Proof-Directed De-compilation of Low-Level Code* **

Shin-ya Katsumata*** [1] and Atsushi Ohori† [2]

[1] Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, UK; sxk@dcs.ed.ac.uk
[2] School of Information Science, Japan Advanced Institute of Science and Technology, Tatsunokuchi, Ishikawa 923-1292, JAPAN; ohori@jaist.ac.jp

Abstract. We present a proof theoretical method for de-compiling low-level code to the typed lambda calculus. We first define a proof system for a low-level code language based on the idea of Curry-Howard isomorphism. This allows us to regard an executable code as a proof in intuitionistic propositional logic. As being a proof of intuitionistic logic, it can be translated to an equivalent proof of natural deduction proof system. This property yields an algorithm to translate a given code into a lambda term. Moreover, the resulting lambda term is not a trivial encoding of a sequence of primitive instructions, but reflects the behavior of the given program. This process therefore serves as *proof-directed de-compilation* of a low-level code language to a high-level language. We carry out this development for a subset of Java Virtual Machine instructions including most of its features such as jumps, object creation and method invocation. The proof-directed de-compilation algorithm has been implemented, which demonstrates the feasibility of our approach.

1 Introduction

The ability to analyze compiled code before its execution is becoming increasingly important due to recently emerging network computing, where pieces of executable code are dynamically exchanged over the Internet and used under the user's own privileges. In such circumstances, it is a legitimate desire to verify that a foreign code satisfies some desired properties before it is executed. This problem has recently attracted the attention of programming language researchers. One notable approach toward verification of properties of compiled code is to construct a formal proof of certain desired properties of a code using a

* Published in Proceedings of Proceedings of European Symposium on Programming, Springer Lecture Notes in Computer Science 2028, pages 352–366. 2001.
** A part of this work was done while both authors were at RIMS, Kyoto University.
*** Shin-ya Katsumata's research was supported in part by a scholarship provided by the Laboratory for Foundations of Computer Science, University of Edinburgh.
† Atsushi Ohori's work was partially supported by the Japanese Ministry of Education Grant-in-Aid for Scientific Research No. 12680345.

theorem prover, and to package the code with its proof to form a *proof-carrying code* [8]. The user of the code can then check the attached proof against the code to ensure that the code satisfies the properties. Another approach is to develop a static type system for compiled code [1, 6, 7, 13]. By checking the type consistency of a code, the user can ensure that the code does not cause any run-time type errors.

Both of these are effective in verifying certain predetermined safety properties. In many cases, however, the user would like to know the actual behavior of a foreign code for ensuring that the code correctly realizes its expected functionality. Moreover, analysis of the exact behavior of a code would open up a new possibility of network computing, where foreign code can be dynamically analyzed and adapted or optimized to suit the user's environment.

An executable code is a large sequence of instructions, which can be rather hard for humans to understand. However this does not mean, at least in principle, that a code is incomprehensible. A properly compiled code of a correct program is a consistent syntactic object that can be interpreted by a machine to perform the function denoted by the original program. This fact suggests that it should be possible to develop a systematic method to extract the logical structure of a code and present it in a high-level language.

The key to developing a code analysis system is the Curry-Howard isomorphism for machine code presented in [11]. In this paradigm, a code language corresponds to a variant of the sequent calculus of intuitionistic propositional logic, called the *sequential sequent calculus*. Being a proof system of the logic, it can be translated to and from other languages corresponding to proof systems of intuitionistic logic. Compilation is one instance, which translates natural deduction to the sequential sequent calculus. It is shown that the converse is also possible, leading to the proof-directed *de-compilation* of machine code. In the next section, we outline our approach based on the Curry-Howard isomorphism.

The purpose of the present paper is to show that this general idea can be used to develop a de-compiler for a low-level code language. Our de-compilation method is most naturally applicable to lambda calculus. We also believe that the general principle of the method can be applicable to a wide range of low-level code languages. To demonstrate the practical feasibility of our method we carry out the development for the Java bytecode language [4], which is the target language of the Java programming language [3]. This language provides several practically useful features such as objects and methods at the bytecode level, for which there is no obvious Curry-Howard isomorphism. The language with these features is a good touchstone for the scalability of our approach to practical extra-logical structures.

We first give, in Section 3, a term language called JAL^0 for a subset of Java Virtual Machine (JVM) assembly language including basic instructions and develop a de-compilation from JAL^0 to a PCF-like language. We show that this de-compilation algorithm preserves both the typing and the semantics of a given bytecode program. We then give an extention, called JAL, of JAL^0 with objects and methods in Section 4. Based on these results, we have implemented a proof-directed de-compiler for the Java bytecode language supporting most of

An input to the de-compiler.

```
.method public
    static fact(I)I
  iconst_1
  istore_1
  goto L12
L5:
  iload_1
  iload_0
  imul
  istore_1
  iinc 0 -1
L12:
  iload_0
  iconst_1
  if_icmpgt L5
  iload_1
  ireturn
.end method
```

Output of the de-compiler.

```
fact(e0) =
   L12(e0, 1)

L12(e0, e1) =
   if(1 < e0) then
      L12(e0 - 1, e1 * e0)
   else
      return e1
```

Fig. 1. An example of the proof-directed de-compilation

its features. Figure 1 shows an actual output of our decompiler. The input JVM bytecode is the result of compiling the following Java program.

```
public static int fact (int n) { int m;
   for(m = 1; n > 1; --n)
      m = m * n;
   return m; }
```

As seen in this example, our de-compiler correctly recovers the factorial program as a tail recursive function without using any knowledge other than the given bytecode. Section 5 describes our prototype de-compiler. Section 6 compares our approach with related work, and Section 7 concludes the paper.

Limitations of space make it difficult to cover the de-compilation method fully; the authors intend to present a more detailed description in another paper.

Conventions and Notations: We count the elements of a list from the left starting with 0. We write $|l|$ for the length of the list l, and write l, e for the list obtained from l by adding the element e at the end of l. We write $\{d_1 : e_1, \cdots, d_n : e_n\}$ for the function which maps d_j to e_j $(1 \leq j \leq n)$, and write $f\{d : e\}$ for the function f' such that $\mathrm{dom}(f') = \mathrm{dom}(f) \cup \{d\}$, $f'(d) = e$ and $f'(x) = f(x)$ for any $x \neq d$.

2 Logical Approach to Code Analysis

In this section, we describe Curry-Howard isomorphism for machine code presented in [11] and outline the proof-directed de-compilation.

We let Δ range over lists of formula representing an assumption set of a logical sequent. The basic observation underlying the logical interpretation of machine code is to consider each instruction I as a logical inference step of the form $\dfrac{\Delta' \rhd B : \tau}{\Delta \rhd I; B : \tau}$ similar to a left rule in Gentzen's sequent calculus. Regarding the assumption set as a description of machine memory (stack), an inference rule of this form represents a primitive machine instruction that transforms a memory state Δ to that of Δ'. The conclusion τ of the judgement $\Delta \rhd B : \tau$ is the type of the value returned by the code B.

A bytecode language corresponds to a proof system consisting of this form of rules, called a sequential sequent calculus, which has the same deducibility as

$$\frac{}{\Delta, \tau \rhd \mathsf{return} : \tau} \qquad \frac{\Delta, \Delta_i \rhd B : \tau \quad 0 \leq i < |\Delta|}{\Delta \rhd \mathsf{acc}\ i; B : \tau} \qquad \frac{\Delta, \mathrm{int} \rhd B : \tau}{\Delta \rhd \mathsf{iconst}\ n; B : \tau}$$

$$\frac{\Delta, \tau' \times \tau'' \rhd B : \tau}{\Delta, \tau', \tau'' \rhd \mathsf{pair}; B : \tau} \qquad \frac{\Delta, \tau' \rhd B : \tau}{\Delta, \tau' \times \tau'' \rhd \mathsf{fst}; B : \tau} \qquad \frac{\Delta, \tau'' \rhd B : \tau}{\Delta, \tau' \times \tau'' \rhd \mathsf{snd}; B : \tau}$$

$$\frac{\Delta, (\Delta_0 \Rightarrow \tau') \rhd B : \tau}{\Delta \rhd \mathsf{code}\ B_0; B : \tau}\ (\text{if } \Delta_0 \rhd B_0 : \tau') \qquad \frac{\Delta, \tau' \rhd B : \tau}{\Delta, (\Delta_0 \Rightarrow \tau'), \Delta_0 \rhd \mathsf{call}\ n; B : \tau}\ (|\Delta_0| = n)$$

Fig. 2. A sequential sequent calculus for a simple bytecode language

intuitionistic propositional logic. One important implication of this result is that a bytecode language can be translated to and from other proof systems of intuitionistic propositional logic. Compilation of lambda terms can be regarded as a proof transformation from natural deduction (i.e. typed lambda calculus) into this proof system. Moreover, it is shown that the converse is also possible. The lambda term obtained through the inverse transformation exhibits the logical structure of the code. This process can therefore be regarded as *de-compilation*. Below, we outline this process using a simple bytecode language.

The set of types (ranged over by τ), *instructions* (ranged over by I) and *code blocks* (ranged over by B) of the bytecode language are given below.

$$\begin{aligned} \tau &::= \mathrm{int} \mid \tau \times \tau \mid \tau, \cdots, \tau \Rightarrow \tau \\ B &::= \mathsf{return} \mid I; B \\ I &::= \mathsf{iconst}\ n \mid \mathsf{acc}\ i \mid \mathsf{pair} \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{code}\ B \mid \mathsf{call}\ n \end{aligned}$$

The type $\tau_1, \cdots, \tau_n \Rightarrow \tau$ is the type of functions which map lists of values of type $\tau_1, \cdots, \tau_n$ to values of type τ. Instructions $\mathsf{iconst}\ n$, $\mathsf{acc}\ i$, and $\mathsf{code}\ B$ push onto the stack the integer value n, the ith element of the stack, and the pointer to the code block B, respectively. pair constructs a pair on the stack, and fst and snd take the first and second element of a pair on the stack. $\mathsf{call}\ n$ pops n elements and a code pointer off the stack, and calls the code with the n arguments.

We consider a list Δ of types as a type of a machine stack with the convention that the right-most formula in a list corresponds to the top of the stack, and interpret a block of this bytecode language as a proof of the sequential sequent calculus. The term assignment system for the sequential sequent calculus is given in Figure 2.

The intended semantics of each instruction should be understood by reading the corresponding proof rule "backward". For example, pair changes the stack state from Δ, τ, τ' to $\Delta, \tau \times \tau'$ indicating the operation that replaces the top-most 2 elements with their product.

We consider the following typed lambda calculus as the target of de-compilation.

$$\begin{aligned} \tau &::= \mathrm{int} \mid \tau \times \tau \mid \tau \to \tau \\ M &::= n \mid x \mid (M, M) \mid \mathsf{fst}(M) \mid \mathsf{snd}(M) \mid \lambda x.M \mid M\ M \end{aligned}$$

$$
\begin{aligned}
[\![\Delta, \tau \rhd \mathsf{return} : \tau]\!] &= s_{|\Delta|} \\
[\![\Delta \rhd \mathsf{acc}\ i; B : \tau]\!] &= [s_i/s_{|\Delta|}][\![\Delta, \Delta_i \rhd B : \tau]\!] \\
[\![\Delta \rhd \mathsf{iconst}\ n; B : \tau]\!] &= [n/s_{|\Delta|}][\![\Delta, \mathrm{int} \rhd B : \tau]\!] \\
[\![\Delta, \tau', \tau'' \rhd \mathsf{pair}; B : \tau]\!] &= [(s_{|\Delta|}, s_{|\Delta|+1})/s_{|\Delta|}][\![\Delta, \tau' \times \tau'' \rhd B : \tau]\!] \\
[\![\Delta, \tau' \times \tau'' \rhd \mathsf{fst}; B : \tau]\!] &= [\mathsf{fst}(s_{|\Delta|})/s_{|\Delta|}][\![\Delta, \tau' \rhd B : \tau]\!] \\
[\![\Delta, \tau' \times \tau'' \rhd \mathsf{snd}; B : \tau]\!] &= [\mathsf{snd}(s_{|\Delta|})/s_{|\Delta|}][\![\Delta, \tau'' \rhd B : \tau]\!] \\
[\![\Delta \rhd \mathsf{code}\ B_0; B : \tau]\!] &= \\
&\quad [\lambda s_0 \cdots s_{|\Delta'|-1}.[\![\Delta' \rhd B' : \tau']\!]/s_{|\Delta|}][\![\Delta, (\Delta' \Rightarrow \tau') \rhd B : \tau]\!] \\
[\![\Delta, (\Delta' \Rightarrow \tau'), \Delta' \rhd \mathsf{call}\ n; B : \tau]\!] &= \\
&\quad [(s_{|\Delta|}\ s_{|\Delta|+1} \cdots\ s_{|\Delta|+n})/s_{|\Delta|}][\![\Delta, \tau' \rhd B : \tau]\!] \quad (n = |\Delta'|)
\end{aligned}
$$

Fig. 3. De-compilation algorithm for a simple bytecode language

Its type system is the standard one. We write $[M/x]N$ for the term obtained from N by substituting M for x (with necessary bound-variable renaming.)

We write $[\![\Delta \rhd B : \tau]\!]$ for the lambda term obtained by transforming the derivation $\Delta \rhd B : \tau$. The general idea behind the transformation is to assign a variable s_i to each element Δ_i in the assumption list Δ, and to proceed by induction on the derivation of the code. The algorithm translates an initial sequent to the variable corresponding to the top of the stack. For a compound proof, the algorithm first obtains the term corresponding to its sub-proof. It then applies the transformation corresponding to the first instruction to obtain the desired lambda term. The set of equations in Figure 3 defines the transformation.

3 JAL^0 : the JVM Assembly Language without Objects

Compared to the simple bytecode language considered above, Java bytecode has the following additional features.

1. *Restricted stack access and local variable support.* JVM does not include an instruction to access an arbitrary stack element. This restriction is compensated by JVM's support for directly accessible mutable local variables.
2. *Labels and jumps.* As in most existing computer architectures, JVM uses labels and jumps to realize control flow.
3. *Classes, objects and methods.* JVM has types and instructions to support object-oriented features.

Since the third feature requires significantly new machinery, we divide our development into two stages. In this section, we define a JVM assembly language without objects, denoted by JAL^0, supporting the first two features, and present its proof system and the proof-directed de-compilation algorithm for JAL^0. Then we state the semantic correctness of the de-compilation algorithm. Later, in Section 4, we describe the necessary extensions to objects and methods.

3.1 Syntax of JAL^0

With the introduction of labels and jumps, an executable program unit is no longer a sequence of instructions, but a collection of labelled blocks, mutually referenced through labels. We let l range over a given countably infinite set of *labels*, and i range over a given countably infinite set of *local variables*. We assume a fixed linear order on the set of local variables, and that any sequence of local variables mentioned in the following development is ordered by this relation.

The syntax of *program units* (ranged over by K), *blocks* (ranged over by B) and *instructions* (ranged over by I) of JAL^0 are given below.

$$\begin{aligned} K &::= \{l : B, \cdots, l : B\} \\ B &::= \mathsf{ireturn} \mid \mathsf{goto}\ l \mid I;B \\ I &::= \mathsf{iconst}\ n \mid \mathsf{pop} \mid \mathsf{dup} \mid \mathsf{swap} \mid \mathsf{iload}\ i \mid \mathsf{istore}\ i \mid \mathsf{iadd} \mid \mathsf{ifzero}\ l \end{aligned}$$

ireturn is for returning an integer value. goto l transfers control to the block labelled l. iconst n is the same as before. pop, dup, swap are the stack operations for popping the stack, for duplicating the top element, and for swapping the top two elements, respectively. iload i pushes the contents of the local variable i onto the stack. istore i pops the top value off the stack and stores it in the local variable i. iadd pops two integers off the stack and pushes back the sum of the two. ifzero l pops the top element off the stack, and transfers control to the code block labelled l if it is 0. A *JAL^0 program* is a program unit K with a distinguished *entry label* l, written $K.l$.

The following example is a JAL^0 program, which takes an integer input through local variable i_0 and returns 1 if it is 0, otherwise returns 0.

$$K_0.l \equiv \left\{ \begin{array}{l} l : \ \mathsf{iload}\ i_0; \mathsf{ifzero}\ l'; \mathsf{iconst}\ 0; \mathsf{goto}\ l'' \\ l' : \ \mathsf{iconst}\ 1; \mathsf{goto}\ l'' \\ l'' : \mathsf{ireturn} \end{array} \right\}.l$$

3.2 The Type System for JAL^0

Each JAL^0 instruction operates on the stack and the local variables, and is represented as an inference rule of the form $\dfrac{\Gamma; \Delta \rhd B : \tau}{\Gamma'; \Delta' \rhd I;B : \tau}$ where Γ is a *local variable context* which maps local variables to value types, Δ is a *stack context* which is a finite sequence of value types, and τ is the value type of the block B. In JAL^0 considered in this section, the only possible value type is "int" of integers. In Section 4, we extend JAL^0 to include object types.

Since blocks in general refer to other blocks through labels, the typing of a block is determined relative to an assumption on the types of blocks assigned to the labels. We define a *block type* to be a logical sequent of the form $\Gamma; \Delta \rhd \tau$ denoting possible blocks B such that $\Gamma; \Delta \rhd B : \tau$. We let Λ range over *label contexts*, which maps labels to block types.

Judgement forms and typing rules of JAL^0 are given in Figure 4. The definition of typing $\Lambda \rhd K$ of a program unit K is essentially the same as the typing

Judgements:

$\Lambda \mid \Gamma; \Delta \rhd B : \tau$	block B has block type $\Gamma; \Delta \rhd \tau$ under Λ
$\Lambda \rhd K$	program unit K is well-typed with Λ
$\Lambda \mid \Gamma; \Delta \rhd K.l : \tau$	the entry point l of a program $K.l$ has block type $\Gamma; \Delta \rhd \tau$ under Λ

Typing rules for blocks:

$$\frac{}{\Lambda \mid \Gamma; \Delta, \mathrm{int} \rhd \mathsf{ireturn} : \mathrm{int}} \quad \frac{\Lambda(l) = \Gamma; \Delta \rhd \tau}{\Lambda \mid \Gamma; \Delta \rhd \mathsf{goto}\ l : \tau} \quad \frac{\Lambda \mid \Gamma; \Delta \rhd B : \tau'}{\Lambda \mid \Gamma; \Delta, \tau \rhd \mathsf{pop}; B : \tau'}$$

$$\frac{\Lambda \mid \Gamma; \Delta, \tau, \tau \rhd B : \tau'}{\Lambda \mid \Gamma; \Delta, \tau \rhd \mathsf{dup}; B : \tau'} \quad \frac{\Lambda \mid \Gamma; \Delta, \tau', \tau \rhd B : \tau''}{\Lambda \mid \Gamma; \Delta, \tau, \tau' \rhd \mathsf{swap}; B : \tau''} \quad \frac{\Lambda \mid \Gamma; \Delta, \mathrm{int} \rhd B : \tau}{\Lambda \mid \Gamma; \Delta \rhd \mathsf{iconst}\ n; B : \tau}$$

$$\frac{\Lambda \mid \Gamma; \Delta, \mathrm{int} \rhd B : \tau \quad \Gamma(i) = \mathrm{int}}{\Lambda \mid \Gamma; \Delta \rhd \mathsf{iload}\ i; B : \tau} \quad \frac{\Gamma\{i : \mathrm{int}\}; \Delta \rhd B : \tau}{\Lambda \mid \Gamma; \Delta, \mathrm{int} \rhd \mathsf{istore}\ i; B : \tau}$$

$$\frac{\Lambda \mid \Gamma; \Delta, \mathrm{int} \rhd B : \tau}{\Lambda \mid \Gamma; \Delta, \mathrm{int}, \mathrm{int} \rhd \mathsf{iadd}; B : \tau} \quad \frac{\Lambda \mid \Gamma; \Delta \rhd B : \tau \quad \Lambda(l) = \Gamma; \Delta \rhd \tau}{\Lambda \mid \Gamma; \Delta, \mathrm{int} \rhd \mathsf{ifzero}\ l; B : \tau}$$

Typing of program units:

$$\frac{\forall l \in \mathrm{dom}(\Lambda). \Lambda \mid \Gamma; \Delta \rhd K(l) : \tau \quad \Lambda(l) = \Gamma; \Delta \rhd \tau}{\Lambda \rhd K}$$

Typing of programs:

$$\frac{\Lambda \rhd K \quad \Lambda(l) = \Gamma; \Delta \rhd \tau}{\Lambda \mid \Gamma; \Delta \rhd K.l : \tau}$$

Fig. 4. Type system of JAL^0

rule for recursive definitions in a functional language. The rules for pop, dup and swap correspond to logical rules for weakening, contraction and exchange, respectively. The rules for iload i and istore i can also be understood as structural rules across two assumptions Γ and Δ. Conditional branch and jump instructions are, as already mentioned, considered as rules referring to other blocks in a program unit. These rules require that the type of the referenced block has the same local variable context, stack context and return type as those of the reference point.

As an example, let Λ_0 be a label context

$$\Lambda_0 = \{l : \{i_0 : \mathrm{int}\}; \emptyset \rhd \mathrm{int}, l' : \{i_0 : \mathrm{int}\}; \emptyset \rhd \mathrm{int}, l'' : \{i_0 : \mathrm{int}\}; \mathrm{int} \rhd \mathrm{int}\}.$$

Then the program unit K_0 given in the previous subsection has the typing $\Lambda_0 \rhd K_0$ and therefore $\Lambda_0 \mid \{i_0 : \mathrm{int}\}; \emptyset \rhd K_0.l : \mathrm{int}$.

3.3 Operational Semantics of JAL^0 and the Type Soundness

The language JAL^0 is intended to model a subset of JVM bytecode. As such, we define its operational semantics by specifying the effect of each instruction on a machine state. A machine state is described by a triple $(E;\ S;\ B)$ of *a local variable environment* E which maps local variables to run-time values, a *stack* S which is a sequence of run-time values, and a *current block* B where the left-most instruction is the next one to be executed. For JAL^0, the possible run-time values (ranged over by v) are integers.

We write $(E;\ S;\ (I; B)) \longrightarrow_K (E';\ S';\ B')$ if the state $(E;\ S;\ (I; B))$ is changed to $(E';\ S';\ B')$ by the execution of I. Figure 5 gives the set of transition

$$
\begin{array}{llll}
(E;\ S;\ \mathsf{iload}\ i; B) & \longrightarrow_K (E;\ S, E(i);\ B) & (E;\ S, v;\ \mathsf{pop}; B) & \longrightarrow_K (E;\ S;\ B) \\
(E;\ S, v;\ \mathsf{dup}; B) & \longrightarrow_K (E;\ S, v, v;\ B) & (E;\ S;\ \mathsf{iconst}\ n; B) & \longrightarrow_K (E;\ S, n;\ B) \\
(E;\ S, v_1, v_2\ \mathsf{swap}; B) & \longrightarrow_K (E;\ S, v_2, v_1;\ B) & (E;\ S, v;\ \mathsf{ireturn}) & \longrightarrow_K (\emptyset;\ v;\ \emptyset) \\
(E;\ S, v;\ \mathsf{istore}\ i; B) & \longrightarrow_K (E\{i : v\};\ S;\ B) & (E;\ S;\ \mathsf{goto}\ l; B) & \longrightarrow_K (E;\ S;\ K(l)) \\
(E;\ S, n, m;\ \mathsf{iadd}; B) & \longrightarrow_K (E;\ S, n+m;\ B) & (E;\ S, 0;\ \mathsf{ifzero}\ l; B) & \longrightarrow_K (E;\ S;\ K(l)) \\
(E;\ S, n;\ \mathsf{ifzero}\ l; B) & \longrightarrow_K (E;\ S;\ B)\ \ (n \neq 0) & &
\end{array}
$$

Fig. 5. Operational semantics of JAL^0

rules. The reflexive transitive closure of $\longrightarrow_K$ is denoted by $\xrightarrow{*}_K$. A program $K.l$ computes a value v from an initial local variable environment E and a stack S, written $(E;\ S;\ K.l) \xrightarrow{*} v$, if $(E;\ S;\ K(l)) \xrightarrow{*}_K (\emptyset;\ v;\ \emptyset)$.

We show that the type system of JAL^0 is sound with respect to this operational semantics. We write $\models v : \tau$ if v has type τ, and define the typing relations for local variable environments and stacks as follows.

$$
\begin{aligned}
&E \models \Gamma \iff \mathrm{dom}(E) = \mathrm{dom}(\Gamma) \text{ and } \forall i \in \mathrm{dom}(E).\ \models E(i) : \Gamma(i) \\
&S \models \Delta \iff |S| = |\Delta| \text{ and } \forall 0 \leq i < |S|.\ \models S_i : \Delta_i
\end{aligned}
$$

Since JAL^0 only contains integers, the typing relation for values is the trivial relation between integers and int, but can easily be extended to other primitive types. We can then show the following.

Theorem 1. *If $E \models \Gamma, S \models \Delta$ and $\Lambda \mid \Gamma; \Delta \triangleright K.l : \tau$ then either $(E;\ S;\ K.l) \xrightarrow{*} v$ such that $\models v : \tau$ or the computation of a program $K.l$ under E, S does not terminate.*

3.4 Proof-Directed De-compilation of JAL^0

To develop a proof-directed de-compilation algorithm for JAL^0, we need to account for jump instructions and local variables. Our strategy is to translate a labelled block to a function from its contexts to its return type, and to translate a jump to a label as a tail call of the function corresponding to the label. Since basic blocks in a program may have jumps mutually calling the other blocks, we assume that the target language supports mutual recursion.

As we mentioned earlier, manipulation of local variables can be modelled by structural rules across a local variable context and a stack context. Their mutability is reduced to introducing a new binding each time a value is stored to a variable, and therefore no additional mechanism is required.

Our target language is the following PCF-like language, which we call λ^{rec}:

$$
\begin{aligned}
\tau &::= \mathrm{int} \mid \tau \to \tau \\
M &::= n \mid x \mid \lambda x.M \mid M\ M \\
&\quad \mid \mathsf{ifzero}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid \mathsf{iadd}\ (M, N) \mid \mathsf{rec}\ \{x = M, \cdots, x = M\}\ \mathsf{in}\ x
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma; \Delta, \mathrm{int} \rhd \mathsf{ireturn} : \mathrm{int}]\!] &= s_{|\Delta|} \\
[\![\Gamma; \Delta, \tau \rhd \mathsf{pop}; B : \tau]\!] &= [\![\Gamma; \Delta \rhd B : \tau]\!] \\
[\![\Gamma; \Delta, \tau \rhd \mathsf{dup}; B : \tau]\!] &= [s_{|\Delta|}/s_{|\Delta|+1}][\![\Gamma; \Delta, \tau, \tau \rhd B : \tau]\!] \\
[\![\Gamma; \Delta, \tau, \tau' \rhd \mathsf{swap}; B : \tau]\!] &= [s_{|\Delta|+1}/s_{|\Delta|}, s_{|\Delta|}/s_{|\Delta|+1}][\![\Gamma; \Delta, \tau', \tau \rhd B : \tau]\!] \\
[\![\Gamma; \Delta \rhd \mathsf{iconst}\ \mathsf{n}; B : \tau]\!] &= [n/s_{|\Delta|}][\![\Gamma; \Delta, \mathrm{int} \rhd B : \tau]\!] \\
[\![\Gamma; \Delta \rhd \mathsf{iload}\ i; B : \tau]\!] &= [i/s_{|\Delta|}][\![\Gamma; \Delta, \mathrm{int} \rhd B : \tau]\!] \\
[\![\Gamma; \Delta, \mathrm{int} \rhd \mathsf{istore}\ i; B : \tau]\!] &= [s_{|\Delta|}/i][\![\Gamma\{i : \mathrm{int}\}; \Delta \rhd B : \tau]\!] \\
[\![\Gamma; \Delta, \mathrm{int} \rhd \mathsf{ifzero}\ l; B : \tau]\!] &= \mathsf{ifzero}(s_{|\Delta|}, \mathrm{apply}(l, \Gamma, \Delta), [\![\Gamma; \Delta \rhd B : \tau]\!]) \\
[\![\Gamma; \Delta, \mathrm{int}, \mathrm{int} \rhd \mathsf{iadd}; B : \tau]\!] &= [\mathsf{iadd}\ (s_{|\Delta|}, s_{|\Delta|+1})/s_{|\Delta|}][\![\Gamma; \Delta, \mathrm{int} \rhd B : \tau]\!] \\
[\![\Gamma; \Delta \rhd \mathsf{goto}\ l : \tau]\!] &= \mathrm{apply}(l, \Gamma, \Delta)
\end{aligned}
$$

Fig. 6. De-compilation algorithm for JAL^0 blocks

$\mathsf{rec}\ \{x_1 = M_1, \cdots, x_n = M_n\}\ \mathsf{in}\ x_j$ denotes the term M_j with mutually recursive definitions, where $x_1, \ldots, x_n$ may appear in $M_1, \ldots, M_n$. The type system of λ^{rec} is standard. We write $\Gamma \rhd M : \tau$ if M has type τ under context Γ.

To present our de-compilation algorithm, we introduce some definitions and notations. We assume that the set of variables of λ^{rec} is the disjoint union of the set of labels, the set of local variables, and a given countably infinite set of *stack variables* indexed with natural numbers. We write s_i for the stack variable of index i. We define the application of all variables in the context $\Gamma; \Delta$ to the label l, written $\mathrm{apply}(l, \Gamma, \Delta)$, as the term $l\ i_0 \cdots i_n\ s_0 \cdots s_{|\Delta|-1}$ for which $\mathrm{dom}(\Gamma) = \{i_0, \cdots, i_n\}$. The de-compilation algorithm for blocks is given by the equations in Figure 6.

We turn to de-compilation of JAL^0 programs. First we define a *closure* of a basic block, written $Cls(\Gamma; \Delta \rhd B : \tau)$, as the λ^{rec} term $\lambda i_0 \cdots i_n s_0 \cdots s_{|\Delta|-1}.[\![\Gamma; \Delta \rhd B : \tau]\!]$. Let $\Lambda = \{l_1 : \Gamma_1; \Delta_1 \rhd \tau_1, \ldots, l_n : \Gamma_n; \Delta_n \rhd \tau_n\}$ and K be a JAL^0 program unit such that $\Lambda \rhd K$. The transformation of a program $\Lambda \mid \Gamma_j; \Delta_j \rhd K.l_j : \tau_j$ is given as follows:

$$
\begin{aligned}
[\![\Lambda \mid \Gamma_j; \Delta_j \rhd K.l_j : \tau_j]\!] = \mathsf{rec}\ \{&l_1 = Cls(\Gamma_1; \Delta_1 \rhd K(l_1) : \tau_1), \cdots, \\
&l_n = Cls(\Gamma_n; \Delta_n \rhd K(l_n) : \tau_n)\}\ \mathsf{in}\ l_j.
\end{aligned}
$$

The above de-compilation algorithm is a proof transformation and as such it preserves types. We establish this statement via the following translation of block types in JAL^0 to types in λ^{rec}.

$$
\overline{\Gamma; \Delta \rhd \tau} = \Gamma(i_0) \to \cdots \to \Gamma(i_n) \to \Delta_0 \to \cdots \to \Delta_{|\Delta|-1} \to \tau
$$

Theorem 2. *If $\Lambda \mid \Gamma; \Delta \rhd K.l : \tau$ then the judgement $\emptyset \rhd [\![\Lambda \mid \Gamma; \Delta \rhd K.l : \tau]\!] : \overline{\Gamma; \Delta \rhd \tau}$ is derivable in λ^{rec}.*

We apply the above transformation to the example $K_0.l$:

$$[\![\Lambda_0 \mid \{i_0 : \mathrm{int}\}; \emptyset \rhd K_0.l : \mathrm{int}]\!] = \mathsf{rec} \left\{ \begin{array}{ll} l & = \lambda i_0.\mathsf{ifzero}\ (i_0,\ l'\ i_0,\ l''\ i_0\ 0) \\ l' & = \lambda i_0.l''\ i_0\ 1 \\ l'' & = \lambda i_0 s_0.s_0 \end{array} \right\} \mathsf{in}\ l.$$

3.5 Correctness of the De-compilation

From Theorem 2, one can see that our de-compilation algorithm is a proof transformation preserving typing. We provide further evidence of the correctness of our de-compilation algorithm by establishing that the algorithm also preserves semantics of JAL^0 programs.

An operational semantics of λ^{rec} is defined in the same way as the standard semantics of call-by-value PCF language. The semantics is determined by the evaluation relation $M \Downarrow W$ which says that the term M evaluates to the value W. A value W is either a natural number n or a term of the form $\lambda x.M$ (i.e. closure). Here we only show the evaluation rule for the mutual recursion operator:

$$\frac{[\cdots \mathsf{rec}\ \{x_1 = M_1, \cdots, x_n = M_n\}\ \mathsf{in}\ x_i/x_i \cdots]M_j \Downarrow W}{\mathsf{rec}\ \{x_1 = M_1, \cdots, x_n = M_n\}\ \mathsf{in}\ x_j \Downarrow W}\ .$$

The other evaluation rules are standard.

We establish the preservation of semantics via the following interpretation of virtual machine states as term substitutions in λ^{rec}. Let Λ be a label context, K be a program unit, E be a local variable environment and S be a stack. The interpretation $\overline{\Lambda \rhd K; E; S}$ of a machine state $\Lambda \rhd K; E; S$ is the term substitution defined by the following partial function from variables to λ^{rec} terms.

$$\overline{\Lambda \rhd K; E; S}(x) = \left\{ \begin{array}{ll} [\![\Lambda \mid \Gamma; \Delta \rhd K.x : \tau]\!] & x \in \mathrm{dom}(K) \wedge \Lambda(x) = \Gamma; \Delta \rhd \tau \\ E(x) & x \in \mathrm{dom}(E) \\ S_j & x \equiv s_j \wedge 0 \leq j < |S| \end{array} \right.$$

Then the preservation of semantics is stated as the following theorem.

Theorem 3. *If* $\Lambda \rhd K$, $\Lambda \mid \Gamma; \Delta \rhd B : \tau$, $E \models \Gamma$, $S \models \Delta$ *and* $(E;\ S;\ B) \xrightarrow{*}_K (\emptyset;\ V;\ \emptyset)$, *then* $\overline{\Lambda \rhd K; E; S}([\![\Lambda \mid \Gamma; \Delta \rhd B : \tau]\!]) \Downarrow V$.

4 Bytecode with Objects and Methods

This section develops the framework for proof-directed de-compilation with object oriented features. We concentrate on the basic mechanism for creating objects and invoking their methods, and leave the other object-oriented features to future research. This development requires us to extend both JAL^0 and the target language λ^{rec}. We extend both of them by adding primitives having the same functionality for object manipulation. In this approach the de-compiler just sends these primitives from the source to the target language. Another approach is to extend the target language with rich types so that the de-compiler can give an encoding of objects. However, we do not adopt this approach, because giving

such an encoding is usually associated with compilation process, which is the inverse of de-compilation.

There still remains one complication in this basic model, which is related to object initialization. As observed by Freund and Mitchell [1], a straightforward formulation of a type system for Java bytecode language is unsound due to the possibility of accessing uninitialized objects. Their solution is to distinguish types of initialized objects from those of uninitialized ones by indexing the type of an uninitialized object with the invocation of the corresponding object creation method. Although their type system is based on the one by Stata and Abadi, this mechanism has sufficient generality that it can be adopted to our framework.

4.1 JAL: JAL^0 with Objects and Classes

We give an extention, called *JAL*, of JAL^0 with object-oriented features. We assume there is a countably infinite set of *class identifiers*(ranged over by c) and a countably infinite set of *object indexes*(ranged over by u). An object index indicates the invocation point of the object creation method. The syntax of types is extended with class identifiers as follows.

$$\kappa ::= c \mid c_u \qquad \tau ::= \mathsf{int} \mid \kappa$$

The type c_u is for a reference to an uninitialized object created at the point u.

We let f and m range over the set of *field names* and the set of *method names*, respectively. We define a *class structure* as a pair of the form $(\{f_1 : \tau_1, \cdots, f_n : \tau_n\}, \{m_1 : \Gamma_1; \emptyset \rhd \tau_1, \cdots m_n : \Gamma_n; \emptyset \rhd \tau_n\})$. A class structure specifies the types of fields and methods in a class. We regard class structures as overloaded functions for field names and method names. We define a *class context*, ranged over by $\mathcal{C}$, as a map from class identifiers to class structures.

The set of instructions are extended with the following new instructions.

$$\mathsf{areturn} \quad \mathsf{aload}\ i \quad \mathsf{astore}\ i \quad \mathsf{new}\ c \quad \mathsf{init}\ c \quad \mathsf{invoke}\ c, m \quad \mathsf{getfield}\ c, f \quad \mathsf{putfield}\ c, f$$

areturn, aload and astore have analogous behavior on object references as the corresponding ones on integers. new c creates an uninitialized object instance of class c and pushes its reference onto the stack. init c pops an uninitialized object reference off the stack and initializes it by replacing each c with c_u. Practically, this instruction corresponds to invoking initialization method of uninitialized object. We omit its arguments for simplicity. invoke c, m invokes a method m on an object of class c by popping m arguments and an object off the stack and transferring control to the method code m of the object. The return value of the method is pushed onto the stack. getfield c, f and setfield c, f reads and writes field f of instance objects of class c respectively. invoke, getfield and setfield instructions fail if they operate on uninitialized object instances.

Judgement forms and typing rules of JAL are given in Figure 7. Typing rules for instructions not included in the figure are the same as those in JAL^0. In the rule for init, $[c/c_u]\Gamma$ or $[c/c_u]\Delta$ is obtained from Γ or Δ by substituting c for each occurrence of c_u respectively. The mechanism for type safe object initialization realized by the rules for new and init is the adaptation of that of [1].

Judgements:

$\mathcal{C}; \Lambda \mid \Gamma; \Delta \rhd B : \tau$	block B has block type $\Gamma; \Delta \rhd \tau$ under $\mathcal{C}$ and Λ
$\mathcal{C}; \Lambda \rhd K$	program unit K is well-typed under $\mathcal{C}$ and Λ
$\mathcal{C}; \Lambda \mid \Gamma; \Delta \rhd K.l : \tau$	the entry point l of the program $K.l$ has block type $\Gamma; \Delta \rhd \tau$ under $\mathcal{C}$ and Λ

Typing rules for blocks involving objects:

$$\frac{}{\mathcal{C}; \Lambda \mid \Gamma; \Delta, c \rhd \mathsf{areturn} : c} \quad \frac{\mathcal{C}; \Lambda \mid \Gamma; \Delta, \kappa \rhd B : \tau \quad \Gamma(i) = \kappa}{\mathcal{C}; \Lambda \mid \Gamma; \Delta \rhd \mathsf{aload}\ i; B : \tau} \quad \frac{\mathcal{C}; \Lambda \mid \Gamma\{i : \kappa\}; \Delta \rhd B : \tau}{\mathcal{C}; \Lambda \mid \Gamma; \Delta, \kappa \rhd \mathsf{astore}\ i; B : \tau}$$

$$\frac{\mathcal{C}; \Lambda \mid \Gamma; \Delta, c_u \rhd B : \tau \quad c \in \mathrm{dom}(\mathcal{C}) \quad u \text{ fresh}}{\mathcal{C}; \Lambda \mid \Gamma; \Delta \rhd \mathsf{new}\ c; B : \tau} \quad \frac{\mathcal{C}; \Lambda \mid [c/c_u]\Gamma; [c/c_u]\Delta \rhd B : \tau}{\mathcal{C}; \Lambda \mid \Gamma; \Delta, c_u \rhd \mathsf{init}\ c; B : \tau}$$

$$\frac{\mathcal{C}; \Lambda \mid \Gamma; \Delta, \mathcal{C}(c)(f) \rhd B : \tau}{\mathcal{C}; \Lambda \mid \Gamma; \Delta, c \rhd \mathsf{getfield}\ c, f; B : \tau} \quad \frac{\mathcal{C}; \Lambda \mid \Gamma; \Delta \rhd B : \tau}{\mathcal{C}; \Lambda \mid \Gamma; \Delta, c, \mathcal{C}(c)(f) \rhd \mathsf{putfield}\ c, f; B : \tau}$$

$$\frac{\mathcal{C}; \Lambda \mid \Gamma; \Delta, \tau \rhd B : \tau' \quad \mathcal{C}(c)(m) = \{i_0 : c,\ i_1 : \tau_1, \cdots, i_n : \tau_n\}; \emptyset \rhd \tau}{\mathcal{C}; \Lambda \mid \Gamma; \Delta, c, \tau_1, \cdots, \tau_n \rhd \mathsf{invoke}\ c, m; B : \tau'}$$

Typing of program units:

$$\frac{\forall l \in \mathrm{dom}(\Lambda). \mathcal{C}; \Lambda \mid \Gamma; \Delta \rhd K(l) : \tau \quad \Lambda(l) = \Gamma; \Delta \rhd \tau}{\mathcal{C}; \Lambda \rhd K}$$

Typing of programs:

$$\frac{\mathcal{C}; \Lambda \rhd K \quad \Lambda(l) = \Gamma; \Delta \rhd \tau}{\mathcal{C}; \Lambda \mid \Gamma; \Delta \rhd K.l : \tau}$$

Fig. 7. Type system of JAL

4.2 De-compilation Algorithm

The target language of the de-compilation is an extention of λ^{rec} with primitives for object manipulation corresponding to those in JAL. We call it λ^{obj}. The set of types and terms of λ^{obj} is the following:

$$\begin{aligned} \tau &::= c \mid \mathrm{int} \mid \tau \to \tau \\ M &::= \cdots \mid \mathsf{let}\ x = \mathsf{new}\ c\ \mathsf{in}\ M \mid x.\mathsf{init}\ c; M \mid \mathsf{let}\ x = x.m(M, \cdots, M)\ \mathsf{in}\ M \\ &\quad \mid \mathsf{let}\ x = x.f\ \mathsf{in}\ M \mid x.f := M; M \end{aligned}$$

The last five terms are those for object creation, object initialization, method invocation, object field extraction and object field update. The typing rules for these additional terms to λ^{rec} are shown in Figure 8. The type system for λ^{obj} is defined relative to a fixed class context $\mathcal{C}$. We should note that this type system does not take into account of uninitialized object types. Because of the higher-order feature, the Freund and Mitchell's technique does not easily extend to the lambda calculus. In this type system, uninitialized object types of the form c_u are identified with c. As a result, we can only show the type preservation up to this identification.

The de-compilation algorithm is obtained by extending the one for JAL^0 presented in the previous section with the equations for object manipulation instructions. Figure 9 shows the additional equations required for this extension. The transformation of JAL programs is given in the same way as the one for JAL^0 using the mutual recursion operator.

$$\frac{\mathcal{C} \mid \Gamma\{x : c\} \rhd M : \tau \quad c \in \mathrm{dom}(\mathcal{C})}{\mathcal{C} \mid \Gamma \rhd \mathsf{let}\ x = \mathsf{new}\ c\ \mathsf{in}\ M : \tau} \qquad \frac{\mathcal{C} \mid \Gamma \rhd M : \tau \quad \Gamma(x) = c}{\mathcal{C} \mid \Gamma \rhd x.\mathsf{init}\ c; M : \tau}$$

$$\frac{\mathcal{C} \mid \Gamma \rhd L_i : \tau_i\ (1 \le i \le n) \quad \mathcal{C} \mid \Gamma\{y : \tau\} \rhd M : \tau' \quad \Gamma(x) = c \quad \mathcal{C}(c)(m) = \{i_0 : c,\ i_1 : \tau_1, \ldots, i_n : \tau_n\}; \emptyset \rhd \tau}{\mathcal{C} \mid \Gamma \rhd \mathsf{let}\ y = x.m(L_1, \ldots, L_n)\ \mathsf{in}\ M : \tau'}$$

$$\frac{\mathcal{C} \mid \Gamma\{y : \tau\} \rhd M : \tau' \quad \Gamma(x) = c \quad \mathcal{C}(c)(f) = \tau}{\mathcal{C} \mid \Gamma \rhd \mathsf{let}\ y = x.f\ \mathsf{in}\ M : \tau'} \qquad \frac{\mathcal{C} \mid \Gamma \rhd M : \tau \quad \mathcal{C} \mid \Gamma \rhd N : \tau' \quad \Gamma(x) = c \quad \mathcal{C}(c)(f) = \tau}{\mathcal{C} \mid \Gamma \rhd x.f := M; N : \tau'}$$

Fig. 8. Additional typing rules for objects

$$\begin{aligned}
[\![\Gamma; \Delta, \tau \rhd \mathsf{areturn} : \tau]\!]_{\mathcal{C}} &= s_{|\Delta|} \\
[\![\Gamma; \Delta \rhd \mathsf{aload}\ i; B : \tau]\!]_{\mathcal{C}} &= [i/s_{|\Delta|}][\![\Gamma; \Delta, \Gamma(i) \rhd B : \tau]\!]_{\mathcal{C}} \\
[\![\Gamma; \Delta, \kappa \rhd \mathsf{astore}\ i; B : \tau]\!]_{\mathcal{C}} &= [s_{|\Delta|}/i][\![\Gamma\{i : \kappa\}; \Delta \rhd B : \tau]\!]_{\mathcal{C}} \\
[\![\Gamma; \Delta \rhd \mathsf{new}\ c; B : \tau]\!]_{\mathcal{C}} &= \mathsf{let}\ s_{|\Delta|} = \mathsf{new}\ c\ \mathsf{in}\ [\![\Gamma; \Delta, c_u \rhd B : \tau]\!]_{\mathcal{C}} \\
[\![\Gamma; \Delta, c_u \rhd \mathsf{init}\ c; B : \tau]\!]_{\mathcal{C}} &= s_{|\Delta|}.\mathsf{init}\ c; [\![[c/c_u]\Gamma; [c/c_u]\Delta \rhd B : \tau]\!]_{\mathcal{C}} \\
[\![\Gamma; \Delta, c, \tau_1, \cdots, \tau_n \rhd \mathsf{invoke}\ c, m; B : \tau']\!]_{\mathcal{C}} &= \mathsf{let}\ s_{|\Delta|} = s_{|\Delta|}.m(s_{|\Delta|+1}, \ldots, s_{|\Delta|+n}) \\
&\qquad \mathsf{in}\ [\![\Gamma; \Delta, \tau \rhd B : \tau']\!]_{\mathcal{C}} \\
&\qquad \text{where } \mathcal{C}(c)(m) = \{i_0 : c, i_1 : \tau_1, \cdots, i_n : \tau_n\}; \emptyset \rhd \tau \\
[\![\Gamma; \Delta, c \rhd \mathsf{getfield}\ c, f; B : \tau]\!]_{\mathcal{C}} &= \mathsf{let}\ s_{|\Delta|} = s_{|\Delta|}.f\ \mathsf{in}\ [\![\Gamma; \Delta, \mathcal{C}(c)(f) \rhd B : \tau]\!]_{\mathcal{C}} \\
[\![\Gamma; \Delta, c, \mathcal{C}(c)(f) \rhd \mathsf{setfield}\ c\ f; B : \tau]\!]_{\mathcal{C}} &= s_{|\Delta|}.f := s_{|\Delta|+1}; [\![\Gamma; \Delta \rhd B : \tau]\!]_{\mathcal{C}}
\end{aligned}$$

Fig. 9. De-compilation algorithm for object primitives in JAL

As in the case for the de-compilation algorithm for JAL^0, this algorithm is a type-preserving proof transformation from a sequential sequent calculus to (an extension of) natural deduction. The following theorem formally establishes this property.

Theorem 4. *If* $\mathcal{C}; \Lambda \mid \Gamma; \Delta \rhd K.l : \tau$ *then the judgement* $\mathcal{C} \mid \emptyset \rhd [\![\Lambda \mid \Gamma; \Delta \rhd K.l : \tau]\!]_{\mathcal{C}} : \overline{\Gamma; \Delta \rhd \tau}$ *is derivable in* λ^{obj} *up to the identification of* c_u *with* c.

5 A Prototype Implementation of a De-compiler

We have implemented a prototype de-compiler, JD, based on the transformation algorithm presented in this paper. Input to JD is an JVM assembly language source file in the format of Jasmin described in [5], which can be mechanically constructed from a JVM class file. JD first parses a given source file to obtain an internal representation of a set of sequences of JVM instructions, each of which corresponds to one method. JD then converts each sequence of instructions into a program unit consisting of blocks. In doing this, JD considers a cascaded block as a collection of blocks connected by implicit jumps, and inserts jumps to make the block structure explicit. This insertion does not change the semantics of the

program. Finally, JD de-compiles each program unit by applying the algorithm presented in this paper to generate a term in the lambda calculus with objects.

JD supports more instructions and types than those we have considered in the formal framework, including those for arithmetics, arrays, double-word types. In addition, JD performs more jobs than we presented in the previous sections. One is removing intermediate labels and temporary variables. Since most Java bytecode programs consist of many small blocks, without this processing, the resulting lambda term would contain many redundant labels and variables. JD achieves this removal by applying a code manipulation which corresponds to some β reductions in the target language. In Figure 1, the block corresponding to label L5 is eliminated by this process.

6 Related Work

The work most relevant to ours is perhaps Stata and Abadi [13] on a type system for Java bytecode subroutines. This work is further refined in [9, 1]. In these approaches, a type system is used to check the consistency of an array of instructions. The result of typechecking is success or failure indicating whether the array of instruction is type consistent or not. In contrast, our approach is to interpret a given code as a constructive proof representing its computation. This allows us to de-compile a code to a lambda term.

Our work is also related to the typed assembly language (TAL) of Morrisett et. al. [6, 7]. Their type system is designed to check the type consistency of a sequence of instruction, and is not intended to serve as a logic. Nonetheless, some of our proof rules are similar to the corresponding ones in their type system. In our proof theory, for example, a jump instruction is interpreted as a rule to refer to an existing proof, which has some similarity to the TAL's treatment of jumps.

Our de-compilation performs proof transformation from a variant of the sequent calculus to natural deduction. Raffalli [12] considers compilation as proof transformation. The TAL approach emphasizes the benefit of compilation as type-preserving transformation, which can be regarded as proof transformation. In the general perspective, our approach shares the same spirit with these approaches. However, the problem of the converse of compilation has not been investigated. From a logical perspective, the relationship between Gentzen's intuitionistic sequent calculus and natural deduction has been extensively studied. (See [2] for a survey.) Our proof system for bytecode languages is similar to the Gentzen's sequent calculus, and therefore some of the cases in de-compilation algorithm have the similar structure to the corresponding cases in proof transformation from the Gentzen's sequent calculus to the natural deduction.

There are a number of works for "reverse engineering" machine code. (See for example [14].) There are also several working de-compilers for Java bytecode language. However, little has been known about the foundation of de-compilation. The major technical contribution of our work is to provide a logical foundation for systematic development of a de-compilation algorithm, for reasoning about the de-compilation process, and for establishing its correctness.

7 Conclusions

We have developed a framework for proof-directed de-compilation of low-level code based on the Curry-Howard isomorphism for machine code, and have presented a proof-directed de-compilation algorithm for a subset of Java bytecode language including integer primitives, stack operations, local variable manipulation, conditional and unconditional jumps. A prototype de-compiler for Java bytecode has been implemented, which demonstrates the feasibility of the proof-directed de-compilation approach presented in this paper. We believe that by combining the existing strategies and heuristic techniques, the method presented in this paper will contribute to developing a practical and robust de-compiler.

Acknowledgements We thank some of anonymous referees for thorough and careful reading of the paper and for providing many helpful comments, which have been very useful for improving the presentation of the paper.

References

1. S. Freund and J. Mitchell. A type system for object initialization in the Java byte code language. In *Proc. OOPSLA'98*, pages 310–328, 1998.
2. J. Gallier. Constructive logics part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science 110*, pages 249–339, 1993.
3. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
4. T. Lindholm and F. Yellin. *The Java virtual machine specification.* Addison Wesley, 2nd edition, 1999.
5. J. Meyer and T. Downing. *Java Virtual Machine.* O'Reilly, 1997.
6. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Proc. Types in Compilation, LNCS 1473*, pages 28-52, 1998.
7. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. POPL'98*, pages 85-97, 1998.
8. G. Necula. Proof-carrying code. In *Proc. POPL'98*, pages 106–119, 1998.
9. R. O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. POPL'99*, pages 70–78, 1999.
10. A. Ohori. A Curry-Howard isomorphism for compilation and program execution. In *Proc. TLCA'99, LNCS 1581*, pages 280–294, 1999.
11. A. Ohori. The logical abstract machine: a Curry-Howard isomorphism for machine code. In *Proc. FLOPS'99, LNCS 1722*, pages 300-318,1999.
12. C. Raffalli. Machine deduction. In *Proc. Types for Proofs and Program, LNCS 806*, pages 333–351, 1994.
13. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. POPL'98*, pages 149–160, 1998.
14. *Proceedings of Working Conference on Reverse Engineering.* IEEE Computer Society Press, 1993–.