Simplification of Abstract Machine for Functional Language and Its Theoretical Investigation

Shin-Ya Nishizaki*, Kensuke Narita, Tomoyuki Ueda
Department of Computer Science, Tokyo Institute of Technology, Ookayama, Meguro, Tokyo, Japan.

* Corresponding author. Tel.: +81-3-5734-2772; email: nisizaki@cs.titech.ac.jp
Manuscript submitted February 18, 2015; accepted March 4, 2015.
doi: 10.17706/jsw.10.10.1148-1159

Abstract: Many researchers have studied abstract machines in order to give operational semantics to various kinds of programming languages. For example, Landin's SECD machine and Curien's Categorical Abstract Machine are proposed for functional programming languages and useful not only for theoretical studies but also implementation of practical language processors. We study simplification of SECD machine and design a new abstract machine, called Simple Abstract Machine. We achieve the simplification of SECD machine by abstracting substitutions for variables. In Simple Abstract Machine, we can formalize first-class continuations more simply and intelligibly than SECD machine.

Key words: Programming language theory, lambda calculus, first-class continuation, abstract machine.

1. Introduction

Many researchers have proposed abstract machines in order to give operational semantics to various kinds of programming languages, such as Landin's SECD machine [1], [2] and Curien's Categorical Abstract machine [3], [4]. From the theoretical viewpoint, there are many kinds of researches of abstract machines.

Graham formalized and verified implementation of SECD machine using the HOL prover [5]. Hardin *et al*. formulated abstract machines for functional languages in the framework of explicit substitutes [6]. Ohori studied abstract machines from the proof-theoretic approach [7]. Curien *et al*. investigated Curry-Howard isomorphism with respect to jump instructions in abstract machines [8].

We study simplification of SECD machine in order to clarify the continuation in the framework of abstract machine. The traditional abstract machines, such as SECD machine, consist of complicated internal configurations. SECD machine has four data sequences: Stack, Environment, Code, and Dump. It is not unclear which components correspond to the continuation. Simplifying SECD machine, we obtain a clear formalization of continuation in the framework of abstract machine. We call it the Simplified Abstract Machine (SAM).

2. Simple Abstract Machine

In this section, we define an abstract machine based on a call-by-value evaluation strategy, called, the Simple Abstract Machine (SAM). We can regard SAM as a simplified version of the SECD machine, which is explained in a later section.

The three kinds of atomic symbols, primitive functions, numerals, and variables, are given in advance of the definition of instructions and codes. Numerals represents integers such as 0,1,2,3,...,-1,-2,-3,... We represent numerals by *n*, *n'*, ..., and variables by *x*, *y*, *z*,..., respectively. In this paper, we consider only the successor function on numerals, s, for simplicity's sake.

Definition 1 (Instructions and Codes of SAM) Instructions and Codes of SAM are defined inductively by the following grammar.

$$I ::= n \mid s \mid \ x \ \mid \ lam(x, C) \mid app$$
$$C ::= I_1 : \ \ I_2 \ : \cdots : I_n$$

The expression lam (*x*, *C*) is called a lambda abstraction and app an application. To simplify the instruction set, we postulate that the primitive functions are unary. We use *I*, *I'*, I_1, I_2,... for instructions and *C*, *C'*, C_1, C_2,... for codes.

Definition 2 (Instructions and Codes of SAM) A set of Values is a subset of the set of instructions, defined by the following grammar. Each element of Values is called a value.

$$V ::= n \ \mid \ s \ \mid \ x \ \mid \ f(V) \ \mid \ \ lam(x, C).$$

Definition 2 (Transition of SAM) A configuration of SAM is represented as a pair (*S*, *C*) of a stack *S*, which is a sequence of value, and a code *C*. Computation of SAM is formulated as transition between configurations defined by the following rules.

$$\begin{array}{rc} num & (S, n{:}\,C) \to (n{:}\,S, C) \\ prim & (S, s{:}\,C) \to (s{:}\,S, C) \\ app\text{-}lam & (S, lam(x, C'){:}\,C) \to (S, C'[x \coloneqq V]{:}\,C) \\ app\text{-}prim & (n{:}\,s{:}\,S, app{:}\,C) \to (n'{:}\,S, C) \end{array}$$

where $n' = n + 1$

The rules num, prim, and lam intend that a value on the code component is pushed to the stack component as an actual parameter which will be bound later.

The rule lam means that if you find an application instruction app at the head of the code component, then the actual parameter V is bound to the formal parameter x and the body of the function *C'*[*x*: = *V*] is evaluated.

We show an example of translation sequence of SAM.

Example 1 (Transition Sequence of SAM) Consider a code

$$lam(x, s{:}\,s{:}\,x{:}\,app{:}\,app){:}\,3{:}\,app.$$

The following is a transition sequence starting with a configuration of an empty stack and the code. We write the empty sequence as { }.

$$\begin{array}{ll} (\{\,\}, lam(x, s{:}\,s{:}\,x{:}\,app{:}\,app){:}\,3{:}\,app) & \\ \to (lam(x, s{:}\,s{:}\,x{:}\,app{:}\,app), 3{:}\,app) & \text{(lam)} \\ \to (3{:}\,lam(x, s{:}\,s{:}\,x{:}\,app{:}\,app), app) & \text{(num)} \\ \to(\{\,\},\ \ s{:}\,s{:}\,3{:}\,app{:}\,app) & \text{(app)} \\ \to (s, s{:}\,3{:}\,app{:}\,app) & \text{(prim)} \\ \to(s{:}\,s, 3{:}\,app{:}\,app) & \text{(num)} \end{array}$$

$$\begin{array}{ll} \to(3\colon s\colon s, app\colon app) & \text{(app-prim)} \\ \to(4\colon s, app) & \text{(app-prim)} \\ \to(5,\ \{\ \}) & \text{(app-prim)} \end{array}$$

Next, we give a translation of SAM into the call-by-value lambda calculus. Before giving the definition of the translation, we introduce the call-by-value lambda calculus.

3. The Call-by-Value Lambda Calculus

In this section, we present the call-by-value lambda calculus.

Definition 4 (Terms) Terms of the lambda calculus are defined inductively by the following grammar:

$$M ::= n \mid s \mid x \mid (\lambda x. M) \mid (M\ N)$$

Definition 5 (Values) Values are defined inductively by the following grammar:

$$V ::= n \mid s \mid x \mid (\lambda x. M)$$

The set of values is a subset of the set of term. The values defined above are the terms that cannot be reduced by the call-by-value reduction introduced below.

Definition 6 (Evaluation context) A context is a term with a hole [ ]. Evaluation contexts for call-by-value evaluation are contexts defined inductively by the following grammar:

$$E[\,] ::= [\,] \mid (E[\,]\ M) \mid (V\ E[\,])$$

The following property on evaluation contexts is trivially derived.

Proposition 1. For every term M, there exists an evaluation context $E[\,]$ and a value V satisfying that

$$M = E[V].$$

Definition 7 (Call-by-value reduction) The call-by-value reduction is a binary relation between terms defined inductively by the following rules.

$$\textit{beta-cbv} \qquad E\big[\,((\lambda x. M)V)\big] \to E\big[M[x := V]\big],$$

In this paper, we assume the successor function on natural numbers as primitive.

$$\textit{prim} \qquad E[(s\ n)] \to E[n'], \textit{where}\ \ n' = n + 1.$$

For example, a function symbol s which means successor function, it is defined as

$$\mathrm{E}[(\mathrm{s}\ 0)] \to \mathrm{E}[1], \mathrm{E}[(\mathrm{s}\ 1)] \to \mathrm{E}[2], \mathrm{E}[(\mathrm{s}\ 2)] \to \mathrm{E}[3], \ldots$$

Since the left-hand side of the reduction rules are not overlapped each other, we have the uniqueness of the call-by-value reduction.

We finish this section with an example of reduction sequence of the call-by-value lambda calculus.

Example 2 (Reduction Sequence) We show a reduction sequence of a term $\Big(\big(\lambda x. \big(s\ (s\ x)\big)\big)\ 3\Big)$.

$$
\begin{array}{ll}
\left(\left(\lambda x.\left(s\ (s\ x)\right)\right)\ 3\right) & \\
\to \left(s\ (s\ 3)\right) & \text{(beta-cbv)} \\
\to (s\ 4) & \text{(prim)} \\
\to 5. & \text{(prim)}
\end{array}
$$

4. Translation of Call-by-Value Lambda Calculus into SAM

The code components of SAM are assumed to be given as results of the following translation of lambda terms.

Definition 8 (Translation of lambda terms to codes)

A translation mapping [| M |] of lambda terms M can be given as

$$
\begin{aligned}
[\![n]\!] &= n, \\
[\![s]\!] &= s, \\
[\![x]\!] &= x, \\
[\![(\lambda x.M)]\!] &= lam(x, [\![M]\!]), \\
[\![(M\ N)]\!] &= [\![M]\!]: [\![N]\!]: app.
\end{aligned}
$$

Example 3 (Translation of Terms into Codes) Consider a lambda term $\left(\left(\lambda x.\left(s\ (s\ x)\right)\right) 3\right)$, where s is a unary primitive function and 3 is a constant. The term is translated as follows.

$$
\begin{aligned}
&[\![\left(\left(\lambda x.\left(s\ (s\ x)\right)\right) 3\right)]\!] \\
&= [\![\lambda x.\left(s\ (s\ x)\right)]\!]: [\![3]\!]: app \\
&= lam(x,)[\![s\ (s\ x)]\!]: 3: app \\
&= lam(x, s: s: x: app: app): 3: app.
\end{aligned}
$$

The translation maps a value of the call-by-value lambda calculus to a value of SAM.

Proposition 2. For a value V of the call-by-value lambda calculus, [| V |] is a value of SAM.

Definition 9 (Translation of Terms into Configurations) For a term M, a sequence of values V_1,..., V_m, and a sequence of terms L_1,...,L_n, we define a mapping

$$\kappa[\![M]\!](V_1: \cdots : V_m, L_1: \cdots : L_n).$$

to a configuration inductively by the following rules.

$$
\begin{aligned}
\kappa[\![V]\!](V_1: \cdots : V_m, L_1: \cdots : L_n) &= ([\![V]\!]: V_1: \cdots : V_m, L_1: \cdots : L_n) \\
\kappa[\![(V\ N)]\!](V_1: \cdots : V_m, L_1: \cdots : L_n) &= \kappa[\![N]\!]([\![V]\!]: V_1: \cdots : V_m, app: L_1: \cdots : L_n) \\
\kappa[\![(M\ N)]\!](V_1: \cdots : V_m, L_1: \cdots : L_n) &= \kappa[\![M]\!](V_1: \cdots : V_m, [\![N]\!]: app: L_1: \cdots : L_n)
\end{aligned}
$$

We can extend the translation on the evaluation contexts by adding the rule for the hole:

$$\kappa[\![[\ ]]\!](V_1: \cdots : V_m, L_1: \cdots : L_n) = \left(V_1: \cdots : V_m,\ [\ ]: L_1: \cdots : L_n\right)$$

Example 4 (Translation of Terms into Configurations) A term $((\lambda x.\ (s\ (s\ x)))\ 3)$ is translated to a configuration as follows.

$$\kappa\left[\!\!\left[\left(\left(\lambda x.\left(s\ (s\ x)\right)\right)3\right)\right]\!\!\right]\ (\{\ \},\{\ \}) = \kappa[\![3]\!]\left([\![\lambda x.\left(s(s\ x)\right)]\!], app\right) = (3, lam(x, s{:}\ s{:}\ x{:}\ app{:}\ app), app).$$

Another example of the translation is as follows.

$$\kappa\left[\!\!\left[\left(\left((\lambda y.y)\left(\lambda x.\left(s(s\ x)\right)\right)\right)3\right)\right]\!\!\right](\{\ \},\{\ \}) = \kappa\left[\!\!\left[\left((\lambda y.y)\left(\lambda x.\left(s(s\ x)\right)\right)\right)\right]\!\!\right](\{\ \}, 3{:}\ app)$$
$$= \kappa[\![\lambda x.\left(s(s\ x)\right)]\!]([\![\lambda y.y]\!], app{:}\ 3{:}\ app) = ([\![(\lambda x.\left(s(s\ x)\right)]\!]{:}\ [\![\lambda y.y]\!], app{:}\ 3{:}\ app)$$
$$= ((lam(x, s{:}\ s{:}\ x){:}\ lam(y, y), app{:}\ 3{:}\ app).$$

This translation maps evaluation contexts to a pair of stack's and code's subsequences. The following proposition is straight-forwardly proved by induction on context $E[\ ]$.

Proposition 3. For an evaluation context $E[V]$ and a value, there exist $V_1'{:}\cdots, V'_{m'}$ and $L'_1{:}\cdots{:}L'_{n'}$,

$$\kappa[\![E[V]]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n) = \kappa[\![V]\!](V_1'{:}\cdots, V'_{m'}{:}\ V_1{:}\cdots{:}V_m, L'_1{:}\cdots{:}L'_{n'}{:}\ L_1{:}\cdots{:}L_n).$$

Proposition 4.

$$(V_1{:}\cdots{:}V_m, [\![E[V]]\!]{:}\ L_1{:}\cdots{:}L_n) \to^* \kappa[\![E[V]]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n).$$

Proof. We prove this proposition on the structure on the evaluation context E [ ].

Case of empty context [ ].

$$(V_1{:}\cdots{:}V_m, [\![V]\!]{:}\ L_1{:}\cdots{:}L_n) \to ([\![V]\!]{:}\ V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n) = \kappa[\![V]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n).$$

Case of (W E[ ]), where W is a value.

$$(V_1{:}\cdots{:}V_m, [\![(W\ E[V])]\!]{:}\ L_1{:}\cdots{:}L_n) = (V_1{:}\cdots{:}V_m, [\![W]\!]{:}\ [\![E[V]]\!]{:}\ app{:}\ L_1{:}\cdots{:}L_n)$$
$$\to ([\![W]\!]{:}\ V_1{:}\cdots{:}V_m, [\![E[V]]\!]{:}\ app{:}\ L_1{:}\cdots{:}L_n) \to^*\ \kappa[\![E[V]]\!]([\![W]\!]{:}\ V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n)$$

because of the induction hypothesis. On the other hand,

$$\kappa[\![(W\ E[V])]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n) = \kappa[\![E[V]]\!]([\![W]\!]{:}\ V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n).$$

Hence,

$$(V_1{:}\cdots{:}V_m, [\![(W\ E[V])]\!]{:}\ L_1{:}\cdots{:}L_n) \to^* \kappa[\![(W\ E[V])]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n)$$

Case of $(E[\ ]\ M)$,

$$(V_1{:}\cdots{:}V_m, [\![(E[V]\ M)]\!]{:}\ L_1{:}\cdots{:}L_n) =$$
$$(V_1{:}\cdots{:}V_m, [\![E[V]]\!]{:}\ [\![M]\!]{:}\ app{:}\ L_1{:}\cdots{:}L_n) \to^* \kappa[\![E[V]]\!](V_1{:}\cdots{:}V_m, [\![M]\!]{:}\ app{:}\ L_1{:}\cdots{:}L_n)$$ because of the induction hypothesis, $$= \kappa[\![(E[V]M)]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n)$$

End of Proof.

Theorem 1 (Soundness of Translation) For any term M, $L'_1{:}\cdots{:}L'_{n'}$, and values $V'_1{:}\cdots{:}V'_{m'}$, if $M \rightarrow M'$, then it holds that

$$\kappa[\![M]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n) \rightarrow^* \kappa[\![M']\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n).$$

Proof. $M \rightarrow M'$ is derived from reduction rule beta-cbv or prim.

Case of beta-cbv:

$$\kappa[\![E[(\lambda x.M)V]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n) \rightarrow^* \kappa[\![E[M[x \coloneqq V]]]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n)$$

This is derived as follows. We abbreviate $L'_1{:}\cdots{:}L'_{n'}$ as $\overline{L'_{n'}}$.

$\kappa[\![E[(\lambda x.M)V]\!](\overline{V_m}, \overline{L_n}) = \kappa[\![E[(\lambda x.M)V]\!]\left(\overline{V'_{m'}}{:}\overline{V_m}, \overline{L'_{n'}}{:}\overline{L_n}\right)$ since Proposition 3, $= \left([\![V]\!]{:}\, lam(x,[\![M]\!]){:}\,\overline{V'_{m'}}{:}\overline{V_m}, app{:}\,\overline{L'_{n'}}{:}\overline{L_n}\right) \rightarrow \left(\overline{V'_{m'}}{:}\overline{V_m}{:}\,[\![M]\!][x \coloneqq [\![V]\!]]{:}\,\overline{L'_{n'}}{:}\overline{L_n}\right) = (\overline{V'_{m'}}{:}\overline{V_m}, [\![M[x \coloneqq$ *V:L'n':Ln) →∗κMx:=V(V'm':Vm, L'n':Ln)* since Proposition 4= *κE[Mx:=V]Vm, Ln.*

Case of prim:

$$\kappa[\![E(s\ n)]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n) \rightarrow^* \kappa[\![E[n']]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n) \ \text{ where } \ n' = n + 1.$$

This is derived as follows.

$$\kappa[\![E[(s\ n)]\!](\overline{V_m}, \overline{L_n}) = \kappa[\![(s\ n)]\!]\left(\overline{V'_{m'}}{:}\overline{V_m}, \overline{L'_{n'}}{:}\overline{L_n}\right) = \kappa[\![n]\!]\left([\![s]\!]{:}\,\overline{V'_{m'}}{:}\overline{V_m}, app{:}\,\overline{L'_{n'}}{:}\overline{L_n}\right)$$
$$= \left([\![n]\!]{:}\,[\![s]\!]{:}\,\overline{V'_{m'}}{:}\overline{V_m}, app{:}\,\overline{L'_{n'}}{:}\overline{L_n}\right)$$
$$= \left(n{:}\,s{:}\,\overline{V'_{m'}}{:}\overline{V_m}, app{:}\,\overline{L'_{n'}}{:}\overline{L_n}\right) \rightarrow \left(n'{:}\,\overline{V'_{m'}}{:}\overline{V_m}, \overline{L'_{n'}}{:}\overline{L_n}\right) \ \textit{where}$$
$$n' = n + 1, = \kappa[\![n']\!]\left(\overline{V'_{m'}}{:}\overline{V_m}, \overline{L'_{n'}}{:}\overline{L_n}\right) = \kappa[\![E[n']]\!](V_1{:}\cdots{:}V_m, L_1{:}\cdots{:}L_n)$$

End of Proof.

5. Simple Type System for SAM

In this section, we give a simple type system to SAM. The point of the simple type system is that functions can be typed of function type. For example, a function of numbers to numbers is of type $(num \rightarrow num)$ and a function of such kind of functions to numbers of type $((num \rightarrow num) \rightarrow num)$.

Definition 10 (Types of SAM) Types of SAM are defined inductively by the following grammar.

$$A \coloneqq num \mid (A \rightarrow B).$$

Type num is a primitive type which represents the set of non-negative integers. Type $(A \rightarrow B)$ represents the set of functions whose domain is A and codomain B.

Definition 11 (Typing of Codes) Type judgment $\Gamma \vdash C : A$ is defined inductively by the following rules, which means that C is of type A under type assignment Γ.

A type assignment Γ is a mapping whose domain is a finite set of variables and codomain is the set of types. If a type assignment maps $x_1,\ldots, x_n$ to $A_1,\ldots, A_n$ respectively, then we write it as $\{x_1{:}\, A_1\}\cdots\{x_n : A_n\}$.

$$\frac{}{\Gamma \vdash n: num}\ ,\ \frac{}{\Gamma \vdash s: num \to num},\quad \frac{\Gamma(x)=A}{\Gamma\vdash x:A}\quad \frac{\Gamma\vdash C_1:(A\to B)\ \ \Gamma\vdash C_2:A}{\Gamma\vdash(C_1:C_2:app):B},\quad \frac{\{x:A\}\Gamma\vdash C:B}{\Gamma\vdash lam(x,C):(A\to B)}$$

where $C_1 : C_2$ is a concatenation of code sequences C_1 and C_2.

Example 5. In Fig. 1 shows a typing derivation tree for the term lam(x, s : s : x : app : app) : 3 : app which appears in Example 1.

$$\frac{\dfrac{\dfrac{\dfrac{}{x: num \vdash x: num \to num}\quad \dfrac{x: num \vdash s: num \to num \quad x: num \vdash s: num}{x: num \vdash s: x: app: num}}{x: num \vdash s: s: x: app: app: num \to num}}{\vdash lam(x, s: s: x: app: app): num \to num}\qquad \dfrac{}{\vdash 3: num}}{\vdash lam(x, s: s: x: app: app): 3: app\ :\ num}$$

Fig. 1. Typing derivation tree.

Definition 12 (Typing of SAM's configurations) Typing of SAM's configurations is defined inductively by the following rules. A typing judgment for configurations is written as

$$\vdash (S, C): A,$$

which is read as "configuration (S , C) is of type A."

$$\frac{\vdash (rev(S): C): A}{\vdash (S, C): A}$$

where $rev(S)$ means the reversed sequence of S.

Reduction on SAM configurations preserves their typing during computation, which is called Subject Reduction Theorem. The following lemma is required in order to prove the subject reduction theorem:

Lemma 1 (Substitution Lemma) If $\{x : A\}$ $\Gamma \vdash C : B$ and $\Gamma \vdash C' : A$, then it holds that $\Gamma \vdash C[x := C'] : B$.

Proof. We prove this lemma by induction on structure on derivation of $\{x : A\}$ $\Gamma \vdash C : B$.

Case $C = n$. Suppose that $\{x : A\}$ $\Gamma \vdash C : B$ and $\Gamma \vdash C' : A$.

$$C[x \coloneqq C'] = n[x \coloneqq C'] = n.$$

By the typing rule, we have $\Gamma \vdash n : num$, that is, $\Gamma \vdash C[\mathrm{x} \coloneqq C'] : B$. Case $C = x$. Suppose that $\{ x : A \} \vdash \ \ x : A$ and $\Gamma' \vdash C' : A$.

$$C[x \coloneqq C'] = x[x \coloneqq C'] = C'.$$

From the assumption, it holds that $\Gamma \vdash C\,[x \coloneqq C'\,] : A$.

Case $C = (C_1 : C_2 : app)$. Suppose that

$$\{x: A\}\Gamma \vdash (C_1: C_2: app): A_2\ \ and\ \ \Gamma \vdash C': A.$$

From the former assumption,

$$\{x: A\}\Gamma \vdash C_1: (A_1 \to A_2)\ \ \text{and}\ \ \{x: A\} \vdash C_1: A_1$$

By the induction hypothesis,

$$\Gamma \vdash C_1[x \coloneqq C]: A_1 \to A_2 \text{ and } \Gamma \vdash C_2[x \coloneqq C] : A_1$$

Therefore,

$$\Gamma \vdash (C_1 : C_2 : app) : A_2.$$

Case $C = lam(y, D)$. We may assume that y does not equal to x, because of bound variable convention. Suppose that

$$\{x: A\}\Gamma' \vdash lam(y, D): A_1 \to A_2 \text{ and } \{y: A_1\}\Gamma' \vdash C': A$$

From the former assumption,

$$\{x: A\}\{y: A_1\}\Gamma' \vdash D: A_2.$$

By the induction hypothesis,

$$\{y: A_1\}\Gamma' \vdash D[x \coloneqq C']: A_2,$$

and by the typing rule, it is derived that

$$\Gamma' \vdash lam(y, D[x \coloneqq C']: A_1 \to A_2.$$

Since

$$lam(y, D)[x \coloneqq C'] = lam(y, D[x \coloneqq C']),$$

we have

$$\Gamma' \vdash C[x \coloneqq C']: A_1 \to A_2.$$

End of Proof.

Theorem 2 (Subject Reduction Theorem) If $(S, C) \to (S', C')$ and $\vdash (S, C) : A$, then it holds that $\vdash (S', C') : A$.

Proof. This theorem is proved by structural induction on transition.

Case of rule num. Suppose that $(S, n : C) \to (n : S, C)$ and $\vdash (S, n : C) : A$. Since $\vdash (S, n : C) : A$, we have $\vdash (rev(S):n : C) : A$. Since $rev(S) : n : C = rev(n : S) : C$, we know $\vdash (rev(n : S) : C) : A$. Hence, $\vdash (rev(n : S), C) : A$.

Case of rules prim and lam. We can prove these cases, similarly to the case of num, since we have

$$rev(S): C = rev(S'): C'.$$

Case of rule app-lam. Suppose that

$$\vdash (S: lam(x, C'): V: app: C) : A$$

By typing rules, we know that code lam(*x, C'*) : *V* : *app* is typable. Therefore, there is some type *B* satisfying that

$$\{x: B\} \vdash C': B \rightarrow D \text{ and } \vdash V: B.$$

On the other hand, by using Lemma 1, we know that ⊢ C' [x ≔ V] : D. Then, replacing lam(x, C') : V : app with this code, we obtain that

$$\vdash (S: C'[x \coloneqq V]: C): A.$$

Case of rule app-prim is similar to this case.

End of Proof.

6. Extension of First-Class Continuations to SAM

In this section, we formulate first-class continuations in the framework of SAM. Continuations have been understood as evaluation contexts in the call-by-value lambda calculus. In SAM, the evaluation contexts are equivalent to the configurations of SAM. In the terminology of reflective programming, the following two notions are fundamental. In order to introduce mechanism of first-class continuation into SAM, we should give them to SAM.

Reification: lowering abstract machine's configuration down to object-level,

Reflection: raising reified abstract machine's configuration upto meta-level.

Programming language Scheme provides reification and reflection through call-with-current-continuation

(abbrev. call/cc) and procedure call, respectively. If we remember that evaluation contexts are equivalent to continuations in the call-by-value lambda calculus, we can introduce call/cc into SAM as follows.

Definition 13 (SAM with First-class Continuations) We define Simple Abstract Machine with First-class Continuations, SAMcall/cc by the following rules.

abort $(S, abort(S', C): C) \rightarrow (S', C')$,

callcc $(lam(x, C'): callcc: S, app: C) \rightarrow \big(S, C'\big[x \coloneqq lam\big(y, abort(S, y: C)\big)\big]: C\big)$.

The former primitive, abort, represents a global exit, like Unix's exit, which is required for eliminating the current continuation.

Definition 14 (Typing Rules for SAM callcc) We define typing for SAM callcc by the typing rules of SAM and the following typing rules for callcc and abort.

$$\Gamma \vdash callcc : \big((A \rightarrow B) \rightarrow A\big) \rightarrow A,$$

$$\frac{\vdash abort(S, C): A}{\vdash (S, C): \phi}$$

where ϕ is the top-level type, that is, the type of the whole code which SAM executes. This technique is also found in the lambda calculus with first-class continuations [9].

7. Comparison between SAM and SECD Machine

In this section, we attempt a comparison between SAM and SECD machine. A configuration of SECD machine is represented as a quadruple of four sequences: Stack, Environment, Code, and Dump, which is the origin of the word "SECD." Computation of SECD machine is formulated as transition between configurations by the following rules.

num	$(S, E, n: C, D) \to (n: S, E, C, D)$,
prim	$(S, E, f: C, D) \to (f: S, E, C, D)$,
ret	$(v: S, E, (\quad), (S', E', C'): D) \to (v: S', E', C', D)$,
var	$(S, E, x: C, D) \to (lookup(x, E): S, E, C, D)$,
lam	$(S, E, lam(x, C'): C, D) \to (lam(x, C')[E']: S, E, app: C, D)$,
app-lam	$(v: lam(x, C')[E']: S, E, app: C, D) \to ((\quad), (x \mapsto v): E', C', (S, E, C): D)$
app-prim	$(v: f: S, E, app: C, D) \to (f(v): S, E, C, D)$.

There are minor difference among the literatures [1][2][5][6][10]. In this paper, we follow the papers [1], [10] and use theirs terminology. Although N is firstly evaluated and then M is secondly done, in evaluation of $(M\ N)$ in many literatures, we assume that M is firstly evaluated, then N is secondly done, and finally the value of N is bound to the formal parameter of M.

There is a crucial difference about variable reference between SAM and SECD machine. In SAM, you formalize variable reference as substitutions. On the other hand, in SECD machine, you formulated it as the second component "Environment" of its configuration. Access of a variable to Environment E is provided by lookup(x, E) and update of binding of x to v by $[x \mapsto v]\ E$.

We can regard an environment in an evaluator of functional languages including SAM as a substitution whose application is delayed. In transition rule app-lam,

$$(v: lam(x, C')[E']: S, E, app: C, D) \to \big((\quad), (x \mapsto v): E', C', (S, E, C): D\big).$$

Body C' of function closure $lam(x, C)[E']$ is evaluated under environment $(x \mapsto v)$:E'. Since the rest of Code C should be evaluated under stack S and environment E, the triple of S, E and C is stored in the dump.

The purpose of environments in SECD is to make rewriting of code sequences needless. If you observe code sequences in SECD's transition rules, we know that codes are truncated or replaced entirely. In other words, codes appearing in SECD are subsequences which shares the tail part of the code sequence given initially. You can represent such tail subsequence as its starting index. Therefore, the third component C of each SECD configuration is not actually required to be represented as a sequence but an index, which is an important optimization in its implementation.

8. Conclusion

In this paper, we proposed the Simple Abstract Machine in which call-by-value evaluation strategy is incorporated, and give the simple type system to it. Further, we showed an extension of SAM by adding first-class continuations. In a previous version of this paper, SAM was proposed and presented the extension of adding first-class continuation. This paper is a further improved version. In this paper, we develop the fundamental mathematical properties such as soundness of the translation of the lambda calculus into SAM, which was not established in the paper.

9. Future Works

We present several future directions of our work.

In this paper, we gave an informal explanation of how to simplify SECD machine to get SAM. We apply such simplification more formally to the other kinds of abstract machine, like FAM [12] and LAM [7].

SAM is based on call-by-value evaluation strategy. On the other hand, Krivine machine [13] is known as an abstract machine based on call-by-name evaluation strategy.

$$\begin{gathered}\big((M\ N), E, S\big) \to (M, E, N[E]\colon S),\\ (lam(x, M), E, N[E']\colon S) \to \big(M\big[x \coloneqq N[E']\big], E, S\big),\\ (x, E, S) \to (N, E', S)\ \text{ where }\ (lookup(E, x) = N[E'].\end{gathered}$$

In Krivine machine, a variable environment provides variable reference. If you use substitution in the SECD machine similarly to SAM, then we can simplify it as

$$\begin{gathered}\big((M\ N), S\big) \to (M, N\colon S),\\ (lam(x, M), N\colon S) \to (M[x \coloneqq N], S).\end{gathered}$$

A major difference between SAM and the simplified Krivine machine is that you formulate the former code as a sequence, however the latter as a tree structure. We may say that the Simple Abstract Machine is much nearer to abstract machine implementation than the Krivine machine.

Many researchers [9], [14]–[17] have studied the duality between call-by-value and call-by-name evaluation strategies. A dual version of call-by-value SAM is also to be studied in future.

Acknowledgment

This paper is based on a previous version [11]. This work was supported by Grants-in-Aid for Scientific Research (C) (23400009).

References

[1] Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal*, *6(4)*, 308– 320.

[2] Henderson, P. (1980). *Functional Programming: Applications and Implementation*. Prentice Hall, 1980.

[3] Cousineau, G., Curien, P. L., & M. Mauny. (1987). The categorical abstract machine. *Science of Computer Programming*, *8(12)*, 173–202.

[4] Leroy, X. From Krivine's machine to the CAML implementations. Retrieved 2005, from http://gallium.inria.fr/ ~xleroy/talks/zam-kazam05.pdf

[5] Graham, B. T. (1992). *The SECD Microprocessor: A Verification Case Study.* Kluwer Academic Publishers.

[6] Hardin, T., Maranget, L., & Pagano, B. (1996). Functional back-ends within the lambda-sigma calculus. *Proceedings of the 1996 International Conference on Functional Programming* (pp. 25–33).

[7] Ohori, A. (1999). The logical abstract machine: A curry-howard isomorphism for machine code. *Proceedings of the Fuji International Symposium on Functional and Logic Programming*.

[8] Curien, P. L. (2000). Abstract machines, control and sequents. *Lecture Notes in Computer Science*.

[9] Griffin, T. G. (1990). A formulae-as-types notion of control. *Proceedings of the Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages* (pp. 47-57).

[10] Field, A. J., & Harrison, P. G. (1988). Functional programming. *International Computer Science Series*.

[11] Narita, K., & Nishizaki, S. (2010). A simple abstract machine for functional first-class continuations. *Proceedings of International Symposium on Communication and Information Technologies* (pp. 111–114)

[12] Cardelli, L. (1984). Compiling a functional language. *Proceedings of the Symposium on LISP and a Functional Programming* (pp. 208–217).

[13] Amadio. R. M., & Curien, P. L. (1998). *Domains and Lambda-Calculi*. Cambridge University Press.

[14] Filinski, A. (1989) Declarative continuations: An investigation of duality in programming language semantics. *Category Theory and Computer Science*.

[15] Curien, P. L., & Herbelin, H. (2000). The duality of computation. *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming* (pp. 233–243). ACM.

[16] Kakutani, Y. (2002) Duality between call-by-name recursion and call-by-value iteration. *Proceedings of CSL 2002*.

[17] Wadler, P. (2003). Call-by-value is dual to call-by-name. *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*. ACM.

Shin-Ya Nishizaki is an associate professor of computer science at Tokyo Institute of Technology, Japan, where he leads a research group on formal theory on software systems. He received his bachelor's, master's and doctorate degrees from Kyoto University, in mathematical science. Before joining Tokyo Institute of Technology in 1998, Dr. Nishizaki held appointments in computer science as an associate professor at Chiba University for 2 years and an assistant professor at Okayama University for 2 years.

Kensuke Narita received his bachelor's and master's degrees from Tokyo Institute of Technology in 2009 and 2011, respectively. He now is working in Hitachi, Ltd.

Tomoyuki Ueda received his bachelor's and master's degrees from Tokyo Institute of Technology in 2001 and 2003, respectively. He now is working in Sanden Shoji, Co. Ltd.