A simply typed context calculus with first-class environments

Masahiko Sato[1], Takafumi Sakurai[2], and Yukiyoshi Kameyama[1]

[1] Graduate School of Informatics, Kyoto University
{masahiko,kameyama}@kuis.kyoto-u.ac.jp
[2] Department of Mathematics and Informatics, Chiba University
sakurai@math.s.chiba-u.ac.jp

Abstract. We introduce a simply typed λ-calculus $\lambda\kappa\varepsilon$ which has both contexts and environments as first-class values. In $\lambda\kappa\varepsilon$, holes in contexts are represented by ordinary variables of appropriate types and hole filling is represented by the functional application together with a new abstraction mechanism which takes care of packing and unpacking of the term which is used to fill in the holes of the context. $\lambda\kappa\varepsilon$ is a conservative extension of the simply typed $\lambda\beta$-calculus, enjoys subject reduction property, is confluent and strongly normalizing.
The traditional method of defining substitution does not work for our calculus. So, we also introduce a new method of defining substitution. Although we introduce the new definition of substitution out of necessity, the new definition turns out to be conceptually simpler than the traditional definition of substitution.

1 Introduction

Informally speaking, a context (in λ-calculus) is a λ-term with some holes in it. For example, writing [ ] for a hole, $\lambda y.\ [\ ]$ is a context, and by filling the hole in it with $x + y$, we get $\lambda y.\ x + y$. By this operation, the variable y in $x + y$ gets *captured* and becomes bound in $\lambda y.\ x + y$, and the variable x remains to be free. So, unlike substitution, hole filling may introduce new and intended bound variables.

Recently there have been several attempts to formalize the notion of context and thereby make computing with contexts possible. For example, Talcott [16], Lee-Friedman [8], Dami [5], Hashimoto-Ohori [7], Sands [13], Mason [9] and Bognar-de Vrijer [3] made notable contributions. However, as far as we know, there is as yet no proposal of a language which has contexts as first-class values and which is at the same time *pure* in the following sense. We understand that a functional language is *pure*[1] if (i) it is a conservative extension of the untyped or simply typed $\lambda\beta$-calculus, (ii) confluent and (iii) strongly normalizing (SN) if the language is typed and has preservation of strong normalization (PSN) property if the language is untyped. The conservative extension property guarantees that

[1] We have introduced this notion of purity in [15].

the language is logically well-behaved and the confluence property and SN or PSN would guarantee that the language is computationally well-behaved.

In this paper, we introduce the calculus $\lambda\kappa\varepsilon$ (κ is for context and ε is for environment) which is pure in the above sense and which has contexts and environments as its first class values, so that we can bind contexts and environments to variables and return them as the values of computations. $\lambda\kappa\varepsilon$ is a simply typed calculus, and in $\lambda\kappa\varepsilon$, holes are represented by ordinary variables of appropriate types (which we will call hole types) and hole filling is represented by the functional application together with a new abstraction mechanism which takes care of packing and unpacking of the term which is used to fill in the holes of the context.

We now illustrate some of the difficulties we face in formalizing the notion of context, and explain our solution informally by relating it to previous works. First, let us consider the context:

$$a[\,] \equiv \lambda x.\ (\lambda y.\ x + [\,])3.$$

If we fill the hole in $a[\,]$ with the term $x + y$, we get

$$a[x + y] \equiv \lambda x.\ (\lambda y.\ x + (x + y))3.$$

By β-reducing it, we can convert $a[x + y]$ to $\lambda x.\ x + (x + 3)$. Since we wish to compute with contexts, we would also like to reduce the β-redex $(\lambda y.\ x + [\,])3$ in $a[\,]$. If we reduce it naïvely, we get $x + [\,]$, so that $a[\,]$ reduces to $\lambda x.\ x + [\,]$. Now, if we fill the hole in $\lambda x.\ x + [\,]$ with $x + y$, we get $\lambda x.\ x + (x + y)$. This shows that hole filling and β-reduction do not commute if we define hole filling and β-reduction as above. In this example, we can note that the hole in the original context $a[\,]$ is within the scope of λx and λy, while the hole in the β-reduced context is only within the scope of λx. This means that a part of the information as to which variables should be captured at the hole is lost if one reduces a β-redex which has a hole in it. Hashimoto-Ohori [7] did not solve this problem. Instead they put restriction on the β-reduction rule in their system and prohibited such β-reductions like the above example.

To solve this problem, we introduce the type A^E which represents the set of objects obtained by abstracting objects of type A with respect to a set $E = \{x_1, \ldots, x_n\}$ of variables. Canonical objects of type A^E are abstracts of the form $\kappa E.\ a$ where a is of type A and the κE binder declares that the variables in E should be understood as local in a. Moreover, E is also a type and its canonical elements are *environments* of the form $\{a_1/x_1, \ldots, a_n/x_n\}$. Then, an object a of type A^E can be instantiated to an object b of type A by applying the abstract a to an environment $e = \{a_1/x_1, \ldots, a_n/x_n\}$. We write $a{\cdot}e$ for the application of the abstract a to the environment e. For example, $(\kappa\{x, y\}.\ x + y){\cdot}\{1/x, 2/y\}$ can be reduced to 3.

In this setting, we can represent the above context $a[\,]$ as

$$C \equiv \lambda x.\ (\lambda y.\ x + X{\cdot}\{x/x, y/y\})3$$

where X represents the hole and its type is of the form $A^{\{x,y\}}$. Now, suppose that we wish to fill the hole X with the term $x + y$. Then, we can achieve this hole filling by substituting $\kappa\{x, y\}.\ x + y$ for X in C. By this substitution, we have:

$$D \equiv \lambda x.\ (\lambda y.\ x + (\kappa\{x, y\}.\ x + y)\cdot\{x/x, y/y\})3.$$

D can be reduced to $\lambda x.\ (\lambda y.\ x + (x + y))3$, which can be further reduced to $\lambda x.\ x + (x + 3)$ as expected. Let us now see what happens if we reduce the β-redex in C first and then fill the hole with $x + y$. By β-reducing C, we get $\lambda x.\ x + X\cdot\{x/x, 3/y\}$. Substituting $\kappa\{x, y\}.\ x + y$ for X in this term, we have

$$\lambda x.\ x + (\kappa\{x, y\}.\ x + y)\cdot\{x/x, 3/y\},$$

which we can further reduce to $\lambda x.\ x + (x + 3)$. We can thus see that hole filling and β-reduction commute in this case.

The idea of decorating a hole with an environment is due to Talcott [16], and Mason [9] also used this idea in his calculus of contexts that has contexts as first-class values. However, in Mason's system, environments appear in a term containing holes only as annotations. This means that such environments are objects outside the system. We present our calculus $\lambda\kappa\varepsilon$ as an extension of $\lambda\varepsilon$ [15] which is a simply typed λ-calculus that has environments as first class values. So, environments are first-class objects in $\lambda\kappa\varepsilon$. Moreover, Mason defines hole filling only as a meta-level operation. Therefore, although contexts are first-class values in his system, one cannot compute hole filling within his system. In contrast to this, we can compute hole filling within our system. For example, we can express the above example of filling $a[\,]$ with $x + y$ as follows:

$$(\lambda X.\ \lambda x.\ (\lambda y.\ x + X\cdot\{x/x, y/y\})3)(\kappa\{x, y\}.\ x + y).$$

We can compute the above term in $\lambda\kappa\varepsilon$, and we get $\lambda x.\ x + (x + 3)$.

We now turn to another problem in the formalization of contexts. Consider the informal context $\lambda x.\ [\,]$. If we fill the hole in this context with x, we get the term $\lambda x.\ x$. This term is α-equivalent to $\lambda y.\ y$. What is the context which is α-equivalent to $\lambda x.\ [\,]$ and which, when filled with x, becomes $\lambda y.\ y$? It is certainly preferable that such a context exists, since, otherwise, hole filling and α-conversion will not always commute. A naïve attempt is to α-convert $\lambda x.\ [\,]$ to $\lambda y.\ [\,]$. But this does not work, since filling $\lambda y.\ [\,]$ with x results in $\lambda y.\ x$ which is not α-equivalent to $\lambda y.\ y$. We can solve this problem easily in our setting as follows. In $\lambda\kappa\varepsilon$, the context $\lambda x.\ [\,]$ is written as $\lambda x.\ X\cdot\{x/x\}$ and this context is α-equivalent to $\lambda y.\ X\cdot\{y/x\}$. Filling these holes in these two contexts with x is achieved by substituting $\kappa\{x\}.\ x$ for X in these contexts, and the results are $\lambda x.\ (\kappa\{x\}.\ x)\cdot\{x/x\}$ and $\lambda y.\ (\kappa\{x\}.\ x)\cdot\{y/x\}$ respectively. Then they are reduced to $\lambda x.\ x$ and $\lambda y.\ y$ as expected.

In this paper we also introduce a new method of defining substitution. As we explain below, the traditional method of defining substitution does not work for our calculus. We are therefore forced to use the new method, but, we believe our new definition of substitution is mathematically cleaner than the traditional

method of defining substitution. We now give an example where the traditional method of defining substitution fails to work. By way of comparison, we first consider the λ term $a \equiv \lambda x.\ x + y$. What is the result of substituting x for y in a? We must be careful enough to avoid the variable clash and rename the bound variable x in a to a fresh variable, say, z, and we get $\lambda z.\ z + x$ as the result of the substitution. Now consider the abstract $b \equiv \kappa\{x\}.\ x + y$. What will be the result c of substituting x for y in b? If we perform substitution by the same method as above, we get $\kappa\{z\}.\ z + x$ which is wrong for the following reason. Note that $b{\cdot}\{2/x\}$ reduces to $2 + y$. So, $c{\cdot}\{2/x\}$ must reduce to $2 + x$. However, we cannot reduce $(\kappa\{z\}.\ z + x){\cdot}\{2/x\}$ since the argument $\{2/x\}$ does not match the binder $\kappa\{z\}$. By the same token, the term $(\kappa\{z\}.\ z + x){\cdot}\{2/x\}$ is not even typable. We can thus see that, unlike variables bound by the λ binder, we cannot rename variables bound by the κ binder. To cope with this situation, we introduce a new method of defining substitution where we rename *free* variables (if necessary) to achieve the capture avoiding substitution. So, our substitution will yield $\kappa\{x\}.\ x + \sharp x$ as the result of substituting x for y in b, where $\sharp x$ in the scope of the $\kappa\{x\}$ binder is a renamed form of the *free* variable x and it stands for the free variable x.

The paper is organized as follows. In section 2, we introduce the type system of $\lambda\kappa\varepsilon$, and introduce derivation rules that are used to define (typed) terms together with their types and free variables. There, we define variables so that they naturally contain both ordinary variables with names and variables as de Bruijn indices. In section 3, we define substitution as a meta-level operation. In section 4, we give reduction rules of $\lambda\kappa\varepsilon$ and give some examples of computations in $\lambda\kappa\varepsilon$. In section 5, we show that $\lambda\kappa\varepsilon$ enjoys a number of desirable properties such as confluence and strong normalizability. In section 6, we give concluding remarks. Due to lack of space, we have omitted almost all proofs. A full version of this paper with proofs is accessible at `http://www.sato.kuis.kyoto-u.ac.jp/~masahiko/index-e.html`.

2 The Type System

In this section, we define the type system of $\lambda\kappa\varepsilon$ by defining the notion of a derivation of a typing judgment. A typing judgement is an expression of the form $\Gamma \vdash a : A$, and if it is derivable then it means that the expression a is a term whose type is A and whose set of free variables is Γ.

In the following we assume that we have given a finite set of atomic types which we do not specify further in this paper. We also assume that we have infinitely many *identifiers* (i). Then, we define *variables* and *types* simultaneously as follows.

A *variable* (we will use x, y, z, u, v as meta-variables for variables) is a triple $\langle k, i, A\rangle$ where k is a natural number, i is an identifier and A is a type. A variable $\langle k, i, A\rangle$ is called a *pure variable* if $k = 0$. *Types* (A, B) are defined by the following grammar:

$$A, B ::= K \mid E \mid A \Rightarrow B \mid A^E$$

where K ranges over atomic types and E over finite sets of pure variables.

In the following, we will use *declaration* as a synonym for a finite set of variables and *pure declaration* as a synonym for a finite set of pure variables. We use Γ, Δ etc. as meta variables for declarations and E, F as meta variables for pure declarations. A pure declaration $\{x_1, \ldots, x_n\}$ will also be called an *environment type* since it is the type of environments whose canonical forms are elements of the form $\{a_1/x_1, \ldots, a_n/x_n\}$.

If $x \equiv \langle k, i, A\rangle$, then we call k the *level* of x, i the *name* of x and A the *type* of x. In this case, we sometimes write x^A for x and also write $\sharp^l x$ for $\langle k+l, i, A\rangle$, $\sharp x$ for $\sharp^1 x$, and $\bar{x}$ for $\langle 0, i, A\rangle$.

We write V for the set of all the variables. Let E be a pure declaration. We define a function $\Uparrow^E : \mathsf{V} \to \mathsf{V} - E$ by $\Uparrow^E(x) := \sharp x$ if $\bar{x} \in E$ and $\Uparrow^E(x) := x$ if $\bar{x} \notin E$. We also define $\Downarrow_E$ as the inverse function of $\Uparrow^E$. Note that $\Downarrow_E(x)$ is defined only when $x \notin E$. For example, if E is empty, then $\Uparrow^E(x) = x$ for any variable x. If E is $\{x, y\}$, then $\Uparrow^E(x) = \sharp^1 x$, $\Uparrow^E(\sharp^1 x) = \sharp^2 x$, $\Uparrow^E(z) = z$, $\Downarrow_E(\sharp^2 x) = \sharp^1 x$, and $\Downarrow_E(x)$ is undefined. We will use $\Uparrow^E$ later to systematically rename variables to avoid collision with the variables in E.

Let Γ be a declaration and E, F be pure declarations. We define the declarations $\Gamma\Uparrow^E$ and $\Gamma\Downarrow_E$ as follows:

$$\Gamma\Uparrow^E := \{\Uparrow^E(x) \mid x \in \Gamma\}, \qquad \Gamma\Downarrow_E := \{\Downarrow_E(x) \mid x \in \Gamma\},$$

where $\Gamma\Downarrow_E$ is defined only when $\Gamma \cap E$ is empty. Furthermore, given two declarations E and F, we define a function $\Updownarrow_F^E : \mathsf{V} \to \mathsf{V}$ as follows.

$$\Updownarrow_F^E(x) := \begin{cases} \Uparrow^E(x) \text{ if } x \in F, \\ \Downarrow_F(x) \text{ if } x \in E\Uparrow^F, \\ x \qquad \text{otherwise.} \end{cases}$$

We give a few examples here. If E is $\{x\}$, then $\Updownarrow_E^E(x) = \sharp x$, $\Updownarrow_E^E(\sharp x) = x$, and $\Updownarrow_E^E(\sharp^2 x) = \sharp^2 x$. If E is $\{x, y\}$ and F is $\{x\}$, then $\Updownarrow_F^E(x) = \sharp x$, $\Updownarrow_F^E(\sharp x) = x$, $\Updownarrow_F^E(y) = y$, and $\Updownarrow_F^E(z) = z$. As we will see in the next section, we will use $\Updownarrow_F^E$ to rename variables when the order of two binders are exchanged.

Using the function $\Updownarrow_F^E$, we define the declaration $\Gamma\Updownarrow_F^E$ as follows.

$$\Gamma\Updownarrow_F^E := \{\Updownarrow_F^E(x) \mid x \in \Gamma\}.$$

Lemma 1. *We have the following equations.*

1. $\Gamma\Uparrow^E \cap E = \emptyset$.
2. $\Gamma\Uparrow^E\Downarrow_E = \Gamma$. *If* $\Gamma \cap E = \emptyset$, *then* $\Gamma\Downarrow_E\Uparrow^E = \Gamma$.
3. $((\Gamma - E)\Downarrow_E - F)\Downarrow_F = ((\Gamma\Updownarrow_E^F - F)\Downarrow_F - E)\Downarrow_E$.

A *typing judgment* is an expression of the form $\Gamma \vdash a : A$ where Γ is a declaration and A is a type. We have the typing rules in Figure 1 that are used to derive typing judgments, where those rules whose names end with 'I' ('E') introduce (eliminate, respectively) the types mentioned in the rule names.

An expression a is said to be a *term* if a typing judgment of the form $\Gamma \vdash a : A$ is derivable for some Γ and A. In this case, we say that Γ is the set of *free*

$$\frac{}{\{x\} \vdash x^A : A}\ (\mathsf{axiom})$$

$$\frac{\Gamma \vdash b : B}{(\Gamma - \{x\})\Downarrow_{\{x\}} \vdash \lambda x^A.\ b : A \Rightarrow B}\ (\Rightarrow I) \qquad \frac{\Gamma \vdash b : A \Rightarrow B \quad \Delta \vdash a : A}{\Gamma \cup \Delta \vdash ba : B}\ (\Rightarrow E)$$

$$\frac{\Gamma \vdash a : A}{(\Gamma - E)\Downarrow_E \vdash \kappa E.\ a : A^E}\ (\mathsf{abs}I) \qquad \frac{\Gamma \vdash a : A^E \quad \Delta \vdash e : E}{\Gamma \cup \Delta \vdash a\bullet e : A}\ (\mathsf{abs}E)$$

$$\frac{\Gamma_1 \vdash a_1 : A_1 \quad \cdots \quad \Gamma_n \vdash a_n : A_n}{\Gamma_1 \cup \ldots \cup \Gamma_n \vdash \{a_1/x_1^{A_1}, \ldots, a_n/x_n^{A_n}\} : \{x_1, \ldots, x_n\}}\ (\mathsf{env}I)$$

$$\frac{\Gamma \vdash e : E \quad \Delta \vdash a : A}{\Gamma \cup (\Delta - E)\Downarrow_E \vdash e[\![a]\!] : A}\ (\mathsf{env}E)$$

In $(\Rightarrow I)$, the variable x must be pure.
In $(\mathsf{env}I)$, the variables $x_1, \ldots, x_n$ must be pure and mutually distinct.

Fig. 1. Typing rules of $\lambda\kappa\varepsilon$

variables in a and write $\mathrm{FV}(a)$ for it and also say that A is the *type* of a and write $\mathrm{TY}(a)$ for it. Note that if $e \equiv \{a_1/x_1, \ldots, a_n/x_n\}$, then $\mathrm{TY}(e)$ is $\{x_1, \ldots, x_n\}$. We will say that these variables are *bound by* e. We also write T for the set of all the terms. A term is *canonical* if it is of the form $\lambda x.b$, $\{a_1/x_1, \ldots, a_n/x_n\}$ or $\kappa E.\ a$, that is, if it is obtained by one of the introduction rules. A term is said to be an *environment term* if its type is an environment type.

In $(\Rightarrow I)$, since free variables in b are within the scope of λx^A, $\Downarrow_{\{x\}}$ should be applied to $\Gamma - \{x\}$ to refer to the variables in b from the outside of the binder. By the same reason, $\Downarrow_E$ is used in $(\mathsf{abs}I)$ and $(\mathsf{env}E)$. We have explained the intuitive meaning of the typing rules $(\mathsf{abs}I)$ and $(\mathsf{abs}E)$ for introducing and eliminating abstractions in section 1. The remaining typing rules come from $\lambda\varepsilon$, and the reader is referred to [15] for the detailed explanation of these rules. Here, we only remark that the term $e[\![a]\!]$ in the $(\mathsf{env}E)$ rule means to evaluate a in the environment e. So, for example, if $e \equiv \{\lambda x.\ \lambda y.\ x + y/z,\ 1/x,\ 2/u\}$, then $e[\![zxy]\!]$ is evaluated to $1 + y$. Note that z and x in zxy are bound by e and y is free in $e[\![zxy]\!]$.

We give below a simple example of a derivation. In the example below, we assume that x and y are distinct pure variables.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{y\} \vdash y : A \Rightarrow A \Rightarrow B}\ (\mathsf{axiom}) \quad \dfrac{}{\{\sharp x\} \vdash \sharp x : A}\ (\mathsf{axiom})}{\{y, \sharp x\} \vdash y(\sharp x) : A \Rightarrow B}\ (\Rightarrow E) \quad \dfrac{}{\{x\} \vdash x : A}\ (\mathsf{axiom})}{\{y, \sharp x, x\} \vdash y(\sharp x)x : B}\ (\Rightarrow E)}{\{y, x\} \vdash \lambda x.\ y(\sharp x)x : A \Rightarrow B}\ (\Rightarrow I)}{\{\} \vdash \kappa\{x, y\}.\ \lambda x.\ y(\sharp x)x : (A \Rightarrow B)^{\{x,y\}}}\ (\mathsf{abs}I)$$

It is easy to see that if $\Gamma \vdash a : A$ is derivable, then we can completely recover the entire derivation tree uniquely by inspecting the typed term a[2].

We have two kinds of abstractions λ and κ — λ abstracts nameless variables and κ abstracts named variables. We can eliminate λ by taking the distinguished named variable ι, replacing λ by $\kappa\{\iota\}$, and using the de Bruijn index method that we explain in the next section. But we did not do so, because we want to design $\lambda\kappa\varepsilon$ so that it extends the traditional λ-calculus directly. (See also the comments at the end of section 3.)

3 Substitution

In this section we define substitution as a meta-level syntactic operation. Our definition is conceptually simpler than the ordinary definition of substitution where α-conversion is sometimes necessary to avoid the unwanted capture of variables. Our method of defining substitution is a simple extension of the method due to de Bruijn [4].

Before going into technical details, we explain our method by comparing it with the traditional method of defining substitution for terms with named variables [1] and also with the method invented by de Bruijn [4]. In the traditional method, for example, substitution of x for y in $\lambda x.\ y$ is done by first α-converting $\lambda x.\ y$ to, say, $\lambda z.\ y$ and then replacing y by x. Thus, the result of substitution is $\lambda z.\ x$. The α-conversion was necessary to avoid unwanted capturing of x by the λx binder in the original term. So, in this approach, one has to define terms as equivalence classes of concrete terms modulo α-equivalence, and therefore, we have to check the well-definedness of the substitution, since we first define the substitution operation on concrete terms. Also, in this approach, one has to define α-equivalence before substitution, but the definition of α-conversion requires the notion of renaming variables which is similar to substitution.

We think that such complication in the traditional definition of substitution comes from the fact that avoidance of capturing free variables was achieved by the renaming of the name of the λ-binder. Our approach here is to avoid the capture of free variables by systematically renaming the free variables which would otherwise be captured[3]. For instance, in case of the above example of substituting x for y in $\lambda x.\ y$, we rename x to $\sharp x$ and substitute $\sharp x$ for y, so that the result of the substitution becomes $\lambda x.\ \sharp x$. We note that in the resulting term $\lambda x.\ \sharp x$, the variable $\sharp x$ is different from x within the scope of λx, and that $\sharp x$ refers to x outside the scope of λx. From this explanation, it should be easy to understand that the result of substituting $x * z$ for y in $\lambda x.\ \lambda x.\ x + y$ is $\lambda x.\ \lambda x.\ x + (\sharp^2 x * z)$. As can be seen from this example, we rename only those variables that would otherwise be captured. Therefore, in case capturing does

[2] Strictly speaking, in order to have this property, we have to identify those derivations which are the same up to the difference of ordering of the premises of the (env$I$) rule.

[3] We have introduced this idea of renaming free variables in [14].

not occur, the result of substitution obtained by our method is the same as that obtained by the traditional method.

We give another example to clarify the intuitive idea. Suppose that x, y, z are distinct pure variables. Then the following terms are all α-equivalent with each other.

$$\begin{aligned}
&\lambda x.\ \lambda y.\ (\lambda z.\ y(zx))(yx)\\
&\equiv_\alpha \lambda \sharp^0 x.\ \lambda \sharp^0 x.\ (\lambda \sharp^0 x.\ \sharp^1 x(\sharp^0 x\ \sharp^2 x))(\sharp^0 x\ \sharp^1 x)\\
&\equiv_\alpha \lambda x.\ \lambda y.\ (\lambda x.\ y(x\ \sharp x))(yx).
\end{aligned}$$

If we write $0, 1$ and 2 for $\sharp^0 x, \sharp^1 x$ and $\sharp^2 x$, respectively, in the second term above, we get $\lambda 0.\ \lambda 0.\ (\lambda 0.\ 1(02))(01)$. Therefore, this term is essentially the same as the representation of the first term in de Bruijn indices. We can therefore see that our terms are natural extensions of both traditional concrete terms with variable names and name free terms á la de Bruijn that use indices.

Let $\phi : \mathsf{V} \to \mathsf{V}$ be a (possibly) partial function such that $\phi(x)$ may be undefined for some $x \in \mathsf{V}$. We extend this function to the function $\overline{\phi} : \mathsf{T} \to \mathsf{T}$ as follows. $\overline{\phi}$ will be total if and only if ϕ is total.

1. $\overline{\phi}(x) := \phi(x)$.
2. $\overline{\phi}(\lambda x.\ a) := \lambda x.\ \overline{\phi_{\{x\}}}(a)$.
3. $\overline{\phi}(ba) := \overline{\phi}(b)\overline{\phi}(a)$.
4. $\overline{\phi}(\kappa F.\ a) := \kappa F.\ \overline{\phi_F}(a)$.
5. $\overline{\phi}(a{\cdot}f) := \overline{\phi}(a){\cdot}\overline{\phi}(f)$.
6. $\overline{\phi}(\{a_1/x_1, \ldots, a_n/x_n\}) := \{\overline{\phi}(a_1)/x_1, \ldots, \overline{\phi}(a_n)/x_n\}$.
7. $\overline{\phi}(e[\![a]\!]) := \overline{\phi}(e)[\![\overline{\phi_{\mathrm{TY}(e)}}(a)]\!]$.

where, for each declaration E, $\phi_E : \mathsf{V} \to \mathsf{V}$ is defined by

$$\phi_E(x) := \begin{cases} x & \text{if } x \in E,\\ \Uparrow^E(\phi(\Downarrow_E(x))) & \text{otherwise.}\end{cases}$$

(We note that if ϕ is not total, ϕ_E is also not total.)

We define the push operation $\uparrow^E$ by putting $a\uparrow^E := \overline{\Uparrow^E}(a)$, the pull operation $\downarrow_E$ by putting $a\downarrow_E := \overline{\Downarrow_E}(a)$, and the exchange operation $\updownarrow_F^E$ by putting $a\updownarrow_F^E := \overline{\Updownarrow_F^E}(a)$.

Let us give a few examples here. Let E be $\{x, y\}$.

$$\begin{aligned}
(\lambda x.\ x(\sharp^3 x))\uparrow^E &\equiv \overline{\Uparrow^E}(\lambda x.\ x(\sharp^3 x))\\
&\equiv \lambda x.\ \overline{\Uparrow^E{}_{\{x\}}}(x(\sharp^3 x))\\
&\equiv \lambda x.\ (\overline{\Uparrow^E{}_{\{x\}}}(x))(\overline{\Uparrow^E{}_{\{x\}}}(\sharp^3 x))\\
&\equiv \lambda x.\ x(\Uparrow^{\{x\}}\Uparrow^E\Downarrow_{\{x\}}(\sharp^3 x))\\
&\equiv \lambda x.\ x(\sharp^4 x).
\end{aligned}$$

Note that $\mathrm{FV}(\lambda x.\ x(\sharp^3 x)) = \{\sharp^2 x\}$, and $\{\sharp^2 x\}\Uparrow^E = \{\sharp^3 x\}$, which is equal to $\mathrm{FV}(\lambda x.\ x(\sharp^4 x))$. Similarly, we have $(\lambda x.\ x(\sharp^3 x))\downarrow_E \equiv \lambda x.\ x(\sharp^2 x)$.

For the exchange operation, we have:

$$(\lambda x.\ (\sharp x)(\sharp^2 x))\updownarrow^{\{x\}}_{\{x\}} \equiv \lambda x.\ (\sharp^2 x)(\sharp x),$$
$$(\lambda x.\ (\sharp x)(\sharp^2 x))\updownarrow^{\{y\}}_{\{x\}} \equiv \lambda x.\ (\sharp x)(\sharp^2 x),$$

where x and y are distinct pure variables.

We now define the substitution operation as follows. Let $s \equiv \{c_1/x_1, \ldots, c_n/x_n\}$ be a canonical environment term. Note that $\mathrm{TY}(s) = \{x_1, \ldots, x_n\}$ in this case. For each term a we define a term $a[s]$ inductively as follows. (We think that the 2nd clause below corresponds to the 2nd clause of the definition of the substitution operation $\sigma : \delta\Lambda \times \Lambda \to \Lambda$ in Fiore-Plotkin-Turi [6], and we think that it should be possible to establish a precise correspondence.)

1. $x[s] := \begin{cases} c_i & \text{if } x \equiv x_i \text{ for some } i, \\ \Downarrow_{\mathrm{TY}(s)}(x) & \text{otherwise.} \end{cases}$
2. $(\lambda x.\ b)[s] := \lambda x.\ b\updownarrow^{\mathrm{TY}(s)}_{\{x\}}[s\uparrow^{\{x\}}]$.
3. $(ba)[s] := b[s]a[s]$.
4. $(\kappa E.\ a)[s] := \kappa E.\ a\updownarrow^{\mathrm{TY}(s)}_{E}[s\uparrow^{E}]$.
5. $(a{\cdot}e)[s] := a[s]{\cdot}e[s]$.
6. $(\{a_1/x_1, \ldots, a_n/x_n\})[s] := \{a_1[s]/x_1, \ldots, a_n[s]/x_n\}$.
7. $(e[\![a]\!])[s] := e[s][\![(a\updownarrow^{\mathrm{TY}(s)}_{\mathrm{TY}(e)})[s\uparrow^{\mathrm{TY}(e)}]]\!]$.

We call $a[s]$ the result of *substituting* $c_1, \ldots, c_n$ *for* $x_1, \ldots, x_n$ *in* a.

Again we give a few examples. Let s be $\{\sharp^3 x/y,\ (x\ \sharp x)/x\}$. Then we have:

$$\begin{aligned}(\lambda x.\ x\ \sharp x)[s] &\equiv \lambda x.\ (x\ \sharp x)\updownarrow^{\{y,x\}}_{\{x\}}[s\uparrow^{\{x\}}] \\ &\equiv \lambda x.\ (\sharp x\ x)[\{\sharp^4 x/y,\ (\sharp x\ \sharp^2 x)/x\}] \\ &\equiv \lambda x.\ (\Downarrow_{\{y,x\}}\sharp x)(\sharp x\ \sharp^2 x) \\ &\equiv \lambda x.\ x(\sharp x\ \sharp^2 x), \\ (\lambda x.\ xy)[\{z/y\}] &\equiv \lambda x.\ xz,\end{aligned}$$

where x, y, z are distinct pure variables.

In the first example, x in $\lambda x.\ x\ \sharp x$ is bound by λx and $\sharp x$ is bound by s. When s goes into the scope of λx, x and $\sharp x$ should be renamed to $\sharp x$ and x, respectively so that they are bound by λx and $s\uparrow^{\{x\}}$.

We can now define the α-equivalence using substitution as follows.

1. $x \equiv_\alpha x$
2. If $a[\{z/x\}] \equiv_\alpha a'[\{z/x'\}]$, then $\lambda x.\ a \equiv_\alpha \lambda x'.\ a'$ where z is a variable such that $z \notin \mathrm{FV}(a) \cup \mathrm{FV}(a') \cup \{x, x'\}$.
3. If $b \equiv_\alpha b'$ and $a \equiv_\alpha a'$, then $ba \equiv_\alpha b'a'$.
4. If $a \equiv_\alpha a'$, then $\kappa E.\ a \equiv_\alpha \kappa E.\ a'$.
5. If $a \equiv_\alpha a'$ and $e \equiv_\alpha e'$, then $a{\cdot}e \equiv_\alpha a'{\cdot}e'$.
6. If $a_1 \equiv_\alpha a'_1, \cdots, a_n \equiv_\alpha a'_n$, then $\{a_1/x_1, \ldots, a_n/x_n\} \equiv_\alpha \{a_1/x_1, \ldots, a_n/x_n\}$.

7. If $a \equiv_\alpha a'$ and $e \equiv_\alpha e'$, then $e[\![a]\!] \equiv_\alpha e'[\![a']\!]$.

Note that variables bound by κ are not renamed in the 4th clause because κ abstracts named variables. On the other hand, variables bound by λ may be renamed in the 2nd clause bacause λ plays the same role as λ in the traditional λ-calculus.

4 Reduction Rules

In this section we give reduction rules of the $\lambda\kappa\varepsilon$ calculus. We first define $\mapsto_{\lambda\kappa\varepsilon}$ as the union of the following three relations $\mapsto_\lambda$, $\mapsto_\kappa$ and $\mapsto_\varepsilon$.

The relation $\mapsto_\lambda$ is defined by the following single rule:

(λ) $(\lambda x.\ b)a \mapsto_\lambda \{a/x\}[\![b]\!]$,

and the relation $\mapsto_\kappa$ is defined by the following single rule:

(κ) $(\kappa E.\ a)\cdot e \mapsto_\kappa e[\![a]\!]$.

The relation $\mapsto_\varepsilon$ is defined by the following 8 conversion rules.

(gc) $e[\![a]\!] \mapsto_\varepsilon a\downarrow_{\mathrm{TY}(e)}$, if $\mathrm{TY}(e) \cap \mathrm{FV}(a) = \emptyset$.
(var) $\{a_1/x_1, \ldots, a_n/x_n\}[\![x_i]\!] \mapsto_\varepsilon a_i$ $(1 \le i \le n)$.
(fun) $e[\![\lambda x.\ b]\!] \mapsto_\varepsilon \lambda x.\ (e\uparrow^{\{x\}})[\![b\updownarrow^{\mathrm{TY}(e)}_{\{x\}}]\!]$
(funapp) $e[\![ba]\!] \mapsto_\varepsilon e[\![b]\!]e[\![a]\!]$.
(abs) $e[\![\kappa E.\ a]\!] \mapsto_\varepsilon \kappa E.\ (e\uparrow^E)[\![a\updownarrow^{\mathrm{TY}(e)}_E]\!]$.
(absapp) $e[\![a\cdot f]\!] \mapsto_\varepsilon e[\![a]\!]\cdot e[\![f]\!]$.
(env) $e[\![\{a_1/x_1, \ldots, a_n/x_n\}]\!] \mapsto_\varepsilon \{e[\![a_1]\!]/x_1, \ldots, e[\![a_n]\!]/x_n\}$.
(eval) $e[\![f[\![x]\!]]\!] \mapsto_\varepsilon e[\![f]\!][\![x]\!]$, if $x \in \mathrm{TY}(f)$.

The rules other than (eval) are internalized forms of the clauses 1–6 of the definition of substitution in section 3. In these rules we have the environment term e in place of the canonical environment term s, and the rule (gc) is a generalization of the second case of clause 1. We can also internalize clause 7 directly and get a correct rule. But, we do not do so since it will result in a system where the strong normalization property does not hold. Instead we have the (eval) rule which corresponds to a special case of clause 7. Although the (eval) rule is a weak version of clause 7, we will see in Theorem 1 that we can faithfully compute substitution internally by using these reduction rules, and at the same time the system enjoys the strong normalizability (Theorem 6). In fact, as can be seen in, e.g., Melliès [10] and Bloo [2], the strong normalizability of calculi of explicit substitutions and explicit environments is a subtle problem. The reader is referred to [15] for a detailed discussion on our choice of the (eval) rule.

We write $a \to_\lambda b$ if b is obtained from a by replacing a subterm c in a by d such that $c \mapsto_\lambda d$. Similarly $\to_\kappa$, $\to_\varepsilon$ and $\to_{\lambda\kappa\varepsilon}$ are defined. The transitive closures of these reductions are denoted with asterisk (*), such as $\overset{*}{\to}_\varepsilon$. The equivalence

relation generated by $\to_{\lambda\kappa\varepsilon}$ is denoted by $=_{\lambda\kappa\varepsilon}$, namely, the reflexive, symmetric, and transitive closure of $\to_{\lambda\kappa\varepsilon}$. Similarly $=_\varepsilon$ is defined.

We give a few examples of reduction sequences. Let $s \equiv \{(x\ \sharp x)/x,\ \sharp^3 x/y\}$ in the second example.

$$\begin{aligned}(\lambda x.\ \lambda y.\ x)y &\to_\lambda \{y/x\}[\![\lambda y.\ x]\!] \\ &\to_\varepsilon \lambda y.\ (\{y/x\}\uparrow^{\{y\}})[\![x\updownarrow^{\{x\}}_{\{y\}}]\!] \\ &\equiv \lambda y.\ \{\sharp y/x\}[\![x]\!] \to_\varepsilon \lambda y.\ \sharp y.\end{aligned}$$

$$\begin{aligned}s[\![\lambda x.\ x\ \sharp x]\!] &\to_\varepsilon \lambda x.\ (s\uparrow^{\{x\}})[\![(x\ \sharp x)\updownarrow^{\{x,y\}}_{\{x\}}]\!] \\ &\equiv \lambda x.\ \{(\sharp x\ \sharp^2 x)/x,\ \sharp^4 x/y\}[\![(\sharp x\ x)]\!] \\ &\overset{*}{\to}_\varepsilon \lambda x.\ (\Downarrow_{\{x,y\}}\sharp x)(\sharp x\ \sharp^2 x) \\ &\equiv \lambda x.\ x(\sharp x\ \sharp^2 x).\end{aligned}$$

$$\begin{aligned}(\lambda X.\ \lambda y.\ X\cdot\{y/y\})(\kappa\{y\}.\ y) &\to_\lambda \{\kappa\{y\}.\ y/X\}[\![\lambda y.\ X\cdot\{y/y\}]\!] \\ &\overset{*}{\to}_\varepsilon \lambda y.\ (\kappa\{y\}.\ y)\cdot\{y/y\} \\ &\to_\kappa \lambda y.\ \{y/y\}[\![y]\!] \overset{*}{\to}_\varepsilon \lambda y.\ y.\end{aligned}$$

In the first example, y is renamed to $\sharp y$ so that it is not captured by the λy binder. The second example corresponds to the example given after the definition of substitution. The third example shows the hole-filling operation where y is captured by the λy binder.

We take an example from Hashimoto-Ohori's paper [7]. Consider the term $(\lambda z.\ C[x+z])x$ where C is an (informal) context $(\lambda x.\ [\ ]+y)3$ and $C[x+z]$ represents the hole-filling operation in the λ-calculus. In Hashimoto-Ohori's calculus, this term can be written as

$$a \equiv (\lambda z.\ (\delta X.(\lambda u.\ X^{\{u/x\}}+y)3)\odot_{\{x/v\}}(v+z))x$$

where X represents a hole, δX abstracts the hole X, and $\odot$ is a hole-filling operator. $\{u/x\}$ and $\{x/v\}$ (called renamers) annotate X and $\odot$ respectively. They are introduced to solve the problem of variable capturing. In our system, the above term can be written as

$$a \equiv (\lambda z.\ (\lambda X.\ (\lambda u.\ X\cdot\{u/x\}+y)3)(\kappa\{x\}.\ (x+z)))x.$$

We can compute this term in many ways, but, here we give two reduction sequences.

$$\begin{aligned}a &\to_\lambda \{x/z\}[\![(\lambda X.\ (\lambda u.\ X\cdot\{u/x\}+y)3)(\kappa\{x\}.\ x+z)]\!] \\ &\overset{*}{\to}_\varepsilon (\lambda X.\ (\lambda u.\ X\cdot\{u/x\}+y)3)(\kappa\{x\}.\ x+\sharp x) \\ &\to_\lambda \{\kappa\{x\}.\ x+\sharp x/X\}[\![(\lambda u.\ X\cdot\{u/x\}+y)3]\!] \\ &\overset{*}{\to}_\varepsilon (\lambda u.\ (\kappa\{x\}.\ x+\sharp x)\cdot\{u/x\}+y)3\end{aligned}$$

$$
\begin{aligned}
&\to_\kappa (\lambda u.\ \{u/x\}[\![x + \sharp x]\!] + y)3 \\
&\overset{*}{\to}_\varepsilon (\lambda u.\ u + x + y)3 \\
&\to_\lambda \{3/u\}[\![u + x + y]\!] \overset{*}{\to}_\varepsilon 3 + x + y.
\end{aligned}
$$

$$
\begin{aligned}
a &\to_\lambda (\lambda z.\ (\lambda X.\ \{3/u\}[\![X\cdot\{u/x\} + y]\!])(\kappa\{x\}.\ x + z))x \\
&\overset{*}{\to}_\varepsilon (\lambda z.\ (\lambda X.\ X\cdot\{3/x\} + y)(\kappa\{x\}.\ x + z))x \\
&\to_\lambda (\lambda z.\ \{\kappa\{x\}.\ x + z/X\}[\![X\cdot\{3/x\} + y]\!])x \\
&\overset{*}{\to}_\varepsilon (\lambda z.\ (\kappa\{x\}.\ x + z)\cdot\{3/x\} + y)x \\
&\to_\kappa (\lambda z.\ \{3/x\}[\![x + z]\!] + y)x \\
&\overset{*}{\to}_\varepsilon (\lambda z.\ 3 + z + y)x \\
&\to_\lambda \{x/z\}[\![3 + z + y]\!] \overset{*}{\to}_\varepsilon 3 + x + y.
\end{aligned}
$$

We remark that, in the second reduction sequence above, we have first reduced the innermost β-redex $(\lambda u.\ X\cdot\{u/x\} + y)3$. Such a reduction is not possible in Hashimoto-Ohori's calculus since in their system the β-conversion is prohibited when the redex contains a free hole. Though the roles of $X^{\{u/x\}}$ and $X\cdot\{u/x\}$ are similar, u in $X^{\{u/x\}}$ should always be a variable, while u in $X\cdot\{u/x\}$ can be substituted by an arbitrary term. This is the reason why our calculus need not put any restriction to the (λ)-reduction rule (the β-conversion).

We also remark on the hole-filling operations without going into the technical details. In Hashimoto-Ohori's calculus, the renamer ν in $\odot_\nu$ works as a variable binder to the second operand of $\odot$ (i.e. to the term to be filled into the hole). Because their typing rule of $M \odot_\nu N$ causes a side effect to the type of the free hole in N, they had to put the restriction that each free hole may occur at most once. Our κE binder, which plays the similar role to the renamer ν in $\odot_\nu$, does not have such a problem, because it is merely an abstraction.

Therefore, our calculus $\lambda\kappa\varepsilon$ can be regarded as a natural and flexible extension to Hashimoto-Ohori's calculus.

5 Properties of $\lambda\kappa\varepsilon$

In this section, we show that $\lambda\kappa\varepsilon$ enjoys a number of desirable properties. We first show that the meta-level operation of substitution is internally realized by the operation of evaluation (Theorem 1), and show some properties of substitution. We also show that $\lambda\kappa\varepsilon$ enjoys subject reduction property (Theorem 3), confluence property (Theorem 4), conservativity over the simply typed $\lambda\beta$-calculus (Theorem 5), and strong normalizability (Theorem 6). Theorems 4–6 establish the purity of $\lambda\kappa\varepsilon$, and as a corollary to the confluence of $\lambda\kappa\varepsilon$, we see that the operations of hole filling and β-reduction always commute.

As we have studied in [15], we can internalize the meta-level operation of substitution by means of evaluation terms which are of the form $e[\![a]\!]$. We can show that the meta-level substitution and the internalized substitution coincide, that is, $a[s] =_{\kappa\varepsilon} s[\![a]\!]$ holds.

Theorem 1. *Let s be a canonical environment term. Then, for any term a, $a[s] =_{\kappa\varepsilon} s[\![a]\!]$ holds.*

Lemma 2 corresponds to the Substitution Lemma [1] in the λ-calculus, that is, $M[x := K][y := L] \equiv M[y := L][x := K[y := L]]$ if $x \not\equiv y$ and $x \notin \mathrm{FV}(L)$.

Lemma 2 (Substitution Lemma). *Let s and t be canonical environment terms. Then, for any term a, $a[s][t] \equiv a\updownarrow^{\mathrm{TY}(t)}_{\mathrm{TY}(s)}[t\uparrow^{\mathrm{TY}(s)}][s[t]]$ holds.*

Note that the effect of exchanging the order of two substitutions s and t is adjusted by applying the exchange operation $\updownarrow^{\mathrm{TY}(t)}_{\mathrm{TY}(s)}$ to a and the push operation $\uparrow^{\mathrm{TY}(s)}$ to t. For example, let a be $x\ \sharp x$, s be $\{z/x\}$, and t be $\{\sharp^3 x/y,\ (x\ \sharp x)/x\}$ in the Lemma. Then, we have

$$\begin{aligned}
(x\ \sharp x)[s][t] &\equiv (z\ x)[t] \\
&\equiv z\ (x\ \sharp x), \\
(x\ \sharp x)\updownarrow^{\{y,x\}}_{\{x\}}[t\uparrow^{\{x\}}][s[t]] &\equiv (\sharp x\ x)[\{\sharp^4 x/y,\ (\sharp x\ \sharp^2 x)/x\}][s[t]] \\
&\equiv (x\ (\sharp x\ \sharp^2 x))[\{z/x\}] \\
&\equiv z\ ((\sharp x\ \sharp^2 x)\downarrow_{\{x\}}) \\
&\equiv z\ (x\ \sharp x).
\end{aligned}$$

(See also the example below the definition of the substitution in seciton 3.)

The reduction is compatible with substitution.

Theorem 2. *If $a \xrightarrow{*}_{\lambda\kappa\varepsilon} b$, then $a[s] \xrightarrow{*}_{\lambda\kappa\varepsilon} b[s]$.*

The following theorems will establish the purity of our calculus.

Theorem 3 (Subject Reduction). *If $\Gamma \vdash a : A$ and $a \to_{\lambda\kappa\varepsilon} b$, then $\Delta \vdash b : A$ for some $\Delta \subseteq \Gamma$.*

Theorem 4 (Confluence). *$\to_{\lambda\kappa\varepsilon}$ on $\lambda\kappa\varepsilon$-terms is confluent.*

Proof. The proof is a straightforward extension of that for $\lambda\varepsilon$, and we omit the details here. □

We remark that from the confluence of $\lambda\kappa\varepsilon$, we see that the operations of hole filling and β-reduction always commute, since in $\lambda\kappa\varepsilon$, hole filling is computed by reducing a term of the form $(\lambda X.\ a)(\kappa E.\ b)$.

We next prove that $\lambda\kappa\varepsilon$ is a conservative extension of the simply typed lambda calculus $\lambda\beta$. For this purpose, we embed the $\lambda\beta$-terms in the $\lambda\kappa\varepsilon$-terms. A $\lambda\beta$-term is a $\lambda\kappa\varepsilon$-term such that its typing derivation uses the (axiom), $(\Rightarrow I)$, $(\Rightarrow E)$ rules only, and all the variables used in the (axiom) rule are pure variables. The α- and β-conversions over $\lambda\beta$ terms are defined as usual.

Theorem 5 (Conservativity). *Let a and b be $\lambda\beta$-terms. We have $a \xrightarrow{*}_{\alpha\beta} b$ in $\lambda\beta$ if and only if $a \xrightarrow{*}_{\lambda\kappa\varepsilon} b'$ and $b \equiv_\alpha b'$ for some term b' in $\lambda\kappa\varepsilon$.*

Proof. (Only-if part) easily follows from the fact that the β-conversion can be simulated by the $\lambda\varepsilon$-reduction rules up to the α-equivalence.

(If-part) can be proved in the same way as in [15], which uses the translation from $\lambda\varepsilon$-terms to $\lambda\beta$-terms. □

Theorem 6 (Strong Normalizability). *If $\Gamma \vdash a : A$, then a is strongly normalizable.*

Proof. We can prove this theorem in the same way as the strongly normalizability theorem of $\lambda\varepsilon$ [15], because we can treat the cases of (abs) and (absapp) similarly to the cases of (fun) and (funapp). □

6 Conclusion

We have introduced a simply typed λ-calculus which has both contexts and environments as first-class values. We have shown that our calculus, $\lambda\kappa\varepsilon$, is a conservative extension of the simply typed $\lambda\beta$-calculus, enjoys subject reduction property, is confluent and strongly normalizing. Thus we have shown that our language is pure in the sense of [15] and also we have realized our hope, which we stated in the conclusion of [15], to design a pure language that has both contexts and environments as first-class values. To the best of our knowledge, $\lambda\kappa\varepsilon$ is the first such language.

We have also introduced a new method of defining substitution which is conceptually simpler than traditional methods. We think that our method is also suitable for representing terms and computing substitutions on the computer.

We conclude the paper by comparing our calculus with some related works. The style of the presentation of our paper is very close to that of Hashimoto-Ohori [7]. Both our calculus and the calculus presented in [7] are simply typed calculi which include simply typed $\lambda\beta$-calculus as their subcalculus. The system in [7] enjoys subject reduction property and is confluent. However, neither conservativity over simply typed $\lambda\beta$-calculus nor strong normalizability are shown in the paper. Therefore, it is not known whether their system is pure in our sense[4]. Also, their calculus has severe restrictions in that (i) each context may have at most one hole in it, and (ii) as we have explained in section 1, the application of the β-reduction is allowed only when the β-redex has no hole in it. Our calculus does not have such restrictions and β-reduction and hole-filling always commute.

Dami's calculus λN [5] is a very simple and powerful calculus with named variables. It is possible to represent both contexts and hole-filling in λN. However, this is done by a translation of $\lambda\beta$ calculus into λN. Therefore, it is hard to read the translated terms as contexts. On the other hand, Mason [9] introduces a system with first-class contexts in which contexts are directly represented as terms in his calculus. However, he defines hole-filling as a meta-level operation.

[4] Sakurada [12] proved the strong normalizability of Hashimoto-Ohori's calculus by interpreting it in $\lambda\varepsilon$.

It is therefore not possible to compute hole-filling within his system. Unlike these systems, in $\lambda\kappa\varepsilon$, contexts are directly representable as terms of $\lambda\kappa\varepsilon$, and we can compute hole-filling within $\lambda\kappa\varepsilon$.

Sands [13] uses Pitts' [11] definition of contexts and shows that hole-filling commutes with many relations on terms including α-equivalence. Pitts defines contexts by representing holes by (higher-order) function variables where each function variable has a fixed arity, and by representing hole-filling by substitution of a meta-abstraction for a function variable. For example, the term

$$\lambda x.\ (\lambda y.\ x + X\cdot\{x/x, y/y\})3$$

in $\lambda\kappa\varepsilon$ can be expressed by

$$\lambda x.\ (\lambda y.\ x + \xi(x, y))3$$

where ξ is a binary function variable, and the substitution

$$\{\kappa\{x, y\}.\ x + y/X\}$$

in $\lambda\kappa\varepsilon$ can be expressed by

$$[(x, y)(x + y)/\xi].$$

As can be seen by this example, Pitts' representation of contexts is structurally similar to ours, but the statuses of contexts are quite different. That is, Pitts' contexts are meta-level objects outside the object language (λ calculus in case of the above example) and our contexts are internal objects of our language $\lambda\kappa\varepsilon$. Because of this meta-level status of Pitts' contexts, Sands [13] could successfully attach contexts to many languages and could prove that hole-filling commutes with many rules in a uniform way. In contrast to this, we have been interested in internalizing such meta-level objects as contexts and environments so that we can enrich λ-calculus to a more powerful programming language.

References

1. Barendregt, H. P., *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, 1981.
2. Bloo, R. and Rose, K.H., Preservation of Strong Normalization in Named Lambda Calculi with Explicit Substitution and Garbage Collection, *Proceedings of CSN'95 (Computer Science in Netherlands)*, van Vliet J.C. (ed.), 1995. (`ftp://ftp.diku.dk/diku/semantics/papers/D-246.ps`)
3. Bognar, M. and de Vrijer, R., A calculus of lambda calculus contexts, available at: `http://www.cs.vu.nl/~mirna/new.ps.gz`.
4. de Bruijn, D. G., Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem, *Indag. Math.* 34, pp. 381-392, 1972.
5. Dami, L., A Lambda-Calculus for Dynamic Binding, pp. 201-231, *Theoretical Computer Science* **192**, 1998.

6. Fiore, M., Plotkin, G., and Turi, D., Abstract Syntax and Variable Binding (Extended Abstract), *Proc. 14th Symposium on Logic in Computer Science*, pp. 193-202, 1999.
7. Hashimoto, M. and Ohori, A., A typed context calculus, Preprint RIMS-1098, Res. Inst. for Math. Sci., Kyoto Univ., 1996, Journal version is to appear in *Theoretical Computer Science*.
8. Lee, S.-R., and D. P. Friedman, Enriching the Lambda Calculus with Contexts: Toward a Theory of Incremental Program Construction, *ACM SIGPLAN Notices, Proc. International Conference on Functional Programming*, pp. 239-250, 1996.
9. Mason, I., Computing with Contexts, *Higher-Order and Symbolic Computation* 12, pp. 171-201, 1999.
10. Melliès, P.-A., Typed λ-calculi with explicit substitutions may not terminate, Typed Lambda Calculi and Applications, *Lecture Notes in Computer Science* **902**, pp. 328-349, 1995.
11. Pitts, A.M., Some notes on inductive and co-inductive techniques in the semantics of functional programs, Notes Series BRICS-NS-94-5, Department of Computer Science, University of Aarhus, 1994.
12. Sakurada, H., An interpretation of a context calculus in an environment calculus, Master Thesis, Dept. of Information Science, Kyoto Univ., 1999 (in Japanese).
13. Sands, D., Computing with Contexts - a simple approach, Proc. Higher-Order Operational Techniques in Semantics, HOOTS II, 16 pages, *Electronic Notes in Theoretical Computer Science* **10**, 1998.
14. Sato, M., Theory of Symbolic Expressions, II, *Publ. of Res. Inst. for Math. Sci.*, Kyoto Univ., **21**, pp. 455-540, 1985.
15. Sato, M., Sakurai T., and Burstall, R., Explicit Environments, Typed Lambda Calculi and Applications, *Lecture Notes in Computer Science* **1581**, pp. 340-354, 1999.
16. Talcott, C., A Theory of binding structures and applications to rewriting, *Theoretical Computer Science* **112**:1, pp. 99-143, 1993.