Java Bytecode as a Typed Term Calculus

Tomoyuki Higuchi
School of Information Sicence
Japan Advanced Institute of
Science and Technology
Tasunokuchi Ishikawa, 923-1292 Japan
thiguchi@jaist.ac.jp

Atsushi Ohori
School of Information Sicence
Japan Advanced Institute of
Science and Technology
Tasunokuchi Ishikawa, 923-1292 Japan
ohori@jaist.ac.jp

ABSTRACT

We propose a type system for the Java bytecode language, prove the type soundness, and develop a type inference algorithm. In contrast to the existing proposals, our type system yields a typed term calculus similar to type systems of lambda calculi. This enables us to transfer existing techniques and results of type theory to a JVM-style bytecode language. We show that ML-style let polymorphism and recursive types can be used to type JVM subroutines, and that there is an ML-style type inference algorithm.The type inference algorithm has beeen implemented. The ability to verify type soundness is a simple corollary of the existence of type inference algorithm. Moreover, our type theoretical approach opens up various type safe extensions including higher-order methods, flexible polymorphic typing through polymorphic type inference, and type-preserving compilation.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Macro and assembly languages, Object-oriented languages*

General Terms

Languages, Theory, Verification

Keywords

Java bytecode, bytecode verifier, type system, type inference

1. INTRODUCTION

Type safety of executable code 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. An important achievement toward this direction is the development of the Java bytecode language [13], which is the target language of the Java programming language [5]. A distinguishing feature is its typing constraint. The system can ensure type correct execution of a given bytecode by checking type constraint before execution.

Type verification of JVM bytecode is essentially static type checking, customary done in high-level typed programming languages. Since JVM is a powerful and complex system, the development of a correct and reliable type checking system requires a formal framework for semantics and typing derivation of the JVM bytecode language. This problem has recently attracted the attention of programming language researchers, and several static type systems have been developed. Stata and Abadi [17] propose a static type system for a subset of JVM bytecode language including subroutines. In this work, a type system checks the satisfiability of constraints on memory states induced by the set of instructions in the given code. This paradigm has been successfully used in subsequent proposals on type-checking the JVM bytecode language.

Freund and Mitchell [4] use this framework to analyze a subtle problem of object initialization and propose a refined type-checking scheme. They further show in [3] that their approach extends to various features of JVM including objects, classes, interfaces, and exceptions. O'Callahan [15] also extends the approach of [17] to allow more flexible typing for subroutines by giving an explicit return type of a subroutine. In addition to this, he introduces polymorphic typing to achieve flexibility in subroutine usage. Hagiya and Tozawa [6] give an alternative approach for subroutines based on dataflow analysis, which yields a simpler soundness proof. Iwama and Kobayashi [7] propose another type system based on [17] to verify the correctness of the usage of lock primitives, where the type of an object has the information about the order in which the object is locked/unlocked. Leroy [12] presents a light-weight verification method based on dataflow analysis.

In those proposals, a type system is designed to check type consistency of memory states imposed by instructions in a given bytecode. While this paradigm is useful in establishing type safety of the JVM bytecode language, some important questions remain. For example, how does this paradigm extend to other useful programming features, or how does this paradigm relates to existing theory of type systems? As we comment later in the next section, O'Callahan's system uses the notion of continuation which represents more information of the behavior of a program than set of memory usage constraints, but its relationship to existing notions in type theory is not entirely clear.

One obstacle in answering these questions appears to be the fact that in this paradigm, types do not represent behavior of bytecode. This is in contrast with type systems of programming languages where types provide a static view of the behavior of a program – a typing $\Gamma \rhd M : \tau$ implies that M yields a value of type τ when it is executed in an environment of type Γ. This view yields a clean type soundness theorem and has been the basis for incorporating various advanced features such as polymorphism and type inference in a language.

Jones [8] suggests representing each JVM instruction as a function and sequencing as function composition. Although semantics of functions is general enough to represent JVM instructions, it does not seem to reflect machine execution directly. As a result, it is not obvious that this approach could be a basis for establishing type soundness or for developing a feasible type-checking algorithm for bytecode including subroutines.

Our goal is to develop a typed calculus for a JVM-style bytecode language, where types represent behavior of code. We base our development on Curry-Howard isomorphism for low-level code [16], where it is shown that a low-level code language corresponds to a sequent style proof system of the intuitionistic propositional logic. Katsumata and Ohori [9] sketch that this idea can be applied to represent JVM-style bytecode. The purpose of [9] is to present a proof-directed de-compilation method, and its treatment of JVM bytecode is only on syntactical typing and is quite limited; it does not consider subroutine or inheritance, and it does not discuss type soundness. In the present paper, we refine the logical approach of [16] and develop a typed calculus of JVM bytecode including most of its features, establish type soundness, and develop a type inference algorithm. We believe that the proposed calculus provides type-theoretical account for JVM, and serves as a framework for extending JVM-style bytecode languages with various advanced features including higher-order methods, flexible polymorphic typing, polymorphic type inference, and type-preserving compilation.

The rest of the paper is organized as follows. Section 2 outlines our approach. Section 3 gives a typed calculus for JVM, and establishes type soundness. Section 4 extends the typed calculus with polymorphism and develops a type inference algorithm. We have implemented a prototype bytecode verifier based on the type inference algorithm, which is described in Section 5. Section 6 discusses some extensions to the type systems, and Section 7 concludes the paper.

2. TYPE-THEORETICAL INTERPRETATION OF BYTECODE

We follow the logical interpretation of low-level code [16] and interpret a bytecode program as a typing derivation. In JVM, a program consists of a collection of labeled blocks ending with a return instruction or a branch instruction. We let I and B range over (non-branching) instructions and code blocks, respectively. A non-branching instruction operates on a local environment and a stack. We use Γ and Δ for types of local environments and stacks respectively. Code blocks, environment types, and stack types are finite sequences, for which we use the following notations. $e{\cdot}S$ is the sequence obtained by adding an element e at the begging of a sequnce S, and $S\{n \leftarrow e\}$ is the sequence obtained by changing the nth element of S to e. The empty sequence is denoted by ϕ.

Applying the idea of [16], we interpret a JVM block B as a judgment of the form $\Gamma, \Delta \rhd B : \tau$ in a sequent style proof system. A return instruction corresponds to an initial sequent in the proof system. For example, $\Gamma, \mathtt{int}{\cdot}\Delta \rhd \mathtt{ireturn} : \mathtt{int}$ indicates that `ireturn` is a complete program returning the top element of the current stack. A $\mathtt{goto(}l\mathtt{)}$ refers to an existing block B named l. So we type $\Gamma, \Delta \rhd \mathtt{goto(}l\mathtt{)} : \tau$ if $\Gamma, \Delta \rhd B : \tau$. An ordinary non-branching instruction I that changes the machine state of type Γ_1, Δ_1 to that of type Γ_2, Δ_2, written as $I : \Gamma_1, \Delta_1 \Longrightarrow \Gamma_2, \Delta_2$, is interpreted as as a left-rule of the form:

$$\frac{\Gamma_2, \Delta_2 \rhd B : \tau}{\Gamma_1, \Delta_1 \rhd I{\cdot}B : \tau}$$

which can be read "backward" saying that the execution of I transforms a bigger proof of $I{\cdot}B$ to a smaller proof of B. Most of the non-branching instructions including one for method invocation can be interpreted in this way.

An important exception is `jsr` for calling a subroutine. A subroutine block (ranged over by SB) does not return a value but returns a modified environment and a modified stack to be used by the block that follows `jsr`. Let l be a label of a subroutine block SB that changes a machine state of type Γ_1, Δ_1 to that of type Γ_2, Δ_2, and let B be a block such that $\Gamma_2, \Delta_2 \rhd B : \tau$. We interpret a subroutine call $\mathtt{jsr(}l\mathtt{)}{\cdot}B$ as a function to transform a judgment of type $\Gamma_2, \Delta_2 \rhd \tau$ to that of type $\Gamma_1, \Delta_1 \rhd \tau$. To represent this intuitive static semantics, we type $\mathtt{jsr(}l\mathtt{)}{\cdot}B$ as $\Gamma_1, \Delta_1 \rhd \mathtt{jsr(}l\mathtt{)}{\cdot}\mathrm{B}\colon \tau$ and assign to SB a type of the form $\langle \Gamma_2, \Delta_2 \rhd \tau // \Gamma_1, \alpha_l{\cdot}\Delta_1 \rhd \tau \rangle$ where α_l denotes the type of a return address.

The use of *continuation* in O'Callahan's work [15] corresponds to introducing the part Γ_1, Δ_1 in the above type. Capturing this part of typing is enough to ensure type safety of a program consisting only of blocks and subroutine blocks. When methods are added, however, some additional machinery seems to be necessary. Our system can serve as a type theoretical framework for representing both methods (or other higher-order objects) and collection of blocks and subroutines.

Further refinement is needed for typing $\mathtt{ret(}i\mathtt{)}$ instruction. This instruction transfers control back to the block whose address is stored in the ith element of a local environment. This means that $\Gamma_2(i)$ is the type of the block B to which it returns, and therefore the equation $\Gamma_2(i) = \Gamma_2, \Delta_2 \rhd \tau$ must hold. We solve this problem by introducing recursive type equations. For each subroutine identified by its entry label l, we introduce a type variable α_l representing the return address with an associated equation, and assign to a subroutine l a type of the form $(\alpha_l = \Gamma_2, \Delta_2 \rhd \tau \text{ in } \langle \alpha_l // \Gamma_1, \Delta_1 \rhd \tau \rangle)$. We treat a recursive type equation $\alpha_l = \Gamma, \Delta \rhd \tau$ as a global declaration similar to ML's `datatype`. When the equation is irrelevant, we simply write $\langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle$.

Figure 1 shows an example of type derivation for bytecode block using a subroutine. Later in section 4 we show that there is an algorithm to infer a polymorphic typing for blocks and subroutine blocks.

3. THE TYPED JVM CALCULUS

Bytecode program:

```
L0: jsr L2        L2: astore_2      L3: iconst_1
L1: iload_1           iload_1           istore_1
    ireturn           ifeq L3           ret 2
                      iload_1
                      ireturn
```

Typing results:

$\alpha_{L2} = \{c, \mathsf{int}, \alpha_{L2}\}, \Delta \rhd \mathsf{int}$ in
- $\mathsf{L0} : \{c, \mathsf{int}, \tau\}, \Delta \rhd \mathsf{int}$
- $\mathsf{L1} : \{c, \mathsf{int}, \alpha_{L2}\}, \mathsf{int}{\cdot}\Delta \rhd \mathsf{int}$
- $\mathsf{L2} : \langle \alpha_{L2} /\!/ \{c, \mathsf{int}, \tau\}, \alpha_{L2}{\cdot}\Delta \rhd \mathsf{int}\rangle$
- $\mathsf{L3} : \langle \alpha_{L3} /\!/ \{c, \mathsf{int}, \alpha_{L2}\}, \Delta \rhd \mathsf{int}\rangle$

Figure 1: Example of Type Derivation

This section defines a typed calculus, Jvmc, for the JVM bytecode language.Since a set of classes and an associated subclass relation are explicitly declared, we define Jvmc relative to a given fixed set of class names (ranged over by c), and a given fixed subclass relation on class names. We write $c_1 <: c_2$ if c_1 is a subclass of c_2.

3.1 The Syntax of Jvmc

A JVM program is a collection of classes. We regards a collection of JVM classes as a pair (Θ, Π) of type specifications Θ and method definitions Π. Θ is a function assigning each class name a set of typed field names (ranged over by f) and a set of typed method names (ranged over by m). Figure 2 gives the syntax of Θ. $\{\tau_1, \ldots, \tau_n\} \Rightarrow \tau$ is a method

$$\begin{array}{rcl}
\Theta & := & \{c = spec, \ldots, c = sepc\} \\
spec & := & \{\mathtt{methods} = \{m : \{\tau_1, \ldots, \tau_n\} \Rightarrow \tau, \cdots\} \\
 & & \ \mathtt{fields} = \{f : \tau, \cdots\}\} \\
\tau & := & \mathsf{int} \mid \mathsf{void} \mid c \mid \alpha_l \mid \top
\end{array}$$

Figure 2: Syntax of Θ

type with argument types $\{\tau_1, \ldots, \tau_n\}$ and the return type τ. $\top$ is a special type representing unused environment entry. In this abstract syntax, we understand that the subclass relation is already incorporated so that the set of field names and the set of method names of a class contain all those defined in its super classes. We also assume that if some common filed names are used in super classes, then those names are properly disambiguated.

Π assign each class name its method definitions. In an actual JVM class file, each method body is a sequence of JVM instructions some of which have labels (ranged over by l) for branch instructions. We regard such a sequence as a labeled collection of *basic blocks*. Furthermore, we divide basic blocks into *code blocks* (ranged over by B) and *subroutine blocks* (ranged over by SB). Subroutine blocks are those that are (originally) invoked by a subroutine call instruction. We identify each subroutine block with its entry label and write $SB(l_s)$ for a subroutine block whose (original) entry point is l_s. With this refinement, the syntax of method definitions Π is given in given in Figure 3, where i and n represent a local variabel and an integer value respectively. Some comments are in order. It is straightforward to

$$\begin{array}{rcl}
\Pi & := & \{c = methods, \ldots, c = methods\} \\
methods & := & \{m = M, \ldots, m = M\} \\
M & := & \{l_b : B, \cdots \mid l_s : SB(l_s), \cdots\} \\
B & := & \mathtt{return} \mid \mathtt{ireturn} \mid \mathtt{areturn} \\
 & \mid & \mathtt{jsr}(l_s, l_b) \mid \mathtt{goto}(l_b) \mid I{\cdot}B \\
SB & := & \mathtt{return} \mid \mathtt{ireturn} \mid \mathtt{areturn} \\
 & \mid & \mathtt{jsr}(l_s, l_s) \mid \mathtt{ret}(i) \mid \mathtt{goto}(l_s) \mid I{\cdot}SB \\
I & := & \mathtt{iconst}(n) \mid \mathtt{iload}(i) \mid \mathtt{aload}(i) \\
 & \mid & \mathtt{istore}(i) \mid \mathtt{astore}(i) \mid \mathtt{dup} \mid \mathtt{pop} \mid \mathtt{iadd} \\
 & \mid & \mathtt{ifeq}(l) \mid \mathtt{new}(c) \mid \mathtt{invoke}(c, m) \\
 & \mid & \mathtt{getfield}(c, f) \mid \mathtt{putfield}(c, f)
\end{array}$$

Figure 3: Syntax of Π

construct a labeled collection of code blocks. Construction of subroutine blocks associated with an entry label can be done by traversing instruction sequence starting from a label l appearing in some $\mathtt{jsr}(l, l')$ and collecting all the reachable basic blocks. For those basic blocks that are associated with more than one entry labels, we consider that separate copies exist for each entry labels. Since code is not mutable, any potions of B_i and SB_i in the result of this construction can be shared. So there is no danger of code size explosion.

In JVM, subroutine call has the form $\mathtt{jsr}(l){\cdot}B$. JVM pushes on the stack the address of B to be used as the return address by the callee. In order to develop a type system, we need to type a return address explicitly. For this reason, we regards $\mathtt{jsr}(l){\cdot}B$ as shorthand for $\mathtt{jsr}(l, l')$ with the introduction of a new labeled block $l' : B$.

3.2 The Type System

The basic typing judgments are those for code blocks and subroutine blocks. As we have outlined in Section 2, they are judgments of the form:

- $\Gamma, \Delta \rhd B : \tau$
- $SB(l) : (\alpha_l = \Gamma_2, \Delta_2 \rhd \tau \text{ in } \langle \alpha_l /\!/ \Gamma_1, \Delta_1 \rhd \tau\rangle)$

Since these blocks contain labels for branch instructions, their derivation are defined relative to a *label environment* $\mathcal{L}$ of the form

$$\begin{array}{l}
\mathcal{L} = \{\ l_b : \Gamma, \Delta \rhd \tau, \ldots \\
\qquad \mid l_s : (\alpha_{l_s} = \Gamma_2, \Delta_2 \rhd \tau \text{ in } \langle \alpha_{l_s} /\!/ \Gamma_1, \Delta_1 \rhd \tau\rangle), \ldots\}
\end{array}$$

specifying a typing of the code blocks and subroutine blocks of a given method. If S is a sequence, we write $S.i$ for the ith element in S. Similarly, if S is a mapping and e is an element in its domain, then $S.e$ denotes the element assigned to e by S.

Using these notations, static semantics of non-branching instructions is given in Figure 4, and the typing rules for blocks are given in Figure 5.

In this definition, we simply assume that $\mathtt{new}(c)$ creates a complete object of class c. In an actual JVM, object creation is done in two stages by first creating a container by $\mathtt{new}$ and then initializing their fields by the constructors. As observed in [4], there is subtle issues associated with this process. We believe that the mechanism proposed in [4] is orthogonal to

$\texttt{iconst}(n) : \Gamma, \Delta \Longrightarrow \Gamma, \mathsf{int}{\cdot}\Delta$

$\texttt{iload}(i) : \Gamma, \Delta \Longrightarrow \Gamma, \mathsf{int}{\cdot}\Delta \quad (\text{if } \Gamma(i) = \mathsf{int})$

$\texttt{aload}(i) : \Gamma, \Delta \Longrightarrow \Gamma, c{\cdot}\Delta \quad (\text{if } \Gamma(i) = c)$

$\texttt{istore}(i) : \Gamma, \mathsf{int}{\cdot}\Delta \Longrightarrow \Gamma\{i \leftarrow \mathsf{int}\}, \Delta$

$\texttt{astore}(i) : \Gamma, c{\cdot}\Delta \Longrightarrow \Gamma\{i \leftarrow c\}, \Delta$

$\texttt{astore}(i) : \Gamma, \alpha_l{\cdot}\Delta \Longrightarrow \Gamma\{i \leftarrow \alpha_l\}, \Delta$

$\texttt{dup} : \Gamma, \tau{\cdot}\Delta \Longrightarrow \Gamma, \tau{\cdot}\tau{\cdot}\Delta$

$\texttt{iadd} : \Gamma, \mathsf{int}{\cdot}\mathsf{int}{\cdot}\Delta \Longrightarrow \Gamma, \mathsf{int}{\cdot}\Delta$

$\texttt{pop} : \Gamma, \tau{\cdot}\Delta \Longrightarrow \Gamma, \Delta$

$\texttt{new}(c) : \Gamma, \Delta \Longrightarrow \Gamma, c{\cdot}\Delta$

$\texttt{getfield}(c_0,f) : \Gamma, c_1{\cdot}\Delta \Longrightarrow \Gamma, \tau{\cdot}\Delta$
$(\text{if } \Theta.c_0.\texttt{fields}.f = \tau \text{ and } c_1 <: c_0)$

$\texttt{putfield}(c_0,f) : \Gamma, \tau{\cdot}c_1{\cdot}\Delta \Longrightarrow \Gamma, \Delta$
$(\text{if } \Theta.c_0.\texttt{fields}.f = \tau \text{ and } c_1 <: c_0)$

$\texttt{invoke}(c_0,m) : \Gamma, \tau_n{\cdot}\ldots\tau_1{\cdot}c_1{\cdot}\Delta \Longrightarrow \Gamma, \tau_0{\cdot}\Delta$
$(\text{if } \Theta.c_0.\texttt{methods}.m = \{\tau'_1, \cdots, \tau'_n\} \Longrightarrow \tau_0, \tau_0 \neq \mathsf{void}$
$\text{and } c_1 <: c_0 \wedge \tau_i <: \tau'_i \text{ for all } 1 \leq i \leq n)$

$\texttt{invoke}(c_0,m) : \Gamma, \tau_n{\cdot}\ldots{\cdot}\tau_1{\cdot}c_1{\cdot}\Delta \Longrightarrow \Gamma, \Delta$
$(\text{if } \Theta.c_0.\texttt{methods}.m = \{\tau'_1, \cdots, \tau'_n\} \Longrightarrow \mathsf{void}$
$\text{and } c_1 <: c_0 \wedge \tau_i <: \tau'_i \text{ for all } 1 \leq i \leq n)$

Figure 4: Static Semantics of Non-branching Instructions

our approach, and can be adopted in our type system as well. Another simplification we made is that all the necessary class files are available at the time of verification. This is reflected in the typing rules of `getfield`, `putfield`, and `invoke`, which refer to Θ. In an actual JVM, classes are dynamically loaded. It is not hard to modify our type system to model dynamic class loading. Since those instructions include the static type of the method or field, we can simply use the specified type to verify the code block containing these instruction without referring to the class files. At the time of executing one of these instructions, which causes a class containing the method to be loaded, we infer the type of the method and verify that it is indeed equal to the one specified in the instruction. This process is easily formalized by using the mechanism of dynamic typing [1].

Typing of a method M is then defined as follows.

$$\begin{aligned}
&\vdash M : \mathcal{L} \\
\Leftrightarrow\ & \forall l_b \in dom(M).\ \mathcal{L}(l_b) = \Gamma, \Delta \rhd \tau \wedge\ \mathcal{L} \vdash \Gamma, \Delta \rhd M(l_b) \ :\ \tau \\
&\text{and } \forall l_s \in dom(M).\ \mathcal{L} \vdash M(l_s) : \mathcal{L}(l_s).
\end{aligned}$$

We assume that a method M contains a block having a unique special label e indicating its entry point, and define the typing of a method as follows:

$$\begin{aligned}
&\vdash M : \{\tau_1, \ldots, \tau_n\} \Rightarrow \tau \\
\Leftrightarrow\ & \exists \mathcal{L} \text{ such that } \vdash M : \mathcal{L} \text{ and} \\
&\mathcal{L} \vdash \Gamma\{0 \leftarrow c, 1 \leftarrow \tau_1, \cdots n \leftarrow \tau_n\}, \phi \rhd M.e \ :\ \tau \\
&\quad (\Gamma = Top(max_M))
\end{aligned}$$

where $Top(n)$ is the type of environment of size n filled with a meaningless type $\top$ and max_M is the number of local variables used in the method M.

We can now define the type correctness of a Jvmc program

Typing rules for code blocks:

$\mathcal{L} \vdash \Gamma, \Delta \rhd \texttt{ireturn} \ :\ \mathsf{int}$

$\mathcal{L} \vdash \Gamma, c{\cdot}\Delta \rhd \texttt{areturn} \ :\ c$

$\mathcal{L} \vdash \Gamma, \Delta \rhd \texttt{return} \ :\ \mathsf{void}$

$\mathcal{L} \vdash \Gamma, \Delta \rhd \texttt{goto}(l) \ :\ \tau \quad (\text{if } \mathcal{L}(l) = \Gamma, \Delta \rhd \tau)$

$$\frac{\mathcal{L} \vdash \Gamma_2, \Delta_2 \rhd B \ :\ \tau}{\mathcal{L} \vdash \Gamma_1, \Delta_1 \rhd I{\cdot}B \ :\ \tau} \quad (\text{if } I : \Gamma_1, \Delta_1 \Longrightarrow \Gamma_2, \Delta_2)$$

$$\frac{\mathcal{L} \vdash \Gamma, \Delta \rhd B \ :\ \tau}{\mathcal{L} \vdash \Gamma, \mathsf{int}{\cdot}\Delta \rhd \texttt{ifeq}(l){\cdot}B \ :\ \tau} \quad (\text{if } \mathcal{L}(l) = \Gamma, \Delta \rhd \tau)$$

$\mathcal{L} \vdash \Gamma_1, \Delta_1 \rhd \texttt{jsr}(l_1,l_2) \ :\ \tau$
$(\text{if } \mathcal{L}(l_1) = (\alpha_{l_1} = \Gamma_2, \Delta_2 \rhd \tau \text{ in } \langle \alpha_{l_1} // \Gamma_1, \alpha_{l_1}{\cdot}\Delta_1 \rhd \tau\rangle)$
$\text{and } \mathcal{L}(l_2) = \Gamma_2, \Delta_2 \rhd \tau)$

Typing rules for subroutine blocks:

$\texttt{ret(x)} : (\alpha_l = \Gamma, \Delta \rhd \tau \text{ in } \langle \alpha_l // \Gamma, \Delta \rhd \tau\rangle)$
$(\text{if } \Gamma(x) = \alpha_l)$

$\texttt{goto}(l) : \langle \alpha_l // \Gamma, \Delta \rhd \tau\rangle \qquad (\text{if } \mathcal{L}(l) = \langle \alpha_l // \Gamma, \Delta \rhd \tau\rangle)$

$\texttt{return} : \langle \alpha_l // \Gamma, \Delta \rhd \mathsf{void}\rangle$

$\texttt{ireturn} : \langle \alpha_l // \Gamma, \mathsf{int}{\cdot}\Delta \rhd \mathsf{int}\rangle$

$\texttt{areturn} : \langle \alpha_l // \Gamma, c{\cdot}\Delta \rhd c\rangle$

$\texttt{jsr}(l_1, l_2) : \langle \alpha_l // \Gamma, \Delta \rhd \tau\rangle$
$(\text{if } \mathcal{L}(l_1) = (\alpha_{l_1} = \Gamma', \Delta' \rhd \tau \text{ in } \langle \alpha_{l_1} // \Gamma, \alpha_{l_1}{\cdot}\Delta \rhd \tau\rangle)$
$\text{and } \mathcal{L}(l_2) = \langle \alpha_l // \Gamma', \Delta' \rhd \tau\rangle)$

$$\frac{SB : \langle \alpha_l // \Gamma_2, \Delta_2 \rhd \tau\rangle}{I{\cdot}SB : \langle \alpha_l // \Gamma_1, \Delta_1 \rhd \tau\rangle} \quad (\text{if } I : \Gamma_1, \Delta_1 \Longrightarrow \Gamma_2, \Delta_2)$$

$$\frac{SB : \langle \alpha_l // \Gamma, \Delta \rhd \tau\rangle}{\texttt{ifeq}(l){\cdot}SB : \langle \alpha_l // \Gamma, \mathsf{int}{\cdot}\Delta \rhd \tau\rangle}$$
$(\text{if } \mathcal{L}(l) = \langle \alpha_l // \Gamma, \Delta \rhd \tau\rangle)$

Figure 5: Typing Rules for Blocks

(Π, Θ) as the following property:

$$\begin{aligned}
\vdash \Pi : \Theta \quad \Leftrightarrow \quad & Dom(\Pi) = Dom(\Theta), \text{and} \\
& \forall c \in Dom(\Pi).\ \vdash \Pi.c.m : \Theta.c.\texttt{methods}.m
\end{aligned}$$

3.3 Operational Semantics

We establish that our type system is correct by formally proving the type soundness theorem with respect to an operational semantics of Jvmc. We let S range over runtime stacks, E range over runtime variable environments, and h range over heaps. Both S and E are finite sequences of runtime values (ranged over by v). A heap h maps an address (ranged over by r) to a runtime representation of an object of the form $\langle f_1 = v_1, \ldots, f_n = v_n\rangle_c$ where c is the class of the object. Possible runtime values are either natural numbers n, an address r in a heap, or a return address $adrs(l)$ which represents the entry address of a block named l and is used by a subroutine. We write

$$I : (S, E), h \Longrightarrow (S', E'), h'$$

to indicate that I changes the machine state $(S, E), h$ to $(S', E'), h'$. Fig. 6 gives this relation. $Update(h, r, f, v)$ in the rule for `putfield` updates the f field of a runtime representation of an object in the heap h pointed by r to v. The object created by `new` consists of default values $\bot_\tau$ of type τ. We assume that these values behave as ordinary values of type τ in subsequent operation. This reflects our simplifying assumption mentioned earlier that `new(`c`)` creates an object of class c and we do not treat the issues of two stage object creation mechanism in JVM.

$$
\begin{array}{l}
\texttt{iconst(}n\texttt{)} : (S, E), h \Longrightarrow (n{\cdot}S, E), h \\
\texttt{iload(}i\texttt{)} : (S, E), h \Longrightarrow (E(i){\cdot}S, E), h \\
\texttt{aload(}i\texttt{)} : (S, E), h \Longrightarrow (E(i){\cdot}S, E), h \\
\texttt{istore(}i\texttt{)} : (n{\cdot}S, E), h \Longrightarrow (S, E\{i \leftarrow n\}), h \\
\texttt{astore(}i\texttt{)} : (r{\cdot}S, E), h \Longrightarrow (S, E\{i \leftarrow r\}), h \\
\texttt{astore(}i\texttt{)} : (adrs(l){\cdot}S, E), h \Longrightarrow (S, E\{i \leftarrow adrs(l)\}), h \\
\texttt{dup} : (v{\cdot}S, E), h \Longrightarrow (v{\cdot}v{\cdot}S, E), h \\
\texttt{iadd} : (n_1{\cdot}n_2{\cdot}S, E), h \Longrightarrow ((n_1 + n_2){\cdot}S, E), h \\
\texttt{pop} : (v{\cdot}S, E), h \Longrightarrow (S, E), h \\
\texttt{new(}c\texttt{)} : (S, E), h \Longrightarrow (r{\cdot}S, E), h' \\
\quad (\text{if } h' = h\{r \leftarrow \langle f_1 = \bot_{\tau_1}, \ldots, f_n = \bot_{\tau_n}\rangle_c\} \\
\qquad \Theta.c.\texttt{fields} = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}, \\
\qquad \text{and } r \notin dom(h)) \\
\texttt{getfield(}c\texttt{,}f\texttt{)} : (r{\cdot}S, E), h \Longrightarrow (h(r).f{\cdot}S, E), h \\
\texttt{putfield(}c\texttt{,}f\texttt{)} : (v{\cdot}r{\cdot}S, E), h \Longrightarrow (S, E), h' \\
\quad (\text{if } h' = h\{r \leftarrow Update(h, r, f, v)\})
\end{array}
$$

Figure 6: Dynamic Semantics of Non-branching Instructions

An operational semantics of Jvmc is given through a set of rules similar to those of SECD machine [10] of the form

$$(S, E, C, D), h \longrightarrow (S', E', C', D'), h'$$

C is a code of the form $M\{B\}$ or $M\{SB\}$ indicating that the machine executes the top instruction of code block B (or subroutine block SB) in a method body M. D is a dump, which is either empty ϕ, or a sequence of saved execution frames of the form $(S, E, C){\cdot}D$. Note that these rules are taken with respect to a given type specifications Θ and method definitions Π. Fig. 7 gives the set of transition rules. In the rule for `invoke`, $TopEnv(max_{(c,m)})$ denotes an environment of size $max_{(c,m)}$ whose elements are special constant $\bot_\top$ of a meaningless value having type $\top$, and $max_{(c,m)}$ is the maximal local variable index used in the method m defined in the class c.

$$
\begin{array}{l}
(S, E, M\{I{\cdot}B\}, D), h \\
\longrightarrow (S', E', M\{B\}, D), h' \quad (\text{if } I : (S, E), h \Longrightarrow (S', E'), h') \\
(n{\cdot}S, E, M\{\texttt{ireturn}\}, (S_0, E_0, M_0\{B_0\}){\cdot}D_0)), h \\
\longrightarrow (n{\cdot}S_0, E_0, M_0\{B_0\}, D_0), h \\
(r{\cdot}S, E, M\{\texttt{areturn}\}, (S_0, E_0, M_0\{B_0\}){\cdot}D_0)), h \\
\longrightarrow (r{\cdot}S_0, E_0, M_0\{B_0\}, D_0), h \\
(S, E, M\{\texttt{return}\}, (S_0, E_0, M_0\{B_0\}){\cdot}D_0)), h \\
\longrightarrow (S_0, E_0, M_0\{B_0\}, D_0), h \\
(0{\cdot}S, E, M\{\texttt{ifeq(}l\texttt{)}{\cdot}B\}, D), h \\
\longrightarrow (S, E, M\{M(l)\}, D), h \\
(n{\cdot}S, E, M\{\texttt{ifeq(}l\texttt{)}{\cdot}B\}, D), h \longrightarrow (S, E, M\{B\}, D), h \\
\quad (\text{if } n \neq 0) \\
(S, E, M\{\texttt{goto(}l\texttt{)}\}, D), h \longrightarrow (S, E, M\{M(l)\}, D), h \\
(v_n{\cdot}\ldots{\cdot}v_1{\cdot}r{\cdot}S, E, M\{\texttt{invoke(}c,m\texttt{)}{\cdot}B\}, D), h \\
\longrightarrow (\phi, E', M'\{M'.entry\}, (S, E, M\{B\}){\cdot}D)), h \\
\quad (\text{if } E' = TopEnv(max_{(c,m)})\{0 \leftarrow c, 1 \leftarrow \tau_1, \cdots n \leftarrow \tau_n\}, \\
\qquad \Theta.c.\texttt{methods}.m = \{\tau_1, \ldots, \tau_n\} \Rightarrow \tau, \\
\qquad \text{and } \Pi.c.m = M') \\
(S, E, M\{\texttt{ret(}i\texttt{)}\}, D), h \longrightarrow (S, E, M\{M(E(i))\}, D), h \\
(S, E, M\{\texttt{jsr(}l_1, l_2\texttt{)}\}, D), h \\
\longrightarrow (adrs(l_2){\cdot}S, E, M\{M(l_1)\}, D), h
\end{array}
$$

Figure 7: Transition Rules of the Jvmc

3.4 Type Soundness

We are now in the position to prove type soundness theorem. To do this, we define typing relations for various runtime objects used in Jvmc. Runtime values may form cycles and sharing through object pointers. To define value typing without resorting to co-induction, we follow [11] and define types of values relative to a *heap type* (ranged over by H) specifying the structure of a heap, which is a function from a finite set of heap addresses to types. Runtime values may also contain return addresses of the form $adrs(l)$ which should be typed with a block type or a subroutine block type. This requires us to define value typing relative to a label environment $\mathcal{L}$ as well. We use the following typing relations.

- $\mathcal{L} \models h : H$ $\;$ h has a heap type H
- $\mathcal{L}; H \models v : \tau$ $\;$ v has type τ under H
- $\mathcal{L}; H \models S : \Delta$ $\;$ S has a stack type Δ under H
- $\mathcal{L}; H \models E : \Gamma$ $\;$ E has an environment type Γ under H

These relations are given in Figure 8. We note that, in the case of JVM, runtime objects are explicitly typed, and therefore one can take H_h for h such that $H_h(r)$ is the runtime tag of $h(r)$. The resulting tying relation is the same as the one defined in [3].

$$
\begin{array}{l}
\mathcal{L}; H \models n : \mathsf{int} \\
\mathcal{L}; H \models r : \tau \quad (\text{if } H(r) <: \tau) \\
\mathcal{L}; H \models adrs(l) : \alpha_{l'} \\
\quad (\text{if } \mathcal{L}(l) = \alpha_{l'} \\
\qquad \text{or } \mathcal{L}(l) = \langle \alpha_{l''} // \Gamma, \Delta \rhd \tau \rangle \text{ and } \alpha_{l'} = \Gamma, \Delta \rhd \tau) \\
\mathcal{L} \models h : H \\
\quad \Leftrightarrow \quad dom(h) = dom(H) \text{ and} \\
\qquad \forall r \in dom(h).\ \text{if } h(r) = \langle f_1 = v_1, \ldots, f_n = v_n \rangle_c \\
\qquad \text{then } c <: H(r) \wedge \mathcal{L}; H \models v_i : \Theta.c.\texttt{fields}.f_i \text{ for each } i. \\
\mathcal{L}; H \models S : \Delta \\
\quad \Leftrightarrow \quad dom(S) = dom(\Delta) \wedge \mathcal{L}; H \models S.i : \Delta.i \text{ for each } i. \\
\mathcal{L}; H \models E : \Gamma \\
\quad \Leftrightarrow \quad dom(E) = dom(\Gamma) \text{ and } \mathcal{L}; H \models E.i : \Gamma.i \text{ for each } i.
\end{array}
$$

Figure 8: Typing of Runtime Values

A key to establish type soundness with respect to SECD-style operational semantics is to define typing relation on dumps. This technique is first used in [16]. We write $H \models D : \tau$ to indicate that D has type τ under H. Its intuitive meaning is that D accepts a value of type τ and resumes the saved computation. This relation is defined inductively on D in Figure 9. Using this relation, we define well typedness

- $H \models \phi : \tau$ for any τ
- $H \models (S, E, M\{B\}) \cdot D : \tau$
  $\Leftrightarrow \exists \Gamma, \Delta, \mathcal{L}, \tau'. \vdash M : \mathcal{L}, \mathcal{L}; H \models S : \Delta', \mathcal{L}; H \models E : \Gamma,$
  $\mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau'$, and $H \models D : \tau'$.
  where $\Delta' = \Delta$ if $\tau = \mathsf{void}$ otherwise $\Delta' = \tau \cdot \Delta$
- $H \models (S, E, M\{SB\}) \cdot D : \tau$
  $\Leftrightarrow \exists \Gamma, \Delta, \mathcal{L}, \tau'. \vdash M : \mathcal{L}, \mathcal{L}; H \models S : \Delta', \mathcal{L}; H \models E : \Gamma,$
  $\mathcal{L} \vdash SB : \langle \alpha_l // \Gamma; \Delta \rhd \tau' \rangle$ and $H \models D : \tau'$.
  where $\Delta' = \Delta$ if $\tau = \mathsf{void}$ otherwise $\Delta' = \tau \cdot \Delta$

Figure 9: Typing of Dump D

of a machine state including a dump as follows.

$H \vdash (S, E, M\{B\}, D), h$
$\Leftrightarrow \exists \mathcal{L}, \Gamma, \Delta$ such that $\vdash M : \mathcal{L}, \mathcal{L} \models h : H, \mathcal{L}; H \models S : \Delta,$
$\mathcal{L}; H \models E : \Gamma, \mathcal{L} \vdash \Gamma, \Delta \rhd B : \tau,$ and $H \models D : \tau$
$H \vdash (S, E, M\{SB\}, D), h$
$\Leftrightarrow \exists \mathcal{L}, \Gamma, \Delta$ such that $\vdash M : \mathcal{L}, \mathcal{L} \models h : H, \mathcal{L}; H \models S : \Delta,$
$\mathcal{L}; H \models E : \Gamma, \mathcal{L} \vdash SB : \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle,$ and $H \models D : \tau$

We can now formally state type soundness as the following theorem.

Theorem 1. *Consider a* Jvmc *program* (Π, Θ) *such that* $\vdash \Pi : \Theta$. *If* $H \vdash (S, E, M\{C\}, D), h$ *then either (1) C is one of* `return`, `ireturn`, `areturn`, *and* $D = \phi$, *or (2) there are some* $S', E', M'\{C'\}, D', h'$, *and* H' *such that* H' *is an extension of* H,

$$(S, E, M\{C\}, D), h \longrightarrow (S', E', M'\{C'\}, D'), h',$$

and $H' \vdash (S', E', M'\{C'\}, D'), h'$. *where C is B or SB.*

Proof. The proof uses following simmple lemma, which can be proved by simple case analysis.

Lemma 1. *If* $H \models v : \tau$ *and* H' *is an extension of* H *then* $H' \models v : \tau$.

We first show the cases where C is a block B. Since $\models \Pi : \Theta$, there is some $\mathcal{L}$ such that $\vdash M : \mathcal{L}$. If $H \vdash (S, E, M\{B\}, D), h$ then there is some Γ, Δ such that $\mathcal{L} \models h : H, \mathcal{L}, H \models E : \Gamma, \mathcal{L}, H \models S : \Delta, \mathcal{L} \vdash \Gamma, \Delta \rhd \tau$ and $H \models D : \tau$. The proof proceeds by cases in terms of the first instruction of B.

Case $B = \mathtt{return}$. $D = \phi$ or there is some S_1,E_1,M_1,B_1,D_1 such that $D = (S_1, E_1, M_1\{B_1\}) \cdot D_1$. The case for $D = \phi$ is trivial. We assume that $D = (S_1, E_1, M_1\{B_1\}) \cdot D_1$. By the transition rule, $(S, E, M\{\mathtt{return}\}, (S_1, E_1, M_1\{B_1\}) \cdot D_1), h \longrightarrow (p \cdot S_1, E_1, M_1\{B_1\}, D_1), h$. By the type system, $\tau = \mathsf{void}$. By the typing rule for D, there are some $\Gamma_1, \Delta_1, \mathcal{L}_1, \tau_1$ such that $\vdash M_1 : \mathcal{L}_1, \mathcal{L}_1, H \models S_1 : \Delta_1, \mathcal{L}_1 \vdash \Gamma_1, \Delta_1 \rhd B_1 : \tau_1$ and $H \models D_1 : \tau_1$.

Case $B = \mathtt{goto(}l\mathtt{)}$. By the transition rule,
$(S, E, M\{\mathtt{goto(}l\mathtt{)}\}, D), h \longrightarrow (S, E, M\{M(l)\}, D), h.$ Since $\mathcal{L}(l) = \Gamma; \Delta \rhd \tau$ by the definition of type system, $\mathcal{L} \vdash \Gamma, \Delta \rhd M(l) : \tau$.

Case $B = \mathtt{jsr(}l_1, l_2\mathtt{)}$. By the transition rule,
$(S, E, M\{\mathtt{jsr(}l_1, l_2\mathtt{)}\}, D), h \longrightarrow (adrs(l_2) \cdot S, E, M\{M(l_1)\}, D), h$. By the definition of the type system, since there are some Γ_1, Δ_1 such that $\mathcal{L}(l_1) = (\alpha_{l_1} = \Gamma_1, \Delta_1 \rhd \tau$ in $\langle \alpha_{l_1} // \Gamma, \alpha_{l_1} \cdot \Delta \rhd \tau \rangle)$, $\mathcal{L}(l_2) = \Gamma_1, \Delta_1 \rhd \tau$, we have $M(l_1) = (\alpha_{l_1} = \Gamma_1; \Delta_1 \rhd \tau$ in $\langle \alpha_{l_1} // \Gamma, \alpha_{l_1} \cdot \Delta \rhd \tau \rangle)$. Therefore we have only to show $\mathcal{L}; H \models adrs(l_2) : \alpha_{l_1}$. Since $\mathcal{L}; H \models adrs(l_2) : \mathcal{L}(l_2)$ and $\alpha_{l_1} = \mathcal{L}(l_2)$, $\mathcal{L}; H \models adrs(l_2) : \alpha_{l_1}$.

Case $B = \mathtt{new(}c\mathtt{)} \cdot B_1$. By the transition rule,
$(S, E, M\{\mathtt{new(}c\mathtt{)} \cdot B_1\}, D), h \longrightarrow (r \cdot S, E, M\{B_1\}, D), h\{r \leftarrow \langle f_1 = \bot_{\tau_1}, \ldots, f_n = \bot_{\tau_n} \rangle_c\}$, $\Theta.c.\mathtt{fields} = \{f_1 : \tau_1, \cdots, f_n : \tau_n\}$ and $r \notin h$. If we take $H' = H\{r \mapsto c\}$ then since r is fresh H' is a extension of H. Since $\mathcal{L}; H' \models \bot_{\tau_i} : \Theta.c.\mathtt{fieldes}.f_i \quad (1 \leq i \leq n)$, $\models h' : H'$. Then the result follows from Lemma 1.

Case $B = \mathtt{getfield(}c, f\mathtt{)} \cdot B_1$. By the definition of type system, there are some τ_0, c_0, Δ' such that $\Delta = \tau_0 \cdot c_0 \cdot \Delta'$, $c_0 <: c_1$, $\tau_0 <: \Theta.c.\mathtt{fields}.f$. Consequently, there are some r_0, s' such that $S = r_0 \cdot S', \mathcal{L}; H \models r_0 : c_0, \mathcal{L}; H \models S' : \Delta'$. By the transition rule, $(r_0 \cdot S', E, M\{\mathtt{getfield(}c, f\mathtt{)} B_1 \cdot\}, D) \longrightarrow (v_1 \cdot S', E, M\{B_1\}, D_1), h$ and $v_1 = h(r_0).f$. By the definition of the type system, there is some τ_1 such that $\mathcal{L} \vdash \Gamma; \tau_1 \cdot \Delta' \rhd B_1 : \tau$ and $\tau_1 = \Theta.c.\mathtt{fields}.f$. Since $c_0 <: c$, $h(r_0).f = \Theta.c.\mathtt{fields}.f$, and therefore $\mathcal{L}; H \models v_1 : \tau_1$.

Case $B = \mathtt{invoke(}c, m\mathtt{)} \cdot B_1$. By the definition of the type system, there are some $\tau_1, \cdots, \tau_n, c_0, \Delta'$ such that $\Delta = \tau_n \cdot \ldots \cdot \tau_1 \cdot c_0 \cdot \Delta'$. Also, there are some $v_n, \cdots, v_n, v_0, S'$ such that $\mathcal{L}; H \models \tau_n \cdot \ldots \cdot \tau_1 \cdot c_0 \cdot \Delta' : v_n \cdot \ldots \cdot v_1 \cdot r \cdot S' \wedge S = v_n \cdot \ldots \cdot v_1 \cdot r \cdot S'$ Then we have: $(v_n \cdot \ldots \cdot v_1 \cdot r \cdot S', E, M\{\mathtt{invoke(}c, m\mathtt{)} \cdot B_1\}, D), h \longrightarrow (\phi, E_1, M_1\{M_1.e\}, (S', E, M\{B_1\}) \cdot D), h$ such that $E_1 = TopEnv(max(c, m))\{0 \mapsto r, 1 \mapsto v_1, \cdots, n \mapsto v_n\}$, and $\Theta.c.\mathtt{methods}.m = \{\tau_1, \cdots, \tau_n\} \Rightarrow \tau'$. We distinguish cases whether τ' is void or not. Here we only show that se for $\tau' = \mathsf{void}$. The other case is similar. Since $\mathcal{L} \vdash \Gamma, \Delta' \rhd B_1 : \tau$ by the definition of type system, $H \models (S', E, M\{B_1\}) \cdot D : \mathsf{void}$. By $\vdash \Pi : \Theta$, there is some $\mathcal{L}_1$ such that $\vdash M_1 : \mathcal{L}_1, \mathcal{L}_1 \vdash Top(max_e)\{0 \mapsto c_0, 1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}, \phi \rhd M_1.entry : \mathsf{void}$. Due to the definition, $\mathcal{L}_1; H_1 \models TopEnv(max(c, m))$ $\{0 \mapsto r, 1 \mapsto v_1, \cdots, n \mapsto v_n\} : Top(max_e)\{0 \mapsto c, 1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}$.

The other cases for blocks are simpler.

The cases when C is a subroutine blocks can be shown similary by case analysis in terms of the first instruction.

We only show the case for $SB = \mathtt{jsr(}l_1, l_2\mathtt{)}$. By the definition of type system, $\mathcal{L} \vdash \mathtt{jsr(}l_1, l_2\mathtt{)} : \langle \alpha_l // \Gamma; \Delta \rhd \tau \rangle$ and $\mathcal{L}(l_1) = (\alpha_{l_1} = \Gamma_1; \Delta_1 \rhd \tau$ in $\langle \alpha_{l_1} // \Gamma; \alpha_{l_1} \cdot \Delta \rhd \tau \rangle)$, $\mathcal{L}(l_2) = \langle \alpha_l // \Gamma_1; \Delta_1 \rhd \tau \rangle$. If $\vdash M : \mathcal{L}, \mathcal{L} \models h : H, \mathcal{L}; H \models S : \Delta, \mathcal{L}; H \models E : \Gamma$, and $H \models D : \tau$ then we have $H \vdash (S, E, M\{\mathtt{jsr(}l_1, l_2\mathtt{)}\}, D), h$. By transition rule,
$(S, E, M\{\mathtt{jsr(}l_1, l_2\mathtt{)}\}, D), h \longrightarrow (adrs(l_2) \cdot S, E, M\{M(l_1)\}, D), h$. Since $\mathcal{L} \vdash M(l_1) : \mathcal{L}(l_1)$, we have only to show $\mathcal{L}; H \models adrs(l_2) : \alpha_{l_1}$. This follows from the above equations for $\mathcal{L}(l_2)$ and $\mathcal{L}(l_1)$. $\square$

This theorem implies that a well typed machine state is either the halting state or a state such that the machine can execute one step transition and produce another well typed machine state. This immediately guarantees that a well typed program never goes wrong, and when the machine terminates, the top element of the stack is a value of correct type specified by the type of the program.

4. POLYMORPHISM AND TYPE INFERENCE

In order to use the type system we have developed as a framework for static verification of type safety of Jvmc code, two further extensions are necessary. One is polymorphic typing of subroutines. The other is the development of a type inference algorithm.

4.1 Polymorphic typing for subroutines

As observed in [17], many useful JVM subroutines are polymorphic with respect to the unused portions of local variable environments. In addition, subroutines can be polymorphic with respect to the data they manipulate. Using typing mechanisms developed for functional languages including ML's polymorphic let, bounded quantification, and record polymorphism, we can define various polymorphic typing of varied strength. The desired strength should be determined in consideration of the feasibility of type inference. Since the JVM code language is implicitly typed in their use of labels, type-checking requires type reconstruction. These two aspect are closely connected. A detailed study of polymorphism and type inference algorithm is beyond the scope of the present article, and we shall present it in a separate paper. In this paper, we present a simple yet powerful version of polymorphic typing and a type inference algorithm which we have implemented.

We regard subroutines as mutually recursive polymorphic function definitions, and treat subroutine labels as *let-bound variables*. We regard a JVMC program $\{l^b_1 : B_1, \dots, l^b_n : B_n \mid l^s_1 : SB_1, \dots, l^s_k : SB_k\}$ as the following

$$\texttt{letrec } l^s_1 \texttt{ = } SB_1 \cdots \texttt{ and } l^s_1 \texttt{ = } SB_k \texttt{ in}$$
$$\texttt{in rec } l^b_1 \texttt{ = } B_1 \cdots \texttt{ and } l^k_1 \texttt{ = } B_k \texttt{ end}$$

where `letrec` is polymorphic binding similar to ML's mutually recursive function definition, and `rec` is monomorphic recursive binding.

Polymorphic types are introduced through type variables (ranged over by t) possibly bounded by a class name. Due to the explicit typing of JVM method invocation, this limited form of bounded quantification [2] is sufficient for wide range of JVM code. We let $\mathcal{K}$ be *a bound environment of type variables* assigning to a type variable t its *bound*, which is either a class name or "$*$" indicating that t has no constraint. We write $\mathcal{K} \vdash \tau <: c$ to indicate that τ is a subclass of a class c under $\mathcal{K}$. We also use *stack type variable* δ and *environment type variables* γ. They corresponds to Stata and Abadi's proposal [17] of ignoring the unused part of an environment in their constraint checking system. O'Callahan [15] proposes to make types of labels polymorphic. However, as he already observed, the resulting system is undecidable. Our proposal achieves just enough polymorphism to type JVM subroutine without incurring the undecidability problem.

To incorporate this form of polymorphism, we extend types, environment types, and stack types as follows.

- $\tau ::= t \mid \mathsf{int} \mid \mathsf{void} \mid c \mid \alpha_l \mid \top$
- $\Gamma ::= \phi \mid \gamma \mid \tau \cdot \Gamma$
- $\Delta ::= \phi \mid \delta \mid \tau \cdot \Delta$

The syntax of a method body M is a compound term `let` M^s `in` M^b of a subroutine body M^s and a block body M^b whose syntax are given below.

- $M^s ::= \{l^s_1 = SB_1, \dots, l^s_n = SB_n\}$
- $M^b ::= \{l^b_1 = B_1, \dots, l^b_k = B_k\}$

In accordance with this, we divide a label environment $\mathcal{L}$ into a subroutine label environment $\mathcal{L}^s$ and a block label environment $\mathcal{L}^b$. Subroutine types in $\mathcal{L}^s$ can be polymorphic types obtained by generalizing type variables. Let σ be a subroutine type. We write $Cls(\mathcal{K}, \sigma)$ for the subroutine type obtained from σ by generalizing all the type variables with their bounds indicated in $\mathcal{K}$. For example,

$$Cls(\{t \mapsto c\}, \langle \alpha_l // t \cdot \gamma, \alpha_l \cdot \delta \rhd \mathsf{int} \rangle) = \forall t < c.\forall\delta.\forall\gamma.\langle \alpha_l // t \cdot \gamma, \alpha_l \cdot \delta \rhd \mathsf{int} \rangle$$

We also write $Cls(\mathcal{K}, \mathcal{L}^s)$ for $\{l : Cls(\mathcal{K}, \mathcal{L}^s(l)) | l \in dom(\mathcal{L}^s)\}$.

Typing rules for this refined language is obtained by making the following modification to the type system defined before.

1. Each typing judgment is refined by incorporating a bound environment $\mathcal{K}$. For example, a judgment of the form $\Gamma, \Delta \rhd B : \tau$ becomes $\mathcal{K}, \Gamma, \Delta \rhd B : \tau$.

2. The rules for instructions of object manipulation are refined so that they can be polymorphic with respect to possible subclasses. The rules for instructions `getfield`, `putfield`, `invoke` are given as follows.

$$\begin{array}{l}
\mathcal{K} \vdash \texttt{getfield}(c_0, f) : \Gamma, c_1 \cdot \Delta \Longrightarrow \Gamma, \Theta.c_0.\texttt{fields}.f \cdot \Delta \\
\quad (\text{if } \mathcal{K} \vdash c_1 <: c_0) \\
\mathcal{K} \vdash \texttt{putfield}(c_0, f) : \Gamma, \tau \cdot c_1 \cdot \Theta \Longrightarrow \Gamma, \Delta \\
\quad (\text{if } \mathcal{K} \vdash c_1 <: c_0 \ \wedge \ \mathcal{K} \vdash \tau <: \Theta.c_1.\texttt{fields}.f) \\
\mathcal{K} \vdash \texttt{invoke}(c_0, m) : \Gamma, \tau_n \cdot \dots \tau_1 \cdot c_1 \cdot \Delta \Longrightarrow \Gamma, \tau_0 \cdot \Delta \\
\quad (\text{if } \Theta.c_0.\texttt{methods}.m = \{\tau'_1, \cdots, \tau'_n\} \Longrightarrow \tau_0, \tau_0 \neq \mathsf{void} \\
\qquad \text{and } \ \mathcal{K} \vdash c_1 <: c_0 \ \wedge \mathcal{K} \vdash \tau_i <: \tau'_i \ \text{ for all } 1 \leq i \leq n) \\
\mathcal{K} \vdash \texttt{invoke}(c_0, m) : \Gamma, \tau_n \cdot \dots \tau_1 \cdot c_1 \cdot \Delta \Longrightarrow \Gamma, \Delta \\
\quad (\text{if } \Theta.c_0.\texttt{methods}.m = \{\tau'_1, \cdots, \tau'_n\} \Longrightarrow \mathsf{void} \\
\qquad \text{and } \ \mathcal{K} \vdash \tau_1 <: c_0 \ \wedge \mathcal{K} \vdash \tau_i <: \tau'_i \ \text{ for all } 1 \leq i \leq n)
\end{array}$$

Calling a polymorphic subroutine by `Jsr` requires instantiation of its type variables. This is represented by the following refined typing rule.

$$\begin{array}{l}
\mathcal{L} \vdash \Gamma_1, \Delta_1 \rhd \texttt{jsr}(l_1, l_2) : \tau \\
\quad (\text{if } \mathcal{L}(l_1) \geq (\alpha_{l_1} = \Gamma_2, \Delta_2 \rhd \tau \text{ in } \langle \alpha_{l_1} // \Gamma_1, \alpha_{l_1} \cdot \Delta_1 \rhd \tau \rangle), \\
\qquad \mathcal{L}(l_2) = \Gamma_2; \Delta_2 \rhd \tau)
\end{array}$$

In this definition, $A \geq B$ denotes that A is a monomorphic type obtained from B by instantiating its type variables.

3. The rules for blocks and programs are given as follows.

$$\frac{\mathcal{K} \vdash M^s : \mathcal{L}^s \ \ Cls(\mathcal{K}, \mathcal{L}^s) \vdash M^b : \mathcal{L}^b}{\mathcal{K} \vdash \texttt{let } M^s \texttt{ in } M^b : \mathcal{L}^b}$$

$$\begin{array}{l}
\mathcal{K}, \mathcal{L}^s \vdash M^b : \mathcal{L}^b \\
\Leftrightarrow \ \ dom(M^b) = dom(\mathcal{L}^b) \text{ and} \\
\quad \text{for any } l \in dom(M^b),\ \mathcal{L}^s \cup \mathcal{L}^b \vdash \Gamma, \Delta_l \rhd M\{M(l)\} : \tau_l
\end{array}$$

4.2 Type inference algorithm

The resulting type system combines ML-style polymorphism and (limited form of) subtype polymorphism. For this type system, we develop a type inference algorithm, which allows us to verify the type safety of JVMC. The algorithm is given in the style of $\mathcal{W}$ [14] with the following treatment specific to JVMC.

In inferring a typing for instructions of object manipulation including `invoke`, `getfield`, and `putfield`, the type

τ of an argument in a stack and the class type c specified in the code must be matched modulo the subclass relation. Let $\mathcal{K}$ be a given bound environment. If $\mathcal{K} \vdash t <: c$ does not hold and $\mathcal{K}(t)$ is either $*$ or some c' such that $c <: c'$ then we change the current bound environment from $\mathcal{K}$ to $\mathcal{K}'$ by modifying t's bound to c. This subclass checking procedure is written as $\mathsf{SubUnify}(\mathcal{K}, C) = \mathcal{K}'$, which takes a set of subclass constraints (of the form $\tau <: c$) and returns a modified bound environment that satisfies the set.

The unification algorithm Unify is also refined to incorporate bound constraints. It takes a set E of pairs of types and a bound environment $\mathcal{K}$ and returns a substitution S such that S is a unifier of E and for all t in $dom(\mathcal{K})$, $\mathcal{K} \vdash S(t) <: \mathcal{K}(t)$. The only necessary refinement is to check the bound constraints and generate a substitution only if the result type is a subtype of the bound. The type inference algorithm Unify is defined as the follows

$$\mathsf{Unify}(E, \mathcal{K}) = \begin{cases} S & ((E, \phi, \mathcal{K}) \Longrightarrow^* (\phi, S, \mathcal{K})) \\ failure & \text{(otherwise)} \end{cases}$$

where $\Longrightarrow^*$ is the reflective transitive closure of the relation $\Longrightarrow$ given below:

$$\begin{array}{l}
(E \cup \{(\tau, \tau)\}, S, \mathcal{K}) \Longrightarrow (E, S, \mathcal{K}) \\
(E \cup \{(t, c)\}, S, \mathcal{K}) \Longrightarrow ([c/t]E, \{(t, c) \cup [c/t]S, \mathcal{K}) \\
\quad \text{(if } c <: \mathcal{K}(t) \text{ or } \mathcal{K}(t) = *) \\
(E \cup \{(t_1, t_2)\}, S, \mathcal{K}) \Longrightarrow ([t_2/t_1]E, \{(t_1, t_2) \cup [t_2/t_1]S, \mathcal{K}) \\
\quad \text{(if } \mathcal{K}(t_1) = * \text{ or } \mathcal{K}(t_2) <: \mathcal{K}(t_1)) \\
(E \cup \{(t_1, t_2)\}, S, \mathcal{K}) \Longrightarrow ([t_1/t_2]E, \{(t_2, t_1) \cup [t_1/t_2]S, \mathcal{K}) \\
\quad \text{(if } \mathcal{K}(t_2) = * \text{ or } \mathcal{K}(t_1) <: \mathcal{K}(t_2)) \\
(E \cup \{(t, \tau)\}, S, \mathcal{K}) \Longrightarrow ([\tau/t]E, \{(t, \tau) \cup [\tau/t]S, \mathcal{K}) \\
\quad \text{(if } \mathcal{K}(t) = *)
\end{array}$$

Unify is also extended to imcorporate environment type variables and stack type variables. Unify treats type variables of the form α_l introduced for each subroutine entry as a unique constant type similar to datatypes in ML.

Using these, we can define a type inference algorithm. Figure 10 shows the main algorithm $\mathcal{WJ}$ It first sets up a skeleton of a subroutine label environment $\mathcal{L}_s$, and infers each subroutine typing using an algorithms $\mathcal{WS}$. It next obtains the correct subroutine label environment $\mathcal{L}_s$ by applying the solution substitutions followed by type generalization. It then sets up a skeleton of a block label environment $\mathcal{L}_b$ incorporating the method type constraint of the entry label declared in a class type specifications Θ, and infers block typings under $\mathcal{L}_s$ using an algorithm $\mathcal{WB}$ to obtain the correct label environment $\mathcal{L}^b$ of the program. In Figure 11 and Figure 12, we only show some cases of the type inference algorithms $\mathcal{WS}$ and $\mathcal{WB}$ for blocks and subroutine blocks. The other cases are easily added. In these algorithm, $\mathcal{K}$ is used as a global environment imperatively modified and sequentially accessed through the entire algorithm. We also assume that any class is a subclass of `Object` class.

5. IMPLEMENTATION AND AN EXAMPLE OUTPUT

We have just finished an implementation of the type inference algorithm. Our implementation takes a JVM class file and a method name with its type, extracts a code sequence

```
$\mathcal{WS}(\mathcal{L}, (\alpha_l = \Gamma_1, \Delta_1 \rhd \tau \text{ in } \langle \alpha_l // \Gamma_2, \Delta_2 \rhd \tau \rangle), \texttt{ret(}i\texttt{)})$
  $= \mathsf{Unify}\{(\Gamma_2(i), \alpha_l), (\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2)\}$
$\mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle, \texttt{areturn}) = \text{let } \mathcal{K} = \mathcal{K}\{t \mapsto \texttt{Object}\}$
                                              $\text{in } \mathsf{Unify}\{(\Delta, t \cdot \delta), (\tau, t)\}$
$\mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma_2, \Delta_2 \rhd \tau \rangle, \texttt{goto(}l'\texttt{)} \cdot SB) =$
  $\text{let } \langle \alpha_{l'} // \Gamma_2', \Delta_2' \rhd \tau \rangle = \mathcal{L}(l')$
  $\text{in if } l = l' \text{ then } \mathsf{Unify}\{(\Gamma_2, \Gamma_2'), (\Delta_2, \Delta_2')\}$
    $\text{else } failure$
$\mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle, \texttt{iconst(}n\texttt{)} \cdot SB)$
  $= \mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma, \mathsf{int} \cdot \Delta \rhd \tau \rangle, SB)$
$\mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle, \texttt{aload(}i\texttt{)} \cdot SB) =$
  $\text{let } \mathcal{K} = \mathcal{K}\{t \mapsto \texttt{Object}\}$
      $S_1 = \mathsf{Unify}\{(\Gamma(i), t)\}$
      $S_2 = \mathcal{WS}(S_1(\mathcal{L}), S_1(\langle \alpha_l // \Gamma, t \cdot \Delta \rhd \tau \rangle), SB)$
  $\text{in } S_2 \circ S_1$
$\mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle, \texttt{dup} \cdot SB) =$
  $\text{let } \mathcal{K} = \mathcal{K}\{t \mapsto *\}$
      $S_1 = \mathsf{Unify}\{(\Delta, t \cdot \delta)\}$
      $S_2 = \mathcal{WS}(S_1(\mathcal{L}), S_2(\langle \alpha_l // \Gamma, t \cdot \Delta \rhd \tau \rangle), SB)$
  $\text{in } S_2 \circ S_1$
$\mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle, \texttt{iadd} \cdot SB) =$
  $\text{let } S_1 = \mathsf{Unify}\{(\Delta, \mathsf{int} \cdot \mathsf{int} \cdot \delta)\}$
      $S_2 = \mathcal{WS}(S_1(\mathcal{L}), S_2(\langle \alpha_l // \Gamma, \mathsf{int} \cdot \delta \rhd \tau \rangle), SB)$
  $\text{in } S_2 \circ S_1$
$\mathcal{WS}(\mathcal{L}, \langle \alpha_l // \Gamma, \Delta \rhd \tau \rangle, \texttt{getfield(}c, f\texttt{)} \cdot SB) =$
  $\text{let } \mathcal{K}\{t \mapsto c\}$
      $S_1 = \mathsf{Unify}\{(\Delta, t \cdot \delta)\}$
      $S_2 = \mathcal{WS}(S_1(\mathcal{L}), S_2(\langle \alpha_l // \Gamma, \delta \rhd \tau \rangle), SB)$
  $\text{in } S_2 \circ S_1$
```

Figure 11: Type inference Algorithm for subroutines (excerpts)

of the method and decomposes it into a collection of code blocks and subroutine blocks. It then infers typing of each blocks. The final result is the label environment.

We show below some example outputs of our type inference system. The first one is the code obtained from compiling the following sample Java program.

```
class C {
    int foo(A a,B b) {
        int sum;
        try { } finally {
                sum = a.bar(3) + b.bar(2);
                if (sum == 0) return 1;}
        return sum; }
}
class A { int bar(int n) { return n; } }
class B extends A { }
```

This program consists of three classes `A`, `B`, and `C`. Note that the method `foo` in class `sample` has a `finally` clause containing method invocation.

From the compiled JVM code of this program, our type inference system generates the following collection of code blocks. (In this version, we ignored exception handler, generated by the compiler.)

```
Entry:  jsr(S1,L1)

L1:     goto(L2)
```

```
$\mathcal{WJ}(\{l_1^s = SB_1(l_1'), \ldots, l_k^s = SB_k(l_k'), l_1^b = B_1, \ldots, l_n^b = B_n\})$
  let $\mathcal{K} = \phi$
      $\mathcal{S}_i = (\alpha_{l_i'} = t_{i_1}^1 \cdot \ldots \cdot t_{i_{max}}^1 \cdot \phi, \delta_i^1 \rhd t$ in $\langle \alpha_{l_i'} // t_{i_1}^2 \cdot \ldots \cdot t_{i_{max}}^2 \cdot \phi, \delta_i^2 \rhd t \rangle)$   $(1 \leq i \leq k)$
      $\mathcal{K} = \mathcal{K}\{t_{i_1}^1 \mapsto *, \cdots, t_{i_{max}}^1 \mapsto *, t_{i_1}^2 \mapsto *, \cdots, t_{i_{max}}^2 \mapsto *, t \mapsto *\}$  $(1 \leq i \leq k)$
      $\mathcal{L} = \{l_1^s = \mathcal{S}_1, \ldots, l_k^s = \mathcal{S}_k\}$
      $S_0 =$ the empty substitution
      $S_i = (\mathcal{WS}(S_{i-1}(\mathcal{L}), S_{i-1}(\mathcal{S}_i), SB_i)) \circ S_{i-1}$  $(1 \leq i \leq k)$   (* infer subroutine types *)
      $\mathcal{B}_{entry} = \tau_1 \cdot \ldots \cdot \tau_n \cdot \top_{n+1} \cdot \ldots \cdot \top_{i_{max}} \cdot \phi, \phi \rhd t_0$    (where $\Theta.c.\texttt{methods}.l_{entry} = \{\tau_1, \ldots, \tau_n\} \Rightarrow \tau_0$)
      $\mathcal{B}_i = t_{i_1}^1 \cdot \ldots \cdot t_{i_{max}}^1 \cdot \phi, \delta_i^1 \rhd t_i$  $(1 \leq i \leq n)$
      $\mathcal{K} = \mathcal{K}\{t_{i_1} \mapsto *, \cdots, t_{i_{max}} \mapsto *, t \mapsto *, t_0 \mapsto \tau_0\}$  $(1 \leq i \leq n)$
      $\mathcal{L}' = \{l_1^s = Cls(\mathcal{K}, S_k(\mathcal{S}_1)), \ldots, l_k^s = Cls(\mathcal{K}, S_k(\mathcal{S}_k)), l_{entry} = \mathcal{B}_{entry}, l_1^b = \mathcal{B}_1, \ldots, l_n^b = \mathcal{B}_n\}$
      $S_0' =$ the empty substitution
      $S_i' = (\mathcal{WB}(S_{i-1}'(\mathcal{L}'), S_{i-1}'(\mathcal{B}_i), B_i)) \circ S_{i-1}'$  $(1 \leq i \leq k)$   (* infer block types *)
  in  $S_n'(\mathcal{L}')$
$i_{max}$ is the maximal number of local variable index used in the method.
```

Figure 10: The main type inference algorithm $\mathcal{WJ}$

```
$\mathcal{WB}(\mathcal{L}, \Gamma, \Delta \rhd \tau, \texttt{jsr(}l_1\texttt{,}l_2\texttt{)}) =$
  let $(\alpha_{l_1} = \Gamma_1, \Delta_1 \rhd \tau_1$ in $\langle \alpha_{l_1} // \Gamma_2, \Delta_2 \rhd \tau_2 \rangle)$
          $= FreshInst(L(l_1))$
      $\Gamma', \Delta' \rhd \tau' = L(l_2)$
  in  $\mathsf{Unify}\{(\Gamma, \Gamma_2), (\alpha_{l_1} \cdot \Delta, \Delta_2), (\Gamma', \Gamma_1), (\Delta', \Delta_1),$
      $(\tau, \tau_2), (\tau', \tau_1)\}$
$\mathcal{WB}(\mathcal{L}, \Gamma, \Delta \rhd \tau, \texttt{iconst(}n\texttt{)} \cdot B) = \mathcal{WB}(\mathcal{L}, \Gamma, \mathsf{int} \cdot \Delta \rhd \tau, B)$
$\mathcal{WB}(\mathcal{L}, \Gamma, \Delta \rhd \tau, \texttt{astore(}i\texttt{)} \cdot B) =$
  let $\mathcal{K}\{t \mapsto *\}$
      $S_1 = \mathsf{Unify}\{(\Delta, t, \delta \cdot)\}$
      $S_2 = \mathcal{WB}(S_1(\mathcal{L}), S_1(\Gamma\{i \leftarrow t\}, \delta \rhd \tau), B)$
  in $S_2 \circ S_1$
$\mathcal{WB}(\mathcal{L}, \Gamma, \Delta \rhd \tau, \texttt{pop} \cdot B) =$
  let $\mathcal{K}\{t \mapsto *\}$
      $S_1 = \mathsf{Unify}\{(\Delta, t \cdot \delta)\}$
      $S_2 = \mathcal{WB}(S_1(\mathcal{L}), S_1(\Gamma, \delta \rhd \tau), B)$
  in  $S_2 \circ S_1$
$\mathcal{WB}(\mathcal{L}, \Gamma, \Delta \rhd \tau, \texttt{putfield(}c, f\texttt{)} \cdot B) =$
  let $\mathcal{K}\{t_1 \mapsto \Theta.c.\texttt{fields}.f, t_2 \mapsto c\}$
      $S_1 = \mathsf{Unify}\{(\Delta, t_1 \cdot t_2 \cdot \delta)\}$
      $S_2 = \mathcal{WB}(S_1(\mathcal{L}), S_1(\Gamma, \delta \rhd \tau), B)$
  in  $S_2 \circ S_1$
$\mathcal{WB}(\mathcal{L}, \Gamma, \Delta \rhd \tau, \texttt{invoke(}c\texttt{,}m\texttt{)} \cdot B) =$
  let $c \cdot \tau_1 \cdot \cdots \cdot \tau_n \cdot \phi \Rightarrow \tau_0 = \mathcal{C}.c.\texttt{methods}.m$
      $\mathcal{K} = \mathcal{K}\{t_0 \mapsto *, t_1 \mapsto *, \cdots, t_n \mapsto *\}$
      $S_1 = \mathsf{Unify}\{t_n \cdot \ldots \cdot t_1 \cdot t_0 \cdot \delta, \Delta\}$
      $\mathcal{K} = \mathsf{SubUnify}(S_1(t_n) \cdot \ldots \cdot S_1(t_1) \cdot S_1(t_0), \tau_n \cdot \ldots \cdot \tau_1 \cdot c)$
      $S_2 =$ if $\tau_0 \neq \mathsf{void}$ then $\mathcal{WB}(S(\mathcal{L}), S(\Gamma, \tau_0 \cdot \delta \rhd \tau), B)$
              else $\mathcal{WB}(S_1(\mathcal{L}), S_1(\Gamma, \delta \rhd \tau), B)$
  in  $S_2 \circ S_1$
```

Figure 12: Type inference algorithm for code blocks (excerpts)

```
L2:     iload(3)
        ireturn

S1:     astore(4)
        aload(1)
        iconst(3)
        invokevirtual(A,bar,(I)I)
        aload(2)
        iconst(2)
        invokevirtual(A,bar,(I)I)
        iadd
        istore(3)
        iload(3)
        ifne(S2)
        iconst(1)
        ireturn

S2:     ret(4)
```

It then infers the label environment of Figure 13. In this output, `'a` etc are type variables and `('a<A)` indicates bounded quantification of the form $\forall$`'a` $<: A$.

The subroutine `S1` is given a bounded polymorphic type reflecting the property that it invokes a method. This example can be type-check with only subtyping. However, the following example bytecode requires parametric polymorphism.

```
L2:     invokevirtual(A,bar,(LA;)I)
        ireturn

L1:     iadd
        pop
        aload_(1)
        jsr(S1,L2)

Entry:  iconst_n(1)
        jsr(S1,L1)

S1:     astore_(2)
        dup
        ret(2)
```

For this, our system infers the typing of Figure 14. There does not seem to exist any other type verification system (including Sun's verifier) that can deal with both examples. Even although the practical usefulness of this generality may not be so obvious, it reflects the general type theoretical nature of our approach.

6. EXTENSIONS AND TREATMENT OF OTHER JVM FEATURES

```
Entry : {CL(C),CL(A),CL(B),-,-}{} => I
L1    : {CL(C),CL(A),CL(B),I,r(S1)}{} => I
L2    : {CL(C),CL(A),CL(B),I,r(S1)}{} => I
S1    : (Ret(S1),{('a<*),('b<A),('c<A),I,Ret(S1)}{'L} => I
              \\ {('a<*),('b<A),('c<A),('e<*),('f<*)}{Ret(S1),'L} => I)
S2    : (Ret(S1),{('a<*),('b<A),('c<A),I,Ret(S1)}{'L} => I
              \\ {('a<*),('b<A),('c<A),I,Ret(S1)}{'L} => I)
```

Figure 13: Infered label environment(1)

```
L2     : {CL(C),CL(A),ret(S1)}{CL(A),CL(A)} => I
L1     : {CL(C),CL(A),ret(S1)}{I,I} => I
Entry  : {CL(C),CL(A),-}{} => I
S1     : (R(S1),{('m<*),('n<*),R(S1)}{('q<*),('q<*),'F} => ('i<*),
             \\ {('m<*),('n<*),('o<*)}{R(S1),('q<*),'F} => ('i<*))
```

Figure 14: Infered label environment(2)

Our claim is that type theoretical framework for presenting a JVM-style bytecode language as a typed calculus allows us to extend the language with various advanced features well studied in type systems of programming languages. Among them the following two are particularly important.

- *Higher-order methods.* We believe that higher-order methods or first-class code blocks should be useful in JVM-style languages. We can easily extend our type system to include first-class code blocks by introducing a type of the form $\Gamma, \Delta \rhd \tau$. Types of first-class methods are special cases where Δ is empty. All the necessary typing mechanisms already exist in our framework, and the type soundness and type verification go through without much new addition.

- *Polymorphism and Type Inference.* Another important addition is polymorphism and type inference. We have so far followed JVM and required that every method is explicitly and monomorphically typed. However, our type inference algorithm is a full-fledged one capable of inferring most general typing of a given program. This implies that if we can extend the definition properly, we obtain a language with polymorphic type inference. Designing such a language requires a careful analysis on the interaction between (parametric) polymorphism subtyping. We are currently designing a polymorphic bytecode language to be used as a common intermediate language.

Several features of JVM that are not covered by the type system and the type inference system we have defined above. We discuss some of them below.

- *Static method.* In addition to those ordinary methods, JVM also support static methods, which are those methods that are associated to a class and are invoked independent of any object. One way to add static method to our type system is to introduce a new typing rule for `invokestatic` which does not take an object argument used through `this` variable, and a restricted type system $\mathcal{L} \vdash_c \Gamma, \Delta \rhd C : \tau$ that prohibits invoking methods in the same class c.

- *Interface classes.* This can be incorporated in our type system by introducing additional structures in Θ for interface declarations. Since interface classes do not have any code, their consistency checking can be done independently of the definition of typing derivation for blocks we have defined. The required procedure to check consistence of Θ including class method can be adopted from existing works such as [3].

- *Overloading.* JVM includes a simple mechanism for overloading based on machine of argument types. Although we believe that it can be added to our system without much difficulty, we have not consider introducing them. As we discuss above, our framework scales up to polymorphism and type inference. Overloading appears to be relate these features. We leave it a future work to design a flexible type system for JVM-style bytecode including polymorphism and overloading.

There are other features of JVM we have not addressed, including *threads*, *exceptions*, *arrays* and *overloading.* These are beyond the scope of the present paper and we leave them for future works.

7. CONCLUSIONS

We have developed a typed calculus for Java Virtual Machine bytecode, and have shown the type soundness with respect to an operational semantics. We have developed and implemented a type inference algorithm and shown that type verification of JVM code is done by ML-style polymorphic type inference.

In contrast to the existing approaches, our type system yields a typed calculus of JVM code blocks just as in type theory of the lambda calculus. This property yields a simple type soundness theorem and allows us to extend the language with polymorphism and type inference.

In addition to the features presented in this abstract, we have also worked out some other features of JVM including *static method, exception*, and *interface classes.* We have also considered several extensions to JVM languages including *higher-order methods* and more refined block-level polymorphism.

8. REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transcations on Programming Languages and Systems*, 13(2):237–268, 1991.

[2] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surverys*, 17(4):471–522, December 1985.

[3] Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–166, 1999.

[4] Stephen N. Freund and John C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[6] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine Subroutines. In *Proceedings of Static Analysis Symposium*, pages 17–32. Springer-Verlag, Berlin Germany, 1998.

[7] Futoshi Iwama and Naoki Kobayashi. A new type system for JVM lock primitives. In *Proceedings of ASIAN PEPM'02.* ACM Press, 2002.

[8] M. Jones. The functions of java bytecode. In *Proc. OOPSLA '98 workshop on Formal Underpinnings of Java*, 1998.

[9] S. Katsumata and A. Ohori. Proof-directed de-compilation of low-level code. In *European Symposium on Programming, Springer LNCS 2028*, pages 352–366, 2001.

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

[11] X. Leroy. *Polymorphic typing of an algorithmic language.* PhD thesis, University of Paris VII, 1992.

[12] Xavier Leroy. On-card bytecode verification for Java card. In *Proceedings of e-Smart 2001, LNCS 2140.*, pages 150–164, 2001.

[13] T. Lindholm and F. Yellin. *The Java virtual machine specification.* Addison Wesley, second edition edition, 1999.

[14] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[15] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 70–78, 1999.

[16] A Ohori. The logical abstract machine: a Curry-Howard isomorphism for machine code. In *Proceedings of International Symposium on Functional and Logic Programming*, 1999.

[17] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 149–160, New York, NY, 1998.