Destructive Environment Operations in the Lambda Calculus with Procedural Features

Kosuke Kaneshita[1] and Shin-ya Nishizaki[2*]
[1] Institute of Science Tokyo, Ookayama Meguro Tokyo, Japan
kosuke.kaneshita@lambda.comp.isct.ac.jp
[2] Institute of Science Tokyo, Ookayama Meguro Tokyo, Japan
nisizaki@comp.isct.ac.jp
*Corresponding author.

Abstract. We have studied the lambda calculus with first-class environments in our previous work. The lambda calculus with first-class environments was an extension of the lambda calculus, but it had no destructive operation on the environment. For example, when we receive an actual parameter as a formal parameter, we only add the binding between them to the environment. In this paper, we try to introduce a mechanism that can destructively manipulate the bindings in the environment.
Also, for the purpose of formalizing the resistance to DoS attacks, we have proposed the Spice-calculus. This calculus modifies the mechanism of argument passing in the pi-calculus and the secure pi-calculus, and it makes it possible to handle memory cost explicitly. In the Spice-calculus, we can explicitly add a binding to the environment and also explicitly remove it. The explicit removal of a binding is a destructive operation on the environment. This study is conducted to further extend and deepen these previous works.
In this paper, we extend the lambda calculus by adding limited procedural programming features, such as explicit memory allocation and deallocation. We design a type system so that the amount of memory required by a subterm can be decided from the type information of its environment. By working in a purely functional framework, we can focus on the mechanism of environment operations without the extra complexity of concurrency. We define the syntax of our calculus and present a big-step operational semantics. Moreover, we propose a simple type system and show that the type preservation theorem holds for the big-step semantics in this system.

Keywords: lambda calculus, first-class environments, destructive operation, memory management, operational semantics, type preservation

1 Background

1.1 Concurrent Computation Model for estimating computational costs

In our previous work, we proposed a concurrent computation model named the *Spice calculus*[1]. This calculus is designed to estimate the resistance of commu-

J. Caro et al. (eds.), *Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2025)*, Atlantis Highlights in Computer Sciences 24,
https://doi.org/10.2991/978-94-6239-638-8_6

nication protocols to Denial-of-Service (DoS) attacks. The key idea behind this approach is to analyze the computation cost of each process. By observing the cost consumed during the execution of protocols, it becomes possible to evaluate their vulnerability to attacks that attempt to exhaust computational resources. The operational semantics of the Spice calculus provides a formal basis for such analysis. This model is particularly useful for capturing the essential behavior of attack and defense mechanisms, such as the SYN-flood attack and the SYN-cookie response, which are commonly found in network protocols. Among various aspects of computation cost, memory cost is closely related to argument passing and variable environments. A notable feature of the Spice calculus is that it enables precise tracking of memory usage by carefully defining the operational semantics.

1.2 Lambda calculus with first-class environments

The lambda calculus with first-class environments, called the environment calculus, was first proposed by Nishizaki [2]. In that paper, it was defined as an extension of the untyped lambda calculus, and weak reduction was introduced. Confluence was proved. Then, a simply typed theory was given, and the strong normalizability theorem was shown. A type inference algorithm and its principal typing theorem were also presented. Later, Nishizaki [3] extended it to an ML-polymorphic type system, where the type inference algorithm and its principal typing theorem were also presented in the same way.

The terms of the environment calculus are given by the following grammar:

$$M ::= x \mid (\lambda x.M) \mid (M\ N) \mid id \mid (M \circ N) \mid (M/x) \cdot N.$$

The semantics of the environment calculus can be defined in a simple way by using records (or labeled products) as follows. It is an extended version of the direct semantics of the ordinary simply-typed lambda calculus. In the direct semantics, a term of type $A \to B$ is interpreted as a function $[\![A]\!] \to [\![B]\!]$. This way is more direct than the continuation semantics.

$$\begin{aligned}
[\![id]\!](\rho) &= \rho, \\
[\![x]\!](\rho) = \mathit{lookup}(\rho, x), \qquad [\![(M \circ N)]\!](\rho) &= [\![M]\!]([\![N]\!](\rho)), \\
[\![(\lambda x.M)]\!](\rho) = \lambda v.[\![M]\!](\mathit{update}(\rho, x, v)), \qquad &= ([\![M]\!] \circ [\![N]\!])(\rho), \\
[\![(MN)]\!](\rho) = \big([\![M]\!](\rho)\big)\big([\![N]\!](\rho)\big), [\![(M/x) \cdot N]\!](\rho) &= \mathit{update}\big([\![N]\!](\rho), x, [\![M]\!](\rho)\big).
\end{aligned}$$

In the above semantics, ρ represents a record. The function $\mathit{lookup}(\rho, x)$ means to access the record ρ with the key x. The function $\mathit{update}(\rho, x, v)$ means to add the pair of the key x and the value v to the record ρ. In this way, we can see that the environment is not treated in a destructive way.

2 Purpose and Contributions

The purpose of this study is to extend the expressive power of environment operations in the lambda calculus, based on our previous works on the environment

calculus and the Spice-calculus. In the original environment calculus, we could add a binding of a variable to a value, but there was no operation to destructively modify or remove an existing binding. On the other hand, in the Spice-calculus, we could both add and remove bindings in the environment, from the viewpoint of argument passing and environment management. This enabled us to formalize the resistance to DoS attacks and to track memory usage explicitly. However, the Spice-calculus was defined on a concurrent computation model, and there remained issues in terms of procedural features and the simplicity of the type system.

In this study, we propose a new calculus, called λ_{de}, which extends the lambda calculus by introducing limited procedural programming features such as explicit memory allocation (`store`), deallocation (`free`), and returning values (`return`). These features allow us to perform destructive operations on environments in a controlled and type-safe manner. Our calculus is designed in a purely functional framework, so that we can focus on the mechanism of environment operations without the additional complexity of concurrency. Furthermore, we design a type system for λ_{de}, in which the memory amount required by a subterm can be decided from the type information of its environment. We also prove that the type preservation theorem holds for the big-step operational semantics of λ_{de}.

3 Lambda Calculus with Destructive Environments

In this chapter, we propose a computational system called the lambda calculus with destructive environments, written as λ_{de}. This system extends the traditional lambda calculus by introducing operations that can destructively manipulate bindings in the environment, such as explicit addition and removal. First, we define the syntax of λ_{de}. Then, we present its operational semantics, which is an extension of the call-by-value evaluation strategy, described in the style of big-step semantics.

3.1 Syntax of λ_{de}

We assume that three sets of symbols called *variables*, *constants*, and *function symbols* written as **Var**, **Const**, and **Fun**, are given in advance. In this paper, we assume that constants consist only of numeric constants and Boolean constants. We use $l, m, n, \ldots$ to denote numeric constants. As for function symbols, we assume only the function symbol intended to represent addition. For readability, we adopt the infix notation and write it as $(M + N)$. Each function symbol is assumed to be assigned a positive integer, called an *arity*, similarly to the first-order predicate logic.

Definition 1 (Terms of λ_{de}). The terms of the λ_{de} calculus are defined by the following grammar.

$$\begin{aligned} M ::= {} & n \mid \mathsf{true} \mid \mathsf{false} \mid (M + N) \mid (\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N) \\ & x \mid (\lambda x.M) \mid (MN) \\ & \mid (\mathsf{store}\ x{=}M;\ N) \mid (\mathsf{free}\ x;\ M) \mid (\mathsf{return}\ x) \end{aligned}$$

As in the usual lambda calculus, x is a variable, $\lambda x.M$ is called a *lambda abstraction*, and (MN) is called a *function application*.

c is a constant. We assume that constants include natural numbers $0, 1, 2, \ldots$ and Boolean values true and false. The term $f(M_1, \ldots, M_n)$ is called a *primitive function application*, where n is the arity of the function symbol f. We assume that primitive function applications include operations like integer addition and subtraction, such as $(M + N)$.

$(\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N)$ is called a *conditional branch*. It first evaluates L, and if the result is true, then it evaluates M and returns its result. If the result is false, then it evaluates N and returns its result.

The term $\mathsf{store}\ x{=}M;\ N$ is called a *store expression*. It is used to allocate a memory space for the variable x, evaluate M, store the result in the memory, and then evaluate N. The term $\mathsf{free}\ x;\ M$ is called a *free expression*. It is used to free the memory space of the variable x, and then evaluate M and return its result. The term $\mathsf{return}\ x$ is called a *return expression*. It is used to free the memory space of the variable x, and then return the value stored in x.

3.2 Operational Semantics of λ_{de}

We extend the syntax of terms by adding a function closure, written as $(\lambda x.M) \circ L$.

Definition 2 (Extended Term of λ_{de}). The terms of the λ_{de} calculus are defined by the following grammar.

$$\begin{aligned} M ::= {} & n \mid \mathsf{true} \mid \mathsf{false} \mid (M_1 + M_2) \mid (\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N) \\ & \mid x \mid (\lambda x.M) \mid (MN) \\ & \mid (\mathsf{store}\ x{=}M;\ N) \mid (\mathsf{free}\ x;\ M) \mid (\lambda x.M)[\rho] \\ \rho ::= {} & [\,] \mid (x, V) \cdot \rho, \\ V ::= {} & n \mid \mathsf{true} \mid \mathsf{false} \mid (\lambda x.M)[\rho] \end{aligned}$$

V is called a value. ρ is called an environment, which is a list of pairs consisting of a variable and a value.

In the following, for clarity, we focus on the binary addition operator $+$ as an example of an n-ary primitive function symbol. Multiplication and subtraction may also be introduced as needed. Among constants, those representing integers are denoted by $n, n_1, n_2, \ldots, m, m_1, m_2, \ldots$.

We do not distinguish between a term and an extended term, as long as there is no confusion. Especially in this chapter, we follow this convention. This extension is necessary for the operational semantics given later.

Definition 3 (Big-step semantics). The four-place relation $(M,\ \rho) \Downarrow (V,\ \rho')$, which relates a term M and an environment ρ to a value V and an updated environment ρ', is defined inductively by the following rules.

$$\frac{}{(n,\ \rho) \Downarrow (n,\ \rho)}\ \textbf{Num} \qquad \frac{}{(\mathsf{true},\ \rho) \Downarrow (\mathsf{true},\ \rho)}\ \textbf{True} \qquad \frac{}{(\mathsf{false},\ \rho) \Downarrow (\mathsf{false},\ \rho)}\ \textbf{False}$$

$$\frac{\mathit{lookup}(\rho,\ x) \Downarrow V}{(x,\ \rho) \Downarrow (V,\ \rho)}\ \textbf{Var}$$

$$\frac{}{((\lambda x.M),\ \rho) \Downarrow ((\lambda x.M)[\rho],\ \rho)}\ \textbf{Lambda}$$

$$\frac{\begin{cases}(M,\ \rho) \Downarrow ((\lambda x.M')[\rho'],\ \rho_1)\\ (N,\ \rho_1) \Downarrow (V_2,\ \rho_2)\\ (M',\ (x, V_2) \cdot \rho') \Downarrow (V_3,\ \rho'')\end{cases}}{((MN),\ \rho) \Downarrow (V_3,\ \rho_2)}\ \textbf{App}$$

$$\frac{(M,\ \rho) \Downarrow (V_1,\ \rho_1) \quad \mathit{overwrite}(\rho_1,\ x,\ V_1) \Downarrow \rho_2 \quad (N,\ \rho_2) \Downarrow (V_2,\ \rho_3)}{((\mathsf{store}\ x{=}M;\ N),\ \rho) \Downarrow (V_2,\ \rho_3)}\ \textbf{Store}$$

$$\frac{\mathit{restrict}(\rho,\ x) \Downarrow \rho' \quad (M,\ \rho') \Downarrow (V,\ \rho'')}{((\mathsf{free}\ x;\ M),\ \rho) \Downarrow (V,\ \rho'')}\ \textbf{Free}$$

$$\frac{\mathit{lookup}(\rho,\ x) \Downarrow V \quad \mathit{restrict}(\rho,\ x) \Downarrow \rho'}{((\mathsf{return}\ \ x),\ \rho) \Downarrow (V,\ \rho')}\ \textbf{Return}$$

$$\frac{(M_1,\ \rho) \Downarrow (n_1,\ \rho_1) \quad (M_2,\ \rho_1) \Downarrow (n_2,\ \rho_2) \quad n = n_1 + n_2}{((M_1 + M_2),\ \rho) \Downarrow (n,\ \rho_2)}\ \textbf{Plus}$$

$$\frac{(L,\ \rho) \Downarrow (\mathsf{true},\ \rho') \quad (M,\ \rho') \Downarrow (V,\ \rho'')}{(((\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N),\ \rho) \Downarrow (V,\ \rho'')}\ \textbf{IfThen}$$

$$\frac{(L,\ \rho) \Downarrow (\mathsf{false},\ \rho') \quad (N,\ \rho') \Downarrow (V,\ \rho'')}{(((\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N),\ \rho) \Downarrow (V,\ \rho'')}\ \textbf{IfElse}$$

Definition 4 (Ternary relation *lookup*$\mathbf{)}$**.** The ternary relation $\mathit{lookup}(\rho,\ x) \Downarrow V$, which relates an environment ρ and a variable x, to a value V, is defined inductively by the following rules.

$$\frac{}{\mathit{lookup}((x, V) \cdot \rho,\ x) \Downarrow V}$$

$$\frac{x \neq y \quad \mathit{lookup}(\rho,\ x) \Downarrow V}{\mathit{lookup}((y, W) \cdot \rho,\ x) \Downarrow V}$$

Definition 5 (Ternary relation *restrict*$\mathbf{)}$**.** The ternary relation $\mathit{restrict}(\rho,\ x) \Downarrow \rho'$, which relates an environment ρ and a variable x, to an environment ρ', is defined inductively by the following rules.

$$\frac{}{\mathit{restrict}((x, V) \cdot \rho,\ x) \Downarrow \rho}$$

$$\frac{\mathit{restrict}(\rho,\ x) \Downarrow \rho'}{\mathit{restrict}((y, W) \cdot \rho,\ x) \Downarrow (y, W) \cdot \rho'}$$

$restrict(\rho,\ x) \Downarrow \rho'$ means that ρ' is obtained from ρ by removing the leftmost binding for x.

Definition 6 (Quaternary relation *overwrite*).
The quaternary relation $overwrite(\rho,\ x,\ V) \Downarrow \rho'$, which relates an environment ρ, a variable x, and a value V to an environment ρ', is defined inductively by the following rules.

$$\frac{}{overwrite((x,V)\cdot\rho,\ x,\ V') \Downarrow (x,V')\cdot\rho}$$

$$\frac{}{overwrite([\,],\ x,\ V') \Downarrow (x,V')\cdot\rho}$$

$$\frac{overwrite(\rho,\ x,\ V') \Downarrow \rho'}{overwrite((y,W)\cdot\rho,\ x,\ V') \Downarrow (y,W)\cdot\rho'}$$

Example 7. We now consider an example illustrating the operational semantics.

Let M_1 be a term,

$$(\lambda x.\mathsf{store}\ y{=}x+1;\ \mathsf{store}\ z{=}y+1;\ \mathsf{free}\ y;\ \mathsf{return}\ z).$$

Let M_2 be the constant 1. Consider the term $M = (M_1\ M_2)$.

Below, we present the general structure of the derivation tree.

$((M_1\ M_2),\ \rho) \Downarrow (3,\ \rho)$
$\triangleright\ (M_1,\ \rho) \Downarrow ((\lambda x.\mathsf{store}\ y{=}x+1;\ \mathsf{store}\ z{=}y+1;\ \mathsf{free}\ y;\ \mathsf{return}\ z)[\rho],\ \rho)$
$\triangleright\ (M_2,\ \rho) \Downarrow (1,\ \rho)$
$\triangleright\ (\mathsf{store}\ y{=}x+1;\ \mathsf{store}\ z{=}y+1;\ \mathsf{free}\ y;\ \mathsf{return}\ z,\ [(x,1)\cdot\rho]) \Downarrow (3,\ [(x,1)\cdot\rho])$
$\triangleright\ \triangleright\ (\mathsf{store}\ z{=}y+1;\ \mathsf{free}\ y;\ \mathsf{return}\ z,\ (y,2)\cdot(x,1)\cdot\rho) \Downarrow (3,\ (x,1)\cdot\rho)$
$\triangleright\ \triangleright\ (\mathsf{free}\ y;\ \mathsf{return}\ z,\ (z,3)\cdot(y,2)\cdot(x,1)\cdot\rho) \Downarrow (3,\ (x,1)\cdot\rho)$
$\triangleright\ \triangleright\ \triangleright\ (\mathsf{return}\ z,\ (z,3)\cdot(x,1)\cdot\rho) \Downarrow (3,\ (x,1)\cdot\rho)$

4 Type system of λ_{de}

First, we define the syntax of types, and then we functionally define the typing judgement for terms based on the type information of environments, using typing rules.

Before defining types, we assume that countably infinite type variables are given.

Definition 8. We define the types of λ_{de} by the following grammar.

$$A ::= \iota \mid \alpha \mid (A \to B)$$

where ι denotes a primitive type and α a type variables. As a primitive type ι, we assume the number type **num** and the boolean type **bool** in this paper.

Definition 9 (Type environment of λ_{de}). We define the type environments of λ_{de} by the following grammar.

$$E ::= \{\ \} \mid \{x : A\}E$$

$\{x_1 : A_1\} \cdots \{x_n : A_1\}\{\ \}$ is written simply as $\{x_1 : A_1\} \cdots \{x_n : A_1\}$.

Definition 10 (Predicate $\in$). We define a ternary predicate $\{x : A\} \in E$ which relates a variable x, a type A and a type environment E as follows.

$$\{x_j : A_j\} \in \{x_1 : A_1\} \cdots \{x_n : A_n\}$$

if and only if $1 \leq j \leq n$ and $x_j \neq x_i$ for $1 \leq i \lneqq j$

Intuitively, $\{x : A\} \in E$ means that $\{x, A\}$ is the first binding of x in E from the left. The predicate $\in$ is related to $\mathit{restrict}(\rho, x)$, which deletes the first x-binding from ρ where ρ is typed by the type environment E.

Definition 11 (Type-binding overwrite $\{x : A\}^\frown E$). We define a function $\{x : A\}^\frown E$, which maps a variable x, a type A, and a type environment E to a type environment, is defined inductively by the following equations.

$$\begin{aligned} \{x : A\}^\frown\{\ \} &= \{x : A\}\{\ \}, \\ \{x : A\}^\frown\{x : B\}E &= \{x : A\}E, \\ \{x : A\}^\frown\{y : C\}E &= \{y : C\}(\{x : A\}E). \end{aligned}$$

The quaternary relation *overwrite* represents an operation on environments. The *type-binding overwrite* is the corresponding operation on type environments.

We also define below a type-binding restriction operation, which corresponds to the ternary relation $\mathit{restrict}(\rho,\ x) \Downarrow \rho'$.

Definition 12 (Type-binding restriction $E|_x$). We define a partial function $E|_x$, which maps an type environment E and a variable x to a type environment, is defined inductively by the following equations.

$$\begin{aligned} (\{x : A\}\rho)|_x &= \rho, \\ (\{y : B\}\rho)|_x &= \{y : B\}(\rho|_x). \end{aligned}$$

Before defining typing judgments, we assume that each constant and each primitive function symbol is assigned a type.

Definition 13 (Typing judgement for terms). We define the typing judgement $E \vdash M : A \dashv H$, which relates a type environment E and H, a term M, and a type A by the following rules.

$$\frac{}{E \vdash c : A^c \dashv E}\ \textbf{Constant}$$

$$\frac{E \vdash M_1 : A_1^f \dashv E_1 \quad \cdots \quad E \vdash M_n : A_n^f \dashv E_n}{E \vdash f(M_1, \ldots, M_n) : A^f \dashv E_n}\ \textbf{PFunc}$$

The type A^c is assigned to the constant c and the types $A_1, \ldots, A_n$ and A^f are assigned to the primitive function symbol f.

$$\frac{\{x : A\} \in E}{E \vdash x : A \dashv E}\ \textbf{Var} \qquad \frac{\{x : A\}E \vdash M : B \dashv \{x : A\}E'}{E \vdash \lambda x.M : (E \triangleright A \to B) \dashv E}\ \textbf{Lam}$$

$$\frac{E \vdash M : E' \triangleright A \to B \dashv H \quad H \vdash N : A \dashv H'}{E \vdash (MN) : B \dashv H'}\ \textbf{App}$$

$$\frac{E \vdash M : A \dashv H \quad \{x : A\}^\frown H \vdash N : B \dashv H'}{E \vdash \mathsf{store}\ x{=}M;\ N : B \vdash H'}\ \textbf{Store}$$

$$\frac{E|_x \vdash M : B \dashv H}{E \vdash \mathsf{free}\ x;\ M : B \dashv H}\ \textbf{Free}$$

$$\frac{\{x : A\} \in E}{E \vdash \mathsf{return}\ x : A \dashv E|_x}\ \textbf{Return}$$

In the judgement $E \vdash M : A \dashv H$, E represents the type of the environment that the term M refers to, and A is the type of M. After the execution of M, the resulting type of the environment is H. The environment type may change when operations such as **store** and **free** are executed in M.

The assumption of the rule **Free**, $E|_x \vdash M : B \dashv H$, implicitly requires the image $E|_x$ of the partial function to be well-defined. The same condition holds for the conclusion of the rule **Return**, $E \vdash \mathsf{return}\ x : A \dashv E|_x$.

Definition 14 (Types of closures). $E \triangleright A \to B$ is called a *type of closure* for an environment type E, types A and B.

The environment type E represents the type of the environment in which the closure is evaluated. The type of closure given here is the same as the one studied in [4] and [5].

Definition 15 (Typing judgement for environment). The typing judgement for environments $\vdash_{env} \rho : H$, which relates an environment ρ and a type environment H is defined inductively by the following rules. This definition is mutually recursive with the typing judgement for terms, which will be defined later.

$$\frac{}{\vdash_{env} [\,] : \{\,\}}\ \textbf{Empty}$$

$$\frac{\{\,\} \vdash V : A \dashv E' \quad \{\,\} \vdash \rho : H}{\vdash_{env} (x, V) \cdot \rho : \{x : A\}H}\ \textbf{Extend}$$

Definition 16 (Typing judgement for closures). The binary relation $\vdash (\lambda x.M)[\rho] : E \triangleright A \to B$ which relates a variable x, a term M and an environment ρ inductively-defined by the following rule.

$$\frac{\vdash_{env} \rho : E \quad \{x : A\}E \vdash M : B \dashv E'}{\vdash (\lambda x.M)[\rho] : E \triangleright A \to B}\ \textbf{Closure}$$

In the **Closure** rule, it is required that the type of the environment remains unchanged before and after the execution of the body M of the lambda abstraction $\lambda x.M$.

Note 17 (Type of Values). Constants, which are values other than closures, are assigned types independently of any environment. The typing of a constant under the empty environment type is written as $\{\ \} \vdash c : A^c \dashv \{\ \}$ This is written as $\vdash_{val} c : A^c$ Together with the typing of closures, we uniformly write $\vdash V : A$ for the typing of any value V.

5 Type preservation of the operational semantics

Proposition 18. For an environment ρ, a variable x, a value V, a type A, and a type environment E, if $\vdash \rho : E$, $\mathit{lookup}(\rho,\ x) \Downarrow V$, and $\{x : A\} \in E$, then $\vdash V : A$.

Proposition 19. If $\vdash \rho : E$, $\mathit{restrict}(\rho,\ x) \Downarrow \rho'$, and $E|_x$ is defined, then $\vdash \rho' : E|_x$.

Theorem 20 (Type Preservation). For a term M, environments ρ, ρ', a value V, and environment types E, H, if

$$(M,\ \rho) \Downarrow (V,\ \rho'), \quad E \vdash M : A \dashv H, \quad \text{and} \quad \vdash \rho : E$$

for some type environment H and some type A, then

$$\vdash V : A \quad \text{and} \quad \vdash \rho' : H$$

.

Proof. We prove this theorem by induction on the structure of the derivation of $(M,\ \rho) \Downarrow (V,\ \rho')$.
Base case Var: Assume that

$$E \vdash x : A \dashv, E \tag{1}$$
$$\vdash \rho : E, \tag{2}$$
$$(x,\ \rho) \Downarrow (V,\ \rho) \tag{3}$$

By applying the typing rule **Var** and (1), we obtain

$$\{x : A\} \in E, \tag{4}$$

and from the operational semantics rule **Var** and (3), we obtain

$$\mathit{lookup}(\rho,\ x) \Downarrow V, \tag{5}$$

From Proposition 18 together with (2), (4), and (5), we obtain $\vdash V : A$. In conjunction with (2), this completes the proof for this case.
Inductive case Lam: We assume that

$$E \vdash (\lambda x.M) : (E \triangleright A \to B) \dashv E, \tag{6}$$
$$\vdash \rho : E, \text{and} \tag{7}$$
$$(\lambda x.M,\ \rho) \Downarrow ((\lambda x.M)[\rho],\ \rho). \tag{8}$$

From (6) and (7), it follows that $\vdash (\lambda x.M)[\rho] : E \triangleright A \to B$. Together with (7) and (8), this concludes the proof for this case.

Inductive case App: Assume that

$$E \vdash (MN) : B \dashv H', \tag{9}$$
$$\vdash \rho : E, \text{and} \tag{10}$$
$$((MN),\ \rho) \Downarrow (V_3,\ \rho_2) \tag{11}$$

By applying the typing rule **App** and (9), we obtain

$$E \vdash M : (E' \triangleright A \to B) \dashv H, \text{and} \tag{12}$$
$$H \vdash N : A \dashv H'. \tag{13}$$

By applying the operational semantics rule **App** and (11), we obtain

$$(M,\ \rho) \Downarrow ((\lambda x.M')[\rho'],\ \rho_1), \tag{14}$$
$$(N,\ \rho_1) \Downarrow (V_2,\ \rho_2), \text{and} \tag{15}$$
$$(M',\ (x_2, V_2) \cdot \rho') \Downarrow (V_3,\ \rho_3). \tag{16}$$

From (12), (14), (10), and together with the induction hypothesis, it follows that

$$\vdash (\lambda x.M')[\rho'], \text{and} \tag{17}$$
$$\vdash \rho_1 : H. \tag{18}$$

From (13), (15), (18), and together with the induction hypothesis, it follows that

$$\vdash V_2 : A, \text{and} \tag{19}$$
$$\vdash \rho_2 : H'. \tag{20}$$

From (17), it follows that

$$E' \vdash \lambda x.M' : E' \triangleright A \to B \tag{21}$$

Therefore, there exists a type environment E'' such that

$$\{x : A\}E' \vdash M' : B \dashv E''. \tag{22}$$

From (17), we also have $\vdash \rho' : E'$.Together with (19), this yields

$$\vdash (x, V_2) \cdot \rho' : \{x : A\}E'., \tag{23}$$

From (22), combined with the induction hypothesis, we obtain

$$\vdash V_3 : B, \text{and} \tag{24}$$
$$\vdash \rho_3 : E''.$$

Finally, (24) and (20) complete the proof for this case.

Inductive case Store: Assume that

$$E \vdash \mathsf{store}\ x{=}M;\ N : B \dashv H'', \tag{25}$$
$$\vdash \rho : E, \text{and} \tag{26}$$
$$(\mathsf{store}\ x{=}M;\ N,\ \rho) \Downarrow (V_2,\ \rho_2). \tag{27}$$

By applying the typing rule **Store** to (25), we obtain

$$E \vdash M : A \dashv H, \tag{28}$$
$$\{x : A\}^\frown H = H', \text{and} \tag{29}$$
$$H' \vdash N : B \dashv H''. \tag{30}$$

By applying the operation semantics rule **Store** to (27), we obtain

$$(M,\ \rho) \Downarrow (V_1,\ \rho_1), \tag{31}$$
$$\mathit{overwrite}(\rho_1,\ x,\ V_1) \Downarrow \rho_1', \text{and} \tag{32}$$
$$(N,\ \rho_1') \Downarrow (V_2,\ \rho_2). \tag{33}$$

From (28), (26), and (31), combined with the induction hypothesis, we obtain

$$\vdash V_1 : A, \text{and} \tag{34}$$
$$\vdash \rho_1 : H. \tag{35}$$

From (32), (34), (35), and (29), it follows that

$$\vdash \rho_1' : H', \tag{36}$$

From (30), (36), and (33), combined with the induction hypothesis, we obtain

$$\vdash V_2 : B \text{ and } \vdash \rho_2 : H''.$$

This completes the proof for this case.

Inductive case Free: Assume that

$$E \vdash \mathsf{free}\ x;\ M : B \dashv H, \tag{37}$$
$$\vdash \rho : E, \text{and} \tag{38}$$
$$(\mathsf{free}\ x;\ M,\ \rho) \Downarrow (V,\ \rho''). \tag{39}$$

By applying the typing rule **Free** to (37), we obtain

$$E|_x \vdash M : B \dashv H. \tag{40}$$

By applying the operational semantics rule **Free** to (39), we obtain

$$\mathit{restrict}(\rho,\ x) \Downarrow \rho', \text{and} \tag{41}$$
$$(M,\ \rho') \Downarrow (V,\ \rho''). \tag{42}$$

Since $E|_x$ is well-defined from , applying (38) and (41) to Proposition 19, we obtain

$$\vdash \rho' : E|_x. \tag{43}$$

Finally, by applying (43) and (42) to the induction hypothesis, we obtain

$$\vdash V : B \text{ and } \vdash \rho'' : H.$$

This completes the proof for this case.
Inductive case Return: Assume that

$$E \vdash \mathsf{return}\ x : A \dashv E_x, \tag{44}$$

$$\vdash \rho : E, \text{and} \tag{45}$$

$$(\mathsf{return}\ x,\ \rho) \Downarrow (V,\ \rho). \tag{46}$$

By applying the typing rule **Return** to (44), we obtain

$$\{x : A\} \in E. \tag{47}$$

By applying the operational semantics rule **Return** to (46), we obtain

$$\mathit{lookup}(\rho,\ x) \Downarrow V \text{ and} \tag{48}$$

$$\mathit{restrict}(\rho,\ x) \Downarrow \rho'. \tag{49}$$

From (47), (45), and (48) Proposition 18 yields $\vdash V : A$.

Futhermore, from (47), (49). and (45), Proposition 19 yields $\vdash \rho' : E|_x$. This completes the proof for this case, and thus the proof of the theorem as a whole is now complete. ■

6 Conclusion and Future Work

In this paper, we have proposed a new lambda calculus, called λ_{de}, which enables type-safe destructive operations on environments. Our calculus is based on our previous works on the environment calculus and the Spice-calculus. In λ_{de}, we introduce limited procedural programming features such as explicit memory allocation (`store`), deallocation (`free`), and returning values (`return`) into a purely functional framework, and we define its big-step operational semantics. We have also designed a simple type system in which the memory amount required by a subterm can be decided from the type information of its environment. Furthermore, we have proved that the type preservation theorem holds for the big-step operational semantics of λ_{de}.

A possible direction of future work is to investigate the applicability of our calculus to resource management other than memory, such as computation time or communication channels. In addition to the present study, there have been works[6][7][8][9] that try to incorporate mechanisms of procedural programming languages into functional languages. We would like to study the relation between those works and our research in a semantic approach. Implementation of λ_{de} and its application to program verification are also interesting subjects for future research.

References

1. Tomioka, D., Nishizaki, S., Ikeda, R.: A cost estimation calculus for analyzing the resistance to denial-of-service attack. In: Software Security – Theories and Systems, *Lecture Notes in Computer Science*, vol. 3233, pp. 25–44. Springer, Berlin, Heidelberg (2004). DOI 10.1007/978-3-540-37621-7_2
2. Nishizaki, S.: Simply typed lambda calculus with first-class environments. Publications of the Research Institute for Mathematical Sciences, Kyoto University **30**(6), 1055–1121 (1994). DOI 10.2977/prims/1195164948. URL https://doi.org/10.2977/prims/1195164948
3. Nishizaki, S.: A polymorphic environment calculus and its type-inference algorithm. Higher-Order and Symbolic Computation **13**(3), 239–278 (2000). DOI 10.1023/A:1010010314528. URL https://doi.org/10.1023/A:1010010314528
4. Nishizaki, S., Takayanagi, Y.: Extracting environments from function closures. In: Proceedings of the 2022 11th International Conference on Software and Computer Applications (ICSCA), pp. 61–68. ACM (2022). DOI 10.1145/3512353.3512361
5. Nishizaki, S.: Transplanting of environments between closures in the lambda calculus. In: Proceedings of the 2023 12th International Conference on Software and Computer Applications (ICSCA), pp. 122–130. ACM (2023). DOI 10.1145/3587716.3587755
6. Mason, I.A., Talcott, C.: Equivalence in functional languages with effects. Journal of Functional Programming **1**(3), 287–327 (1991). DOI 10.1017/S0956796800000125. URL https://doi.org/10.1017/S0956796800000125
7. Talcott, C.: Reasoning about programs with effects. Electronic Notes in Theoretical Computer Science **20**, 1–23 (1998). DOI 10.1016/S1571-0661(05)80243-9. URL https://doi.org/10.1016/S1571-0661(05)80243-9
8. Wadler, P.: Monads for functional programming. In: J. Jeuring, E. Meijer (eds.) Advanced Functional Programming, *Lecture Notes in Computer Science*, vol. 925, pp. 24–52. Springer, Berlin, Heidelberg (1995). DOI 10.1007/3-540-59451-5_2. URL https://doi.org/10.1007/3-540-59451-5_2
9. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: Proceedings of the 18th European Symposium on Programming (ESOP 2009), *Lecture Notes in Computer Science*, vol. 5502, pp. 80–94. Springer (2009). DOI 10.1007/978-3-642-00590-9_7. URL https://doi.org/10.1007/978-3-642-00590-9_7