A Proof Theory for Machine Code

ATSUSHI OHORI
Japan Advanced Institute of Science and Technology

This paper develops a proof theory for low-level code languages. We first define a proof system, which we refer to as the *sequential sequent calculus*, and show that it enjoys the cut elimination property and that its expressive power is the same as that of the natural deduction proof system. We then establish the Curry-Howard isomorphism between this proof system and a low-level code language by showing the following properties: (1) the set of proofs and the set of typed codes is in one-to-one correspondence, (2) the operational semantics of the code language is directly derived from the cut elimination procedure of the proof system, and (3) compilation and de-compilation algorithms between the code language and the typed lambda calculus are extracted from the proof transformations between the sequential sequent calculus and the natural deduction proof system. This logical framework serves as a basis for the development of type systems of various low-level code languages, type-preserving compilation, and static code analysis.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.4 [**Programming Languages**]: Processors—*Compiler*; *Code generator*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*; F.3.4 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Languages; Theory;

Additional Key Words and Phrases: Curry-Howard isomorphism

1. INTRODUCTION

The general motivation of this study is to establish logical foundations for robust compilation and static code analysis.

Logical foundations for high-level languages have been well established through the Curry-Howard isomorphism [Curry and Feys 1968; Howard 1980] between the typed lambda calculus and the natural deduction proof system. (See, for example, [Gallier 1993] and [Girard et al. 1989] for tutorials on this topic.) In this framework, not only the syntax of the typed terms corresponds to that of the natural deduction proofs, but also the reduction relation underlying the semantics of the lambda calculus corresponds to the proof normalization in the natural deduction

***** Unpublished manuscript. Submitted for publication. *****

A preliminary summary of some of the results were presented at the Fuji International Symposium on Functional and Logic Programming, 1999, as an invited paper entitled: "The Logical Abstract Machine: A Curry-Howard Isomorphism for Machine Code," Springer LNCS 1722, pages 300–318. This work was partially supported by Grant-in-aid for scientific research on the priority area of "informatics" A01-08, grants no:15017239, 16016240; Grant-in-aid for scientific research (B), grant no:15300006; and the Japan MEXT (Ministry of Education, Culture, Sports, Science and Technologies) leading project of "Comprehensive Development of Foundation Software for E-Society" under the title "dependable software development technology based on static program analysis."
Author's current address. Research Institute of Electrical Communication, Tohoku University, Katahira 2-1-1, Aobaku, Sendai 980-8577, Japan. ohori@riec.tohoku.ac.jp.

system. Moreover, the isomorphism is direct and natural so that we can regard the natural deduction proof system as the essence of the type structure of the lambda calculus. This correspondence has provided significant insight into the research on type systems of high-level programming languages, leading to the development of various advanced type systems such as those based on linear logic and classical logic [Wadler 1990; Abramsky 1993; Griffin 1990; Parigot 1992] to name a few.

In contrast with these developments, however, logical foundations for low-level code languages have not been well investigated and as a consequence, the relationship between type structures of high-level languages and those of low-level code languages has not been well understood. Recently, motivated by the need for static verification of low-level code, several type systems for code languages have been proposed. Notable examples include the type system for the Java bytecode language by Stata and Abadi [1998] and the typed assembly language by Morrisett et al [1998; 1998]. They have been successfully used for static verification of low-level code and also for type-safe code generation. However, there does not seem to exist Curry-Howard correspondence for those type systems. Raffalli [1994] has proposed a proof system for a simple code language and has studied its relationship to the natural deduction system. However, its relationship to the (dynamic) semantics of the code language has not been investigated. It is also not entirely clear whether or not this formalism can be applied to practical code languages such as those considered in [Stata and Abadi 1998; Morrisett et al. 1998; Morrisett et al. 1998].

The aim of the present study is to develop a proof theory for low-level code languages and to establish the Curry-Howard isomorphism for the proof system. In particular, we would like to achieve both a rigorous logical correspondence and a natural and direct representation of low-level code. In order to achieve this aim, we first define a proof system of the intuitionistic propositional logic with implication, conjunction and disjunction. An inference rule of this proof system is similar to a left rule in Gentzen's sequent calculus and represents a primitive instruction that changes a machine state. A proof composed of these inference rules corresponds to a code block consisting of instructions in a conventional sequential machine. For this reason, we refer to this calculus as the *sequential sequent calculus*. We prove that this calculus has the cut elimination property by giving a cut elimination procedure. We also show that the expressive power of the proof system is the same as that of the natural deduction proof system by giving proof transformations (in both directions) between the sequential sequent calculus and the natural deduction system.

The proof system is designed to represent both the static and dynamic semantics of a low-level code language. By decorating each inference step with an instruction name, we obtain a typed code language. The set of typable codes is in one-to-one correspondence to the set of proofs. The operational semantics of the code language is directly derived from the cut elimination procedure. These results establish the desired Curry-Howard isomorphism. Moreover, from the relationship between the sequential sequent calculus and the natural deduction system, we derive both compilation and de-compilation algorithms between the typed lambda calculus and the code language. In addition to these rigorous logical correspondences, the obtained type system captures the properties of low-level code in a direct and natural manner

so that it can be applied to various code languages. For example, this framework has already been successfully used for register allocation by proof transformation [Ohori 2004], the development of a type system of Java bytecode [Higuchi and Ohori 2002], proof-directed de-compilation of Java bytecode [Katsumata and Ohori 2001], and static access control for Java bytecode [Higuchi and Ohori 2003].

1.1 Related works

In addition to the works mentioned above [Raffalli 1994; Stata and Abadi 1998; Morrisett et al. 1998; Morrisett et al. 1998], there are other relevant works, which we review in this subsection.

There have been a number of approaches to systematic implementation of a functional programming language using an abstract machine. Notable results include SECD machine [Landin 1964], SK-reduction machine [Turner 1979], and the categorical abstract machine [Cousineau et al. 1987]. From a general perspective, all those approaches are a source of inspiration of this work. Our new contribution is a proof-theoretical account for low-level machine code based on the principle of Curry-Howard isomorphism. In [Cousineau et al. 1987], the authors emphasize that the categorical abstract machine is a low-level machine manipulating a stack. However, their stack-based sequential machine is ad-hoc in the sense that it is outside the categorical model on which their formalism is based. In contrast, our low-level machine is directly derived from a proof theory that models sequential execution of primitive instructions using a stack or registers and therefore its sequential control structure is an essential part of our logical framework.

With regard to this logical correspondence, the SK-reduction machine [Turner 1979] deserves special comments. The core of this approach is the translation from the lambda calculus to the combinatory logic [Curry and Feys 1968]. As observed in [Curry and Feys 1968; Lambek 1980], in the typed setting, this translation algorithm corresponds to a proof of a well known theorem in propositional logic, which states that any formula provable in the natural deduction is also provable in the Hilbert system. This can be regarded as the first example of the correspondence between compilation and proof transformation. Our framework achieves an analogous rigorous logical correspondence for a low-level code language.

1.2 Paper Organization

The proof system we shall develop in this paper differs from existing ones including the natural deduction and the sequent calculus. In order to provide the motivation of the development of our non-conventional proof system, in Section 2, we outline the key insight into logical interpretation of low-level code. Section 3 defines the sequential sequent calculus. Section 4 proves the cut elimination theorem. Section 5 shows that the calculus is equivalent to the natural deduction system by giving proof transformations in both directions. Section 6 extracts a typed low-level code language and its operational semantics from the proof system and its cut elimination procedure. Section 7 derives compilation and de-compilation algorithms for the code language from the proof transformations given in Section 5. Section 8 shows that the framework can be refined to represent various low-level code languages including a register-transfer language, a code language with jumps, and a machine code language with a finite number of registers. Section 9 discusses applications and

extensions of the framework with suggestion for further investigation. Section 10 concludes the paper.

2. TOWARDS PROOF THEORY FOR LOW-LEVEL CODE

In order to establish logical foundations for machine code, we must develop a proof system of the intuitionistic logic that reflects both the syntax and the (operational) semantics of machine code. This section reviews basic properties of machine code and outlines our logical framework.

A code language for a conventional sequential machine has the following general properties.

(1) A code block is a sequence of instructions.
(2) Each instruction performs a primitive operation on machine memory.
(3) The final result of a code block is determined by the return instruction at the end of the block.
(4) A function is implemented by a code pointer optionally with a local environment.

To represent such a machine code language, we interpret a sequent (logical judgment) $\Delta \vdash A$ as a specification of a code block that computes a value of type A from a given machine state represented by Δ and interpret each primitive instruction I that changes a machine state from Δ_1 to Δ_2 as a logical inference rule of the form

$$(\text{Rule-}I)\quad \frac{\Delta_2 \vdash A}{\Delta_1 \vdash A}$$

similar to a left-rule in Gentzen's sequent calculus. Machine oriented meaning can be understood by reading this rule "backward", i.e., it represents the execution of an instruction I that transforms the machine state Δ_1 to Δ_2 and continues the execution of the following instruction sequence represented by the premise. The last instruction that returns the computed value in a code block is represented by the following initial sequent (axiom) in the proof system

$$(\text{taut})\quad A \cdot \Delta \vdash A\ .$$

A code block consisting of a sequence of instructions is represented by a proof of the form

$$\cfrac{\cfrac{A \cdot \Delta_n \vdash A}{\cdots}}{\Delta_1 \vdash A}\ .$$

The execution of the first instruction in a code block corresponds to the elimination of the last inference step. This models sequential execution of a machine, where the "program counter" is incremented to point to the next instruction in the code block.

On the basis of this general strategy, we can construct a proof system for a code language by introducing an appropriate inference rule for each primitive instruction. Instructions that destruct a data structure are represented by left rules in Gentzen's sequent calculus. For example, if we interpret Δ as a stack of values, then the Conjunction-Left rule

$$\frac{A \cdot \Delta \vdash C}{A \wedge B \cdot \Delta \vdash C}$$

represents the instruction that pops a pair from a stack and pushes back the first element of the pair onto the stack. In order to represent instructions that construct a data structure, we need left rules that eliminate a logical connective. For example, the rule

$$\frac{A \wedge B \cdot \Delta \vdash C}{B \cdot A \cdot \Delta \vdash C}$$

represents the instruction that pops two values from a stack, makes a pair consisting of the two values, and pushes the pair back onto the stack.

To represent the feature of using a code pointer (the item (4) above), we need to introduce an inference rule that refers to an existing proof. The rule

$$\frac{\langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash A \qquad \Delta_0 \vdash A_0}{\Delta \vdash A}$$

represents the instruction that pushes onto a stack a pointer to the code represented by the right premise and continues the execution of the code block represented by the left premise. Note that a formula $\langle \Delta_0 \Rightarrow A_0 \rangle$ represents a proof (a code block) which uses an environment Δ_0 different from the current one Δ. This implies that, in order to use this code, it is necessary for the caller to set up an environment Δ_0, to save the current environment, and to transfer control to the code block.

A complete program is represented by a special cut rule between a proof of Δ and a proof of a sequent $\Delta \vdash A$. A top-level proof constructed by the cut rule corresponds to a machine configuration in which a code block of type $\Delta \vdash A$ is bound to a machine state of type Δ. The sequential execution of a code block is modeled by the cut elimination process of the proof system, which successively eliminates the last inference step of the code block proof until it reaches the initial sequent.

On the basis of the above general observation, we construct a sequent-style proof system of the intuitionistic logic that models a code language of a sequential machine.

3. THE SEQUENTIAL SEQUENT CALCULUS

We assume that there is a given set Ax of non-logical axioms (ranged over by a), which corresponds to a set of base (atomic) types. The set of formulas (ranged over by A, B, etc) is given by the following syntax.

$$A ::= a \mid A \wedge A \mid A \vee A \mid \langle \Delta \Rightarrow A \rangle$$

$A \wedge B$ and $A \vee B$ are conjunction and disjunction, corresponding to product and disjoint union, respectively. Δ ranges over (ordered) finite sequences of formulas. We write $[A_1, \ldots, A_n]$ for the sequence consisting of $A_1, \ldots, A_n$; we write $\Delta_1 \cdot \Delta_2$ for the concatenation of Δ_1 and Δ_2; and we write $A \cdot \Delta$ for the sequence obtained from Δ by adding A. $\langle \Delta \Rightarrow A \rangle$ is implication from a sequence of formulas Δ to a formula A and corresponds to function closures containing a code pointer.

Rules for code blocks:

$$(\mathbf{S}\text{-taut})\ \ A \cdot \Delta \vdash_c A \qquad (\mathbf{S}\text{-C})\ \ \frac{A \cdot \Delta \vdash_c B}{\Delta \vdash_c B}\ \ (A \in \Delta)$$

$$(\mathbf{S}\text{-C-true})\ \ \frac{a \cdot \Delta \vdash_c A}{\Delta \vdash_c A}\ \ (a \in Ax) \qquad (\mathbf{S}\text{-C-}\Rightarrow)\ \ \frac{\langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A \quad \Delta_0 \vdash_c A_0}{\Delta \vdash_c A}$$

$$(\mathbf{S}\text{-}\Rightarrow\text{-I1})\ \ \frac{A_0 \cdot \Delta \vdash_c A}{\Delta_0 \cdot \langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A} \qquad (\mathbf{S}\text{-}\Rightarrow\text{-I2})\ \ \frac{\langle \Delta_1 \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A}{\Delta_0 \cdot \langle \Delta_1 \cdot \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A}$$

$$(\mathbf{S}\text{-}\wedge\text{-I1})\ \ \frac{A \cdot \Delta \vdash_c C}{A \wedge B \cdot \Delta \vdash_c C} \qquad (\mathbf{S}\text{-}\wedge\text{-I2})\ \ \frac{B \cdot \Delta \vdash_c C}{A \wedge B \cdot \Delta \vdash_c C}$$

$$(\mathbf{S}\text{-}\wedge\text{-E})\ \ \frac{A \wedge B \cdot \Delta \vdash_c C}{B \cdot A \cdot \Delta \vdash_c C} \qquad (\mathbf{S}\text{-}\vee\text{-I})\ \ \frac{C \cdot \Delta \vdash_c D \quad A \cdot \Delta \vdash_c C \quad B \cdot \Delta \vdash_c C}{A \vee B \cdot \Delta \vdash_c D}$$

$$(\mathbf{S}\text{-}\vee\text{-E1})\ \ \frac{A \vee B \cdot \Delta \vdash_c C}{A \cdot \Delta \vdash_c C} \qquad (\mathbf{S}\text{-}\vee\text{-E2})\ \ \frac{A \vee B \cdot \Delta \vdash_c C}{B \cdot \Delta \vdash_c C}$$

Rules for values

$$(\mathbf{S}\text{-Ax})\ \ \vdash_v a \quad (a \in Ax) \qquad (\mathbf{S}\text{-}\wedge\text{-V})\ \ \frac{\vdash_v A \quad \vdash_v B}{\vdash_v A \wedge B}$$

$$(\mathbf{S}\text{-}\vee\text{-V1})\ \ \frac{\vdash_v A}{\vdash_v A \vee B} \qquad (\mathbf{S}\text{-}\vee\text{-V2})\ \ \frac{\vdash_v B}{\vdash_v A \vee B}$$

$$(\mathbf{S}\text{-}\Rightarrow\text{-V})\ \ \frac{\vdash_e \Delta_1 \quad \Delta_2 \cdot \Delta_1 \vdash_c A}{\vdash_v \langle \Delta_2 \Rightarrow A \rangle}$$

Rules for environments

$$(\mathbf{S}\text{-Seq})\ \ \frac{\vdash_v A \text{ for all } A \in \Delta}{\vdash_e \Delta}$$

The cut rule for top-level proofs

$$(\mathbf{S}\text{-Cut})\ \ \frac{\vdash_e \Delta \quad \Delta \vdash_c A}{\vdash A}$$

Fig. 1. The Sequential Sequent Calculus : **S**

Corresponding to low-level machine code and machine states, the sequential sequent calculus, which we refer to as **S**, has the following forms of judgments:

$$\begin{aligned} \Delta \vdash_c A &\quad \text{for code blocks,} \\ \vdash_v A &\quad \text{for values,} \\ \vdash_e \Delta &\quad \text{for environments, and} \\ \vdash A &\quad \text{for top-level configurations.} \end{aligned}$$

The set of proof rules is given in Fig. 1. In this proof system, logical rules (other than axioms and structural rules) are all left rules, which introduce or eliminate logical connectives. For this reason, we use a label of the form (**S**-$\Box$-I) for a left rule that introduces a connective $\Box$ and (**S**-$\Box$-E) for a left rule that eliminates a connective $\Box$.

We use $\mathcal{C}$, $\mathcal{V}$, $\mathcal{E}$ and $\mathcal{P}$ for meta variables ranging over code block proofs, value proofs, environment proofs, and top-level proofs, respectively. We write $\mathcal{C}(\Delta \vdash_c A)$ for a proof $\mathcal{C}$ whose end sequent (conclusion) is $\Delta \vdash_c A$. Similarly, $\mathcal{V}(\vdash_v A)$ and $\mathcal{E}(\vdash_e \Delta)$ are proofs of $\vdash_v A$ and $\vdash_e \Delta$, respectively. $\mathcal{E}$ is a sequence of value proofs.

We write $\mathcal{E}_1 \cdot \mathcal{E}_2$ for the environment proof obtained by concatenating $\mathcal{E}_1$ and $\mathcal{E}_2$, and write $\mathcal{V} \cdot \mathcal{E}$ for the environment proof obtained by adding $\mathcal{V}$ to $\mathcal{E}$. By definition, if $\mathcal{E}_1(\vdash_e \Delta_1)$ and $\mathcal{E}_1(\vdash_e \Delta_1)$ then $\mathcal{E}_1 \cdot \mathcal{E}_2$ is a proof of $\vdash_e \Delta_1 \cdot \Delta_2$, and if $\mathcal{V}(\vdash_v A)$ and $\mathcal{E}(\vdash_e \Delta)$ then $\mathcal{V} \cdot \mathcal{E}$ is a proof of $\vdash_e A \cdot \Delta$.

Some explanations of the proof rules are in order.

—*Structural rules.* Treatment of structural rules – those for exchange, weakening and contraction – determines the properties of machine states represented by Δ. We do not include the general exchange rule that is found in the sequent calculus and restrict each rule to introduce or eliminate a logical connective at the left-most position in Δ. A limited form of exchange is included in (**S**-C), which can access any formula in Δ. Weakening is implicit in (**S**-taut). There are three forms of contraction rules: (**S**-C), (**S**-C-true) and (**S**-C-⇒). (**S**-C-true) discards a true formulas and (**S**-C-⇒) discards an implication formula that corresponds to a provable sequent. These two rules can be regarded as variants of contraction. Under these rules, Δ corresponds to a runtime stack and a proof corresponds to a code block of a stack machine. Contraction rules represent the instructions to push a constant (**S**-C-true), to push a code pointer (**S**-C-⇒), and to copy an arbitrary element in the stack to the stack top (**S**-C).
In Section 8, we show that by changing the treatment of these structural rules we can represent a register transfer language with unbounded number of registers and a code language for a machine with a fixed number of registers and unbounded amount of memory for saving and restoring registers.

—*Implication rule.* Both the natural deduction system and the sequent calculus contain the implication introduction rule

$$\frac{A \cdot \Delta \vdash B}{\Delta \vdash A \Rightarrow B}$$

which corresponds to the lambda abstraction. However, in a low-level machine, there is no primitive instruction to create a function. A function is implemented by referring to an existing code. As we have explained in the previous section, rule (**S**-C-⇒) models an instruction that pushes a pointer to an existing code onto the current runtime stack. Rules (**S**-⇒-I1) and (**S**-⇒-I2) correspond to invoking a function and making a closure, respectively. We note that, in an actual machine, each code block is assigned a label (address) and a jump instruction is provided to transfer control to a labeled code block. In this setting, passing parameters must be coded explicitly using inference rules. In Section 8.3, we describe how to represent this feature in our logical approach. The rules for implication we consider in this section represent a particular calling convention that is suitable in establishing the correspondence to the natural deduction system.

—*Cut rule.* In the sequent calculus, the cut rule is an ordinary inference rule that can appear anywhere in a proof and the cut elimination procedure inductively transforms the subproofs of a given proof to eliminate all the cuts in the proof. In our calculus, a proof represents a code block consisting of machine instructions. The execution of a code block is a process to transform the machine state successively by the instructions. Reflecting this structure, the calculus restricts the cut rule to be the top-level rule between a proof of a code block and a proof

of a machine state. We shall show that the elimination of this top-level cut rule precisely corresponds to the execution of a code block.

In each inference rule, we refer to the left-most premise as the *major premise*. The major premise corresponds to the "continuation" of the current instruction. For example, in rule (**S**-C-⇒), the right premise represents an existing code block whose pointer is pushed onto the stack and the left premise is the code to be executed after this operation.

This proof system has the desired properties as a proof system of the intuitionistic propositional logic. It is easily checked that it is sound with respect to its semantics, i.e., any provable sequent is valid. Since the current definition of the set of formulas is generated from a set of axioms by conjunction, disjunction and a variant of implication, any formula is valid. For the sake of the argument of soundness, we can consider the set of formulas obtained by adding a given set of propositional variables. For this extended proof system, the soundness can be proved by simple induction on the structure of a proof using the standard definition of the semantics of the logical connectives. Instead of pursuing model theoretical investigation, we focus on its syntactic properties. Among them, the following two are the most important in establishing the Curry-Howard isomorphism for machine code.

(1) The proof system has the cut elimination property.
(2) Its provability is equivalent to that of the natural deduction system of the intuitionistic propositional logic with conjunction, disjunction, and implication.

We shall establish these properties in the subsequent two sections.

4. CUT ELIMINATION

In Gentzen's sequent calculus [Gentzen 1969], cut elimination is proved by pushing cuts towards the leaves of the proof. Since our purpose is not only to show cut elimination property but also to establish a correspondence between cut elimination process and machine execution, we cannot adopt this approach. Our calculus only allows cut at the top level proof between a code proof and an environment proof. Cut elimination need to proceed by transforming a top level cut to another one with a shorter code proof, as depicted below.

$$\dfrac{\mathcal{E}(\vdash_e \Delta) \quad \dfrac{\mathcal{C}_1(\Delta_1 \vdash_c A)}{\Delta \vdash_c A}}{\vdash A} \quad \Longrightarrow \quad \dfrac{\mathcal{E}(\vdash_e \Delta_1) \quad \mathcal{C}_1(\Delta_1 \vdash_c A)}{\vdash A}$$

This process reflects the structure of machine, which executes the first instruction of a given code to update the machine state and proceeds.

There are two difficulties in showing the termination of this form of cut elimination procedure. The first is due to a special form of construction rule (**S**-C-⇒). When we eliminate this inference step, the cut elimination process needs to "save" the right subproof and to proceed with the major premise. The saved subproof will be resumed later by the rules of (**S**-⇒-I1) and (**S**-⇒-I2). By this way, cut elimination is explicitly sequentialized. To show termination of the resumed subproof, we need to "remember" the induction hypothesis of the saved subproof. Another problem is due to the contraction rule (**S**-C). When we eliminate this inference step,

the cut formula will be duplicated. This requires to propagate those remembered property.

We note that in Gentzen's sequent calculus, the first problem does not arise, and the second problem is solved by enhancing the cut rule to an extended rule, often called "mix", which cuts multiple occurrences of formula at one inference step. With this enhancement, one can define a complexity measure on proofs on which cut elimination proof can proceed by induction.

Mainly due to the problem of (**S**-C-⇒) rule, defining a complexity measure for our calculus appears to be difficult. Our strategy is to show a stronger property on value proofs returned by the cut elimination procedure so that the cut elimination proof can proceed by induction on the structure of the given proof. For this purpose, we set up a families of predicates on value proofs. They resemble those of *reducibility* [Tait 1966] (see also [Mitchell 1996] for the detailed exposition of this and related topics), so we call them $\{Red_v(A)\}$ for value proofs and $\{Red_e(\Delta)\}$ for environment proofs. Their definitions are given below.

— $\vdash_v a \in Red_v(a)$.

— $\dfrac{\mathcal{V}_1(\vdash_v A) \quad \mathcal{V}_2(\vdash_v B)}{\vdash_v A \wedge B} \in Red_v(A \wedge B)$ if $\mathcal{V}_1 \in Red_v(A)$ and $\mathcal{V}_2 \in Red_v(B)$.

— $\dfrac{\mathcal{V}(\vdash_v A)}{\vdash_v A \vee B} \in Red_v(A \vee B)$ if $\mathcal{V} \in Red_v(A)$.

— $\dfrac{\mathcal{V}(\vdash_v B)}{\vdash_v A \vee B} \in Red_v(A \vee B)$ if $\mathcal{V} \in Red_v(B)$.

— $\dfrac{\mathcal{E}_0(\vdash_e \Delta_0) \quad \mathcal{C}(\Delta \cdot \Delta_0 \vdash_c A)}{\vdash_v \langle \Delta \Rightarrow A \rangle} \in Red_v(\langle \Delta \Rightarrow A \rangle)$

if for any $\mathcal{E} \in Red_e(\Delta)$, the top level proof $\dfrac{\mathcal{E}(\vdash_e \Delta) \cdot \mathcal{E}_0(\vdash_e \Delta_0) \quad \mathcal{C}(\Delta \cdot \Delta_0 \vdash_c A)}{\vdash A}$

can be transformed to some proof $\mathcal{V} \in Red_v(A)$.

— $\dfrac{\mathcal{V}_1(\vdash A_1) \cdots \mathcal{V}_n(\vdash A_n)}{\vdash_e [A_1, \cdots, A_n]} \in Red_e([A_1, \cdots, A_n])$ if $\mathcal{E}_i \in Red_v(A_i)$ $(1 \leq i \leq n)$

$\mathcal{V} \in Red_v(\langle \Delta \Rightarrow A \rangle)$ intuitively means that the code block proof saved in $\mathcal{V}$ has the cut elimination property. This allows us to remember the induction hypothesis for the right subproof of the rule (**S**-C-⇒), and properly copy the remembered hypothesis when processing the rule (**S**-C).

With these preparations, we now show the cut-elimination theorem.

THEOREM 4.1. *If $\mathcal{C}(\Delta \vdash_c A)$ is a code block proof then, for any environment proof $\mathcal{E} \in Red_e(\Delta)$, the top level proof $\dfrac{\mathcal{E}(\vdash_e \Delta) \quad \mathcal{C}(\Delta \vdash_c A)}{\vdash A}$ can be transformed to some value proof $\mathcal{V} \in Red_v(A)$.*

PROOF. We prove this theorem by giving a cut elimination procedure that transforms a top-level proof to a value proof. The proof is by induction on the structure of $\mathcal{C}$.

We let $\mathcal{E}$ be any environment proof such that $\mathcal{E} \in Red_e(\Delta)$ and consider the top-level proof

$$\mathcal{P} \equiv \quad \frac{\mathcal{E}(\vdash_e \Delta) \quad \mathcal{C}(\Delta \vdash_c A)}{\vdash A} \quad .$$

We proceeds by case analysis in terms of the last inference step in $\mathcal{C}$.

Case (**S**-taut). At this point, we eliminate the cut. $\mathcal{P}$ must be of the form:

$$\frac{\dfrac{\mathcal{V}_1(\vdash_v A) \cdot \mathcal{E}_1(\vdash_e \Delta_1)}{\vdash_e A \cdot \Delta_1} \qquad A \cdot \Delta_1 \vdash_c A}{\vdash A} \; .$$

The cut elimination procedure returns $\mathcal{V}_1$. By the assumption on $\mathcal{E}$, $\mathcal{V}_1 \in Red_v(A)$.

Case (**S**-C). $\mathcal{C}$ must be of the form $\dfrac{\mathcal{C}_1(B \cdot \Delta \vdash_c A)}{\Delta \vdash_c A}$. Since $\mathcal{E} \in Red_e(\Delta)$ and $B \in \Delta$, there exists some $\mathcal{V}_1(\vdash_v B) \in \mathcal{E}$ such that $\mathcal{V}_1 \in Red_v(B)$. We perform the following transformation:

$$\frac{\mathcal{E}(\vdash_e \Delta) \qquad \dfrac{\mathcal{C}_1(B \cdot \Delta \vdash_c A)}{\Delta \vdash_c A}}{\vdash A} \qquad \Longrightarrow \qquad \frac{\mathcal{V}_1(\vdash_v B) \cdot \mathcal{E}(\vdash_e \Delta) \quad \mathcal{C}_1(B \cdot \Delta \vdash_c A)}{\vdash A} \; .$$

Since $\mathcal{V}_1 \cdot \mathcal{E} \in Red_e(B \cdot \Delta)$, the result follows from the induction hypothesis.

Case (**S**-C-true). We perform the following transformation and apply the induction hypothesis.

$$\frac{\mathcal{E}(\vdash_e \Delta) \qquad \dfrac{\mathcal{C}_1(a \cdot \Delta \vdash_c A)}{\Delta \vdash_c A}}{\vdash A} \qquad \Longrightarrow \qquad \frac{a(\vdash_v a) \cdot \mathcal{E}(\vdash_e \Delta) \quad \mathcal{C}_1(a \cdot \Delta \vdash_c A)}{\vdash A}$$

Case (**S**-C-$\Rightarrow$). The given proof must be of the form:

$$\frac{\mathcal{E}(\vdash_e \Delta) \qquad \dfrac{\mathcal{C}_1(\langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A) \quad \mathcal{C}_2(\Delta_0 \vdash_c A_0)}{\Delta \vdash_c A}}{\vdash A}$$

By the induction hypothesis applied to $\mathcal{C}_2$, $\dfrac{\vdash_e \emptyset \quad \mathcal{C}_2(\Delta_0 \vdash_c A_0)}{\vdash_v \langle \Delta_0 \Rightarrow A_0 \rangle} \in Red_v(\langle \Delta_0 \Rightarrow A_0 \rangle)$, and therefore

$$\frac{\vdash_e \emptyset \quad \mathcal{C}_2(\Delta_0 \vdash_c A_0)}{\vdash_v \langle \Delta_0 \Rightarrow A_0 \rangle} \cdot \mathcal{E} \in Red_v(\langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta)$$

The desired result then follows from the induction hypothesis applied to $\mathcal{C}_1$ in the following top-level proof.

$$\frac{\dfrac{\vdash_e \emptyset \quad \mathcal{C}_2(\Delta_0 \vdash_c A_0)}{\vdash_v \langle \Delta_0 \Rightarrow A_0 \rangle} \cdot \mathcal{E}(\vdash_e \Delta) \qquad \mathcal{C}_1(\langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A)}{\vdash A}$$

Case (**S**-⇒-I1). The given derivation must be of the following form:

$$\begin{array}{rcl}\Delta & = & \Delta_1 \cdot \langle \Delta_1 \Rightarrow A_0 \rangle \cdot \Delta_2 \\ \mathcal{E} & = & \mathcal{E}_1(\vdash_e \Delta_1) \cdot \mathcal{V}_1(\langle \Delta_1 \Rightarrow A_0 \rangle) \cdot \mathcal{E}_2(\vdash_e \Delta_2) \\ \mathcal{P} & = & \dfrac{\mathcal{E} \quad \dfrac{\mathcal{C}_1(A_0 \cdot \Delta_2 \vdash_c A)}{\Delta_1 \cdot \langle \Delta_1 \Rightarrow A_0 \rangle \cdot \Delta_2 \vdash_c A}}{\vdash A}\ .\end{array}$$

Since $\mathcal{E} \in Red_e(\Delta)$, $\mathcal{E}_1 \in Red_e(\Delta_1)$ and $\mathcal{V}_1 \in Red_v(\langle \Delta_1 \Rightarrow A_0 \rangle)$. By the definition of Red_v, $\mathcal{V}_1$ must be of the form $\dfrac{\mathcal{E}_0(\vdash_e \Delta_0) \quad \mathcal{C}_2(\Delta_1 \cdot \Delta_0 \vdash_c A_0)}{\vdash_v \langle \Delta_1 \Rightarrow A_0 \rangle}$ such that the top level proof $\dfrac{\mathcal{E}_1 \cdot \mathcal{E}_0 \quad \mathcal{C}_2}{\vdash A_0}$ can be transformed to some proof $\mathcal{V}_0 \in Red_v(A_0)$. Then $\mathcal{V}_0 \cdot \mathcal{E}_2 \in Red_e(A_0 \cdot \Delta_2)$. By the induction hypothesis applied to $\mathcal{C}_1$, the top-level proof

$$\frac{\mathcal{V}_0(\vdash_v A_0) \cdot \mathcal{E}_2(\vdash_e \Delta_2) \quad \mathcal{C}_1(A_0 \cdot \Delta_2 \vdash_c A)}{\vdash_v A}$$

can be transformed to some proof $\mathcal{V}_2 \in Red_v(A)$, as desired.

Case (**S**-⇒-I2). The given derivation must be of the form:

$$\begin{array}{rcl}\Delta & = & \Delta_1 \cdot \langle \Delta_2 \cdot \Delta_1 \Rightarrow A_0 \rangle \cdot \Delta_3 \\ \mathcal{E} & = & \mathcal{E}_1(\vdash_e \Delta_1) \cdot \mathcal{V}_1(\langle \Delta_2 \cdot \Delta_1 \Rightarrow A_0 \rangle) \cdot \mathcal{E}_2(\vdash_e \Delta_3) \\ \mathcal{P} & = & \dfrac{\mathcal{E} \quad \dfrac{\mathcal{C}_1(\langle \Delta_2 \Rightarrow A_0 \rangle \cdot \Delta_3 \vdash_c A)}{\Delta_1 \cdot \langle \Delta_2 \cdot \Delta_1 \Rightarrow A_0 \rangle \cdot \Delta_3 \vdash_c A}}{\vdash A}\ .\end{array}$$

Since $\mathcal{E} \in Red_e(\Delta)$, $\mathcal{E}_1 \in Red_e(\Delta_1)$ and $\mathcal{V}_1 \in Red_v(\langle \Delta_2 \cdot \Delta_1 \Rightarrow A_0 \rangle)$. By the definition of Red_v, $\mathcal{V}_1$ must be of the form $\dfrac{\mathcal{E}_0(\vdash_e \Delta_0) \quad \mathcal{C}_2(\Delta_2 \cdot \Delta_1 \cdot \Delta_0 \vdash_c A_0)}{\vdash_v \langle \Delta_2 \cdot \Delta_1 \Rightarrow A_0 \rangle}$ such that, for any environment proof $\mathcal{E}' \in Red_e(\Delta_2 \cdot \Delta_1)$, the top level proof $\dfrac{\mathcal{E}' \cdot \mathcal{E}_0 \quad \mathcal{C}_2}{\vdash A_0}$ can be transformed to some proof $\mathcal{V}_0 \in Red_v(A_0)$. Let $\mathcal{E}''$ be any environment proof in $Red_e(\Delta_2)$. Since $\mathcal{E}_1 \in Red_e(\Delta_1)$, $\mathcal{E}' \cdot \mathcal{E}_1 \in Red_e(\Delta_2 \cdot \Delta_1)$. Then the top level proof $\dfrac{\mathcal{E}'' \cdot \mathcal{E}_1 \cdot \mathcal{E}_0 \quad \mathcal{C}_2}{\vdash A_0}$ can be transformed to some value proof $\mathcal{V}' \in Red_v(A_0)$. Let $\mathcal{V}_2 = \dfrac{\mathcal{E}_1 \cdot \mathcal{E}_0 \quad \mathcal{C}_2}{\vdash_v \langle \Delta_2 \Rightarrow A_0 \rangle}$. The above property implies that $\mathcal{V}_2 \in Red_v(\langle \Delta_2 \Rightarrow A_0 \rangle)$, and therefore $\mathcal{V}_2 \cdot \mathcal{E}_2 \in Red_v(\langle \Delta_2 \Rightarrow A_0 \rangle \cdot \Delta_3)$. The desired result then follows from the induction hypothesis applied to $\mathcal{C}_1$ for the top level proof $\dfrac{\mathcal{V}_2 \cdot \mathcal{E}_2 \quad \mathcal{C}_1}{\vdash A}$.

Case (**S**-∧-I1). We perform the following transformation and apply the induction

hypothesis.

$$\dfrac{\dfrac{\dfrac{\mathcal{V}_1(\vdash_v B)\quad \mathcal{V}_2(\vdash_v C)}{\vdash_v B\wedge C}\quad \cdot\mathcal{E}_1(\vdash_e \Delta_1)\qquad \dfrac{\mathcal{C}_1(B\cdot\Delta_1\vdash_c A)}{B\wedge C\cdot\Delta_1\vdash_c A}}{\vdash A}}{}$$

$$\Longrightarrow\quad \dfrac{\mathcal{V}_1(\vdash_v B)\cdot\mathcal{E}_1(\vdash_e \Delta_1)\quad \mathcal{C}_1(B\cdot\Delta_1\vdash_c A)}{\vdash A}$$

The case for (**S**-∧-I2) is similar.

Case (**S**-∧-E). We perform the following transformation.

$$\dfrac{\dfrac{\mathcal{V}_2(\vdash_v C)\cdot\mathcal{V}_1(\vdash_v B)\cdot\mathcal{E}_1(\vdash_e \Delta_1)\qquad \dfrac{\mathcal{C}_1(B\wedge C\cdot\Delta_1\vdash_c A)}{C\cdot B\cdot\Delta_1\vdash_c A}}{\vdash A}}{}$$

$$\Longrightarrow\quad \dfrac{\dfrac{\dfrac{\mathcal{V}_1(\vdash_v B)\quad \mathcal{V}_2(\vdash_v C)}{\vdash_v B\wedge C}\quad \cdot\mathcal{E}_1(\vdash_e \Delta_1)\qquad \mathcal{C}_1(B\wedge C\cdot\Delta_1\vdash_c A)}{\vdash A}}{}$$

Since $\dfrac{\mathcal{V}_1(\vdash_v B)\quad \mathcal{V}_2(\vdash_v C)}{\vdash_v B\wedge C}\ \in Red_v(B\wedge C)$, the result follows from the induction hypothesis.

Case (**S**-∨-I). The given derivation must be either of the form:

$$\mathcal{P}\equiv\quad \dfrac{\dfrac{\mathcal{V}_1(\vdash_v B)}{\vdash_v B\vee C}\quad \cdot\mathcal{E}_1(\vdash_e \Delta_1)\qquad \dfrac{\mathcal{C}_1(D\cdot\Delta_1\vdash_c A)\quad \mathcal{C}_2(B\cdot\Delta_1\vdash_c D)\quad \mathcal{C}_3(C\cdot\Delta_1\vdash_c D)}{B\vee C\cdot\Delta_1\vdash_c A}}{\vdash A}$$

or of the form:

$$\mathcal{P}\equiv\quad \dfrac{\dfrac{\mathcal{V}_1(\vdash_v C)}{\vdash_v B\vee C}\quad \cdot\mathcal{E}_1(\vdash_e \Delta_1)\qquad \dfrac{\mathcal{C}_1(D\cdot\Delta_1\vdash_c A)\quad \mathcal{C}_2(B\cdot\Delta_1\vdash_c D)\quad \mathcal{C}_3(C\cdot\Delta_1\vdash_c D)}{B\vee C\cdot\Delta_1\vdash_c A}}{\vdash A}\ .$$

For the former case, by the induction hypothesis applied to $\mathcal{C}_2$ for the following top-level proof

$$\dfrac{\mathcal{V}_1(\vdash_v B)\cdot\mathcal{E}_1(\vdash_e \Delta_1)\quad \mathcal{C}_2(B\cdot\Delta_1\vdash_c D)}{\vdash D}$$

we have a proof $\mathcal{V}_2\in Red_v(D)$. We then apply the induction hypothesis to $\mathcal{C}_1$ for the following top-level proof to obtain the desired proof.

$$\dfrac{\mathcal{V}_2(\vdash_v D)\cdot\mathcal{E}_1(\vdash_e \Delta_1)\quad \mathcal{C}_1(D\cdot\Delta_1\vdash_c A)}{\vdash A}$$

The latter case is similar.

$$(\mathbf{N}\text{-taut})\ \Delta \vdash A \quad (A \in \Delta) \qquad (\mathbf{N}\text{-axiom})\ \Delta \vdash a \quad (a \in Ax)$$

$$(\mathbf{N}\text{-}\Rightarrow\text{-I})\ \frac{A \cdot \Delta \vdash B}{\Delta \vdash \langle A \Rightarrow B\rangle} \qquad (\mathbf{N}\text{-}\Rightarrow\text{-E})\ \frac{\Delta \vdash \langle A \Rightarrow B\rangle \quad \Delta \vdash A}{\Delta \vdash B}$$

$$(\mathbf{N}\text{-}\wedge\text{-I})\ \frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \wedge B} \qquad (\mathbf{N}\text{-}\wedge\text{-E1})\ \frac{\Delta \vdash A \wedge B}{\Delta \vdash A}$$

$$(\mathbf{N}\text{-}\wedge\text{-E2})\ \frac{\Delta \vdash A \wedge B}{\Delta \vdash B} \qquad (\mathbf{N}\text{-}\vee\text{-I1})\ \frac{\Delta \vdash A}{\Delta \vdash A \vee B}$$

$$(\mathbf{N}\text{-}\vee\text{-I2})\ \frac{\Delta \vdash B}{\Delta \vdash A \vee B} \qquad (\mathbf{N}\text{-}\vee\text{-E})\ \frac{\Delta \vdash A \vee B \quad A \cdot \Delta \vdash C \quad B \cdot \Delta \vdash C}{\Delta \vdash C}$$

Fig. 2. The Natural Deduction Proof System : **N**

Case (**S**-∨-E1). We perform the following transformation and apply the induction hypothesis.

$$\frac{\mathcal{V}_1(\vdash_v B) \cdot \mathcal{E}_1(\vdash_e \Delta_1) \qquad \dfrac{\mathcal{C}_1(B \vee C \cdot \Delta_1 \vdash_c A)}{B \cdot \Delta_1 \vdash_c A}}{\vdash A}$$

$$\Longrightarrow \quad \frac{\dfrac{\mathcal{V}_1(\vdash_v B)}{\vdash_v B \vee C} \cdot \mathcal{E}_1(\vdash_e \Delta_1) \qquad \mathcal{C}_1(B \vee C \cdot \Delta_1 \vdash_c A)}{\vdash A}$$

The case for (**S**-∨-I2) is similar.

We have exhausted all the cases. □

As a special case, we have the following.

Corollary 4.2. *Any closed program of the form*

$$\frac{\vdash_e \emptyset \quad \mathcal{C}(\emptyset \vdash_c A)}{\vdash A}$$

can be transformed to a value proof of $\vdash_v A$.

5. RELATIONSHIP TO THE NATURAL DEDUCTION SYSTEM

This section establishes that the sequential sequent calculus is equivalent to the natural deduction proof system.

We consider the natural deduction system with the following set of formulas.

$$A ::= a \mid \langle A \Rightarrow A\rangle \mid A \wedge A \mid A \vee A$$

Fig. 5 gives the set of proof rules. This system is denoted by **N**.

Our goal in this section is to show that $\Delta \vdash A$ is provable in **S** if and only if $\Delta \vdash A$ is provable in **N**. We first consider the if part.

To construct a proof transformation from **N** to **S**, we first define the notion of proof transformers in **S**. The *major-premise path* of a proof is the path of the sequents obtained by traversing the major premise of each inference step in the proof from the end sequent to the initial sequent. We let $\mathcal{C}[\,]$ be a *partial proof* in **S** having a hole $[\,]$ at the position of the initial sequent on the major-premise path in a proof and write $\mathcal{C}[\mathcal{C}_1]$ for the proof obtained by filling the hole with a proof $\mathcal{C}_1$.

We note that all the sequents appearing on the major-premise path in a proof have the same conclusion formula, which is uniquely determined by the initial sequent on the path. Due to this property, we regard a partial proof $\mathcal{C}[\ ]$ as a transformer of a proof of the form $\mathcal{C}_1(\Delta_1 \vdash_c A)$ to a proof of the form $\mathcal{C}[\mathcal{C}_1](\Delta_2 \vdash_c A)$ *for any formula* A and we write

$$\mathcal{C}[\ ] : \Delta_2 \Longrightarrow \Delta_1.$$

We refer to such a partial proof $\mathcal{C}[\ ]$ as a *proof transformer* of type $\Delta_2 \Longrightarrow \Delta_1$. From this definition, if $\mathcal{C}_1[\ ] : \Delta_1 \Longrightarrow \Delta_2$ and $\mathcal{C}_2$ is a proof of $\Delta_2 \vdash_c A$ then $\mathcal{C}_1[\mathcal{C}_2]$ is a proof of $\Delta_1 \vdash_c A$. In writing a concrete proof transformer, we write $*$ for the conclusion formula that will be determined by the proof being filled in the hole. Following is a simple example of a proof transformer.

$$\frac{[\ ]}{A{\wedge}B \cdot \Delta \vdash_c *}\ (\mathbf{S}\text{-}\wedge\text{-I1}) \quad : A{\wedge}B \cdot \Delta \Longrightarrow A \cdot \Delta$$

We write $\Delta \subseteq \Delta'$ if Δ can be embedded in Δ', i.e., $\Delta \subseteq \Delta'$ if $\Delta = [A_1, \ldots, A_n]$, $\Delta' = [B_1, \ldots, B_m]$, $n \leq m$, and there is some injective function f from $\{1, \ldots, n\}$ to $\{1, \ldots, m\}$ such that $A_i = B_{f(i)}$ and if $i < j$ then $f(i) < f(j)$. Note that if $\Delta \subseteq \Delta'$ then the order of the formulas in Δ is preserved in Δ'.

Our goal is to develop a proof transformation algorithm that transforms a proof of the form $\mathcal{D}(\Delta \vdash A)$ in $\mathbf{N}$ to a proof in $\mathbf{S}$. To do this by induction on $\mathcal{D}$, we generalize the problem to the one to construct a proof transformer of the form $\mathcal{C}[\ ] : \Delta' \Longrightarrow A \cdot \Delta'$ in $\mathbf{S}$ for any Δ' such that $\Delta \subseteq \Delta'$. This process involves serialization of proof transformations of subproofs in rules ($\mathbf{N}$-$\Rightarrow$-E), ($\mathbf{N}$-$\wedge$-I) and ($\mathbf{N}$-$\vee$-E). Extending the environment Δ to some Δ' is necessary to process these rules. The following lemma solves this generalized problem.

LEMMA 5.1. *If there is a proof* $\mathcal{D}(\Delta \vdash A)$ *in* $\mathbf{N}$ *and* $\Delta \subseteq \Delta'$*, then there is a proof transformer* $\mathcal{C}[\ ] : \Delta' \Longrightarrow A \cdot \Delta'$ *in* $\mathbf{S}$.

PROOF. Let Δ' be environment such that $\Delta \subseteq \Delta'$. The proof is by induction on $\mathcal{D}$.

Case ($\mathbf{N}$-taut). We take $\mathcal{C}[\ ]$ to be the proof transformer.

$$\frac{[\ ]}{\Delta' \vdash_c *}\ (A \in \Delta') \text{ by rule } (\mathbf{S}\text{-C}) \quad : \Delta' \Longrightarrow A \cdot \Delta'$$

Case ($\mathbf{N}$-axiom). Similarly to the above using the rule ($\mathbf{S}$-C-true)

Case ($\mathbf{N}$-$\Rightarrow$-I). $\mathcal{D}$ must be of the form:

$$\frac{\mathcal{D}_1(A \cdot \Delta \vdash B)}{\Delta \vdash \langle A{\Rightarrow}B\rangle}\ .$$

Since $A \cdot \Delta \subseteq A \cdot \Delta'$, by the induction hypothesis applied to $\mathcal{D}_1$, we have a proof transformer $\mathcal{C}_1[\ ] : A \cdot \Delta' \Longrightarrow B \cdot A \cdot \Delta'$. By applying this to the initial sequent $B \cdot A \cdot \Delta' \vdash_c B$, we have a proof $\mathcal{C}_1[B \cdot A \cdot \Delta' \vdash_c B](A \cdot \Delta' \vdash_c B)$. We then have the

following partial proof in **S**

$$\dfrac{\dfrac{\dfrac{[\,]}{\Delta' \cdot \langle A \cdot \Delta' \Rightarrow B\rangle \cdot \Delta' \vdash_c *}\ (\mathbf{S}\text{-}\Rightarrow\text{-I2})}{\langle A \cdot \Delta' \Rightarrow B\rangle \cdot \Delta' \vdash_c *}\ (\mathbf{S}\text{-C}) \qquad \mathcal{C}_1[B \cdot A \cdot \Delta' \vdash_c B](A \cdot \Delta' \vdash_c B)}{\Delta' \vdash_c *}\ (\mathbf{S}\text{-C-}\Rightarrow)$$

which is a desired proof transformer of type $\Delta' \Longrightarrow \langle A \Rightarrow B\rangle \cdot \Delta'$.

Case (**N**-⇒-E). $\mathcal{D}$ must be of the form:

$$\dfrac{\mathcal{D}_1(\Delta \vdash \langle A \Rightarrow B\rangle) \quad \mathcal{D}_2(\Delta \vdash A)}{\Delta \vdash B}\ .$$

By the induction hypothesis applied to $\mathcal{D}_1$ and $\mathcal{D}_2$, we have

$$\mathcal{C}_1[\,] \,:\, \Delta' \Longrightarrow \langle A \Rightarrow B\rangle \cdot \Delta'\ ,\ \text{and}$$
$$\mathcal{C}_2[\,] \,:\, \langle A \Rightarrow B\rangle \cdot \Delta' \Longrightarrow A \cdot \langle A \Rightarrow B\rangle \cdot \Delta'\ .$$

By filling the hole of $\mathcal{C}_1$ with $\mathcal{C}_2[\,]$ we have

$$\mathcal{C}_1[\mathcal{C}_2[\,]] : \Delta' \Longrightarrow A \cdot \langle A \Rightarrow B\rangle \cdot \Delta'$$

Then we have

$$\mathcal{C}_1\left[\mathcal{C}_2\left[\ \dfrac{[\,]}{A \cdot \langle A \Rightarrow B\rangle \cdot \Delta' \vdash_c *}\ (\mathbf{S}\text{-}\Rightarrow\text{-I1})\ \right]\right] : \Delta' \Longrightarrow B \cdot \Delta'$$

as desired.

Case (**N**-∧-I). $\mathcal{D}$ must be of the form:

$$\dfrac{\mathcal{D}_1(\Delta \vdash A) \quad \mathcal{D}_2(\Delta \vdash B)}{\Delta \vdash A \wedge B}$$

By the induction hypothesis applied to $\mathcal{D}_1$ and $\mathcal{D}_2$, we have

$$\mathcal{C}_1[\,] \,:\, \Delta' \Longrightarrow A \cdot \Delta',\ \text{and}$$
$$\mathcal{C}_2[\,] \,:\, A \cdot \Delta' \Longrightarrow B \cdot A \cdot \Delta'\ .$$

By filling the hole of $\mathcal{C}_1$ with $\mathcal{C}_2[\,]$ we have

$$\mathcal{C}_1[\mathcal{C}_2[\,]] : \Delta' \Longrightarrow B \cdot A \cdot \Delta'.$$

Then we have

$$\mathcal{C}_1\left[\mathcal{C}_2\left[\ \dfrac{[\,]}{B \cdot A \cdot \Delta' \vdash_c *}\ (\mathbf{S}\text{-}\wedge\text{-E})\ \right]\right] : \Delta' \Longrightarrow A \wedge B \cdot \Delta'$$

as desired.

Case (**N**-∧-E1). $\mathcal{D}$ must be of the form:

$$\dfrac{\mathcal{D}_1(\Delta \vdash A \wedge B)}{\Delta \vdash A}\ .$$

By the induction hypothesis, we have

$$\mathcal{C}_1[\ ] : \Delta' \Longrightarrow A \wedge B \cdot \Delta'.$$

From this we have

$$\mathcal{C}_1 \left[\ \frac{[\ ]}{A \wedge B \cdot \Delta' \vdash_c *} \ (\mathbf{S}\text{-}\wedge\text{-I1}) \ \right] : \Delta' \Longrightarrow A \cdot \Delta'$$

as desired. The case for (**N**-∧-E2) is similar.

Case (**N**-∨-I1). $\mathcal{D}$ must be of the form:

$$\frac{\mathcal{D}_1(\Delta \vdash A)}{\Delta \vdash A \vee B} \ .$$

By the induction hypothesis, we have

$$\mathcal{C}_1[\ ] : \Delta' \Longrightarrow A \cdot \Delta'.$$

From this we have

$$\mathcal{C}_1 \left[\ \frac{[\ ]}{A \cdot \Delta' \vdash_c *} \ (\mathbf{S}\text{-}\vee\text{-E1}) \ \right] : \Delta' \Longrightarrow A \vee B \cdot \Delta'$$

as desired. The case for (**N**-∨-I2) is similar.

Case (**N**-∨-E). $\mathcal{D}$ must be of the form:

$$\frac{\mathcal{D}_1(\Delta \vdash A \vee B) \quad \mathcal{D}_2(A \cdot \Delta \vdash C) \quad \mathcal{D}_3(B \cdot \Delta \vdash C)}{\Delta \vdash C} \ .$$

By the induction hypotheses, we have

$$\begin{aligned} &\mathcal{C}_1[\ ] \ : \ \Delta' \Longrightarrow A \vee B \cdot \Delta' \ , \\ &\mathcal{C}_2[\ ] \ : \ A \cdot \Delta' \Longrightarrow C \cdot A \cdot \Delta' \ , \text{ and} \\ &\mathcal{C}_3[\ ] \ : \ B \cdot \Delta' \Longrightarrow C \cdot B \cdot \Delta' \ . \end{aligned}$$

By filling the holes of $\mathcal{C}_2$ and $\mathcal{C}_3$ with the corresponding axions, we have the following proofs.

$$\begin{aligned} \mathcal{C}'_2(A \cdot \Delta' \vdash_c C) \ &= \ \mathcal{C}_2[C \cdot A \cdot \Delta' \vdash_c C \ (\mathbf{S}\text{-taut})](A \cdot \Delta' \vdash_c C) \\ \mathcal{C}'_3(B \cdot \Delta' \vdash_c C) \ &= \ \mathcal{C}_3[C \cdot B \cdot \Delta' \vdash_c C \ (\mathbf{S}\text{-taut})](B \cdot \Delta' \vdash_c C) \end{aligned}$$

From these, we construct the following partial proof

$$\mathcal{C}_1 \left[\ \frac{[\ ] \quad \mathcal{C}'_2(A \cdot \Delta' \vdash_c C) \quad \mathcal{C}'_3(B \cdot \Delta' \vdash_c C)}{A \vee B \cdot \Delta' \vdash_c *} \ (\mathbf{S}\text{-}\vee\text{-I}) \ \right]$$

which is the desired proof transformer of type $\Delta' \Longrightarrow C \cdot \Delta'$. □

THEOREM 5.2. *If $\Delta \vdash A$ is provable in* **N** *then $\Delta \vdash_c A$ is provable in* **S**.

PROOF. Let $\mathcal{D}(\Delta \vdash A)$ be a given proof in **N**. By Lemma 5.1, there exists a proof transformer $\mathcal{C}[\ ] : \Delta \Longrightarrow A \cdot \Delta$. Then $\mathcal{C}[A \cdot \Delta \vdash_c A \ (\mathbf{S}\text{-taut})]$ is a proof of $\Delta \vdash_c A$ in **S**. □

The above theorem states that any natural deduction proof corresponds to some code block proof in **S**. A closed proof $\mathcal{D}(\emptyset \vdash A)$ in **N** corresponds to the following top-level proof in **S**.

$$\frac{\vdash_e \emptyset \quad \mathcal{C}[A \vdash_c A](\emptyset \vdash_c A)}{\vdash A}\ (\mathbf{S}\text{-cut})$$

where $\mathcal{C}[\ ]$ is the proof transformer of type $\emptyset \Longrightarrow A$ obtained from $\mathcal{D}$.

The converse of Theorem 5.2 also holds. To formally state this connection, we define the type $\overline{A}$ of **N** corresponding to type A of **S** as follows.

$$\begin{aligned}
\overline{b} &= b \\
\overline{\langle [A_1, \ldots, A_n] \Rightarrow A \rangle} &= (\overline{A_n} \Rightarrow \cdots (\overline{A_1} \Rightarrow \overline{A})) \\
\overline{A_1 \wedge A_2} &= \overline{A_1} \wedge \overline{A_2} \\
\overline{A_1 \vee A_2} &= \overline{A_1} \vee \overline{A_2}
\end{aligned}$$

We also write $\overline{\Delta}$ for the sequence of formulas obtained from Δ by applying the above translation elementwise. We now show the following.

THEOREM 5.3. *If $\Delta \vdash_c A$ is provable in* **S** *then $\overline{\Delta} \vdash \overline{A}$ is provable in* **N**.

PROOF. We use the following lemmas in the natural deduction system.

LEMMA 5.4. *If $A \cdot \Delta \vdash B$ and $\Delta \vdash A$ then $\Delta \vdash B$*

LEMMA 5.5. *If $\Delta \vdash A$ and $\Delta \subseteq \Delta'$ then $\Delta' \vdash A$.*

Let $\mathcal{C}$ be any proof of $\Delta \vdash_c A$ in **S**. The proof of the theorem is by induction on $\mathcal{C}$. We proceed by case analysis in terms of the last inference step.

Case (**S**-taut). $\overline{A} \cdot \overline{\Delta} \vdash \overline{A}$ is also a proof in **N**.

Case (**S**-C). $\mathcal{C}$ must be of the form:

$$\frac{\mathcal{C}_1(B \cdot \Delta \vdash_c A)}{\Delta \vdash_c A}\ .$$

for some $B \in \Delta$. By the induction hypothesis, there exists a proof $\mathcal{D}_1$ of $\overline{B} \cdot \overline{\Delta} \vdash \overline{A}$. Since $\overline{\Delta} \vdash \overline{B}$ is a proof in **N**, by Lemma 5.4, we have $\overline{\Delta} \vdash \overline{A}$. The case for (**S**-C-true) is similar.

Case (**S**-C-⇒). $\mathcal{C}$ must be of the form:

$$\frac{\mathcal{C}_1(\langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A) \quad \mathcal{C}_2(\Delta_0 \vdash_c A_0)}{\Delta \vdash_c A}\ .$$

By the induction hypothesis applied to $\mathcal{C}_1$ and $\mathcal{C}_2$, $\overline{\langle \Delta_0 \Rightarrow A_0 \rangle} \cdot \overline{\Delta} \vdash \overline{A}$ and $\overline{\Delta_0} \vdash \overline{A_0}$ are provable in **N**. By repeated applications of (**N**-⇒-I) in **N**, $\emptyset \vdash \overline{\langle \Delta_0 \Rightarrow A_0 \rangle}$ is provable in **N**. Then by Lemma 5.5 and Lemma 5.4, $\overline{\Delta} \vdash \overline{A}$ is provable in **N**.

Case (**S**-⇒-I1). $\mathcal{C}$ must be of the form:

$$\frac{\mathcal{C}_1(A_0 \cdot \Delta \vdash_c A)}{\Delta_0 \cdot \langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A}\ .$$

By the induction hypothesis, $\overline{A_0} \cdot \overline{\Delta} \vdash \overline{A}$ in **N**. By Lemma 5.5, $\overline{A_0} \cdot \overline{\Delta_0} \cdot \overline{\langle \Delta_0 \Rightarrow A_0 \rangle} \cdot \overline{\Delta} \vdash \overline{A}$ in **N**. Let $[A_1, \ldots, A_n] = \Delta_0$, and $\Delta_i = [A_1, \ldots, A_{n-i}]$. Also let $\Delta' = \Delta_0 \cdot \langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta$. We construct the following proof in **N**.

$$\dfrac{\dfrac{\dfrac{\overline{\Delta'} \vdash \overline{\langle \Delta_0 \Rightarrow A_0 \rangle} \quad \overline{\Delta'} \vdash \overline{A_n}}{\overline{\Delta'} \vdash \overline{\langle \Delta_1 \Rightarrow A_0 \rangle}} \quad \overline{\Delta'} \vdash \overline{A_{n-1}}}{\vdots} \quad \vdots}{\dfrac{\overline{\Delta'} \vdash \overline{\langle A_1 \Rightarrow A_0 \rangle} \quad \overline{\Delta'} \vdash \overline{A_1}}{\overline{\Delta'} \vdash \overline{A_0}}}$$

By Lemma 5.4, we have $\overline{\Delta_0} \cdot \overline{\langle \Delta_0 \Rightarrow A_0 \rangle} \cdot \overline{\Delta} \vdash \overline{A}$ in **N** as desired.

Case (**S**-⇒-I2). $\mathcal{C}$ must be of the form:

$$\dfrac{\mathcal{C}_1(\langle \Delta' \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A)}{\Delta_0 \cdot \langle \Delta' \cdot \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash_c A} .$$

By the induction hypothesis, $\overline{\langle \Delta' \Rightarrow A_0 \rangle} \cdot \overline{\Delta} \vdash \overline{A}$ in **N**. By Lemma 5.5, $\overline{\langle \Delta' \Rightarrow A_0 \rangle} \cdot \overline{\Delta_0} \cdot \overline{\langle \Delta' \cdot \Delta_0 \Rightarrow A_0 \rangle} \cdot \overline{\Delta} \vdash \overline{A}$ in **N**. Let $[A_1, \ldots, A_n] = \Delta_0$, and $\Delta_i = [A_1, \ldots, A_{n-i}]$. Also let $\Delta'' = \Delta_0 \cdot \langle \Delta' \cdot \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta$. We construct the following proof in **N**.

$$\dfrac{\dfrac{\dfrac{\overline{\Delta''} \vdash \overline{\langle \Delta' \cdot \Delta_0 \Rightarrow A_0 \rangle} \quad \overline{\Delta''} \vdash \overline{A_n}}{\overline{\Delta''} \vdash \overline{\langle \Delta' \cdot \Delta_1 \Rightarrow A_0 \rangle}} \quad \overline{\Delta''} \vdash \overline{A_{n-1}}}{\vdots} \quad \vdots}{\dfrac{\overline{\Delta''} \vdash \overline{\langle \Delta' \cdot A_1 \Rightarrow A_0 \rangle} \quad \overline{\Delta''} \vdash \overline{A_1}}{\overline{\Delta''} \vdash \overline{\langle \Delta' \Rightarrow A_0 \rangle}}}$$

By Lemma 5.4, we have $\overline{\Delta_0} \cdot \overline{\langle \Delta' \cdot \Delta_0 \Rightarrow A_0 \rangle} \cdot \overline{\Delta} \vdash \overline{A}$ in **N** as desired.

Case (**S**-∧-I1). $\mathcal{C}$ must be of the form:

$$\dfrac{\mathcal{C}_1(A \cdot \Delta \vdash_c C)}{A \wedge B \cdot \Delta \vdash_c C} .$$

By the induction hypothesis, $\overline{A} \cdot \overline{\Delta} \vdash \overline{C}$ in **N**. By Lemma 5.5, $\overline{A} \cdot \overline{A \wedge B} \cdot \overline{\Delta} \vdash \overline{C}$ in **N**. We have the following proof in **N**.

$$\dfrac{\overline{A \wedge B} \cdot \overline{\Delta} \vdash \overline{A \wedge B}}{\overline{A \wedge B} \cdot \overline{\Delta} \vdash \overline{A}}$$

Then by Lemma 5.4, we have $\overline{A \wedge B} \cdot \overline{\Delta} \vdash \overline{C}$ in **N**. The case for (∧-I2) is similar.

Case (**S**-∧-E). $\mathcal{C}$ must be of the form:

$$\dfrac{\mathcal{C}_1(A \wedge B \cdot \Delta \vdash_c C)}{B \cdot A \cdot \Delta \vdash_c C} .$$

By the induction hypothesis, $\overline{A \wedge B} \cdot \overline{\Delta} \vdash \overline{C}$ in **N**. By Lemma 5.5, $\overline{A \wedge B} \cdot \overline{B} \cdot \overline{A} \cdot \overline{\Delta} \vdash \overline{C}$ in **N**. We have the following proof in **N**.

$$\dfrac{\overline{B} \cdot \overline{A} \cdot \overline{\Delta} \vdash \overline{A} \quad \overline{B} \cdot \overline{A} \cdot \overline{\Delta} \vdash \overline{B}}{\overline{B} \cdot \overline{A} \cdot \overline{\Delta} \vdash \overline{A \wedge B}}$$

Then by Lemma 5.4, we have $\overline{B} \cdot \overline{A} \cdot \overline{\Delta} \vdash \overline{C}$ in $\mathbf{N}$.

Case ($\mathbf{S}$-$\vee$-I). $\mathcal{C}$ must be of the form:

$$\frac{\mathcal{C}_1(C \cdot \Delta \vdash_c D) \quad \mathcal{C}_2(A \cdot \Delta \vdash_c C) \quad \mathcal{C}_3(B \cdot \Delta \vdash_c C)}{A \vee B \cdot \Delta \vdash_c D} \ .$$

By the induction hypotheses together with Lemma 5.5, there exist some proofs $\mathcal{D}_1(\overline{C} \cdot \overline{A} \vee \overline{B} \cdot \overline{\Delta} \vdash \overline{D})$, $\mathcal{D}_2(\overline{A} \cdot \overline{A} \vee \overline{B} \cdot \overline{\Delta} \vdash \overline{C})$, and $\mathcal{D}_3(\overline{B} \cdot \overline{A} \vee \overline{B} \cdot \overline{\Delta} \vdash \overline{C})$ in $\mathbf{N}$. Using the latter two, we construct the following proof in $\mathbf{N}$.

$$\frac{\overline{A} \vee \overline{B} \cdot \overline{\Delta} \vdash \overline{A} \vee \overline{B} \quad \mathcal{D}_2(\overline{A} \cdot \overline{A} \vee \overline{B} \cdot \overline{\Delta} \vdash \overline{C}) \quad \mathcal{D}_3(\overline{B} \cdot \overline{A} \vee \overline{B} \cdot \overline{\Delta} \vdash \overline{C})}{\overline{A} \vee \overline{B} \cdot \overline{\Delta} \vdash \overline{C}}$$

By Lemma 5.4 applied to this proof and $\mathcal{D}_1$, we have $\overline{A \vee B} \cdot \overline{\Delta} \vdash \overline{D}$ in $\mathbf{N}$.

Case ($\mathbf{S}$-$\vee$-E1). $\mathcal{C}$ must be of the form:

$$\frac{\mathcal{C}_1(A \vee B \cdot \Delta \vdash_c C)}{A \cdot \Delta \vdash_c C} \ .$$

By the induction hypotheses, $\overline{A \vee B} \cdot \overline{\Delta} \vdash \overline{C}$ in $\mathbf{N}$. We have the following proof in $\mathbf{N}$.

$$\frac{\overline{A} \cdot \overline{\Delta} \vdash \overline{A}}{\overline{A} \cdot \overline{\Delta} \vdash \overline{A \vee B}}$$

Then by Lemma 5.5 and Lemma 5.4, we have $\overline{A} \cdot \overline{\Delta} \vdash \overline{C}$ in $\mathbf{N}$. The case for ($\mathbf{S}$-$\vee$-E2) is similar. □

6. THE LOGICAL ABSTRACT MACHINE

The previous development is entirely constructive, from which we can systematically extract a machine code language and an execution model of the machine. We refer to the resulting system as the *logical abstract machine*. This section presents its simple variant. In Section 8 we shall show a few code languages closer to practical machine code.

6.1 The Code Language

By regarding the environment Δ in each code block proof $\Delta \vdash_c A$ as a description of a runtime stack, we obtain a stack-based code language from the definition of $\mathbf{S}$. We write $\Delta(n)$ for the n-th element in Δ counting from the right, starting with 0, and write $|\Delta|$ for the length of Δ. In particular, $\Delta(0)$ is the bottom element of the stack and A in $A \cdot \Delta$ is the $|\Delta|$-th element in the stack.

We let I and $\mathcal{B}$ range over *instructions* and *code blocks* (lists of instructions), respectively. We adopt the convention that the left-most instruction of $\mathcal{B}$ is the first one to execute and write $I \cdot \mathcal{B}$ (and $\mathcal{B}' \cdot \mathcal{B}$) for the code block obtained by appending I ($\mathcal{B}'$) at the front of $\mathcal{B}$. The code language is defined by the following syntax.

$$\begin{aligned} I &::= \mathsf{Acc}(n) \mid \mathsf{Const}(c^a) \mid \mathsf{Code}(\mathcal{B}) \mid \mathsf{Call(n)} \mid \mathsf{App}(n) \mid \mathsf{Fst} \mid \mathsf{Snd} \mid \mathsf{Pair} \\ &\quad \mid \mathsf{Case}(\mathcal{B}, \mathcal{B}) \mid \mathsf{Inl} \mid \mathsf{Inr} \\ \mathcal{B} &::= \mathsf{Return} \mid I \cdot \mathcal{B} \end{aligned}$$

$$\text{(s-taut)}\quad A \cdot \Delta \vdash \mathsf{Return} : A$$

$$\text{(s-C)}\quad \frac{A \cdot \Delta \vdash \mathcal{B} : B}{\Delta \vdash \mathsf{Acc}(n) \cdot \mathcal{B} : B} \quad (\Delta(n) = A)$$

$$\text{(s-C-true)}\quad \frac{a \cdot \Delta \vdash \mathcal{B} : A}{\Delta \vdash \mathsf{Const}(c^a) \cdot \mathcal{B} : A} \quad (\text{if } c^a \text{ is a constant of base type } a)$$

$$\text{(s-C-}\Rightarrow\text{)}\quad \frac{\langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash \mathcal{B} : A \qquad \Delta_0 \vdash \mathcal{B}_0 : A_0}{\Delta \vdash \mathsf{Code}(\mathcal{B}_0) \cdot \mathcal{B} : A}$$

$$\text{(s-}\Rightarrow\text{-I1)}\quad \frac{A_0 \cdot \Delta \vdash \mathcal{B} : A}{\Delta_0 \cdot \langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash \mathsf{Call}(n) \cdot \mathcal{B} : A} \quad (|\Delta_0| = n)$$

$$\text{(s-}\Rightarrow\text{-I2)}\quad \frac{\langle \Delta_1 \Rightarrow A_0 \rangle \cdot \Delta \vdash \mathcal{B} : A}{\Delta_0 \cdot \langle \Delta_1, \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta \vdash \mathsf{App}(n) \cdot \mathcal{B} : A} \quad (|\Delta_0| = n)$$

$$\text{(s-}\wedge\text{-I1)}\quad \frac{A \cdot \Delta \vdash \mathcal{B} : C}{A \wedge B \cdot \Delta \vdash \mathsf{Fst} \cdot \mathcal{B} : C}$$

$$\text{(s-}\wedge\text{-I2)}\quad \frac{B \cdot \Delta \vdash \mathcal{B} : C}{A \wedge B \cdot \Delta \vdash \mathsf{Snd} \cdot \mathcal{B} : C}$$

$$\text{(s-}\wedge\text{-E)}\quad \frac{A \wedge B \cdot \Delta \vdash \mathcal{B} : C}{B \cdot A \cdot \Delta \vdash \mathsf{Pair} \cdot \mathcal{B} : C}$$

$$\text{(s-}\vee\text{-I)}\quad \frac{C \cdot \Delta \vdash \mathcal{B} : D \qquad A \cdot \Delta \vdash \mathcal{B}_1 : C \qquad B \cdot \Delta \vdash \mathcal{B}_2 : C}{A \vee B \cdot \Delta \vdash \mathsf{Case}(\mathcal{B}_1, \mathcal{B}_2) \cdot \mathcal{B} : D}$$

$$\text{(s-}\vee\text{-E1)}\quad \frac{A \vee B \cdot \Delta \vdash \mathcal{B} : C}{A \cdot \Delta \vdash \mathsf{Inl} \cdot \mathcal{B} : C}$$

$$\text{(s-}\vee\text{-E2)}\quad \frac{A \vee B \cdot \Delta \vdash \mathcal{B} : C}{B \cdot \Delta \vdash \mathsf{Inr} \cdot \mathcal{B} : C}$$

Fig. 3. The Type System of the Logical Abstract Machine

The intended meaning of instructions are as follows. Return returns the stack top element to the caller. $\mathsf{Acc}(n)$ copies the n-th element of the stack to the stack top. $\mathsf{Const}(c^a)$ pushes the constant c^a onto the stack. $\mathsf{Code}(\mathcal{B})$ pushes a pointer to the code block $\mathcal{B}$. $\mathsf{Call}(n)$ pops n arguments and a function closure from the stack, applies the function closure to the arguments, and pushes the result onto the stack. $\mathsf{App}(n)$ pops n arguments and a function closure from the stack and pushes onto the stack the function closure obtained by extending the closure's environment with the n arguments. The other instructions are primitive operations for products and sums.

The type system of this language is obtained by decorating each inference rule of **S** with the corresponding instruction name. Fig. 3 shows the typing rules. If we erase the code blocks from the rules, then the resulting system is exactly **S**.

Proposition 6.1. *There is one-to-one correspondence between the set of code block proofs in* **S** *and the set of typing derivations in the code language.*

6.2 Operational Semantics

The operational semantics of the code language is derived from the proof of the cut elimination theorem (Theorem 4.1).

Corresponding to value proofs in **S**, the set of *runtime values* (ranged over by v)

Value typings

$$(\mathbf{s}\text{-Ax})\ \vdash c^a : a \quad (\text{if } c^a \text{ is a constant of type } a) \qquad (\mathbf{s}\text{-}\wedge\text{-V})\ \frac{\vdash v_1 : A \quad \vdash v_2 : B}{\vdash (v_1, v_2) : A \wedge B}$$

$$(\mathbf{s}\text{-}\vee\text{-V1})\ \frac{\vdash v : A}{\vdash inl(v) : A \vee B} \qquad (\mathbf{s}\text{-}\vee\text{-V2})\ \frac{\vdash v : B}{\vdash inr(v) : A \vee B}$$

$$(\mathbf{s}\text{-}\Rightarrow\text{-V})\ \frac{\vdash \delta : \Delta_1 \quad \Delta_2, \Delta_1 \vdash \mathcal{B} : A}{\vdash cls(\delta, \mathcal{B}) : \langle \Delta_2 \Rightarrow A \rangle}$$

Stack typings

$$(\mathbf{s}\text{-seq})\ \frac{\vdash v_i : A_i\ (1 \leq i \leq n)}{\vdash [v_1, \ldots, v_n] : [A_1, \ldots, A_n]}$$

Dump typings

$$(\mathbf{s}\text{-dump1})\ \vdash \emptyset : \langle A \Rightarrow A \rangle \qquad (\mathbf{s}\text{-dump2})\ \frac{\vdash cls(\delta, \mathcal{B}) : \langle A \Rightarrow B \rangle \quad \vdash \gamma : \langle B \Rightarrow C \rangle}{\vdash (\delta, \mathcal{B}) \cdot \gamma : \langle A \Rightarrow C \rangle}$$

Typing rules for machine configurations

$$(\mathbf{s}\text{-top})\ \frac{\vdash \delta : \Delta \quad \Delta \vdash \mathcal{B} : A \quad \vdash \gamma : \langle A \Rightarrow B \rangle}{\vdash (\delta, \mathcal{B}, \gamma) : B}$$

Fig. 4. Typing Rules for Runtime Structures

is defined by the following syntax.

$$v ::= c^a \mid (v, v) \mid inl(v) \mid inr(v) \mid cls(\delta, \mathcal{B})$$

(v_1, v_2) is a pair of values. $inl(v)$ and $inr(v)$ are left and right injections to a union type, respectively. $cls(\delta, \mathcal{B})$ is a function closure consisting of a stack δ and a code block $\mathcal{B}$. A *runtime stack* (ranged over by δ) is an ordered finite sequence of values, and corresponds to an environment proof in **S**.

In addition to runtime stacks, the machine requires the following data structure

$$\gamma ::= \emptyset \mid (\delta, \mathcal{B}) \cdot \gamma$$

which represents suspended computation (continuation) that should be resumed when the current code block terminates. γ plays the same role as the dump of the SECD machine in Landin [1964], so in what follows we call them *dumps*. A *machine configuration* is a triple $(\delta, \mathcal{B}, \gamma)$ consisting of a stack δ, a code block $\mathcal{B}$, and a dump γ.

The set of typing rules for the runtime objects is given in Fig. 4. Parallel to the correspondence between the set of code block proofs in **S** and the set of typing derivations in the code language, the set of typing derivations of values and that of stacks are in one-to-one correspondence to the set of values proofs and that of environment proofs in **S**, respectively. A top-level proof in **S** corresponds to a machine configuration with the empty dump.

Proposition 6.2. *Any top-level proof*

$$\frac{\mathcal{E}(\vdash_e \Delta) \quad \mathcal{C}(\Delta \vdash_c A)}{\vdash A}$$

$$(v \cdot \delta,\ \mathsf{Return},\ \emptyset) \longrightarrow (v,\ \emptyset, \emptyset)$$
$$(v \cdot \delta,\ \mathsf{Return},\ (\delta_0, \mathcal{B}_0) \cdot \gamma) \longrightarrow (v \cdot \delta_0,\ \mathcal{B}_0,\ \gamma)$$
$$(\delta,\ \mathsf{Acc}(n) \cdot \mathcal{B},\ \gamma) \longrightarrow (\delta(n) \cdot \delta,\ \mathcal{B},\ \gamma)$$
$$(\delta,\ \mathsf{Const}(c^a) \cdot \mathcal{B},\ \gamma) \longrightarrow (c^a \cdot \delta,\ \mathcal{B},\ \gamma)$$
$$(\delta,\ \mathsf{Code}(\mathcal{B}_0) \cdot \mathcal{B},\ \gamma) \longrightarrow (cls(\emptyset, \mathcal{B}_0) \cdot \delta,\ \mathcal{B},\ \gamma)$$
$$(\delta_1 \cdot cls(\delta_0, \mathcal{B}_0) \cdot \delta,\ \mathsf{Call}(n) \cdot \mathcal{B},\ \gamma) \longrightarrow (\delta_1 \cdot \delta_0,\ \mathcal{B}_0,\ (\delta, \mathcal{B}) \cdot \gamma) \qquad (|\delta_1| = n)$$
$$(\delta_1 \cdot cls(\delta_0, \mathcal{B}_0) \cdot \delta,\ \mathsf{App}(n) \cdot \mathcal{B},\ \gamma) \longrightarrow (cls(\delta_1 \cdot \delta_0, \mathcal{B}_0) \cdot \delta,\ \mathcal{B},\ \gamma) \qquad (|\delta_1| = n)$$
$$((v_1, v_2) \cdot \delta,\ \mathsf{Fst} \cdot \mathcal{B},\ \gamma) \longrightarrow (v_1 \cdot \delta,\ \mathcal{B},\ \gamma)$$
$$((v_1, v_2) \cdot \delta,\ \mathsf{Snd} \cdot \mathcal{B},\ \gamma) \longrightarrow (v_2 \cdot \delta,\ \mathcal{B},\ \gamma)$$
$$(v_2 \cdot v_1 \cdot \delta,\ \mathsf{Pair} \cdot \mathcal{B},\ \gamma) \longrightarrow ((v_1, v_2) \cdot \delta,\ \mathcal{B},\ \gamma)$$
$$(inl(v) \cdot \delta,\ \mathsf{Case}(\mathcal{B}_1, \mathcal{B}_2) \cdot \mathcal{B},\ \gamma) \longrightarrow (v \cdot \delta,\ \mathcal{B}_1,\ (\delta, \mathcal{B}) \cdot \gamma)$$
$$(inr(v) \cdot \delta,\ \mathsf{Case}(\mathcal{B}_1, \mathcal{B}_2) \cdot \mathcal{B},\ \gamma) \longrightarrow (v \cdot \delta,\ \mathcal{B}_2,\ (\delta, \mathcal{B}) \cdot \gamma)$$
$$(v \cdot \delta,\ \mathsf{Inl} \cdot \mathcal{B},\ \gamma) \longrightarrow (inl(v) \cdot \delta,\ \mathcal{B},\ \gamma)$$
$$(v \cdot \delta,\ \mathsf{Inr} \cdot \mathcal{B},\ \gamma) \longrightarrow (inr(v) \cdot \delta,\ \mathcal{B},\ \gamma)$$

Fig. 5. The Operational Semantics of the Logical Abstract Machine

in **S** *uniquely determines a machine configuration* $(\delta, \mathcal{B}, \emptyset)$ *such that*

$$\frac{\vdash \delta : \Delta \qquad \Delta \vdash \mathcal{B} : A \qquad \vdash \emptyset : \langle A \Rightarrow A \rangle}{\vdash (\delta, \mathcal{B}, \emptyset) : A}$$

Conversely, any typed machine configuration with the empty dump corresponds to a proof of a top-level derivation.

The operational semantics of the code language is defined through a set of rules to transform a machine configuration. We write

$$(\delta,\ \mathcal{B},\ \gamma) \longrightarrow (\delta',\ \mathcal{B}',\ \gamma')$$

if $(\delta,\ \mathcal{B},\ \gamma)$ is transformed to $(\delta',\ \mathcal{B}',\ \gamma')$. Fig. 5 gives the set of transformation rules. The reflexive, transitive closure of $\longrightarrow$ is denoted by $\xrightarrow{*}$. We then define the top-level evaluation relation as follows.

$$\frac{(\delta, \mathcal{B}, \gamma) \xrightarrow{*} (v, \emptyset, \emptyset)}{(\delta, \mathcal{B}, \gamma) \Downarrow v}$$

The evaluation always terminates on any type correct machine configuration, yielding a value of the same type. This desired property is extracted form the cut elimination theorem of **S**.

Corresponding to $\{Red\}$ in **S**, we define a family of predicates $\{red\}$ on the runtime objects. The definitions of $\{red_v\}$ and $\{red_e\}$ on values and stacks are obtained from the corresponding definitions of $\{Red_v\}$ and $\{Red_e\}$ on value proofs and environment proofs, respectively. From the perspective of cut elimination proof, the role of a dump is to record the induction hypothesis to be applied to the major premise after evaluating the case branch in (**S**-$\vee$-I) and the code invocation in (**S**-$\Rightarrow$-I1). So we have the following definition for $\{red_d\}$ on dumps.

—$\emptyset \in red_d(\langle A \Rightarrow A \rangle)$.

—$(\delta, \mathcal{B}) \cdot \gamma \in red_d(\langle A \Rightarrow C \rangle)$ if $cls(\delta, \mathcal{B}) \in red_v(\langle A \Rightarrow B \rangle)$ and $\gamma \in red_d(\langle B \Rightarrow C \rangle)$.

We have the following type soundness theorem, which corresponds to the cut elimination theorem in **S**.

THEOREM 6.3. *If $\Delta \vdash \mathcal{B} : A$ then for any $\delta \in red_e(\Delta)$ and $\gamma \in red_d(\langle A \Rightarrow B \rangle)$, we have $(\delta,\ \mathcal{B},\ \gamma) \Downarrow v$ such that $v \in red_v(B)$.*

PROOF. The proof is by induction on the structure of $\mathcal{B}$.

Let δ and γ be any stack and dump such that $\delta \in red_e(\Delta)$ and $\gamma \in red_d(\langle A \Rightarrow B \rangle)$. The proof stepwise corresponds to the proof of the cut elimination theorem (Theorem 4.1) in **S**.

We first consider the cases other than (**s**-taut), (**s**-$\Rightarrow$-I1), (**s**-$\vee$-I1), and (**s**-$\vee$-I2). In those cases, there must be some I and $\mathcal{B}_0$ such that $\mathcal{B} = I \cdot \mathcal{B}_0$ and

$$\frac{\vdash \delta : \Delta \quad \dfrac{\Delta_0 \vdash \mathcal{B}_0 : A}{\Delta \vdash I \cdot \mathcal{B}_0 : A} \quad \vdash \gamma : \langle A \Rightarrow B \rangle}{\vdash (\delta, I \cdot \mathcal{B}_0, \gamma) : B}\ .$$

From the proof of the case for I in the cut elimination theorem, we can construct δ_0 such that $\vdash \delta_0 : \Delta_0$ and

$$\frac{\vdash \delta_0 : \Delta_0 \quad \Delta_0 \vdash \mathcal{B}_0 : A \quad \vdash \gamma : \langle A \Rightarrow B \rangle}{\vdash (\delta_0, \mathcal{B}_0, \gamma) : B}\ .$$

It can be verified that the following transformation is defined.

$$(\delta, I \cdot \mathcal{B}_0, \gamma) \longrightarrow (\delta_0, \mathcal{B}_0, \gamma),$$

The desired result then follows from the induction hypothesis. If we ignore the unchanged dump, then this proof precisely corresponds to the corresponding case of the proof of the cut elimination theorem.

Each of the cases (**s**-taut), (**s**-$\Rightarrow$-I1), (**s**-$\vee$-I1), and (**s**-$\vee$-I2) manipulates the dump. The first one resumes the computation stored in the dump. The other three save the execution of the current code (the code corresponding to the major premise) in the dump and starts the executing of the new code block given in the instruction or in the stack. In the cut elimination theorem for these three cases, the cut elimination procedure is applied twice: firstly to the new code block proof; secondly to the code block proof of the major premise after the first application. The machine sequentializes these two invocations by recording the code block corresponding to the major premise in the dump. This corresponds to remembering the induction hypothesis of the major premise in the cut elimination proof.

We give the cases for (**s**-taut) and (**s**-$\Rightarrow$-I1) below. The cases for (**s**-$\vee$-I1) and (**s**-$\vee$-I2) are similarly constructed from the corresponding cases of the cut elimination proof.

Case (**s**-taut). $\mathcal{B}$ must be an axiom $A \cdot \Delta_1 \vdash \mathsf{Return} : A$. Then $\delta = v_1 \cdot \delta_1$ such that $v_1 \in red_v(A)$. The case for $\gamma = \emptyset$ is trivial. Suppose $\gamma = (\delta_0, \mathcal{B}_0) \cdot \gamma_1$. Then the machine makes the following transition.

$$(v_1 \cdot \delta_1, \mathsf{Return}, (\delta_0, \mathcal{B}_0) \cdot \gamma_1) \longrightarrow (v_1 \cdot \delta_0, \mathcal{B}_0, \gamma_1).$$

Since $\gamma \in red_d(\langle A \Rightarrow B \rangle)$, $cls(\delta_0, \mathcal{B}_0) \in red_v(\langle A \Rightarrow C \rangle)$ and $\gamma_1 \in red_d(\langle C \Rightarrow B \rangle)$ for some C. Since $v_1 \in red_v(A)$, the desired result follows from the definition of red_v.

Case (**s**-⇒-I1). The typing derivation of $\mathcal{B}$ must be of the form:

$$\frac{A_0 \cdot \Delta_1 \vdash \mathcal{B}_1 \,:\, A}{\Delta_0 \cdot \langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta_1 \vdash \mathsf{Call}(n) \cdot \mathcal{B}_1 \,:\, A}\ .$$

Since $\delta \in red_e(\Delta_0 \cdot \langle \Delta_0 \Rightarrow A_0 \rangle \cdot \Delta_1)$, the given stack δ must be of the form: $\delta_0 \cdot cls(\delta_2, \mathcal{B}_0) \cdot \delta_1$ such that $\delta_1 \in red_e(\Delta_1)$, $cls(\delta_2, \mathcal{B}_0) \in red_v(\langle \Delta_0 \Rightarrow A_0 \rangle)$, $\delta_0 \in red_e(\Delta_0)$, and $|\delta_0| = n$. Then the machine makes the following transition.

$$(\delta_0 \cdot cls(\delta_2, \mathcal{B}_0) \cdot \delta_1, \mathsf{Call}(n) \cdot \mathcal{B}_1, \gamma) \longrightarrow (\delta_0 \cdot \delta_2, \mathcal{B}_0, (\delta_1, \mathcal{B}_1) \cdot \gamma).$$

Since $\delta_1 \in red_e(\Delta_1)$, by the induction hypothesis, $cls(\delta_1, \mathcal{B}_1) \in red_v(\langle A_0 \Rightarrow A \rangle)$. Since $\gamma \in red_d(\langle A \Rightarrow B \rangle)$, $(\delta_1, \mathcal{B}_1) \cdot \gamma \in red_d(\langle A_0 \Rightarrow B \rangle)$. The result then follows from the definition of red_d for $cls(\delta_2, \mathcal{B}_0)$. Under our interpretation of a dump to be a recorded induction hypothesis, this proof precisely corresponds to the case (⇒-I1) in the proof of the cut elimination theorem. □

Since $\emptyset \in red_e(\emptyset)$ and $\emptyset \in red_d(\langle A \Rightarrow A \rangle)$, the above theorem implies the following.

COROLLARY 6.4. *If* $\vdash (\emptyset, \mathcal{B}, \emptyset) : A$ *then* $(\emptyset, \mathcal{B}, \emptyset) \Downarrow v$ *such that* $\vdash v : A$.

7. COMPILATION AND DE-COMPILATION AS PROOF TRANSFORMATIONS

We have so far established the relationship between the sequential sequent calculus **S** and the natural deduction system **N**. We have also established the Curry-Howard isomorphism between **S** and the logical abstract machine. By combining these results, we immediately obtain compilation and de-compilation algorithms between the lambda calculus and the logical abstract machine code.

We consider the following set of lambda terms.

$$\begin{aligned} M \ ::= \ & c^a \mid x \mid \lambda x : A.M \mid M\ M \mid (M, M) \mid M.1 \mid M.2 \\ & \mid \ inl(M) \mid inr(M) \mid \mathit{case}\ M_1\ \mathit{of}\ x : A.M_2, y : A.M_3 \end{aligned}$$

A *type assignment* Γ is a function from a finite set of variables to formulas. The type system of the lambda terms is obtained from the natural deduction proof system by decorating each inference rule with the corresponding term constructor. Fig. 6 gives the set of typing rules.

To establish a correspondence between a nameless environment Δ in **N** and a named type assignment Γ in the lambda calculus, we assume a linear order in the set of variables and regard Γ as a sequence. Furthermore, we assume that, in the judgments of form $\Gamma \vdash \lambda x : A.M : \langle A \Rightarrow B \rangle$ and $\Gamma \vdash \mathit{case}\ M_1\ \mathit{of}\ x : A.M_2, y : B.M_3 : C$, the bound variables x, y have higher indexes than those of variables appear in the type assignment Γ. For a type assignment Γ in **N** we let Δ_Γ be the corresponding environment obtained by the linear ordering on variables, i.e., if $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ such that $x_i < x_j$ for any $1 \leq i < j \leq n$, then $\Delta_\Gamma = [A_n, \ldots, A_1]$. If $x \in dom(\Gamma)$ then we write $lookup(\Gamma, x)$ for the position in Δ_Γ corresponding to x. Note that this convention makes a new local variable always allocated on the top of the stack having the highest index.

We first show that any lambda term $\Gamma \vdash M : A$ can be translated to a code block of the logical abstract machine, which, when executed with a stack of type Δ_Γ, extends Δ_Γ with a value of type A. Corresponding to partial proofs in **S**, we refer to a sequence of instructions that does not terminate with Return as a *partial*

$$(\mathbf{n}\text{-taut})\ \ \Gamma \vdash x : A \quad (x : A \in \Gamma) \qquad (\mathbf{n}\text{-axiom})\ \ \Gamma \vdash c^a : a$$

$$(\mathbf{n}\text{-}\Rightarrow\text{-I})\ \ \frac{\Gamma \cup \{x : A\} \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \Rightarrow B} \qquad (\mathbf{n}\text{-}\Rightarrow\text{-E})\ \ \frac{\Gamma \vdash M_1 : A \Rightarrow B \quad \Gamma \vdash M_2 : A}{\Gamma \vdash M_1\ M_2 : B}$$

$$(\mathbf{n}\text{-}\wedge\text{-I})\ \ \frac{\Gamma \vdash M_1 : A \quad \Gamma \vdash M_2 : B}{\Gamma \vdash (M_1, M_2) : A \wedge B} \qquad (\mathbf{n}\text{-}\wedge\text{-E1})\ \ \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash M.1 : A}$$

$$(\mathbf{n}\text{-}\wedge\text{-E2})\ \ \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash M.2 : B} \qquad (\mathbf{n}\text{-}\vee\text{-I1})\ \ \frac{\Gamma \vdash M : A}{\Gamma \vdash inl(M) : A \vee B}$$

$$(\mathbf{n}\text{-}\vee\text{-I2})\ \ \frac{\Gamma \vdash M : B}{\Gamma \vdash inr(M) : A \vee B}$$

$$(\mathbf{n}\text{-}\vee\text{-E})\ \ \frac{\Gamma \vdash M_1 : A \vee B \quad \Gamma \cup \{x : A\} \vdash M_2 : C \quad \Gamma \cup \{y : B\} \vdash M_3 : C}{\Gamma \vdash case\ M_1\ of\ x : A.M_2, y : B.M_3 : C}$$

Fig. 6. The Type System of the Lambda Calculus with Products and Sums

code block. We let $\mathcal{B}$ range over partial code blocks as well as code blocks. We define the following predicate on partial code blocks.

$$\mathcal{B} : \Delta_1 \Rightarrow \Delta_2 \Longleftrightarrow \text{for any } \mathcal{B}' \text{ and } A, \text{ if } \Delta_2 \vdash \mathcal{B}' : A \text{ then } \Delta_1 \vdash \mathcal{B} \cdot \mathcal{B}' : A.$$

LEMMA 7.1. *If* $\Gamma \vdash M : A$ *then there is a partial code block* $comp(\Gamma, M)$ *such that* $comp(\Gamma, M) : \Delta_\Gamma \Rightarrow A \cdot \Delta_\Gamma$.

PROOF. The proof is obtained by decorating the proof of Lemma 5.1 with term constructors in the lambda calculus and instructions in the logical abstract machine code. Fig. 7 shows the algorithm of $comp(\Gamma, M)$ extracted from the proof of Lemma 5.1. □

The algorithm *comp* constructed in the proof of this lemma recursively traverses subterms and concatenates the instructions. For the case of lambda abstraction, the algorithm first compiles the body M_1 and obtains a code and it then makes a closure by applying the code to the set of free variables using App instruction. Those who have written a byte-code compiler would immediately recognize the similarity between this proof and a compilation algorithm.

A complete compilation algorithm is obtained by adding the Return instruction at the end of the code sequence as shown in the following.

THEOREM 7.2. *If* $\Gamma \vdash M : A$ *then there is a code block* $\mathcal{B}_M$ *such that* $\Delta_\Gamma \vdash \mathcal{B}_M : A$.

PROOF. By Lemma 7.1, there exists some partial code block $comp(\Gamma, M)$ such that $comp(\Gamma, M) : \Delta_\Gamma \Rightarrow A \cdot \Delta_\Gamma$. Take $\mathcal{B}_M$ to be $comp(\Gamma, M)$·Return. Since $A \cdot \Delta_\Gamma \vdash \mathsf{Return} : A$, we have $\Delta_\Gamma \vdash \mathcal{B}_M : A$. □

The converse of Theorem 7.2 also holds. We write $[N/x]M$ for the lambda term obtained from M by substituting N for x. We assume that there is a set of variables indexed by natural numbers. Let Γ_Δ be the type assignment $\{x_i : \overline{\Delta(i)} | 0 \leq i < |\Delta|\}$, where x_i is the variable whose index is i.

THEOREM 7.3. *If* $\Delta \vdash \mathcal{B} : A$ *then there exists a lambda term* $M_\mathcal{B}$ *such that* $\Gamma_\Delta \vdash M_\mathcal{B} : \overline{A}$.

$$
\begin{array}{l}
comp(\Gamma, x) = \mathsf{Acc}(lookup(\Gamma, x)) \\
comp(\Gamma, c^a) = \mathsf{Const}(c^a) \\
comp(\Gamma, \lambda x : A.M_1) \\
\quad = \text{let } n = |\Gamma| \\
\qquad \text{in } \mathsf{Code}(comp(\Gamma \cup \{x : A\}, M_1) \cdot \mathsf{Return}) \cdot \mathsf{Acc}(0) \cdots \mathsf{Acc}(n-1) \cdot \mathsf{App}(n) \\
comp(\Gamma, M_1\ M_2) = comp(\Gamma, M_1) \cdot comp(\Gamma, M_2) \cdot \mathsf{Call}(1) \\
comp(\Gamma, (M_1, M_2)) = comp(\Gamma, M_1) \cdot comp(\Gamma, M_2) \cdot \mathsf{Pair} \\
comp(\Gamma, M_1.1) = comp(\Gamma, M_1) \cdot \mathsf{Fst} \\
comp(\Gamma, M_1.2) = comp(\Gamma, M_1) \cdot \mathsf{Snd} \\
comp(\Gamma, inl(M_1)) = comp(\Gamma, M_1) \cdot \mathsf{Inl} \\
comp(\Gamma, inr(M_1)) = comp(\Gamma, M_1) \cdot \mathsf{Inr} \\
comp(\Gamma, case\ M_1\ of\ x : A.M_2, y : B.M_3) \\
\quad = comp(\Gamma, M_1) \cdot \mathsf{Case}(comp(\Gamma \cup \{x : A\}, M_2) \cdot \mathsf{Return}, comp(\Gamma \cup \{y : B\}, M_3) \cdot \mathsf{Return})
\end{array}
$$

Fig. 7. Compilation Algorithm From the Typed Lambda Calculus to the Logical Abstract Machine

PROOF. By decorating the proof of Theorem 5.3 with term constructors in the lambda calculus and instructions in the logical abstract machine code, we obtain an algorithm $decomp\,(\mathcal{C})$ which constructs a code block $M_{\mathcal{B}}$ from any given proof $\mathcal{C}$ of a sequent $\Delta \vdash \mathcal{B} : A$ satisfying the property $\Gamma_\Delta \vdash decomp\,(\mathcal{C}) : \overline{A}$. Fig. 8 shows the algorithm extracted from the proof of Theorem 5.3. □

The decompilation algorithm recursively decompiles the code sequence starting from Return and then applies the substitution that represents the effect of the last instruction. In the case of $\mathsf{Code}(\mathcal{B}_0) \cdot \mathcal{B}$, for example, it first decompiles the code $\mathcal{B}_0$ referenced in the first instruction and obtains a corresponding function term. It then decompiles the code $\mathcal{B}$ to obtain a main term. Finally, it substitutes the function term for the variable $x_{|\Delta|}$, which denotes the function, in the main term Different from the relationship between the lambda calculus and the combinatory logic (Hilbert system), the term obtained from the proof of this theorem is not a trivial one, but reflects the logical structure of the program realized by the code. This property opens up the possibility of analyzing low-level code by transforming it to a high-level proof system that is more suitable for various static analysis. Based on this general observation, in [Katsumata and Ohori 2001], a proof-directed de-compilation method for Java bytecode has been developed.

8. REPRESENTING CODE LANGUAGES CLOSER TO MACHINE CODE

The logical abstract machine we defined in Section 6 is directly derived from the proof system **S** by regarding the environment Δ as a representation of a runtime stack. However, if we interpret an environment as a description of a set of variables, then we obtain a code language for machine with registers. Moreover, we can represent a variety of code languages by changing the treatment of structural rules on environments. This section describes typical ones.

$$decomp\,(A \cdot \Delta \vdash \mathsf{Return} \,:\, A) = x_{|\Delta|}$$
$$decomp\,(\Delta \vdash \mathsf{Acc}(n) \cdot \mathcal{B} \,:\, B) = [x_n/x_{|\Delta|}](decomp\,(\Delta(n) \cdot \Delta \vdash \mathsf{Acc}(n) \cdot \mathcal{B} \,:\, B))$$
$$decomp\,(\Delta \vdash \mathsf{Const}(c^a) \cdot \mathcal{B} \,:\, A) = [c^a/x_{|\Delta|}](decomp\,(a \cdot \Delta \vdash \mathcal{B} \,:\, A))$$
$$decomp\left(\frac{\langle \Delta_0 \Rightarrow A_0\rangle \cdot \Delta \vdash \mathcal{B} \,:\, A \quad \Delta_0 \vdash \mathcal{B}_0 \,:\, A_0}{\Delta \vdash \mathsf{Code}(\mathcal{B}_0) \cdot \mathcal{B} \,:\, A}\right)$$
$$= \text{let } M_0 = \lambda x_0 \ldots \lambda x_{|\Delta_0|-1}.decomp\,(\Delta_0 \vdash \mathcal{B}_0 \,:\, A_0) \text{ in } [M_0/x_{|\Delta|}](decomp\,(\langle \Delta_0 \Rightarrow A_0\rangle \cdot \Delta \vdash \mathcal{B} \,:\, A))$$
$$decomp\left(\frac{A_0 \cdot \Delta \vdash \mathcal{B} \,:\, A}{\Delta_0 \cdot \langle \Delta_0 \Rightarrow A_0\rangle \cdot \Delta \vdash \mathsf{Call}(n) \cdot \mathcal{B} \,:\, A}\right)$$
$$= \text{let } M_0 = (\cdots (x_n\ x_{n+1}) \cdots\ x_{n+m}) \ \ (|\Delta| = n, |\Delta_0| = m) \text{ in } [M_0/x_n](decomp\,(A_0 \cdot \Delta \vdash \mathcal{B} \,:\, A))$$
$$decomp\left(\frac{\langle \Delta_1 \Rightarrow A_0\rangle \cdot \Delta \vdash \mathcal{B} \,:\, A}{\Delta_0 \cdot \langle \Delta_0, \Delta_1 \Rightarrow A_0\rangle \cdot \Delta \vdash \mathsf{App}(n) \cdot \mathcal{B} \,:\, A}\right) =$$
$$= \text{let } M_0 = (\cdots (x_n\ x_{n+1}) \cdots\ x_{n+m}) \ \ (|\Delta| = n, |\Delta_0| = m) \text{ in } [M_0/x_n](decomp\,(\langle \Delta_1 \Rightarrow A_0\rangle \cdot \Delta \vdash \mathcal{B} \,:\, A))$$
$$decomp\left(\frac{A \cdot \Delta \vdash \mathcal{B} \,:\, C}{A \wedge B \cdot \Delta \vdash \mathsf{Fst} \cdot \mathcal{B} \,:\, C}\right) = [x_{|\Delta|}.1/x_{|\Delta|}](decomp\,(A \cdot \Delta \vdash \mathcal{B} \,:\, C))$$
$$decomp\left(\frac{B \cdot \Delta \vdash \mathcal{B} \,:\, C}{A \wedge B \cdot \Delta \vdash \mathsf{Snd} \cdot \mathcal{B} \,:\, C}\right) = [x_{|\Delta|}.2/x_{|\Delta|}](decomp\,(B \cdot \Delta \vdash \mathcal{B} \,:\, C))$$
$$decomp\left(\frac{A \wedge B \cdot \Delta \vdash \mathcal{B} \,:\, C}{B \cdot A \cdot \Delta \vdash \mathsf{Pair} \cdot \mathcal{B} \,:\, C}\right) = [(x_{|\Delta|}, x_{|\Delta|+1})/x_{|\Delta|}](decomp\,(A \wedge B \cdot \Delta \vdash \mathcal{B} \,:\, C))$$
$$decomp\left(\frac{C \cdot \Delta \vdash \mathcal{B} \,:\, D \quad A \cdot \Delta \vdash \mathcal{B}_1 \,:\, C \quad B \cdot \Delta \vdash \mathcal{B}_2 \,:\, C}{A \vee B \cdot \Delta \vdash \mathsf{Case}(\mathcal{B}_1,\mathcal{B}_2) \cdot \mathcal{B} \,:\, D}\right) =$$
$$= \text{let } M_0 = \mathit{case}\ x_{|\Delta|}\ \mathit{of}\ x_{|\Delta|}.decomp\,(A \cdot \Delta \vdash \mathcal{B}_1 \,:\, C), x_{|\Delta|}.decomp\,(B \cdot \Delta \vdash \mathcal{B}_2 \,:\, C) \text{ in } [M_0/x_{|\Delta|}](decomp\,(C \cdot \Delta \vdash \mathcal{B} \,:\, D))$$
$$decomp\left(\frac{A \vee B \cdot \Delta \vdash \mathcal{B} \,:\, C}{A \cdot \Delta \vdash \mathsf{Inl} \cdot \mathcal{B} \,:\, C}\right) = [inl(x_{|\Delta|})/x_{|\Delta|}](decomp\,(A \vee B \cdot \Delta \vdash \mathcal{B} \,:\, C))$$
$$decomp\left(\frac{A \vee B \cdot \Delta \vdash \mathcal{B} \,:\, C}{B \cdot \Delta \vdash \mathsf{Inr} \cdot \mathcal{B} \,:\, C}\right) = [inr(x_{|\Delta|})/x_{|\Delta|}](decomp\,(A \vee B \cdot \Delta \vdash \mathcal{B} \,:\, C))$$

Fig. 8. De-compilation Algorithm of the Logical Abstract Machine

8.1 A Register Transfer Language

The simplest variant of a code language with variables is a register transfer language with unbounded number of registers. In this language, an instruction has typically the following "three-address" format

$$x = op(y, z)$$

which uses y, z and assigns the result to x. By specifying the input and output in this way for each instructions of the code language consider in the previous section, we obtain the following set of instructions and code blocks.

$$\begin{aligned} I \ ::= \ & x = y \mid x = \mathsf{Const}(c^a) \mid x = \mathsf{Code}(\mathcal{B}) \mid y = \mathsf{Call}\ x\ \mathsf{with}\ (y_1 = x_1, \ldots, y_n = x_n) \\ & \mid y = \mathsf{App}\ x\ \mathsf{to}\ (y_1 = x_1, \ldots, y_n = x_n) \mid x = \mathsf{Fst}(y) \mid x = \mathsf{Snd}(y) \mid x = \mathsf{Pair}(y,z) \\ & \mid x = \mathsf{Case}(y,(z).\mathcal{B}_1,(w).\mathcal{B}_2) \mid x = \mathsf{Inl}(y) \mid x = \mathsf{Inr}(y) \\ \mathcal{B} \ ::= \ & \mathsf{Return}(x) \mid I \cdot \mathcal{B} \end{aligned}$$

In Call and App, $\{x_1, \ldots, x_n\}$ is the set of actual parameter names and $\{y_1, \ldots, y_n\}$ is the corresponding set of formal parameter names used in the callee code.

Suppose an instruction $x = op(y, z)$ requires inputs of type A and B and returns the result of type C, then its inference rule is of the form

$$\frac{\Gamma_1 \cup \{x : C\} \vdash \mathcal{B} \,:\, D}{\Gamma_2 \cup \{y : A, z : B\} \vdash x = op(y, z) \cdot \mathcal{B} \,:\, D}\ .$$

A precise formulation depends on when the input variables y, z and the output variable x are allocated. The simplest variant is to assume that the target x is always distinct from any other variables and y, z are kept until the end of the block $\mathcal{B}$. From the perspective of logic, this corresponds to a proof system in which the weakening rule is implicitly included in the initial sequent in the style of Kleene's G3 system [Kleene 1952]. In this formulation, the above inference rule becomes the following

$$\frac{\Gamma \cup \{y : A, z : B, x : C\} \vdash \mathcal{B} \,:\, D}{\Gamma \cup \{y : A, z : B\} \vdash x = op(y, z) \cdot \mathcal{B} \,:\, D}\ .$$

This approach yields a register transfer language with single assignment property. Fig. 9 gives the type system. The major results we have established in the previous sections on evaluation, compilation and de-compilation are mechanically transferable to this code language.

8.2 Code Languages with Explicit Register Manipulation

The previous language uses unbounded number of variables and it does not re-use these variables. By introducing explicit structural rules, i.e., those for contraction, weakening and exchange, we can also represent a code language with explicit allocation and deallocation of variables.

Allocating a new variable can be modeled by a special form of contraction. In **S**, we already have three forms of contraction rules: (C), (C-⇒), and (C-true). Under the interpretation that an environment represents a set of typed variables, each of these rules performs both allocation of a new variable and assigning a value to it. In order to separate these roles, we introduce a special formula $\top$ whose interpretation is always true. Unlike $a \in Ax$, however, a proof of this formula has no computational meaning. The following contraction rule for $\top$ can then be interpreted as the instruction to allocate a fresh variable with the empty content.

$$\frac{\Gamma \cup \{x : \top\} \vdash \mathcal{B} \,:\, A}{\Gamma \vdash \mathsf{Alloc}(x) \cdot \mathcal{B} \,:\, A} \qquad (x \notin dom(\Gamma))$$

We require that a variable must be present in Γ before it is loaded. This is represented by changing the rules involving assignment as follows

$$\frac{\Gamma \cup \{x : C, y : A, z : B\} \vdash \mathcal{B} \,:\, D}{\Gamma \cup \{x : E, y : A, z : B\} \vdash x = op(y, z) \cdot \mathcal{B} \,:\, D}$$

where C is the result type of op and E can be any formula including $\top$.

Deallocating (freeing) a variable is represented by weakening. In the type system of the register transfer language, weakening is included in the axiom (taut). This means that all the variables used during the execution of the block are kept until

$$(\mathbf{rs}\text{-taut})\quad \Gamma \cup \{x : A\} \vdash \mathsf{Return}(x) \ : \ A$$

$$(\mathbf{rs}\text{-C})\quad \frac{\Gamma \cup \{x : A, y : A\} \vdash \mathcal{B} \ : \ B}{\Gamma \cup \{x : A\} \vdash y = x \cdot \mathcal{B} \ : \ B}$$

$$(\mathbf{rs}\text{-C-true})\quad \frac{\Gamma \cup \{x : b\} \vdash \mathcal{B} \ : \ A}{\Gamma \vdash x = c^b \cdot \mathcal{B} \ : \ A}$$

$$(\mathbf{rs}\text{-C-}\Rightarrow)\quad \frac{\Gamma \cup \{x : \langle \Gamma_0 \Rightarrow A_0 \rangle\} \vdash \mathcal{B} \ : \ A \qquad \Gamma_0 \vdash \mathcal{B}_0 \ : \ A_0}{\Gamma \vdash x = \mathsf{Code}(\mathcal{B}_0) \cdot \mathcal{B} \ : \ A}$$

$$(\mathbf{rs}\text{-}\Rightarrow\text{-I1})\quad \frac{\Gamma_1 \cup \{x : \langle \Gamma_2 \Rightarrow A_0 \rangle ; y : A_0\} \vdash \mathcal{B} \ : \ A}{\begin{array}{l}\Gamma_1 \cup \{x : \langle \Gamma_2 \Rightarrow A_0 \rangle\} \vdash y = \mathsf{Call}\ x\ \mathsf{with}\ (y_1 = x_1, \ldots, y_n = x_n) \cdot \mathcal{B} \ : \ A \\ (\Gamma_2 = \{y_1 : A_1; \cdots ; y_n : A_n\}, \Gamma_1(x_i) = A_i, 1 \leq i \leq n)\end{array}}$$

$$(\mathbf{rs}\text{-}\Rightarrow\text{-I2})\quad \frac{\Gamma_1 \cup \{x : \langle \Gamma_2 \cup \Gamma_3 \Rightarrow A_0 \rangle ; y : \langle \Gamma_3 \Rightarrow A_0 \rangle\} \vdash \mathcal{B} \ : \ A}{\begin{array}{l}\Gamma_1 \cup \{x : \langle \Gamma_2 \cup \Gamma_3 \Rightarrow A_0 \rangle\} \vdash y = \mathsf{App}\ x\ \mathsf{to}\ (y_1 = x_1, \ldots, y_n = x_n) \cdot \mathcal{B} \ : \ A \\ (\Gamma_2 = \{y_1 : A_1; \cdots ; y_n : A_n\}, \Gamma_1(x_i) = A_i, 1 \leq i \leq n)\end{array}}$$

$$(\mathbf{rs}\text{-}\wedge\text{-E})\quad \frac{\Gamma \cup \{x : A, y : B, z : A \wedge B\} \vdash \mathcal{B} \ : \ C}{\Gamma \cup \{x : A, y : B\} \vdash z = \mathsf{Pair}(x,y) \cdot \mathcal{B} \ : \ C}$$

$$(\mathbf{rs}\text{-}\wedge\text{-I1})\quad \frac{\Gamma \cup \{x : A \wedge B, y : A\} \vdash \mathcal{B} \ : \ C}{\Gamma \cup \{x : A \wedge B\} \vdash y = \mathsf{Fst}(x) \cdot \mathcal{B} \ : \ C}$$

$$(\mathbf{rs}\text{-}\wedge\text{-I2})\quad \frac{\Gamma \cup \{x : A \wedge B, y : B\} \vdash \mathcal{B} \ : \ C}{\Gamma \cup \{x : A \wedge B\} \vdash y = \mathsf{Snd}(x) \cdot \mathcal{B} \ : \ C}$$

$$(\mathbf{rs}\text{-}\vee\text{-E1})\quad \frac{\Gamma \cup \{x : A, y : A \vee B\} \vdash \mathcal{B} \ : \ C}{\Gamma \cup \{x : A\} \vdash y = \mathsf{Inl}(x) \cdot \mathcal{B} \ : \ C}$$

$$(\mathbf{rs}\text{-}\vee\text{-E2})\quad \frac{\Gamma \cup \{x : B, y : A \vee B\} \vdash \mathcal{B} \ : \ C}{\Gamma \cup \{x : B\} \vdash y = \mathsf{Inl}(x) \cdot \mathcal{B} \ : \ C}$$

$$(\mathbf{rs}\text{-}\vee\text{-I})\quad \frac{\Gamma \cup \{x : A \vee B, y : C\} \vdash \mathcal{B} \ : \ D \qquad \Gamma \cup \{z_1 : A\} \vdash \mathcal{B}_1 \ : \ C \qquad \Gamma \cup \{z_2 : B\} \vdash \mathcal{B}_2 \ : \ C}{\Gamma \cup \{x : A \vee B\} \vdash y = \mathsf{Case}(x,\ (z_1).\mathcal{B}_1,\ (z_2).\mathcal{B}_2) \cdot \mathcal{B} \ :}$$

Fig. 9. The Type System of the Register Transfer Language

the block exits with the $\mathsf{Return}(x)$ instruction. A tighter control of variable usage is enforced by restricting (taut) as:

$$\{x : A\} \vdash \mathsf{Return}(x) \ : \ A$$

so that all the variables other than x must be deallocated after their use by the following weakening rule

$$\frac{\Gamma \vdash \mathcal{B} \ : \ A}{\Gamma \cup \{x : A\} \vdash \mathsf{Discard}(x) \cdot \mathcal{B} \ : \ A} \ .$$

The above refinement achieves tighter control of variable usage, but the number of variables remains unbounded. To represent a machine with a fixed number of registers, we decompose a type assignment Γ into two sets: Σ for memory and Π for registers and require the length of Π to be less than a given constant k of the number of registers. All the primitive operations must only use Π. The inference rule for *op* above now becomes the following

$$\frac{\Sigma \mid \Pi \cup \{x : C, y : A, z : B\} \vdash_k \mathcal{B} \ : \ D}{\Sigma \mid \Pi \cup \{x : E, y : A, z : B\} \vdash_k x = op(y,z) \cdot \mathcal{B} \ : \ D} \ .$$

To load and store registers, the following exchange rules are introduced.

$$(\mathbf{rs}\text{-load})\ \frac{\Sigma \mid \Pi \cup \{x : A\} \vdash_k \mathcal{B} : B}{\Sigma \cup \{x : A\} \mid \Pi \vdash_k \mathsf{Load}(x) \cdot \mathcal{B} : B}$$

$$(\mathbf{rs}\text{-store})\ \frac{\Sigma \cup \{x : A\} \mid \Pi \vdash_k \mathcal{B} : B}{\Sigma \mid \Pi \cup \{x : A\} \vdash_k \mathsf{Store}(x) : B}\quad (\text{if } |\Pi \cup \{x : A\}| \leq k)$$

By incorporating these refined structural rules into the type system of the register transfer language, we obtain a type system that faithfully represents a machine language with a fixed number of registers. The resulting formalization is suitable for analyzing various properties of machine code, and can serve as a basis for designing type safe manipulation of machine code. As we have mentioned, this idea has been used to develop a register allocation algorithm [Ohori 2004] as a proof transformation. Since it only changes the treatment of structural rules, the preservation of static and dynamic semantics of code directly follows from the construction. For a conventional algorithm based on graph coloring, establishing those properties may be difficult.

8.3 Representing Jumps and Loops

So far, we have based our development on proof theoretical principles of intuitionistic logic, and have developed proof systems and the corresponding code languages for low-level machines. The resulting proof systems have the cut elimination property and the corresponding code languages have the property that the execution of any code block terminates. Although this property is desirable for proof theory, this formalism is clearly too weak for modeling actual machine code. To model a practical low-level language, we must extend our formalism with some mechanism to represent jumps and loops.

In an actual machine code, a program consists of a collection of labeled basic blocks and each block contains jump instructions to transfer control to some other block. We take the register transfer language we have defined as an example and extend it with a mechanism for jumps. We assume that there is a given set of labels (ranged over by l) and extend the syntax of instructions and code blocks as follows.

$$\begin{aligned} I &::= \cdots \mid \mathsf{Ifzero}\ (x)\ \mathsf{goto}\ l \\ \mathcal{B} &::= \mathsf{Return} \mid \mathsf{goto}\ l \mid I \cdot \mathcal{B} \end{aligned}$$

$\mathsf{Ifzero}(x)\ \mathsf{goto}\ l$ tests whether the value of x is 0 or not and if it is 0 then jumps to the block labeled l, otherwise continues the execution of the following instruction. $\mathsf{goto}\ l$ is unconditional jump.

A typing judgment is refined to the following form

$$\mathcal{L}\,;\, \Gamma \vdash \mathcal{B} : A$$

indicating the fact that $\mathcal{B}$ is a code block computing a value of A from the environment of type Γ *possibly using existing blocks described in* $\mathcal{L}$. $\mathcal{L}$ is a function of the form

$$\mathcal{L} = \{l_1 : \Gamma_1 \vdash A_1, \ldots, l_n : \Gamma_n \vdash A_n\}$$

from a finite set of labels to logical sequents. The rules for goto and Ifzero are defined as follows.

$$(\mathbf{s}\text{-ref})\quad \mathcal{L}\,;\,\Gamma \vdash \mathsf{Goto}(l)\ :\ A \quad (\text{if } \mathcal{L}(l) = \Gamma \vdash A)$$

$$(\mathbf{s}\text{-if})\quad \frac{\mathcal{L}\,;\,\Gamma \cup \{x : Int\} \vdash \mathcal{B}\ :\ A}{\mathcal{L}\,;\,\Gamma \cup \{x : Int\} \vdash \mathsf{Ifzero}(x)\ \mathsf{goto}\ l \cdot \mathcal{B}\ :\ A} \quad (\text{if } \mathcal{L}(l) = \Gamma \cup \{x : Int\} \vdash A)$$

All the other rules do not use $\mathcal{L}$ and are mechanically obtained from the corresponding rules by adding $\mathcal{L}$ in each sequent.

A top-level program $\mathcal{P}$ is a labeled set of code blocks of the form

$$\mathcal{P} = \{l_1 = \mathcal{B}_1, \ldots, l_n = \mathcal{B}_n\}\ .$$

The typing rule for a top-level program is defined as follows.

$$\frac{\{l_1 : \Gamma_1 \vdash A_1, \ldots, l_n : \Gamma_n \vdash A_n\}\,;\,\Gamma_i \vdash \mathcal{B}_i\ :\ A_i \qquad (1 \le i \le n)}{\vdash \{l_1 = \mathcal{B}_1, \ldots, l_n = \mathcal{B}_n\} : \{l_1 : \Gamma_1 \vdash A_1, \ldots, l_n : \Gamma_n \vdash A_n\}}$$

It is this top-level rule that creates loops. Each basic block still corresponds to a intuitionistic proof under the interpretation that the additional assumption $\mathcal{L}$ indicates existence of proofs. This allows us to show its type soundness property similarly to the code languages we have developed so far. The situation is analogous to the relationship between a typed functional language with recursive function definitions and the typed lambda calculus.

9. DISCUSSIONS ON APPLICATIONS AND EXTENSIONS

This study is the first step toward a proof theory for machine code, and the presented formalism contains some limitations. Also, a number of interesting issues on possible applications and extensions remain to be investigated. In this section, we review some of them and suggest further work.

—*Compilation by Proof Transformation.*
The general motivation of this work is not merely to understand the correspondence between proof theory and low-level code, but also to provide a basis for robust compilation and systematic code analysis.
In our development, we have used the natural deduction system **N** as a given proof system and have shown that any proof in **N** can be transformed to a proof in **S**. From this result, we have obtained a type-preserving compilation algorithm from the lambda calculus to the logical abstract machine code. However, an actual implementation of a functional language performs much finer translation consisting of a number of compilation steps. One approach to developing a practical compilation algorithm based on our logical framework would be to define each of intermediate languages as a proof system and to give a proof transformation for each compilation step. Development of a series of proof transformation steps that cover the entire process of an actual compiler is a challenging future work. One relevant result toward this direction is a logical interpretation of A-normal transformation [Flanagan et al. 1993]. In [Ohori 1999], the author shows that translating the lambda calculus into A-normal forms corresponds to transforming the natural deduction system into Kleene's G3 proof system. Since the sequential sequent calculus is closer to G3 system than the natural deduction system,

we believe that a proof transformation from G3 system to **S** can be developed yielding a finer compilation step. From a more general perspective, such a proof transformation process would complement the recent study on type-preserving compilation such as [Morrisett et al. 1998] and [Minamide et al. 1996] by giving additional insights beyond the preservation of typing. The possibility of reverse transformation (decompilation) shown in this paper is one such example. Another is the observation made in [Ohori 2004] about the relationship between various forms of structural rules and register manipulation.

—*Analysis of Low-level Code.*
Another area of application of the logical framework developed in this paper is low-level code analysis. The general idea underlying the logical abstract machine developed in this paper can be applied to various other code languages, yielding their static type systems. As an example, in [Higuchi and Ohori 2002], a type system for the Java bytecode language has been developed by extending the stack-based logical abstract machine with jumps and the primitives to manipulate objects. This would provide an alternative basis for bytecode verification. Different from the existing type systems such as [Stata and Abadi 1998; Freund and Mitchell 2003], the type system based on our logical interpretation not only checks type consistency but it also represents constructive meaning of code in the sense of Curry-Howard correspondence, as the type system of the lambda calculus does. This property allows one to apply results shown in this paper to those type systems. For example, in [Katsumata and Ohori 2001], it has been shown that a de-compilation algorithm from the Java bytecode to the lambda term with recursion can be constructed. Since the underlying constructive interpretation is similar to the one that underlies the typed lambda calculus, this approach would also allow one to transfer some results of type-based analysis for the lambda calculus to low-level code languages. Recently, it has been shown in [Higuchi and Ohori 2003] that the static verification of access control for the lambda calculus by Skalka and Smith [2000] can be transfered to the Java bytecode language by refining the type system of [Higuchi and Ohori 2002].

—*Various language extensions.*
We have shown that our logical approach can be used to represent several code languages, including a code language with fixed number of registers. However, the current formalism lacks several features of actual machine code languages.

Among them, a particularly important one in practice is mutable memory. There does not seem to exist any satisfactory attempt to establish a Curry-Howard isomorphism for a language with mutable memory cells. One possibility would be to combine our proof system for low-level code with the idea underlying the logic for "bunched implication" [O'Hearn and Pym 1999]. One strategy would be to extend a sequent of the form $\Delta \vdash_c A$ to $\Theta; \Delta \vdash_c A$ where Θ is a bunched environment representing mutable memory.

Another direction toward extending our logical approach is to enhance the underlying logic. Our development has been based entirely on intuitionistic propositional logic. By extending the underlying logic, it should be possible to incorporate various language features such as control structures through classical logic

[Griffin 1990] and resource management through sub-structural logic [Ono and Komori 1985] and linear logic [Girard 1987].

10. CONCLUSIONS

We have developed a proof theory for low-level code languages. We have defined a proof system, which we have referred to as the *sequential sequent calculus*, and have establish the Curry-Howard correspondence between this proof system and a low-level code language by showing the following properties: (1) the set of proofs and the set of typed codes is in one-to-one correspondence, (2) the operational semantics of the code language is directly derived from the cut elimination procedure of the proof system, and (3) compilation and de-compilation algorithms between the code language and the typed lambda calculus are extracted from the proof transformations between the sequential sequent calculus and the natural deduction system. We have then shown that a bytecode language, a register transfer language, and a machine code with a finite number of registers are represented as typed term calculi.

With the extensions and further works we have discussed in the previous section, we believe the logical framework presented in this paper can serve as a basis for robust and efficient compilation and for static code analysis.

Acknowledgments

The author would like to thank Shin'ya Katsumata and Masahito Hasegawa for helpful discussion on the relationship between logic and low-level machine and on cut elimination. The author also thanks Yukiyoshi Kameyama for pointing out the existence of Raffalli's paper after the presentation of the preliminary result of this work at FLOPS '99 conference. The author thanks anonymous referees for their through and careful reading and for numerous helpful comments that were very useful for improving the paper. The author is particularly grateful to one of the referees who pointed out an error in the proof of the cut elimination proof in an earlier version of the paper.

REFERENCES

Abramsky, S. 1993. Computational interpretation of linear logic. *Theoretical Computer Science 3,* 57, 3–57.

Cousineau, G., Curien, P.-L., and Mauny, M. 1987. The categorical abstract machine. *Science of Computer Programming 8,* 2, 173–202.

Curry, H. B. and Feys, R. 1968. *Combinatory Logic.* Vol. 1. North-Holland, Amsterdam.

Flanagan, C., Sabry, A., Duba, B., and Felleisen, M. 1993. The essence of compiling with continuation. In *Proc. ACM PLDI Conference.* ACM, New York, 237–247.

Freund, S. and Mitchell, J. 2003. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning 30,* 3–4, 271–321.

Gallier, J. 1993. Constructive logics part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science 110*, 249–339.

Gentzen, G. 1969. Investigation into logical deduction. In *The Collected Papers of Gerhard Gentzen*, M. Szabo, Ed. North-Holland, Amsterdam.

Girard, J., Lafont, Y., and Taylor, P. 1989. *Proofs and Types.* Cambridge University Press, Cambridge, U.K.

Girard, J.-Y. 1987. Linear logic. *Theoretical Computer Science 50,* 1, 1–102.

Griffin, T. 1990. A formulae-as-types notion of control. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM, New York, 47–58.

Higuchi, T. and Ohori, A. 2002. Java bytecode as a typed term calculus. In *Proceedings of ACM Conference on Principles and Practice of Declarative Programming*. ACM, New York, 201–211.

Higuchi, T. and Ohori, A. 2003. A static type system for JVM access control. In *Proc. ACM International Conference on Functional Programming*. ACM, New York. An extended version submitted for publication, 2004.

Howard, W. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, London, 476–490.

Katsumata, S. and Ohori, A. 2001. Proof-directed de-compilation of low-level code. In *Proceedings of European Symposium on Programming*. Lecture Notes in Computer Science, vol. 2028. Springer-Verlag, Berlin, 352–366.

Kleene, S. 1952. *Introduction to Metamathematics*. North-Holland, Amsterdam. 7th edition.

Lambek, J. 1980. From λ-calculus to cartesian closed categories. In *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, London, 375–402.

Landin, P. J. 1964. The mechanical evaluation of expressions. *Computer Journal 6*, 308–320.

Minamide, Y., Morrisett, J. G., and Harper, R. 1996. Typed closure conversion. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM, New York, 271–283.

Mitchell, J. 1996. *Foundations for Programming Languages*. MIT Press, Boston, MA.

Morrisett, G., Crary, K., Glew, N., and Walker, D. 1998. Stack-based typed assembly language. In *Proceedings of International Workshop on Types in Compilation*. Lecture Notes in Computer Science, vol. 1473. Springer-Verlag, Berlin, 28–52.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1998. From system F to typed assembly language. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM, New York, 85–7.

O'Hearn, P. and Pym, D. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic 5,* 2, 215–244.

Ohori, A. 1999. A Curry-Howard isomorphism for compilation and program execution. In *Proceedings of Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science, vol. 1581. Springer-Verlag, Berlin, 258–179.

Ohori, A. 2004. Register allocation by proof transformation. *Journal of Science of Computer Programming 50,* 1-3, 161 – 187.

Ono, H. and Komori, Y. 1985. Logics without the contraction rule. *Journal of Symbolic Logic 50,* 1, 169–201.

Parigot, M. 1992. $\lambda\mu$-calculus: an alorithmic interpretation of classical natural deduction. In *Proceedings of Logic Programming and Automated Reasoning*. Lecture Notes in Computer Science, vol. 624. Springer-Verlag, Berlin, 190–201.

Raffalli, C. 1994. Machine deduction. In *Proceedings of Types for Proofs and Program*. Lecture Notes in Computer Science, vol. 806. Springer-Verlag, Berlin, 333–351.

Skalka, S. and Smith, S. 2000. Static enforcement of security with types. In *Proceedings of International Conference on Functional Programming (ICFP)*. ACM, New York, 34–45.

Stata, R. and Abadi, M. 1998. A type system for Java bytecode subroutines. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM, New York, 149–160.

Tait, W. 1966. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic 32,* 2, 198–212.

Turner, D. 1979. A new implementation technique for applicative languages. *Software Practice and Experience 9*, 31–49.

Wadler, P. 1990. Linear types can change the world! In *Progarmming Concepts and Methods*, M. Broy and C. Jones, Eds. IFIP TC 2 Working Conference. North Holland, Sea of Galilee, Israel, 561–581.