A logical analysis for Java Virtual Machine and its application to implementation

Tomoyuki Higuchi (010093)

School of Information Science,
Japan Advanced Institute of Science and Technology

February 15, 2002

Keywords: Java virtual machine,bytecode verifier,type system,type inference.

1 Background and aims

JAVA has became one of most widely used programming language due to its strong support for network computing. A JAVA program is compiled to Java bytecode which is executed by the Java virtual machine. This structure achieves architecture independent network programming. Moreover, JVM contains a *bytecode verifier* which ensure security of a program by examining the compiled bytecode sequence and detecting various inconsistencies before its execution.

Despite its importance, however, the specification of the verifier is written in English admitting certain degree of ambiguity and its mathematical correctness is not proven.

To resolve this problem, some researchers have attempted to formalize the JVM verifier. The most notable one is the work of Stata and Abadi, where they defined a type system for Java bytecode to analyze the bytecode verifier. Since it, various type systems based on their work have been proposed.

Although thease type systems are useful for verifying correctness of program, their various formal properties are not yet well understood. In particular, their relationship to the well established type theory of the lambda calculus. As a consequence, useful concepts and techniques developed in the lambda calculus are not directly applicable.

The purpose of this thesis is to analyze Java bytecode and to establish type theoretical basis for design and implementation of Java bytecode verifiers.

2 The contributions of the thesis

To achieve the purpose, we hava solved the following problems.

1. We define the a type system for Java bytecode based on the proof system called sequential sequent calculus and prove the type soundness.

2. We show that bytecode verifier can be formalized by type inference algorithm.

3. This type inference algorithm is implemented and compared with Sun's bytecode verifier.

In the following, we outline each of them.

2.1 The type sysmte for Java bytecode

Sequential sequent calculus is a proof system which corresponds to machine code. In this calculus, each instruction I of machine code is represented as an inference rule of the following form:

$$\frac{\Delta_2 \rhd I : \tau}{\Delta_1 \rhd I \cdot C : \tau}$$

where C is a code sequence and Δ indicates machine memory. This typing rule represents the property that I changes machine memory Δ_1 to Δ_2. A return instruction corresponds to an initial sequent of the form $\tau \cdot \Delta \rhd \texttt{return} : \tau$ in the proof system. A program (code sequence) corresponds to a proof composed of thease inference rule.

In JVM, Java bytecode operates on local variables and operand stack. To model this structure, each JVM instruction I is interpreted as a typing rule a form:

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

where Γ, Δ is a local environment and a stack environment respectively. A JVM program (method) is not a simple code sequence but a set of labeled code blocks. A label is used as an argument of jump an instruction. We interpret such a jump instruction as reference to an existing proof. For instance, $\texttt{goto}(l)$ which jumps to the code block B labeled l is represented as $\Gamma, \Delta \rhd \texttt{goto}(l) : \tau$ (if $\Gamma, \Delta \rhd B : \tau$).

One further refinement is required. In JVM, a program contains *subroutine* – a kind of internal procedure – other than general code blocks. A subroutine can be considered as a code sequence which changes the machine state. To subroutine SB that changes the machine state Γ_2, Δ_2 to Γ_1, Δ_1, we given following type:

$$SB : \langle \Gamma_1, \Delta_1 \rhd \tau // \Gamma_2, \Delta_2 \rhd \tau \rangle$$

This indicates that SB is the function which extends a proof $\Gamma_1, \Delta_1 \rhd \tau$ to another proof $\Gamma_2, \Delta_2 \rhd \tau$.

2.2 Type inference

Java bytecode verification problem is reduced to the typability problem in the JVM type system. Solving the typability problem requires to construct a type inference algorithm, since a code block refers to other code blocks through untyped labels. Furthermore, a subroutine label must be given a polymorphic type because it can be referenced from various different contexts in a method. To represent this polymorphic nature, we divide a program into a set M^s of subroutines and a set M^b of code blocks and intepret the program as a term similar to an ML's polymorphic let expression as follows.

$$\text{let } M^s \text{ in } M^b$$

For this refined program, type inference algorithm can be constructed by applying the idea of ML's type inference algorithm $\mathcal{W}$.

2.3 Implementation and Evaluation

The type inference algorithm is implemented in Standard ML. This system reads the bytecode sequence from a given class file and divides it into a set of code blocks. It then infers a type of method by applying the type inference algorithm. Using this implementation, we have examined the type system by comparing with Sun's bytecode verifier. Our initial results show that they are orthogonal in expressive power.

3 Furture work

To establish the a basis for analysis and implementation of JVM based on result shown in this thesis, further research is needed. The following three are particularly important.

- **Extending type sysmte to various features**
  In the thesis, we defined the type system for a subset of JVM. We should investigate the excluded features such as exception, interface, static method and object initialization.
- **Beyond the JVM**
  We would like to extend JVM with various advanced feature such as higher-order methods which take a method as argument or return a method.
- **Stronger type inference algorithm**
  The type inference algorithm developed in this thesis is still incompleteness and some extension to it is remain. Possible enhancement is to allow block to have a polymorphic type.