The Logical Abstract Machine: a Curry-Howard isomorphism for machine code

Atsushi Ohori

Research Institute for Mathematical Sciences
Kyoto University, Kyoto 606-8502, Japan
ohori@kurims.kyoto-u.ac.jp

Abstract. This paper presents a logical framework for low-level machine code and code generation. We first define a calculus, called *sequential sequent calculus*, of intuitionistic propositional logic. A proof of the calculus only contains left rules and has a linear (non-branching) structure, which reflects the properties of sequential machine code. We then establish a Curry-Howard isomorphism between this proof system and machine code based on the following observation. An ordinary machine instruction corresponds to a polymorphic proof transformer that extends a given proof with one inference step. A return instruction, which turns a sequence of instructions into a program, corresponds to a logical axiom (an initial proof tree). Sequential execution of code corresponds to transforming a proof to a smaller one by successively eliminating the last inference step. This logical correspondence enables us to present and analyze various low-level implementation processes of a functional language within the logical framework. For example, a code generation algorithm for the lambda calculus is extracted from a proof of the equivalence theorem between the natural deduction and the sequential sequent calculus.

1 Introduction

Theoretical foundations of syntax and semantics of functional languages have been well established through extensive studies of typed lambda calculi. (See [14, 6] for surveys in this area.) Unfortunately, however, those theoretical results are not directly applicable to implementation of a functional language on a low-level sequential machine because of the mismatch between models of typed lambda calculi and actual computer hardware. For example, consider a very simple program $(\lambda x.\lambda y.(x, y))\ 1\ 2$. In the typed lambda calculus with products, this itself is a term and denotes an element in a domain of integer products. In an actual language implementation, however, this program is transformed to an intermediate expression and then compiled to a sequence of machine instructions. The details of the compiled code depend on the target computer architecture. Fig. 1 shows pseudo codes we consider in this paper for a stack-based machine and for a register based machine. The properties of these codes are radically different from those of existing semantic models of the lambda calculus. As a result, theoretical analysis of a programming language based on typed lambda calculi does not directly extend to implementation of a programming language.

```
        Code(label1)
        Const(1)
        Const(2)                        r0 <- Code(label1)
        Call(2)                         r1 <- Const(1)
        Return                          r2 <- Const(2)
                                        r0 <- call r0 with (r1,r2)
label1: Acc(0)                          Return(r0)
        Acc(1)
        Pair                    label1: r2 <- Pair(r1,r0)
        Return                          Return(r2)
```

In a stack-based machine. In a register-based machine.

Fig. 1. Machine codes for $(\lambda x.\lambda y.(x, y))\ 1\ 2$

The general motivation of this study is to establish logical foundations for implementing a functional language on conventional computer hardware. To achieve this goal, we need to find a constructive logic appropriate for machine code, and to establish a logical interpretation of a compilation process from a high-level language to machine code, possibly using an intermediate language. In a previous work [17], the author has shown that the process of translating the typed lambda calculus into an intermediate language called "A-normal forms" [4] is characterized as a proof transformation from the natural deduction proof system to a variant of Gentzen's intuitionistic sequent calculus (Kleene's G3a proof system [9]). However, A-normal forms are still high-level expressions, which must be compiled to machine code. A remaining technical challenge is to establish a logical framework for machine code and code generation. An attempt is made in the present paper to develop such a framework.

We first define a proof system, called *sequential sequent calculus*, of intuitionistic propositional logic, whose proofs closely reflect the properties of low-level machine code. By applying the idea of Curry-Howard isomorphism, the *logical abstract machine* (LAM for short) is derived from the calculus. When we regard the set of assumptions in a sequent as a list of formulas, then the derived machine corresponds to a stack-based abstract machine similar to those used, for example, in Java [13] and Camllight [12] implementation. We call this machine the stack-based logical abstract machine (SLAM). When we regard the set of assumptions as an association list of variables and formulas, then the derived machine corresponds to a "register transfer language" which is used as an abstract description of machine code in a native code compiler. We call this machine the register-based logical abstract machine (RLAM).

We prove that the sequential sequent calculus is equivalent to existing proof systems of intuitionistic propositional logic. From the proofs of the equivalence, we can extract code generation algorithms. The choice of the source proof system determines the source language, and the choice of the representation of an assumption set determines the target machine code. For example, if we choose the

natural deduction as the source proof system and the association list representation of an assumption set, then the extracted algorithm generates RLAM code for a register machine from a lambda term. Other combinations are equally possible. This framework enables us to analyze compilation process and to represent various optimization methods within a logical framework.

Backgrounds and related works.

There have been a number of approaches to systematic implementation of a functional programming language using an abstract machine. Notable results include Landin's SECD machine [11], Turner's SK-reduction machine [20], and the categorical abstract machine of Cousineauet et. al. [2]. In a general perspective, all those approaches are source of inspiration of this work. The new contribution of our approach is to provide a formal account for low-level machine code based on the principle of Curry-Howard isomorphism [3, 7]. In [2], it is emphasized that the categorical abstract machine is a law-level machine manipulating a stack or registers. However, its stack based sequential machine is ad-hoc in the sense that it is outside the categorical model on which it is based. In contrast, our LAM is directly derived from a logic that models sequential execution of primitive instructions using stack or registers, and therefore its sequential control structure is an essential part of our formal framework. We establish that a LAM program is isomorphic to a proof in our sequential sequent calculus, and that a code generation algorithm is isomorphic to a proof of a theorem stating that any formula provable in the natural deduction (or a sequent calculus) is also provable in the sequential sequent calculus. This will enable us to extend various high-level logical analyses to low-level machine code and code generation process.

With regard to this logical correspondence, the SK-reduction machine [20] deserves special comments. The core of this approach is the translation from the lambda calculus to the combinatory logic [3]. As observed in [3, 10], in the typed setting, this translation algorithm corresponds exactly to a proof of a well know theorem in propositional logic stating that any formula provable in the natural deduction is also probable in the Hilbert system. The bracket abstraction used in the translation algorithm (to simulate lambda abstraction in the combinatory logic) is isomorphic to a proof of the deduction theorem in Hilbert system stating that if τ_1 is provable from $\Delta \cup \tau_2$ then $\tau_2 \supset \tau_1$ is provable from Δ. Although this connection does not appear to be well appreciated in literature, this can be regarded as the first example of Curry-Howard isomorphism for compilation of a functional language to an abstract machine. However, the SK-reduction machine is based on the notion of functions (primitive combinators) and does not directly model computer hardware. We achieve the same rigorous logical correspondence for machine code and code generation.

Another inspiration of our work comes from recent active researches on defining a type system for low-level machine languages. Morrisett et. al. [16, 15] define a "typed assembly language." Stata and Abadi [19] define a type system for Java byte-code. In those works, a type system is successfully used to prove soundness of code execution. However, the type system is defined by considering each in-

struction's effect on the state of memory (registers or stack), and its logical foundations have not been investigated. In our approach, the logical abstract machine is derived from a logic. This enables us to present various implementation processes such as code generation entirely within a logical framework.

A note added 24th November, 1999 on one more related work.

After the Fuji Symposium on Functional and Logic Programming, Novemver, 1999, it was pointed out to the author that the paper by Christophe Raffalli on "Machine Deduction" (in Types for Proofs and Program, LNCS 806, pp. 333-351, 1994) discusses the related problems. Although the proof systems and the results presented in the present paper are different from those in the above paper, the general motivations and approaches of this work are also related to the present paper. The author is grateful to Yukiyoshi Kameyama for pointing out the existence of Raffalli's paper.

The author intends to prepare an extended version including appropriate comparison to Raffalli's work and others.

Paper Organization. Section 2 analyzes the nature of machine code and defines the sequential sequent calculus. Section 3 extracts the logical abstract machine from the proof system of the sequential sequent calculus and establish a Curry-Howard isomorphism between the calculus and the machine. Section 4 shows that the sequential sequent calculus is equivalent to other formalisms for intuitionistic propositional logic, and extracts compilation algorithms. Section 5 discusses some issues in implementation of a functional language.

Limitations of space make it difficult to present the framework fully; the author intends to present a more detailed description in another paper.

2 The Sequential Sequent Calculus

The first step in establishing logical foundations for machine code and code generation is to find a constructive logic that reflects the essential natures of conventional sequential machines.

By examining machine code such as those shown in Fig. 1, we observe the following general properties.

1. Machine code is constructed by sequential composition of instructions, each of which performs an action on machine memory. The machine is run by mechanically executing the first instruction of the code.
2. An instruction is polymorphic in the sense that it may be a part of various different programs computing difference values. The result of the entire code is determined by a special instruction that returns a value to the caller.
3. To represent arbitrary computation, the machine supports an operation to call a pre-compiled code.

In the following discussion, we write $\Delta \rhd \tau$ for a logical sequent with a set Δ of assumptions and a conclusion τ.

Logical inference rules that reflect item 1 are those of the form

$$\frac{\Delta_2 \rhd \tau}{\Delta_1 \rhd \tau}$$

which have only one premise and do not change the conclusion τ. By interpreting an assumption set as a description of the state of machine memory, a proof composed by this form of rules would yields a sequence of primitive instructions, each of which operates on machine memory. This is in contrast with the natural deduction where rules combine arbitrary large proofs. Moreover, sequential execution of machine is modeled by the (trivial) transformation of a proof to a smaller one by successively eliminating the last inference step.

In the context of proof system, the item 2 is understood as the property that each instruction corresponds not to a complete proof but to an inference step that extends a proof. This observation leads us to interpret each machine instruction I as a polymorphic function that transforms a proof of the form

$$\begin{array}{c} \vdots \\ \Delta_1 \rhd \tau \end{array}$$

to a proof of the form

$$\begin{array}{c} \vdots \\ \dfrac{\Delta_1 \rhd \tau}{\Delta_2 \rhd \tau}\ R_I \end{array}$$

for *arbitrary result type* τ. At run-time, the instruction corresponding to this rule transforms the memory state represented by Δ_2 to the one represented by Δ_1. Note that the direction in which machine performs the memory state transformation is the opposite to the direction in logical inference. For example, an instruction that creates a pair of integer is modeled by the following inference step.

$$\frac{\Delta; \tau_1 \wedge \tau_2 \rhd \tau}{\Delta; \tau_1; \tau_2 \rhd \tau}\ (pair)$$

A return instruction in a program computing a value of type τ corresponds to an axiom $\Delta \rhd \tau$ such that $\tau \in \Delta$.

The logical interpretation of item 3 of the ability to call a pre-compiled code is the availability of a proof of a sequent, and it can be modeled by a rule to assume an already proved formula as an axiom.

The above analysis leads us to define the sequential sequent calculus of intuitionistic propositional logic. We first consider the calculus for stack-based machines where an assumption set is a list of formulas.

We write $\langle \tau_1; \ldots; \tau_n \rangle$ for the list containing $\tau_1, \ldots, \tau_n$, and write $\Delta; \tau$ for the list obtained from Δ by appending τ to Δ. We also use the notation $\Delta(n)$ for the n^{th} element in the list Δ (counting from the left starting with 0,) and $|\Delta|$ for the length of Δ. The empty list is denoted by ϕ. A list of assumptions describes the type of a stack. We adopt the convention that the right-most formula in a list corresponds to the top of the stack. We assume that there is a given set of

non-logical axioms (ranged over by b), and consider the following set of formulas (ranged over by τ).

$$\tau ::= b \mid \tau \wedge \tau \mid \tau \vee \tau \mid (\Delta \Rightarrow \tau)$$

$\tau_1 \wedge \tau_2$ and $\tau_1 \vee \tau_2$ are conjunction and disjunction, corresponding to product and disjoint union, respectively. A formula $(\Delta \Rightarrow \tau)$ intuitively denotes the fact that τ is derived from Δ. Its constructive interpretation is the availability of a proof of the sequent $\Delta \rhd \tau$. This serves as an alternative to implication formula. The set of proof rules is given in Fig. 2. The rationale behind the rather unusual names of the rules will become clear later when we show a correspondence between this logic and an abstract machine. We write $\mathcal{S}_S$ for this proof system, and write $\mathcal{S}_S \vdash \Delta \rhd \tau$ if $\Delta \rhd \tau$ is provable in this proof system.

The axiom:

(return) $\Delta; \tau \rhd \tau$

Inference rules.

(acc) $\dfrac{\Delta; \tau_1 \rhd \tau}{\Delta \rhd \tau}$ $(\tau_1 \in \Delta)$ (const) $\dfrac{\Delta; b \rhd \tau}{\Delta \rhd \tau}$ (if b is a non-logical axiom)

(pair) $\dfrac{\Delta; \tau_1 \wedge \tau_2 \rhd \tau}{\Delta; \tau_1; \tau_2 \rhd \tau}$ (fst) $\dfrac{\Delta; \tau_1 \rhd \tau}{\Delta; \tau_1 \wedge \tau_2 \rhd \tau}$

(snd) $\dfrac{\Delta; \tau_2 \rhd \tau}{\Delta; \tau_1 \wedge \tau_2 \rhd \tau}$ (inl) $\dfrac{\Delta; \tau_1 \vee \tau_2 \rhd \tau}{\Delta; \tau_1 \rhd \tau}$

(inr) $\dfrac{\Delta; \tau_1 \vee \tau_2 \rhd \tau}{\Delta; \tau_2 \rhd \tau}$ (case) $\dfrac{\Delta; \tau_3 \rhd \tau}{\Delta; \tau_1 \vee \tau_2 \rhd \tau}$ (if $\vdash \Delta; \tau_1 \rhd \tau_3$ and $\vdash \Delta; \tau_2 \rhd \tau_3$)

(code) $\dfrac{\Delta; (\Delta_0 \Rightarrow \tau_0) \rhd \tau}{\Delta \rhd \tau}$ (if $\Delta_0 \rhd \tau_0$) (call) $\dfrac{\Delta; \tau_0 \rhd \tau}{\Delta; (\Delta_0 \Rightarrow \tau_0); \Delta_0 \rhd \tau}$

(app) $\dfrac{\Delta; (\Delta_1 \Rightarrow \tau_0) \rhd \tau}{\Delta; (\Delta_0; \Delta_1 \Rightarrow \tau_0); \Delta_0 \rhd \tau}$

Fig. 2. $\mathcal{S}_S$: the Sequential Sequent Calculus for Stack Machines

---

The sequential sequent calculus suitable for register-based machines is obtained by changing the representation of an assumption list Δ to a named assumption set. We let Γ range over functions representing named assumption sets, and write $\Gamma, x : \tau$ for the function Γ' such that $dom(\Gamma') = dom(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for any $y \in dom(\Gamma), y \neq x$. The set of proof rules is given in Fig. 3. We write $\mathcal{S}_R$ for this proof system, and write $\mathcal{S}_R \vdash \Gamma \rhd \tau$ if $\Gamma \rhd \tau$ is provable in this proof system.

The axiom:

(return) $\Gamma, x : \tau \rhd \tau$

Inference rules.

(acc) $\dfrac{\Gamma, x : \tau_1, y : \tau_1 \rhd \tau}{\Gamma, x : \tau_1 \rhd \tau}$ (const) $\dfrac{\Gamma, x : b \rhd \tau}{\Gamma \rhd \tau}$ (if b is a non-logical axiom)

(pair) $\dfrac{\Gamma, x : \tau_1, y : \tau_2, z : \tau_1 \wedge \tau_2 \rhd \tau}{\Gamma, x : \tau_1, y : \tau_2 \rhd \tau}$ (fst) $\dfrac{\Gamma, x : \tau_1 \wedge \tau_2, y : \tau_1 \rhd \tau}{\Gamma, x : \tau_1 \wedge \tau_2 \rhd \tau}$

(snd) $\dfrac{\Gamma, x : \tau_1 \wedge \tau_2, y : \tau_2 \rhd \tau}{\Gamma, x : \tau_1 \wedge \tau_2 \rhd \tau}$ (inl) $\dfrac{\Gamma, x : \tau_1, y : \tau_1 \vee \tau_2 \rhd \tau}{\Gamma, x : \tau_1 \rhd \tau}$

(inr) $\dfrac{\Gamma, x : \tau_2, y : \tau_1 \vee \tau_2 \rhd \tau}{\Gamma, x : \tau_2 \rhd \tau}$

(case) $\dfrac{\Gamma, x : \tau_1 \vee \tau_2, y : \tau_3 \rhd \tau}{\Gamma, x : \tau_1 \vee \tau_2 \rhd \tau}$ (if $\Gamma, z_1 : \tau_1 \rhd \tau_3$, $\Gamma, z_2 : \tau_2 \rhd \tau_3$)

(code) $\dfrac{\Gamma, x : (\Gamma_0 \Rightarrow \tau_0) \rhd \tau}{\Gamma \rhd \tau}$ (if $\vdash \Gamma_0 \rhd \tau_0$)

(call) $\dfrac{\Gamma_1, x : (\Gamma_2 \Rightarrow \tau_0), y : \tau_0 \rhd \tau}{\Gamma_1, x : (\Gamma_2 \Rightarrow \tau_0) \rhd \tau}$
$(\Gamma_2 = \{y_1 : \tau_1; \cdots; y_n : \tau_n\}, \Gamma_1(x_i) = \tau_i, 1 \leq i \leq n)$

(app) $\dfrac{\Gamma_1, x : (\Gamma_2, \Gamma_3 \Rightarrow \tau_0); y : (\Gamma_3 \Rightarrow \tau_0) \rhd \tau}{\Gamma_1, x : (\Gamma_2, \Gamma_3 \Rightarrow \tau_0) \rhd \tau}$
$(\Gamma_2 = \{y_1 : \tau_1; \cdots; y_n : \tau_n\}, \Gamma_1(x_i) = \tau_i, 1 \leq i \leq n)$

Fig. 3. $\mathcal{S}_R$: the Sequential Sequent Calculus for Register Machines

3 The Logical Abstract Machines

Guided by the notion of Curry-Howard isomorphism [3, 7], machine instructions are extracted from the sequential sequent calculi.

3.1 SLAM: the Stack Based LAM

The stack-based logical abstract machine, SLAM, is obtained from the calculus $\mathcal{S}_S$. The set of instructions (ranged over by I) is given as follows

$$\begin{aligned} I ::= {} & \mathsf{Return} \mid \mathsf{Acc}(n) \mid \mathsf{Const}(c^b) \mid \mathsf{Pair} \mid \mathsf{Fst} \mid \mathsf{Snd} \\ & \mid \mathsf{Inl} \mid \mathsf{Inr} \mid \mathsf{Case} \mid \mathsf{Code}(C) \mid \mathsf{Call(n)} \mid \mathsf{App}(n) \end{aligned}$$

where C ranges over lists of instructions. We adopt the convention that the left-most instruction of C is the first one to execute, and write $I; C$ (and $C'; C$) for the list obtained by appending an instruction I (a list C' of instructions) at the front of C. The type system of these instruction is obtained by decorating each rule of $\mathcal{S}_S$ with the corresponding instruction. Fig. 4 shows the typing rule for SLAM, which is the term calculus of $\mathcal{S}_S$.

$$\text{(return)}\ \ \Delta;\tau \rhd \mathsf{Return} \ :\ \tau \qquad \text{(acc)}\ \ \frac{\Delta;\tau_1 \rhd C \ :\ \tau}{\Delta \rhd \mathsf{Acc}(n); C \ :\ \tau}\ \ (\Delta(n) = \tau_1)$$

$$\text{(const)}\ \ \frac{\Delta; b \rhd C \ :\ \tau}{\Delta \rhd \mathsf{Const}(c^b); C \ :\ \tau} \qquad \text{(pair)}\ \ \frac{\Delta;\tau_1 \wedge \tau_2 \rhd C \ :\ \tau}{\Delta;\tau_1;\tau_2 \rhd \mathsf{Pair}; C \ :\ \tau}$$

$$\text{(fst)}\ \ \frac{\Delta;\tau_1 \rhd C \ :\ \tau}{\Delta;\tau_1 \wedge \tau_2 \rhd \mathsf{Fst}; C \ :\ \tau} \qquad \text{(snd)}\ \ \frac{\Delta;\tau_2 \rhd C \ :\ \tau}{\Delta;\tau_1 \wedge \tau_2 \rhd \mathsf{Snd}; C \ :\ \tau}$$

$$\text{(inl)}\ \ \frac{\Delta;\tau_1 \vee \tau_2 \rhd C \ :\ \tau}{\Delta;\tau_1 \rhd \mathsf{Inl}; C \ :\ \tau} \qquad \text{(inr)}\ \ \frac{\Delta;\tau_1 \vee \tau_2 \rhd C \ :\ \tau}{\Delta;\tau_2 \rhd \mathsf{Inr}; C \ :\ \tau}$$

$$\text{(case)}\ \ \frac{\Delta;\tau_3 \rhd C \ :\ \tau}{\Delta;\tau_1 \vee \tau_2 \rhd \mathsf{Case}(C_1,C_2); C \ :\ \tau}\ \ (\text{if} \vdash \Delta;\tau_1 \rhd C_1 \ :\ \tau_3 \text{ and} \vdash \Delta;\tau_2 \rhd C_2 \ :\ \tau_3)$$

$$\text{(code)}\ \ \frac{\Delta;(\Delta_0 \Rightarrow \tau_0) \rhd C \ :\ \tau}{\Delta \rhd \mathsf{Code}(C_0); C \ :\ \tau}\ \ (\text{if} \vdash \Delta_0 \rhd C_0 \ :\ \tau_0)$$

$$\text{(call)}\ \ \frac{\Delta;\tau_0 \rhd C \ :\ \tau}{\Delta;(\Delta_1 \Rightarrow \tau_0);\Delta_1 \rhd \mathsf{Call}(n); C \ :\ \tau}\ \ (|\Delta_1| = n)$$

$$\text{(app)}\ \ \frac{\Delta;(\Delta_1 \Rightarrow \tau_0) \rhd C \ :\ \tau}{\Delta;(\Delta_2;\Delta_1 \Rightarrow \tau_0);\Delta_2 \rhd \mathsf{App}(n); C \ :\ \tau}\ \ (|\Delta_2| = n)$$

Fig. 4. Type System of SLAM

Each instruction "pops" arguments (if any) from the top of the stack and pushes the result on top of the stack. $\mathsf{Code}(C)$ pushes a pointer to code C. $\mathsf{Call}(n)$ pops n arguments and a function, calls the function with the arguments, and pushes the result. $\mathsf{App}(n)$ pops n arguments and a function closure and pushes the function closure obtained by extending the closure's stack with the n arguments. The meanings of the other instructions are obvious from their typing rules. Their precise behavior is defined by an operational semantics, given below.

The set of run-time values (ranged over by v) is defined as follows

$$v ::= c^b \mid (v, v) \mid inl(v) \mid inr(v) \mid cls(S, C)$$

where $cls(S, C)$ is a function closure consisting of stack S, which is a list of values, and code C. The operational semantics is defined as a set of rules to transform a configuration

$$(S, C, D)$$

where D is a "dump," which is a list of pairs of S and C. Fig. 5 gives the set of transformation rules. The reflexive, transitive closure of the relation $\longrightarrow$ is denoted by $\stackrel{*}{\longrightarrow}$.

Evaluation of code under this operational semantics corresponds to the trivial proof transformation by successively eliminating the last inference step. To formally establish this relation, we define the typing relations for values and for stack (written $\models v : \tau$ and $\models S : \Delta$ respectively) as follows.

- $\models c^b : b$.

$$
\begin{array}{rcl}
(S; v, \mathsf{Return}, \phi) & \longrightarrow & (v, \phi, \phi) \\
(S; v, \mathsf{Return}, D; (S_0, C_0)) & \longrightarrow & (S_0; v, C_0, D) \\
(S, \mathsf{Acc}(n); C, D) & \longrightarrow & (S; S(n), C, D) \\
(S, \mathsf{Const}(c); C, D) & \longrightarrow & (S; c, C, D) \\
(S; v_1; v_2, \mathsf{Pair}; C, D) & \longrightarrow & (S; (v_1, v_2), C, D) \\
(S; (v_1, v_2), \mathsf{Fst}; C, D) & \longrightarrow & (S; v_1, C, D) \\
(S; (v_1, v_2), \mathsf{Snd}; C, D) & \longrightarrow & (S; v_2, C, D) \\
(S; v, \mathsf{Inl}; C, D) & \longrightarrow & (S; inl(v), C, D) \\
(S; v, \mathsf{Inr}; C, D) & \longrightarrow & (S; inr(v), C, D) \\
(S; inl(v), \mathsf{Case}(C_1,C_2); C, D) & \longrightarrow & (S; v, C_1, D; (S, C)) \\
(S; inr(v), \mathsf{Case}(C_1,C_2); C, D) & \longrightarrow & (S; v, C_2, D; (S, C)) \\
(S, \mathsf{Code}(C_0); C, D) & \longrightarrow & (S; cls(\phi, C_0), C, D) \\
(S; cls(S_0, C_0); v_1; \cdots; v_m, \mathsf{Call}(m); C, D) & \longrightarrow & (S_0; v_1; \cdots; v_m, C_0, D; (S, C)) \\
(S; cls(S_0, C_0); v_1; \cdots; v_m, \mathsf{App}(m); C, D) & \longrightarrow & (S; cls(S_0; v_1; \cdots; v_m, C_0), C, D)
\end{array}
$$

Fig. 5. Operational Semantics of SLAM

- $\models (v_1, v_2) : \tau_1 \wedge \tau_2$ if $\models v_1 : \tau_1$ and $\models v_2 : \tau_2$.
- $\models inl(v) : \tau_1 \vee \tau_2$ if $\models v : \tau_1$
- $\models inr(v) : \tau_1 \vee \tau_2$ if $\models v : \tau_2$.
- $\models cls(S, C) : (\Delta \Rightarrow \tau)$ if $\models S : \Delta_1$ and $\Delta_1; \Delta \rhd C \ : \ \tau$ for some Δ_1.
- $\models S : \Delta$ if $S = \langle v_1; \ldots; v_n \rangle, \Delta = \langle \tau_1; \ldots; \tau_n \rangle$ such that $\models v_i : \tau_i (1 \leq i \leq n)$.

We can then show the following.

Theorem 1. *If there is a proof of the form*

$$
\dfrac{\dfrac{\vdots}{\Delta_2 \rhd C_2 \ : \ \tau}}{\dfrac{\vdots}{\Delta_1 \rhd C_1; C_2 \ : \ \tau}}
$$

$S_1 \models \Delta_1$, *and* $(S_1, C_1; C_2, D) \xrightarrow{*} (S_2, C_2, D)$ *then* $S_2 \models \Delta_2$.

Proof. This is shown by induction on the number of reduction steps of the machine. The proof proceeds by cases in terms of the first instruction of C. For the instructions other than $\mathsf{Case}(C_1,C_2)$ and $\mathsf{Call}(n)$, the property is show by checking the rule of the instruction against the proof rule and then applying the induction hypothesis. The cases for $\mathsf{Case}(C_1,C_2)$ and $\mathsf{Call}(n)$ can be shown by using the fact that the only instruction that decreases (pops) the dump D is Return. □

We can also show that the machine does not halt unexpectedly. To show this property, we need to define the type correctness of a dump. We write $\models D : \tau$ to represent the property that D is a type correct dump expecting a return value of type τ. This relation is given below.

- $\models \phi : \tau$.
- $\models D; (S, C) : \tau$ if there are some Δ and τ' such that $S \models \Delta$, $\Delta; \tau \rhd C : \tau'$ and $\models D : \tau'$.

Theorem 2. *If $\Delta \rhd C : \tau$, $S \models \Delta$ and $\models D : \tau$ then $(S, C, D) \longrightarrow (S', C', D')$ and if $C' \neq \phi$ then there are some Δ', τ' such that $S' \models \Delta'$, $\Delta' \rhd C' : \tau'$ and $\models D' : \tau'$.*

Proof. By cases in terms of the first instruction of C.

Combining the two theorems, we have the following.

Corollary 1. *If $\Delta \rhd C : \tau$, $S \models \Delta$, $(S, C, \phi) \xrightarrow{*} (S', C', D)$ and there is no (S'', C'', D') such that $(S', C', D) \longrightarrow (S'', C'', D')$ then $(S', C', D) = (v, \phi, \phi)$ such that $\models v : \tau$.*

Proof. By the property of the proof system, there are Δ', C_0 such tha $C = C_0; \mathsf{Return}$ and $\Delta'; \tau \rhd \mathsf{Return} : \tau$. It is easily checked that if $D \neq \phi$ then $C \neq \phi$. Then by Theorem 2, the property that $(S, C, \phi) \xrightarrow{*} (S', C', D)$ and there is no (S'', C'', D') such that $(S', C', D) \longrightarrow (S'', C'', D')$ implies that $(S, C_0; \mathsf{Return}, \phi) \xrightarrow{*} (S_1, \mathsf{Return}, \phi)$ for some S_1. By Theorem 1, $\models S_1 : \Delta'; \tau$. Thus $S_1 = S_1'; v$ such that $\models v : \tau$. Then $(S_1'; v, \mathsf{Return}, \phi) \longrightarrow (v, \phi, \phi)$. □

3.2 RLAM: the Register-Based LAM

Parallel to the stack-based logical abstract machine, SLAM, we can also construct the register based machine, RLAM, from the sequential sequent calculus, $\mathcal{S}_R$, and we can show the properties analogous to those of SLAM. Here we only give the set of instructions and the typing rules in Fig. 6, where $x \leftarrow I$ intuitively means that the value denoted by I is assigned to register x. The intuitive meaning of each instruction can then be understood analogously to the corresponding instruction of SLAM.

4 Code Generation as Proof Transformation

The sequential sequent calculus is shown to be equivalent to existing intuitionistic propositional calculi with conjunction and disjunction. Moreover, the proof of the equivalence is effective. For example, any proof in the natural deduction of intuitionistic propositional logic can be transformed to a proof in the sequential sequent calculus, and vice versa. The same also holds for the intuitionistic propositional sequent calculus. Through Curry-Howard isomorphism, these results immediately yield compilation algorithms from the lambda calculus or the A-normal forms to the logical abstract machine.

Here we only consider the typed lambda calculus (with products and sums.) The set of types is given by the following grammar.

$$\tau ::= b \mid \tau \supset \tau \mid \tau \wedge \tau \mid \tau \vee \tau$$

$$I ::= \mathsf{Return}(x) \mid x{\leftarrow}y \mid x{\leftarrow}\mathsf{Const}(c^b) \mid x{\leftarrow}\mathsf{Pair}(y,z) \mid x{\leftarrow}\mathsf{Fst}(y) \mid x{\leftarrow}\mathsf{Snd}(y)$$
$$\mid x{\leftarrow}\mathsf{Inl}(y) \mid x{\leftarrow}\mathsf{Inr}(y) \mid x{\leftarrow}\mathsf{Case}(y,(x).C_1,(x).C_2) \mid x{\leftarrow}\mathsf{Code}(C)$$
$$\mid y{\leftarrow}\mathsf{Call}\ x\ \mathsf{with}\ (y_1{\leftarrow}x_1,\ldots,y_n{\leftarrow}x_n)$$
$$\mid y{\leftarrow}\mathsf{App}\ x\ \mathsf{to}\ (y_1{\leftarrow}x_1,\ldots,y_n{\leftarrow}x_n)$$

(return) $\Gamma, x:\tau \rhd \mathsf{Return}(x) : \tau$ (acc) $\dfrac{\Gamma, x:\tau_1; y:\tau_1 \rhd C : \tau}{\Gamma, x:\tau_1 \rhd y{\leftarrow}x; C : \tau}$

(const) $\dfrac{\Gamma, x:b \rhd C : \tau}{\Gamma \rhd x{\leftarrow}c^b; C : \tau}$ (pair) $\dfrac{\Gamma, x:\tau_1; y:\tau_2; z:\tau_1{\wedge}\tau_2 \rhd C : \tau}{\Gamma, x:\tau_1; y:\tau_2 \rhd z{\leftarrow}\mathsf{Pair}(x,y); C : \tau}$

(fst) $\dfrac{\Gamma, x:\tau_1{\wedge}\tau_2; y:\tau_1 \rhd C : \tau}{\Gamma, x:\tau_1{\wedge}\tau_2 \rhd y{\leftarrow}\mathsf{Fst}(x); C : \tau}$ (snd) $\dfrac{\Gamma, x:\tau_1{\wedge}\tau_2; y:\tau_2 \rhd C : \tau}{\Gamma, x:\tau_1{\wedge}\tau_2 \rhd y{\leftarrow}\mathsf{Snd}(x); C : \tau}$

(inl) $\dfrac{\Gamma, x:\tau_1; y:\tau_1{\vee}\tau_2 \rhd C : \tau}{\Gamma, x:\tau_1 \rhd y{\leftarrow}\mathsf{Inl}(x); C : \tau}$ (inr) $\dfrac{\Gamma, x:\tau_2; y:\tau_1{\vee}\tau_2 \rhd C : \tau}{\Gamma, x:\tau_2 \rhd y{\leftarrow}\mathsf{Inr}(x); C : \tau}$

(case) $\dfrac{\Gamma, x:\tau_1{\vee}\tau_2; y:\tau_3 \rhd C : \tau}{\Gamma, x:\tau_1{\vee}\tau_2 \rhd y{\leftarrow}\mathsf{Case}(x,\ (z_1).C_1,\ (z_2).C_2); C : \tau}$
(if $\Gamma, z_1:\tau_1 \rhd C_1 : \tau_3,\ \Gamma, z_2:\tau_2 \rhd C_2 : \tau_3$)

(code) $\dfrac{\Gamma, x:(\Gamma_0 \Rightarrow \tau_0) \rhd C : \tau}{\Gamma \rhd x{\leftarrow}\mathsf{Code}(C'); C : \tau}$ (if $\vdash \Gamma_0 \rhd C' : \tau_0$)

(call) $\dfrac{\Gamma_1, x:(\Gamma_2 \Rightarrow \tau_0); y:\tau_0 \rhd C : \tau}{\Gamma_1, x:(\Gamma_2 \Rightarrow \tau_0) \rhd y{\leftarrow}\mathsf{Call}\ x\ \mathsf{with}\ (y_1{\leftarrow}x_1,\ldots,y_n{\leftarrow}x_n); C : \tau}$
$(\Gamma_2 = \{y_1:\tau_1;\cdots;y_n:\tau_n\}, \Gamma_1(x_i) = \tau_i, 1 \le i \le n)$

(app) $\dfrac{\Gamma_1, x:(\Gamma_2, \Gamma_3 \Rightarrow \tau_0); y:(\Gamma_3 \Rightarrow \tau_0) \rhd C : \tau}{\Gamma_1, x:(\Gamma_2, \Gamma_3 \Rightarrow \tau_0) \rhd y{\leftarrow}\mathsf{Appl}\ x\ \mathsf{to}\ (y_1{\leftarrow}x_1,\ldots,y_n{\leftarrow}x_n); C : \tau}$
$(\Gamma_2 = \{y_1:\tau_1;\cdots;y_n:\tau_n\}, \Gamma_1(x_i) = \tau_i, 1 \le i \le n)$

Fig. 6. The Register-Based Logical Abstract Machine, RLAM

The set of typing rules of the lambda calculus is given in Fig. 7. We denote this proof system by $\mathcal{N}$ and write $\mathcal{N} \vdash \Gamma \rhd M : \tau$ if $\Gamma \rhd M : \tau$ is provable in this proof system.

We identify implication type $\tau_1 \supset \tau_2$ of $\mathcal{N}$ with code type $(\langle\tau_1\rangle \Rightarrow \tau_2)$ of $\mathcal{S}_S$. For a type assignment Γ in $\mathcal{N}$ we let Δ_Γ be the corresponding assumption list in $\mathcal{S}_S$ obtained by a linear ordering on variables, i.e., if $\Gamma = \{x_1:\tau_1,\ldots,x_n:\tau_n\}$ such that $x_i < x_j$ for any $1 \le i < j \le n$, then $\Delta_\Gamma = \langle\tau_1;\ldots;\tau_n\rangle$. If $x \in dom(\Gamma)$ then we write $lookup(\Gamma, x)$ for the position in Δ_Γ corresponding to x.

We first show that any lambda term M can be translated to SLAM code $[\![M]\!]$ that extends a given stack with the value denoted by M. To establish this connection, we define the following relation for SLAM code.

$$C : \Delta_1 \Rightarrow \Delta_2 \iff \text{for any } C', \tau \text{ if } \mathcal{S}_S \vdash \Delta_2 \rhd C' : \tau \text{ then } \mathcal{S}_S \vdash \Delta_1 \rhd C; C' : \tau.$$

$$\text{(taut)}\ \ \Gamma, x:\tau \rhd x\ :\ \tau \qquad \text{(axiom)}\ \ \Gamma \rhd c^{\tau}\ :\ \tau \qquad (\supset\text{:I})\ \ \frac{\Gamma, x:\tau_1 \rhd M\ :\ \tau_1}{\Gamma \rhd \lambda x.M\ :\ \tau_1 \supset \tau_2}$$

$$(\supset\text{:E})\ \ \frac{\Gamma \rhd M_1\ :\ \tau_1 \supset \tau_2 \quad \Gamma \rhd M_2\ :\ \tau_1}{\Gamma \rhd M_1\ M_2\ :\ \tau_2} \qquad (\wedge\text{:I})\ \ \frac{\Gamma \rhd M_1\ :\ \tau_1 \quad \Gamma \rhd M_2\ :\ \tau_2}{\Gamma \rhd (M_1, M_2)\ :\ \tau_1 \wedge \tau_2}$$

$$(\wedge\text{:E1})\ \ \frac{\Gamma \rhd M\ :\ \tau_1 \wedge \tau_2}{\Gamma \rhd M.1\ :\ \tau_1} \qquad (\wedge\text{:E2})\ \ \frac{\Gamma \rhd M\ :\ \tau_1 \wedge \tau_2}{\Gamma \rhd M.2\ :\ \tau_2}$$

$$(\vee\text{:I1})\ \ \frac{\Gamma \rhd M\ :\ \tau_1}{\Gamma \rhd inl(M)\ :\ \tau_1 \vee \tau_2} \qquad (\vee\text{:I2})\ \ \frac{\Gamma \rhd M\ :\ \tau_2}{\Gamma \rhd inr(M)\ :\ \tau_1 \vee \tau_2}$$

$$(\vee\text{:E})\ \ \frac{\Gamma \rhd M_1\ :\ \tau_1 \vee \tau_2 \quad \Gamma, x:\tau_1 \rhd M_2\ :\ \tau_3 \quad \Gamma, y:\tau_2 \rhd M_3\ :\ \tau_3}{\Gamma \rhd case\ M_1\ of\ x.M_2, y.M_3\ :\ \tau_3}$$

Fig. 7. $\mathcal{N}$: Typed Lambda Calculus with Products and Sums

From this definition, it is immediate that each inference rule of SLAM of the form $(\mathrm{R}_I)\ \ \frac{\Delta_2 \rhd C\ :\ \tau}{\Delta_1 \rhd I; C\ :\ \tau}$ is regarded as the relation $I : \Delta_1 \Rightarrow \Delta_2$, and that this relation is "transitive", i.e., if $C_1 : \Delta_1 \Rightarrow \Delta_2$ and $C_2 : \Delta_2 \Rightarrow \Delta_3$ then $C_1; C_2 : \Delta_1 \Rightarrow \Delta_3$.

The following lemma plays a central role in establishing the logical correspondence. Since a proof of this lemma gives a code generation algorithm and its type correctness proof, we show it in some detail.

Lemma 1. *If $\mathcal{N} \vdash \Gamma \rhd M\ :\ \tau$ then there is SLAM code $[\![M]\!]$ such that $[\![M]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau$.*

Proof. By induction on M.

Case x. Since $x : \tau \in \Gamma$, by rule (acc), $\mathsf{Acc}(lookup(\Gamma, x)) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau$.

Case c^b. By rule (const), $\mathsf{Push}(c^b) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; b$.

Case $\lambda x.M'$. By the typing rule, there are some τ_1, τ_2 such that $\tau = \tau_1 \supset \tau_2$, and $\mathcal{N} \vdash \Gamma, x : \tau_1 \rhd M'\ :\ \tau_2$. By the induction hypothesis, $[\![M']\!] : \Delta_{\Gamma, x:\tau_1} \Rightarrow \Delta_{\Gamma, x:\tau_1}; \tau_2$. Since $\mathcal{S}_S \vdash \Delta_{\Gamma, x:\tau_1}; \tau_2 \rhd \mathsf{Return}\ :\ \tau_2$, by definition, $\mathcal{S}_S \vdash \Delta_{\Gamma, x:\tau_1} \rhd [\![M']\!]; \mathsf{Return}\ :\ \tau_2$. By the bound variable convention in $\mathcal{N}$ we can assume that x is larger than any variables in $dom(\Gamma)$ and therefore $\Delta_{\Gamma, x:\tau_1} = \Delta_\Gamma; \tau_1$. Thus we have $\mathsf{Code}([\![M']\!];\mathsf{Return}) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; (\Delta_\Gamma; \tau_1 \Rightarrow \tau_2)$. Let $n = |\Delta_\Gamma|$. By repeated applications of rule (acc),

$$\mathsf{Code}([\![M']\!];\mathsf{Return});\mathsf{Acc}(0);\cdots;\mathsf{Acc}(n-1) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; (\Delta_\Gamma; \tau_1 \Rightarrow \tau_2); \Delta_\Gamma$$

By rule (app), we have

$$\mathsf{Code}([\![M']\!];\mathsf{Return}));\mathsf{Acc}(0);\cdots;\mathsf{Acc}(n-1);\mathsf{App}(n) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; (\langle \tau_1 \rangle \Rightarrow \tau_2)$$

Case $M_1\ M_2$. By the type system of $\mathcal{N}$, there is some τ_1 such that $\mathcal{N} \vdash \Gamma \rhd M_1\ :\ \tau_1 \supset \tau$ and $\mathcal{N} \vdash \Gamma \rhd M_2\ :\ \tau_1$. By the induction hypothesis for M_1,

$[\![M_1]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; (\langle \tau_1 \rangle \Rightarrow \tau)$. Let x be a variable larger than any variables in $dom(\Gamma)$. By the property of $\mathcal{N}$, $\mathcal{N} \vdash \Gamma, x : \tau_1 \supset \tau \rhd M_2 : \tau_1$. By the choice of x, $\Delta_{\Gamma,x:(\langle \tau_1 \rangle \Rightarrow \tau)} = \Delta_\Gamma; (\langle \tau_1 \rangle \Rightarrow \tau)$. Then by the induction hypothesis for M_2, $[\![M_2]\!] : \Delta_\Gamma; (\langle \tau_1 \rangle \Rightarrow \tau) \Rightarrow \Delta_\Gamma; (\langle \tau_1 \rangle \Rightarrow \tau); \tau_1$ Then we have

$$[\![M_1]\!];[\![M_2]\!];\mathsf{Call(1)} : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau$$

Case (M_1, M_2). By the type system of $\mathcal{N}$, there are some τ_1, τ_2 such that $\tau = \tau_1 \wedge \tau_2$, $\mathcal{N} \vdash \Gamma \rhd M_1 : \tau_1$ and $\mathcal{N} \vdash \Gamma \rhd M_2 : \tau_2$. By the induction hypothesis for M_1, $[\![M_1]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1$. Let x be a variable larger than any variables in $dom(\Gamma)$. By the property of $\mathcal{N}$, $\mathcal{N} \vdash \Gamma, x : \tau_1 \rhd M_2 : \tau_2$. By the choice of x, $\Delta_{\Gamma,x:\tau_1} = \Delta_\Gamma; \tau_1$. Then by the induction hypothesis for M_2, $[\![M_2]\!] : \Delta_\Gamma; \tau_1 \Rightarrow \Delta_\Gamma; \tau_1; \tau_2$ Thus we have

$$[\![M_1]\!];[\![M_2]\!];\mathsf{Pair} : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1 \wedge \tau_2$$

Case $M.1$. By the type system of $\mathcal{N}$, there is some τ_1 such that $\mathcal{N} \vdash \Gamma \rhd M_1 : \tau \wedge \tau_1$. By the induction hypothesis, $[\![M_1]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau \wedge \tau_1$. Thus we have $[\![M_1]\!];\mathsf{Fst} : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau$

The case for $M.2$ is similar.

Case $inl(M_1)$. By the type system of $\mathcal{N}$, there are some τ_1, τ_2 such that $\tau = \tau_1 \vee \tau_2$ and $\mathcal{N} \vdash \Gamma \rhd M_1 : \tau_1$. By the induction hypothesis, $[\![M_1]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1$. Then we have $[\![M_1]\!];\mathsf{Inl} : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1 \vee \tau_2$

The case for $inr(M)$ is similar.

Case *case* M_1 *of* $x.M_2, y.M_3$. There are some τ_1, τ_2 such that $\Gamma \rhd M_1 : \tau_1 \vee \tau_2$, $\Gamma, x : \tau_1 \rhd M_2 : \tau$, and $\Gamma, y : \tau_2 \rhd M_3 : \tau$. By the induction hypothesis for M_1, $[\![M_1]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau_1 \vee \tau_2$. By the bound variable convention in the lambda calculus, we can assume that x, y are larger than any variables in $dom(\Gamma)$. Thus $\Delta_{\Gamma,x:\tau_1} = \Delta_\Gamma; \tau_1$ and $\Delta_{\Gamma,y:\tau_2} = \Delta_\Gamma; \tau_2$. Then by the induction hypotheses for M_2 and M_3, $[\![M_2]\!] : \Delta_\Gamma; \tau_1 \Rightarrow \Delta_\Gamma; \tau_1; \tau$ and $[\![M_3]\!] : \Delta_\Gamma; \tau_2 \Rightarrow \Delta_\Gamma; \tau_2; \tau$ Then by definition $\mathcal{S}_S \vdash \Delta_\Gamma; \tau_1 \rhd [\![M_2]\!];\mathsf{Return} : \tau$ and $\mathcal{S}_S \vdash \Delta_\Gamma; \tau_2 \rhd [\![M_3]\!];\mathsf{Return} : \tau$, and therefore

$$[\![M_1]\!];\mathsf{Case}([\![M_2]\!];\mathsf{Return},[\![M_3]\!];\mathsf{Return}) : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau$$

□

Theorem 3. *If* $\mathcal{N} \vdash \Gamma \rhd M : \tau$ *then there is some SLAM program* C_M *such that* $\mathcal{S}_S \vdash \Delta_\Gamma \rhd C_M : \tau$.

Proof. By Lemma 1, there is some $[\![M]\!]$ such that $[\![M]\!] : \Delta_\Gamma \Rightarrow \Delta_\Gamma; \tau$. Take C_M to be $[\![M]\!];\mathsf{Return}$. Since $\mathcal{S}_S \vdash \Delta_\Gamma; \tau \rhd \mathsf{Return} : \tau$, we have $\mathcal{S}_S \vdash \Delta_\Gamma \rhd C_M : \tau$.
□

Those who have written a byte-code compiler would immediately recognize the similarity between this proof and a compilation algorithm; but they would

also recognize some redundancy this translation produces. This point will be taken up in Section 5 when we discuss implementation issues.

The sequential sequent calculus is sound with respect to the intuitionistic propositional logic, and any sequent provable in the sequential sequent calculus is also provable in other proof systems including the natural deduction and a sequent calculus. Under our Curry-Howard isomorphism, this means that the above theorem is reversible. To formally state this connection, we define the type $\overline{\tau}$ of $\mathcal{N}$ corresponding to a type τ of $\mathcal{S}_S$ as follows.

$$\begin{aligned} \overline{b} &= b \\ \overline{(\langle \tau_1; \ldots; \tau_n \rangle \Rightarrow \tau)} &= \overline{\tau_1} \supset \cdots \overline{\tau_n} \supset \overline{\tau} \\ \overline{\tau_1 \wedge \tau_2} &= \overline{\tau_1} \wedge \overline{\tau_2} \\ \overline{\tau_1 \vee \tau_2} &= \overline{\tau_1} \vee \overline{\tau_2} \end{aligned}$$

Let Δ be an assumption set of $\mathcal{S}_S$ and let $n = |\Delta|$. We write Γ_Δ for the type assignment of $\mathcal{N}$ such that $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma_\Delta(x_i) = \overline{\Delta(i)}$, where $x_1, \ldots, x_n$ are distinct variables chosen in some arbitrary but fix way (we can use integer i itself for x_i.) We can then show the following.

Theorem 4. *If $\mathcal{S}_S \vdash \Delta \rhd C : \tau$ then there is a lambda term M_C such that $\mathcal{N} \vdash \Gamma_\Delta \rhd M_C : \overline{\tau}$.*

Proof. By induction on the derivation of $\mathcal{S}_S \vdash \Delta \rhd C : \tau$ using the substitution lemma in the lambda calculus. □

Different from the relationship between the lambda calculus (natural deduction) and the combinatory logic (Hilbert system), the term obtained by a proof of this theorem is not a trivial one, but reflects the logical structure of the program realized by the code. This has the important implication of opening up the possibility of high-level code analysis. This topic is outside the scope of the current paper. In near future, we shall report on this topic elsewhere.

The above two theorems are proved for the natural deduction. The same results can be proved for Kleene's G3a, yielding translation algorithms between A-normal forms and LAM code.

5 Implementing a functional language using LAM

In this section, we consider some issues in implementing a functional language based on a logical framework presented in this paper.

A typical implementation of a functional language consists of the following steps.

1. A-norm transformation.
2. Lambda lifting and closure optimization.
3. Pseudo code generation.
4. Register allocation and native code generation.

These processes are usually thought of as those required for converting a heigh-level programming language into a low-level machine language. A machine language is indeed low-level for the programmer in the sense that it consists of simple and primitive operations. However, this does not means that code generation and optimization are inevitably "low-level" and ad-hoc. As we have demonstrated, a machine language is understood as a proof system, and code generation process is a proof transformation. These results enables us to analyze various implementation issues in a rigorous logical framework.

In a logical perspective, A-normal transformation is to transform the elimination rules in the natural deduction system to combinations of left rules and cut rules in a sequent calculus. The resulting sequent calculus (Kleene's G3a) is much closer to the sequential sequent calculus, and therefore the subsequent code generation process become much simpler. This topic is outside the scope of the present paper. The interested reader is referred to [17] for Curry-Howard isomorphism for A-normal translation. In the rest of this section, we consider some issues in the context of direct-style compilation for the lambda calculus.

5.1 Lambda lifting and closure optimization

Lambda lifting and associated closure optimization are the processes of absorbing the difference between functions and machine code. (See [18, 1] for the details.) In the lambda calculus, one-argument function abstraction and function application are the primitives. On the other hand, in computer hardware, the unit of execution is a pre-compiled code requiring multiple arguments. Our logical abstract machine properly reflects this situation, and is therefore a suitable formalism to analyze the efficiency of these processes.

Examining the code generation algorithm given in the proof of theorem 3 reveals the redundancy in treating nested lambda abstractions and nested lambda applications. For example, consider the lambda term $(\lambda x.\lambda y.(x, y))\ 1\ 2$. A straightforward application of the algorithm yields the following code in SLAM (written in a linear fashion using labels).

```
          Code(label1);Const(1);Call(1);Const(2);Call(1);Return
label1:   Code(label2);Acc(0);App(1);Return
label2:   Acc(0);Acc(1);Pair;Return
```

Apparently, the intermediate closure creation is redundant. One way of eliminating this redundancy is to translate the lambda calculus to an intermediate language that is closer to the logical abstract machine. We show one possible definition of an intermediate language below.

$$
\begin{aligned}
P &::= \mathit{decl}\ D\ \mathit{in}\ M \\
D &::= \phi \mid D; L = \lambda\langle x_1, \ldots, x_n\rangle.M \\
M &::= x \mid L \mid (MM\ \cdots\ M) \mid (M, M) \mid M.1 \mid M.2 \\
&\quad \mid \mathit{inl}(M) \mid \mathit{inr}(M) \mid \mathit{case}\ M_1\ \mathit{of}\ \lambda x.M_2, \lambda x.M_3
\end{aligned}
$$

In this definition, a program P consists of a sequence of declarations D and a program body. A declaration is an association list of a label (denoted by L) and a multi-argument function $\lambda\langle x_1, \ldots, x_n\rangle.M$ such that $FV(M) = \{x_1, \ldots, x_n\}$, and corresponds to a "super combinator" which is a target expression of lambda lifting [18]. Instead of lambda abstraction and lambda application, the set of terms contains a label L referring to a code, and a function call with multiple arguments.

A type system for this language and a compilation algorithm from this language to the logical abstract machine are easily given. However, a non-trivial static analysis is needed to translate the lambda calculus to this language. This is outside the scope of the present paper. We plan to present a detailed account for the type-directed translation from the lambda calculus to LAM code using this intermediate language elsewhere. Here we only show an example of the compilation process using this intermediate language. Fig. 8 is an actual output of a prototype compiler the author has implemented based on the logical approach presented in this paper. In this example, `$1` is a label and `[$1 1 2]` is function application to multiple arguments. One can see that the generated code does not contain the redundancy mentioned above. Also note that the compiler calculates the maximum height of the stack needed for each code.

```
Source expr:
 (fn x=>fn y=>(x,y)) 1 2

Typecheked and transformed to:
 decls
   $1 = (fn <x,y> => (x,y)) : ('a.'b.<'a;'b> => 'a * 'b)
 in
   [|$1| 1 2]
 end
  : int * int

Compiled to SLAM code:
Start at :label0

Max stack size : 3          Max stack size : 4
label0: Code(label1)        label1: Acc(0)
        Const(1)                    Acc(1)
        Const(2)                    Pair
        App(2)                      Return
        Return
```

Fig. 8. Compilation process for $(\lambda x.\lambda y.(x, y))$ 1 2

5.2 Generating Efficient RLAM code

We have explained our logic-based approach using the stack-based machine SLAM, which is suitable for a byte-code interpreter. However, some inefficiency is inevitable in a stack-based code mainly because its restricted memory access through the stack-top pointer. For example, accessing and pushing two arguments Acc(0) and Acc(1) in the code shown in Fig. 8 for (x, y) is redundant if arguments can be passed through registers. By choosing a register-based machine RLAM we defined in Section 3.2 we can solve this problem. As already mentioned, all the major results so far presented hold for RLAM. Our prototype compiler also generate RLAM machine code. Fig. 9 shows the RLAM code generated for $(\lambda x.\lambda y.(x, y))$ 1 2.

```
Compiled to RLAM code:
Start at :label0

label0: r0 <- Code(label1)                    label1: r5<- Pair(r3,r4)
        r1 <- Const(1)                                Return(r5)
        r2 <- Const(2)
        r0 <- Call r0 with (r3<-r1, r4<-r2)
        Return(r0)
```

Fig. 9. RLAM code for $(\lambda x.\lambda y.(x, y))$ 1 2

In RLAM, an assumption list Γ in each sequent $\Gamma \rhd C : \tau$ is a mapping from variable names to formula, and models the registers used at the time when the first instruction of C is executed. By this interpretation, RLAM is regarded as a register transfer language where an inference step of the form $\dfrac{\Gamma_2 \rhd C : \tau}{\Gamma_1 \rhd I; C : \tau}$ indicates that the instruction I modifies the set of registers indicated by Γ_1 to those of Γ_2. In particular, if Γ_2 extends Γ_1 then I "loads" an empty register with a value. Note that the notation $\Gamma, x : \tau$ may override x in Γ. This corresponds to re-using a register which is no longer "live," i.e. if x is not free in the premise.

By the above simple analysis, it is expected that RLAM can serve as a formal basis to analyze and design register allocation. Katsumata's recent result [8] indicates that this is indeed the case; he have shown that by incorporating the mechanism of linear logic, a calculus similar to RLAM is used to perform precise analysis on register liveness.

6 Conclusions and further investigations

We have developed a logical framework for machine code and code generation. As a logic for machine code, we have define the *sequential sequent calculus*, which is a variant of a sequent calculus whose proof only involves left rules and has a linear (non-branching) structure. We have established a Curry-Howard

isomorphism between this proof system and a sequential machine language. We have then shown that code generation algorithm is extracted from a proof of the equivalence theorem between the natural deduction and the sequential sequent calculus. We have also demonstrated that various implementation issues can be analyzed and expressed in our logical framework.

This is a first step towards a logical approach to implementation of a high-level programming language, and there are a number of interesting issues remain to be investigated. Here we briefly mention some of them.

- Various language extensions.
  In order to apply the framework presented in this paper to actual language implementation, we need to extend it with various features of practical programming languages. Adding recursion is relatively straightforward (though the resulting proof system no longer corresponds to the intuitionistic propositional logic.) A more challenging issue would be to extend the framework to classical logic for representing various control structures such as those investigated in [5]. Another interesting issue in this direction is to consider higher-order logic to represent polymorphism.
- Static analysis of machine code.
  As we have mentioned, we are developing a method to reconstruct the logical structure of a program from machine code. Other possibility would be to develop a framework for type inference and abstract interpretation for machine code. These static analyses would be particularly important in ensuring security in mobile and network programming.
- Development of a practical compiler.
  Another important topic is to consider proof reduction and equational theory for machine code. They will provide a firm basis for code optimization.

We believe that with further research for optimizations, the logical framework presented in this paper will serve as a basis for efficient and robust implementation of high-level programming languages.

Acknowledgments

The author would like to thank Shin'ya Katsumata for stimulating discussion on logic and computer hardware, and for comments on a draft of this paper.

References

1. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
2. G. Cousineau, P-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2), 1987.
3. H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1968.

4. C. Flanagan, A. Sabry, B.F. Duba, and M. Felleisen. The essence of compiling with continuation. In *Proc. ACM PLDI Conference*, pages 237–247, 1993.
5. T. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, 1990.
6. C.A. Gunter. *Semantics of Programming Languages – Structures and Techniques*. The MIT Press, 1992.
7. W. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 476–490. Academic Press, 1980.
8. S. Katsumata, 1999. Personal communication.
9. S. Kleene. *Introduction to Metamathematics*. North-Holland, 1952. 7th edition.
10. J. Lambek. From λ-calculus to cartesian closed categories. In *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 375–402. Academic Press, 1980.
11. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
12. X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1992.
13. T. Lindholm and F. Yellin. *The Java virtual machine specification*. Addison-Wesley, 1996.
14. J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
15. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Proc. International Workshop on Types in Compilation, Springer LNCS 1478*, 1998.
16. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. ACM Symposium on Principles of Programming Languages*, 1998.
17. A Ohori. A Curry-Howard isomorphism for compilation and program execution. In *Proc. Typed Lambda Calculi and Applications, Springer LNCS 1581*, pages 258–179, 1999.
18. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice-Hall, 1987.
19. B. Stata and M. Abadi. A type system for java bytecode subroutines. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 149–160, 1998.
20. D.A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.