^{v}\!\downarrow\!(N^\wedge) & \text{if } M = \lambda v.L, \mathrm{Lv}(v) = \ell,\\ M^\wedge \mathbin{@_\ell} N^\wedge & \text{otherwise,}\end{cases}\\
\mathrm{id}^\wedge &\coloneqq \mathrm{id},\\
({}^{v}\!\downarrow\!(M) \cdot \sigma)^\wedge &\coloneqq {}^{v}\!\downarrow\!(M^\wedge) \cdot \sigma^\wedge,\\
(\uparrow_v \cdot \sigma)^\wedge &\coloneqq \uparrow_v \cdot \sigma^\wedge.
\end{aligned}
$$

Lemma 3.28. *Let M and N be terms such that $M \Rrightarrow N$. Then $N \Rrightarrow \cdot \simeq M^\wedge$ holds.*

Proof. By straightforward induction on $\mathrm{Ht}(M)$. □

Theorem 3.29. *Let M and N be terms. If $M \leftarrow_\beta^* \cdot \simeq \cdot \rightarrow_\beta^* N$, then $M \rightarrow_\beta^* \cdot \simeq \cdot \leftarrow_\beta^* N$ holds.*

Proof. By Lemma 3.26 and Lemma 3.28, we can easily prove that $M \Lleftarrow \cdot \simeq \cdot \Rrightarrow N$ implies $M \Rrightarrow \cdot \simeq \cdot \Lleftarrow N$ for any terms M and N. By this proposition and Lemma 3.26, the above theorem is proved. □

Corollary 3.30. *Let M and N be terms, and $\rightarrow_{\beta\varepsilon}$ be the union of reduction relations $\rightarrow_\beta$ and $\rightarrow_\varepsilon$. If $M \leftarrow_{\beta\varepsilon}^* \cdot \rightarrow_{\beta\varepsilon}^* N$, then we have $M \rightarrow_{\beta\varepsilon}^* \cdot \leftarrow_{\beta\varepsilon}^* N$.*

4. An Application

Calculus λ^* provides us a way to manipulate binding structure flexibly with dynamic binding via meta-level variables. We illustrate the feature through an application of the calculus to a procedural language.

4.1 Procedural Language PROC

We introduce a simple procedural language PROC as an extension of imperative language IMP in [20]. PROC permits us to store and retrieve not only numbers but also procedures. We implement PROC in λ^* by exploiting the feature to manipulate binding structure dynamically. The implementation of PROC demonstrates that some notions related to names in procedural languages, such as stores, recursion, and localization of names, can be realized by dynamic binding via meta-level variables.

Definition 4.1. We define the syntax of PROC by the following BNF:

$$
\begin{aligned}
\text{N-expressions } E &::= x \mid n \mid \texttt{plus}\ E\ E \mid \texttt{minus}\ E\ E\\
\text{P-expressions } F &::= z \mid \texttt{proc}\ P\\
\text{Commands } C &::= x \texttt{ = } E \mid z \texttt{ = } F \mid \texttt{exec}\ F \mid \texttt{if}\ E\ P\ P \mid\\
&\qquad \texttt{while}\ E\ P \mid \texttt{local}\ P\ \texttt{export}\ {}^{x_1 x_2 \cdots x_h}_{z_1 z_2 \cdots z_k}\\
\text{Procedures } P &::= \mathrm{id} \mid C \texttt{ ; } P
\end{aligned}
$$

where n, h, k range over the set $\mathbb{N}$ of nonnegative integers, x, x_1, $x_2, \ldots, x_h$ range over a set $\mathcal{N}_\mathrm{N}$ of names for numbers, z, z_1, $z_2, \ldots$, z_k range over a set $\mathcal{N}_\mathrm{P}$ of names for procedures, and id signifies the empty sequence. The sets $\mathcal{N}_\mathrm{N}$ and $\mathcal{N}_\mathrm{P}$ are disjoint. We define $\mathbb{P}$ as the set of all procedures. Note that a procedure P is a finite sequence of commands. We represent concatenation of procedures P_1 and P_2 as $P_1 \texttt{ ; } P_2$.

Definition 4.2. A pair $\langle \varphi_\mathrm{N}, \varphi_\mathrm{P} \rangle$ of a function φ_N of $\mathcal{N}_\mathrm{N}$ into $\mathbb{N}$ and a function φ_P of $\mathcal{N}_\mathrm{P}$ into $\mathbb{P}$, is called a **state** of stores. We define $\mathbb{S}$ as the set of all states of stores, and call an element of the set $\mathbb{S}$ simply a state. Let $\varphi = \langle \varphi_\mathrm{N}, \varphi_\mathrm{P} \rangle$ be a state. Then the value E^φ of n-expression E in state φ is the nonnegative integer defined inductively as follows:

$$
\begin{aligned}
x^\varphi &\coloneqq \varphi_\mathrm{N}(x),\\
n^\varphi &\coloneqq n,\\
(\texttt{plus}\ E_1\ E_2)^\varphi &\coloneqq {E_1}^\varphi + {E_2}^\varphi,\\
(\texttt{minus}\ E_1\ E_2)^\varphi &\coloneqq \max\{{E_1}^\varphi - {E_2}^\varphi, 0\}.
\end{aligned}
$$

Similarly, the value F^φ of p-expression F in state φ is the procedure defined as follows:

$$
\begin{aligned}
z^\varphi &\coloneqq \varphi_\mathrm{P}(z),\\
(\texttt{proc}\ P)^\varphi &\coloneqq P.
\end{aligned}
$$

Definition 4.3. Let $\varphi = \langle \varphi_\mathrm{N}, \varphi_\mathrm{P} \rangle$ be a state. For nonnegative integers n and names x in $\mathcal{N}_\mathrm{N}$, state $\varphi\{n/x\}$ is defined as follows:

$$
\varphi\{n/x\} \coloneqq \langle \varphi'_\mathrm{N}, \varphi_\mathrm{P} \rangle \ \text{ where } \varphi'_\mathrm{N}(x') = \begin{cases} n & \text{if } x' = x,\\ \varphi_\mathrm{N}(x') & \text{otherwise,}\end{cases}
$$

for names x' in $\mathcal{N}_\mathrm{N}$. Similarly, state $\varphi\{P/z\}$ is defined for procedures P and names z in $\mathcal{N}_\mathrm{P}$ as follows:

$$
\varphi\{P/z\} \coloneqq \langle \varphi_\mathrm{N}, \varphi'_\mathrm{P} \rangle \ \text{ where } \varphi'_\mathrm{P}(z') = \begin{cases} P & \text{if } z' = z,\\ \varphi_\mathrm{P}(z') & \text{otherwise,}\end{cases}
$$

for names z' in $\mathcal{N}_\mathrm{P}$.

Definition 4.4. For procedures P, P', and states φ, φ', we define transition relation $\langle P \mid \varphi \rangle \rightarrow \langle P' \mid \varphi' \rangle$ as follows:

$$
\begin{aligned}
\langle x \texttt{ = } E \texttt{ ; } P \mid \varphi \rangle &\rightarrow \langle P \mid \varphi\{E^\varphi/x\} \rangle,\\
\langle z \texttt{ = } F \texttt{ ; } P \mid \varphi \rangle &\rightarrow \langle P \mid \varphi\{F^\varphi/z\} \rangle,\\
\langle \texttt{exec}\ F \texttt{ ; } P \mid \varphi \rangle &\rightarrow \langle F^\varphi \texttt{ ; } P \mid \varphi \rangle,\\
\langle \texttt{if}\ E\ P_1\ P_2 \texttt{ ; } P \mid \varphi \rangle &\rightarrow \langle P_1 \texttt{ ; } P \mid \varphi \rangle \quad \text{if } E^\varphi > 0,\\
\langle \texttt{if}\ E\ P_1\ P_2 \texttt{ ; } P \mid \varphi \rangle &\rightarrow \langle P_2 \texttt{ ; } P \mid \varphi \rangle \quad \text{if } E^\varphi = 0,\\
\langle \texttt{while}\ E\ P_1 \texttt{ ; } P \mid \varphi \rangle &\rightarrow \langle \texttt{if}\ E\ \{P_1 \texttt{ ; while } E\ P_1\}\ \mathrm{id} \texttt{ ; } P \mid \varphi \rangle,
\end{aligned}
$$

and, if $\langle P_1 \mid \varphi \rangle \rightarrow^* \langle \mathrm{id} \mid \varphi' \rangle$ for some state φ' then

$$
\langle \texttt{local}\ P_1\ \texttt{export}\ {}^{x_1 \cdots x_h}_{z_1 \cdots z_k} \texttt{ ; } P \mid \varphi \rangle \rightarrow \langle P \mid \psi \rangle
$$

with $\psi = \varphi\{{x_1}^{\varphi'}/x_1\} \cdots \{{x_h}^{\varphi'}/x_h\}\{{z_1}^{\varphi'}/z_1\} \cdots \{{z_k}^{\varphi'}/z_k\}$.

4.2 Implementation of PROC in λ^*

In the subsequent discussion, we assume that the set $\mathcal{N}_\mathrm{N}$ of names for numbers and the set $\mathcal{N}_\mathrm{P}$ of names for procedures are disjoint finite sets $\{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_a\}$ and $\{\bar{z}_1, \bar{z}_2, \ldots, \bar{z}_b\}$ respectively for some nonnegative integers a and b. Note that we can adopt such assumption when computing a procedure in PROC, since every procedure contains only finitely many names. Furthermore, we assume that the set of names of level 1 in calculus λ^* contains all names in PROC.

Notation. In this section, applications of each level are left-associative, and the body of an abstraction extends as far right as possible, in a customary way. Upper-case letters are names of level 2, and lower-case letters are names of level 1. We omit marks '$@_1$' for applications of level 1. For instance, $(\lambda \mathtt{M.Mxy}) \mathbin{@_2} \mathtt{M} \mathbin{@_2} \mathtt{N}$ signifies $((\lambda \mathtt{M.}((\mathtt{M} \mathbin{@_1} \mathtt{x}) \mathbin{@_1} \mathtt{y})) \mathbin{@_2} \mathtt{M}) \mathbin{@_2} \mathtt{N}$.

Definition 4.5. To n-expressions E, p-expressions F, commands C and procedures P in PROC, we assign terms $[\![E]\!]$, $[\![F]\!]$, $(\!|C|\!)$ and

$[\![P]\!]$ in calculus λ^* respectively, as follows:

$$[\![x]\!] \coloneqq x, \quad [\![0]\!] \coloneqq \mathit{zero}, \quad [\![n+1]\!] \coloneqq \mathit{scc}[\![n]\!],$$
$$[\![\texttt{plus}\ E_1\ E_2]\!] \coloneqq \mathit{plus}[\![E_1]\!][\![E_2]\!],$$
$$[\![\texttt{minus}\ E_1\ E_2]\!] \coloneqq \mathit{minus}[\![E_1]\!][\![E_2]\!],$$
$$[\![z]\!] \coloneqq z, \quad [\![\texttt{proc}\ P]\!] \coloneqq \mathit{block}\ @_2[\![P]\!],$$
$$(\!|x = E|\!) \coloneqq \mathit{set}_x\ @_2[\![E]\!], \quad (\!|z = F|\!) \coloneqq \mathit{set}_z\ @_2[\![F]\!],$$
$$(\!|\texttt{exec}\ F|\!) \coloneqq \mathit{unblock}\ @_2[\![F]\!],$$
$$(\!|\texttt{if}\ E\ P_1\ P_2|\!) \coloneqq \mathit{if}\ @_2[\![E]\!]\ @_2[\![P_1]\!]\ @_2[\![P_2]\!],$$
$$(\!|\texttt{while}\ E\ P|\!) \coloneqq \mathit{while}\ @_2[\![E]\!]\ @_2[\![P]\!],$$
$$(\!|\texttt{local}\ P\ \texttt{export}\ {}^{x_1 \ldots x_k}_{z_1 \ldots z_k}|\!) \coloneqq \mathit{local}^{x_1 \ldots x_h}_{z_1 \ldots z_k}\ @_2[\![P]\!],$$
$$[\![\mathrm{id}]\!] \coloneqq \mathit{id}, \quad [\![C\,;P]\!] \coloneqq \mathit{comp}\ @_2(\!|C|\!)\ @_2[\![P]\!],$$

where

$$\mathit{zero} \coloneqq \lambda\mathtt{s}.\lambda\mathtt{z}.\mathtt{z}, \quad \mathit{fst} \coloneqq \lambda\mathtt{a}.\lambda\mathtt{b}.\mathtt{a}, \quad \mathit{snd} \coloneqq \lambda\mathtt{a}.\lambda\mathtt{b}.\mathtt{b},$$
$$\mathit{scc} \coloneqq \lambda\mathtt{n}.\lambda\mathtt{s}.\lambda\mathtt{z}.\mathtt{s}(\mathtt{nsz}),$$
$$\mathit{prd} \coloneqq \lambda\mathtt{n}.\mathtt{n}(\lambda\mathtt{p}.\lambda\mathtt{c}.\mathtt{c}(\mathtt{p}\ \mathit{snd})(\mathit{scc}(\mathtt{p}\ \mathit{snd})))(\lambda\mathtt{c}.\mathtt{c}(\mathit{zero})\mathit{zero})\mathit{fst},$$
$$\mathit{plus} \coloneqq \lambda\mathtt{m}.\lambda\mathtt{n}.\mathtt{n}(\mathit{scc})\mathtt{m}, \quad \mathit{minus} \coloneqq \lambda\mathtt{m}.\lambda\mathtt{n}.\mathtt{n}(\mathit{prd})\mathtt{m},$$
$$\mathit{block} \coloneqq \lambda\mathtt{Y}.\lambda\mathtt{x}.(\mathtt{x}\ @_2\ \mathtt{Y}), \quad \mathit{unblock} \coloneqq \lambda\mathtt{Z}.\mathtt{Z}(\lambda\mathtt{Y}.\mathtt{Y}),$$
$$\mathit{set}_v \coloneqq \lambda\mathtt{E}.\lambda\mathtt{K}.(\lambda v.\mathtt{K})\mathtt{E},$$
$$\mathit{if} \coloneqq \lambda\mathtt{E}.\lambda\mathtt{P}.\lambda\mathtt{Q}.\lambda\mathtt{K}.\mathtt{E}(\lambda\mathtt{z}.\mathit{fst})\mathit{snd}(\mathtt{P}\ @_2\ \mathtt{K})(\mathtt{Q}\ @_2\ \mathtt{K}),$$
$$\mathit{while} \coloneqq \mathit{fix}\ @_2(\lambda\mathtt{W}.\lambda\mathtt{E}.\lambda\mathtt{P}.(\mathit{if}\ @_2\ \mathtt{E}\ @_2\ \mathit{loop}\ @_2\ \mathrm{id})),$$
$$\mathit{fix} \coloneqq \lambda\mathtt{F}.((\lambda\mathtt{X}.(\mathtt{F}\ @_2(\mathtt{X}\ @_2\ \mathtt{X})))\ @_2(\lambda\mathtt{X}.(\mathtt{F}\ @_2(\mathtt{X}\ @_2\ \mathtt{X})))),$$
$$\mathit{loop} \coloneqq \mathit{comp}\ @_2\ \mathtt{P}\ @_2(\mathtt{W}\ @_2\ \mathtt{E}\ @_2\ \mathtt{P}),$$
$$\mathit{local}^{x_1 \ldots x_h}_{z_1 \ldots z_k} \coloneqq \lambda\mathtt{P}.\lambda\mathtt{K}.(\mathit{export}^{x_1 \ldots x_h}_{z_1 \ldots z_k})(\lambda\mathtt{K}.\lambda x_1 \ldots \lambda z_k.\mathtt{K})\mathtt{P},$$
$$\mathit{export}^{x_1 \ldots x_h}_{z_1 \ldots z_k} \coloneqq \lambda\mathtt{e}.\lambda\mathtt{r}.(\mathtt{e}\ @_2\ \mathtt{K})(\mathtt{r}\ @_2\ x_1) \cdots (\mathtt{r}\ @_2\ z_k),$$
$$\mathit{id} \coloneqq \lambda\mathtt{K}.\mathtt{K}, \quad \mathit{comp} \coloneqq \lambda\mathtt{P}.\lambda\mathtt{Q}.\lambda\mathtt{K}.(\mathtt{P}\ @_2(\mathtt{Q}\ @_2\ \mathtt{K})).$$

Definition 4.6. For each state φ in PROC, we define $[\![\varphi]\!]$ as the substitution $\sigma_{\mathrm{N}} \cdot \sigma_{\mathrm{P}}$ in calculus λ^*, where σ_{N} and σ_{P} are the substitutions defined as follows:

$$\sigma_{\mathrm{N}} \coloneqq {}^{\bar{x}_1}\!\downarrow([\![\bar{x}_1^{\varphi}]\!]) \cdot \cdots \cdot {}^{\bar{x}_a}\!\downarrow([\![\bar{x}_a^{\varphi}]\!]),$$
$$\sigma_{\mathrm{P}} \coloneqq {}^{\bar{z}_1}\!\downarrow(\mathit{block}\ @_2[\![\bar{z}_1^{\varphi}]\!]) \cdot \cdots \cdot {}^{\bar{z}_b}\!\downarrow(\mathit{block}\ @_2[\![\bar{z}_b^{\varphi}]\!]).$$

We also define a relation $\sigma \approx_0 \tau$ for substitutions σ and τ as follows:

$$\sigma \approx_0 \tau \quad \text{if and only if} \quad \sigma\langle v, 0\rangle = \tau\langle v, 0\rangle \text{ for each name } v.$$

Notation. In what follows, the relation $\to_\beta^* \cdot \simeq$ is denoted by $\rightsquigarrow_\beta$, and the relation $\to_\beta^* \cdot \simeq \cdot \leftarrow_\beta^*$ is denoted by $\cong_\beta$.

With the above setting, a transition sequence in PROC is simulated by the corresponding reduction sequence in calculus λ^*, as stated in the following propositions.

Proposition 4.7. *Let P be a procedure, φ and φ' states, and σ a substitution. If $\langle P \mid \varphi\rangle \to^* \langle \mathrm{id} \mid \varphi'\rangle$ and $\sigma \approx_0 [\![\varphi]\!]$ hold, then there exists a substitution σ' such that $[\![P]\!] * \sigma \rightsquigarrow_\beta [\![\mathrm{id}]\!] * \sigma'$ and $\sigma' \approx_0 [\![\varphi']\!]$.*

Proof. By straightforward induction on size of transition sequence $\langle P \mid \varphi\rangle \to^* \langle \mathrm{id} \mid \varphi'\rangle$. □

Corollary 4.8. *Let P and P' be procedures, φ and φ' states, and σ a substitution. If $\langle P \mid \varphi\rangle \to \langle P' \mid \varphi'\rangle$ and $\sigma \approx_0 [\![\varphi]\!]$ hold, then there exists a substitution σ' such that $[\![P]\!] * \sigma \cong_\beta [\![P']\!] * \sigma'$ and $\sigma' \approx_0 [\![\varphi']\!]$.*

Remark 4.9. Stated differently, we have the following as an alternative to Proposition 4.7. Let P be a procedure, φ and φ' states, and σ a substitution. If $\langle P \mid \varphi\rangle \to^* \langle \mathrm{id} \mid \varphi'\rangle$ and $\sigma \approx_0 [\![\varphi]\!]$ hold, then we have

$$\mathit{comp}\ @_2(\lambda\mathtt{K}.\mathtt{K}[\sigma])\ @_2[\![P]\!] \rightsquigarrow_\beta \lambda\mathtt{K}.\mathtt{K}[\sigma']$$

for some substitution σ' such that $\sigma' \approx_0 [\![\varphi']\!]$.

Example 4.10. Let P be the following procedure in PROC that corresponds to the Ruby program in Figure 3:

```
sum = proc { n = ar;
  ct = plus ct 1;
  if n {
    local {ar = minus n 1; exec sum} export rv, ct;
    rv = n + rv
  } { rv = 0 }
}; ct = 0; local {ar = 3; exec sum} export rv, ct
```

Then $[\![P]\!]$ is reduced to the following term in normal form:

$$[\![P]\!] \rightsquigarrow_\beta \lambda\mathtt{K}.\mathtt{K}[{}^{\mathtt{ct}}\!\downarrow(\hat{4}) \cdot {}^{\mathtt{ct}}\!\downarrow(\hat{0}) \cdot {}^{\mathtt{rv}}\!\downarrow(\hat{6}) \cdot {}^{\mathtt{sum}}\!\downarrow(M)]$$

where $\hat{0}$, $\hat{4}$ and $\hat{6}$ are the Church numerals of 0, 4 and 6 respectively, and M is the term that corresponds to the procedure assigned to name `sum`.

Figure 3. Sum.rb

```
1: def sum(n)
2:   $ct = $ct + 1
3:   if n > 0
4:     n + sum(n - 1)
5:   else 0 end
6: end
7: $ct = 0
8: sum(3)
```

Figure 4. DynamicLiar.rb

```
1: p = "not(eval(p))"
2: print eval(p)
```

Figure 5. StaticLiar.rb

```
1: p = not(p)
2: print p
```

4.3 Recursion via Names and Textual Substitution

In PROC, we can write recursive procedures in the usual manner. This feature stems from the fact that we can store and retrieve procedures textually via names, and the feature is implemented with terms *block* and *unblock* in calculus λ^*. Actually, the terms *block* and *unblock* behave as quotes and 'eval' command in programming languages. For example, consider the following term L that corresponds to the Ruby program in Figure 4:

$$L = (\lambda\mathtt{p}.(\mathit{unblock}\ @_2\ \mathtt{p}))(\mathit{block}\ @_2(\mathtt{n}(\mathit{unblock}\ @_2\ \mathtt{p}))).$$

Then we have the following infinite reduction sequence that corresponds to the infinite loop caused by the program in Figure 4:

$$L \rightsquigarrow_\beta M \rightsquigarrow_\beta \mathtt{n}M \rightsquigarrow_\beta \mathtt{n}(\mathtt{n}M) \rightsquigarrow_\beta \mathtt{n}(\mathtt{n}(\mathtt{n}M)) \rightsquigarrow_\beta \cdots$$

where M is the term defined as follows:

$$M = (\lambda\mathtt{Z}.\mathtt{Z}[{}^{\mathtt{p}}\!\downarrow(N)](\lambda\mathtt{Y}.\mathtt{Y}[{}^{\mathtt{p}}\!\downarrow(N)]))\ @_2\ \mathtt{p},$$
$$N = \mathit{block}\ @_2(\mathtt{n}(\mathit{unblock}\ @_2\ \mathtt{p})).$$

Note that the terms *block* and *unblock* play fundamental roles to generate the infinite reduction sequence. In fact, if we remove the applications of the terms *block* and *unblock* in term L, then we get the term $(\lambda\mathtt{p}.\mathtt{p})(\mathtt{np})$, which corresponds to the Ruby program in Figure 5 that causes no infinite loops. From viewpoint of variable binding, quotes are considered as blocks to variable binding discussed in Section 2.5, and 'eval' command destroys blocks to variable binding. The terms *block* and *unblock* behave in the same way.

4.4 A Comment from Viewpoint of Type Systems

To make a brief comment about properties of calculus λ^*, we introduce a rough type system similar to the simple type system for the lambda calculus.

Definition 4.11. We assume that we are given a set of atomic types. The set Typ of types is defined inductively as follows:

$$T \in \mathit{Typ} \quad \text{if } T \text{ is an atomic type,}$$
$$T_1 \to_\ell T_2 \in \mathit{Typ} \quad \text{if } \ell \text{ is a level and } T_1, T_2 \in \mathit{Typ}.$$

Definition 4.12. A function of $\mathcal{N}$ into Typ is called a **type assignment**. For type assignments ξ, terms M, substitutions σ, and types T, we define typing relations $\xi \vdash M : T$ and $\xi \vdash \sigma$ inductively as follows:

$\xi \vdash v^d[\sigma] : \xi(v)$ if $\xi \vdash \sigma$,
$\xi \vdash \lambda v.M : \xi(v) \to_\ell T$ if $\ell = \mathrm{Lv}(v)$ and $\xi \vdash M : T$,
$\xi \vdash M_1 @_\ell M_2 : T'$ if $\xi \vdash M_1 : T \to_\ell T'$ and $\xi \vdash M_2 : T$,
$\xi \vdash \mathrm{id}$,
$\xi \vdash {}^v\!\downarrow(M) \cdot \sigma$ if $\xi \vdash M : \xi(v)$ and $\xi \vdash \sigma$,
$\xi \vdash \uparrow_v \cdot \sigma$ if $\xi \vdash \sigma$.

With the rough type system defined above, we can have subject reduction property stated by the following proposition.

Proposition 4.13. *Let ξ be a type assignment, T a type, M and M' terms. If $\xi \vdash M : T$ and $M \to_\beta M'$, then $\xi \vdash M' : T$ holds.*

However, the rough type system does not provide us strong normalization property, in contrast to the type system for calculus $\lambda\mathcal{M}$ [18]. For example, consider a type assignment $\xi_{\mathtt{x}}$ satisfying the following conditions:

$$\xi_{\mathtt{x}}(\mathtt{n}) = B \to_1 B, \quad \xi_{\mathtt{x}}(\mathtt{p}) = \mathit{Block}(B), \quad \xi_{\mathtt{x}}(\mathtt{x}) = B \to_2 B,$$
$$\xi_{\mathtt{x}}(\mathtt{Y}) = B, \quad \xi_{\mathtt{x}}(\mathtt{Z}) = \mathit{Block}(B),$$

where B is a type, and $\mathit{Block}(B)$ signifies $(B \to_2 B) \to_1 B$. Then the following expressions hold:

$$\xi_{\mathtt{x}} \vdash \mathit{block} : B \to_2 \mathit{Block}(B),$$
$$\xi_{\mathtt{x}} \vdash \mathit{unblock} : \mathit{Block}(B) \to_2 B,$$
$$\xi_{\mathtt{x}} \vdash L : B,$$

where L is the term defined in Section 4.3. The term L has type B, and thus we may expect that the term L signifies a value of type B. However, the term L actually has no normal form and hence signifies nothing, as in the case of the liar paradox.

In order to eliminate such unwanted situations and achieve strong normalization property, we need a more elaborate type system. In fact, the rough type system permits term $L' = \lambda\mathtt{n}.L$ to have type $(B \to_1 B) \to_1 B$ in the type assignment $\xi_{\mathtt{x}}$, whereas L' is a fixed-point operator of level 1 in a sense that we have $L'M \cong_\beta M(L'M)$ for any term M. A key difference from calculus $\lambda\mathcal{M}$ providing strong normalization property is caused by the existence of cross-level terms that bring on level-increasing reduction, such as object-level β-reduction generating new meta-level β-redexes. A term is said to be **cross-level** if the term can be typed only in cross-level type assignments. A type assignment ξ is said to be cross-level if there exists a name v such that the type $\xi(v)$ of v in the type assignment ξ contains an arrow '$\to_\ell$' of level ℓ greater than $\mathrm{Lv}(v)$. For instance, the above $\xi_{\mathtt{x}}$ is a cross-level type assignment, and the terms block and $\mathit{export}_{z_1 \ldots z_k}^{x_1 \ldots x_h}$ are cross-level terms as well as the term L mentioned above. Cross-level type assignments and cross-level terms are seemingly meaningless. Hence, cross-level terms have been left out of consideration by type systems or by level-controlled reductions in previous meta lambda calculi. However, we consider that cross-level terms may be worth investigating, since cross-level terms seem to have connections with notions related to names and bindings in programming languages, such as stores, quotes, recursion, and localization of names, as demonstrated in this section. We leave further research about this topic as a future work.

5. Related Works

5.1 Meta Lambda Calculi

We discuss connections with previous meta lambda calculi that include inherently textual substitution via meta-level variables.

Calculus $\lambda\mathcal{M}$ proposed by Sato, Sakurai, Kameyama and Igarashi [18] is a meta lambda calculus with infinitely hierarchical levels. $\lambda\mathcal{M}$ adopts level-controlled reductions and a type system to achieve preferable properties in coexistence with textual substitution. The type system eliminates cross-level terms mentioned in Section 4.4. $\lambda\mathcal{M}$ is actually a subsystem of λ^* consisting only of annotation-free terms. The level-controlled reduction in $\lambda\mathcal{M}$ is viewed as a restriction of β-reduction in λ^* on a set of annotation-free terms.

Calculus LamCC in Gabbay and Lengrand [10] also includes infinitely hierarchical levels. LamCC adopts level-controlled reductions and explicit substitutions. Unlike other meta lambda calculi, LamCC does not have the notion of level of application. In other words, the level of each application occurring in a term is determined dynamically in the process of computation. One of the features of λ^* different from LamCC is that substitutions in λ^* have canonical representation. By this feature, terms signifying the same substitution, for example, $(\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{M})\mathtt{Nz}$ and $(\lambda\mathtt{y}.\lambda\mathtt{x}.\mathtt{M})\mathtt{zN}$ are reduced to the same term $\mathtt{M}[{}^{\mathtt{x}}\!\downarrow(\mathtt{N}) \cdot {}^{\mathtt{y}}\!\downarrow(\mathtt{z})]$ by β-reduction and ε-reduction in λ^*. In LamCC, the above two terms are reduced to distinct two terms in normal form.

Gabbay's NEW calculus of contexts [9] and two-level lambda calculus [11] adopt level-controlled reductions and freshness contexts. A freshness context is regarded as an assumption about freshness conditions for meta-level variables. Reductions are controlled by freshness contexts. In a word, a term is reduced under some assumption about freshness conditions for meta-level variables.

The NEW calculus and LamCC mentioned above include NEW binders '$\mathit{И}$' separately from ordinary lambda binders 'λ'. A NEW binder indicates that an object-level variable is fresh for meta-level variables. For instance, $\mathit{И}\mathtt{x}.\lambda\mathtt{x}.\mathtt{Mx} \to_\alpha \mathit{И}\mathtt{y}.\lambda\mathtt{y}.\mathtt{My}$ holds, since the object-level variables '$\mathtt{x}$' and '$\mathtt{y}$' in the above terms are considered to be fresh for meta-level variable $\mathtt{M}$. In calculus λ^*, such information is represented by pop-elements. The above two terms and the α-renaming are represented in λ^* as $\lambda\mathtt{x}.\mathtt{M}[\uparrow_{\mathtt{x}}]\mathtt{x} \to_\alpha \to_\varepsilon \lambda\mathtt{y}.\mathtt{M}[\uparrow_{\mathtt{y}}]\mathtt{y}$.

Bekki's meta-lambda calculus [2] is a study of categorical semantics for metavariables. The type system in the calculus eliminates cross-level terms mentioned in Section 4.4. The formalization shown in Bekki and Asai [3] and in Masuko and Bekki [13] adopts assumption about free variables for meta-level variables in order to perform α-renaming in the presence of meta-level variables. The notion of blocks to variable binding discussed in Section 2.5 is pointed out in [13].

5.2 Other Works

Attempts to model textual substitution via metavariables in a calculus are originated from Hashimoto and Ohori's typed context calculus [12], which is designed to internalize the notion of lambda contexts. Their calculus adopts level-controlled reductions and a mechanism, called renamers, to manage binding structure consistently with the notion of holes to represent lambda contexts. Holes are regarded as meta-level variables, and lambda contexts are regarded as meta-level abstractions from viewpoint of meta lambda calculi. The technique of renamers is considered a kind of approaches by interfaces assigned to meta-level variables mentioned in Section 1.3. Sato, Sakurai and Kameyama's simply typed context calculus [16], calculus λm in Sato et al. [18], Nanevski, Pientka and Pfenning's calculi [14, 15] with modal types, and Boespflug and Pientka [4] with multi-level modal types are also designed by approaches of interfaces assigned to metavariables. These calculi are called lambda

calculi with interfaces distinctively from meta lambda calculi in this paper. The goal of these calculi to provide type systems for meta-level variables seems to lead the design to use information about interfaces assigned to meta-level variables not only in type systems but also in rewriting systems. This feature of the design makes the difference between these calculi and meta lambda calculi as illustrated in Section 1.3. The technique of indexed variables is adopted in [16].

The syntax of calculus λ^{m} in Davies [7] is similar to the syntax of λ^* in a sense that variables and applications are assigned with numbers. The numbers in calculus λ^* signify levels of variables, which determine *strength* of substitution so that substitution of level ℓ is performed by regarding variables of level less than ℓ as mere texts. In contrast, the numbers in calculus λ^{m} signify stages of computation, which determine *time* of substitution. In λ^{m}, all variables occurring in a term are bound statically as usual, unlike meta lambda calculi.

Methods to deal with substitutions as syntactic objects date back to Abadi, Cardelli, Curien and Lévy's explicit substitutions [1]. Dowek, Hardin and Kirchner [8] applies the method to the nameless lambda calculus with meta-level variables. The formalization of substitutions with push-elements and pop-elements in λ^* is almost the same as their formalization, although substitutions in λ^* are sorted by names and defined to occur only as suspended substitutions on meta-level variables. The slight difference from their formalization stems from the purpose to make calculus λ^* become a supersystem of $\lambda\mathcal{M}$ [18] as well as the ordinary lambda calculus. In other words, this paper does not concern itself about modeling concrete way and cost of performing substitution, which is the original purpose of explicit substitutions in Abadi et al.

Calculus $\lambda\mathrm{N}$ proposed by Dami [6] is an extension of the lambda calculus to model dynamic binding. The syntax of $\lambda\mathrm{N}$ includes additional constructs called labels separately from ordinary variables. A main difference between $\lambda\mathrm{N}$ and λ^* is their style of dynamic binding. Calculus λ^* deals with dynamic binding by lambda binders via meta-level variables, whereas $\lambda\mathrm{N}$ deals with dynamic binding by labels, not by lambda binders.

6. Conclusion

We have proposed meta lambda calculus λ^*, in which any β-redex of any level can be reduced to perform substitution for variables. This feature makes it possible to advance computation even in the presence of meta-level variables, and hence provides us new possibilities for reduction strategies that have been restricted by level-controlled reductions in previous meta lambda calculi.

Also, we have shown a procedural language as an application of calculus λ^*. Through the implementation of the procedural language, we have observed the connections between dynamic binding via meta-level variables and the notions of stores and recursion in procedural languages. We hope that calculus λ^* contributes toward understanding metavariables and the association with notions related to names and bindings in programming languages.

Acknowledgments

The author wishes to thank his supervisor, Ryu Hasegawa, sincerely for the fruitful discussions. The author also thanks Brigitte Pientka and anonymous reviewers for helpful comments.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien and J.-J. Lévy. 1991. Explicit substitutions. *Journal of Functional Programming* 1, 4, 375–416.

[2] Daisuke Bekki. 2009. Monads and meta-lambda calculus. In *New Frontiers in Artificial Intelligence (JSAI 2008)*, LNAI 5447, 193–208.

[3] Daisuke Bekki and Kenichi Asai. 2010. Representing covert movements by delimited continuations. In *New Frontiers in Artificial Intelligence (JSAI-isAI 2009)*, LNAI 6284, 161–180.

[4] Mathieu Boespflug and Brigitte Pientka. 2011. Multi-level contextual type theory. In *Proceedings of the 6th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2011)*, EPTCS 71, 29–43.

[5] N. G. de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 75, 5, 381–392.

[6] Laurent Dami. 1998. A lambda-calculus for dynamic binding. *Theoretical Computer Science* 192, 2, 201–231.

[7] Rowan Davies. 1996. A temporal-logic approach to binding-time analysis. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, 184–195.

[8] Gilles Dowek, Thérèse Hardin and Claude Kirchner. 2000. Higher order unification via explicit substitutions. *Information and Computation* 157, 183–235.

[9] Murdoch J. Gabbay. 2005. A NEW calculus of contexts. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '05)*, 94–105.

[10] Murdoch J. Gabbay and Stéphane Lengrand. 2009. The lambda-context calculus (extended version). *Information and Computation* 207, 12, 1369–1400.

[11] Murdoch J. Gabbay and Dominic P. Mulligan. 2009. Two-level lambda-calculus. *Electronic Notes in Theoretical Computer Science* 246, 107–129.

[12] Masatomo Hashimoto and Atsushi Ohori. 2001. A typed context calculus. *Theoretical Computer Science* 266, 249–272.

[13] Moe Masuko and Daisuke Bekki. 2011. Categorical semantics of meta-lambda calculus (in Japanese). In *Informal Proceedings of the 13th JSSST Workshop on Programming and Programming Languages*, 60–74.

[14] Aleksandar Nanevski, Brigitte Pientka and Frank Pfenning. 2003. A modal foundation for meta-variables. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding (MERLIN '03)*, 1–6.

[15] Aleksandar Nanevski, Frank Pfenning and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic* 9, 3, Article 23, 49 pages.

[16] Masahiko Sato, Takafumi Sakurai and Yukiyoshi Kameyama. 2002. A simply typed context calculus with first-class environments. *Journal of Functional and Logic Programming* 2002, 1–41.

[17] Masahiko Sato, Takafumi Sakurai, Yukiyoshi Kameyama and Atsushi Igarashi. 2003. Calculi of meta-variables. In *Computer Science Logic (CSL 2003)*, LNCS 2803, 484–497.

[18] Masahiko Sato, Takafumi Sakurai, Yukiyoshi Kameyama and Atsushi Igarashi. 2008. Calculi of meta-variables. *Frontiers of Computer Science in China* 2, 1, 12–21.

[19] Masako Takahashi. 1995. Parallel reductions in λ-calculus. *Information and Computation* 118, 1, 120–127.

[20] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages*. The MIT Press.