Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 7 pages

A Type System for Object Initialization In the Java™ Bytecode Language

Stephen N. Freund John C. Mitchell

Department of Computer Science
Stanford University
Stanford, CA 94305-9045
{freunds, mitchell}@cs.stanford.edu

Abstract

In the standard Java implementation, a Java language program is compiled to Java bytecode and this bytecode is then interpreted by the Java Virtual Machine. Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a *bytecode verifier* that performs a number of consistency checks before code is interpreted. However, there is no formal specification of the verifier. As one-step towards such a specification, we develop a precise specification of a subset of the bytecode language dealing with object creation and initialization. For this subset, we prove that for every Java bytecode program that satisfies our typing constraints, every object is initialized before it is used.

1 Introduction

The Java programming language is a statically-typed general-purpose programming language with an implementation architecture that is designed to facilitate transmission of compiled code across a network. In the standard implementation, a Java language program is compiled to Java bytecode and this bytecode is then interpreted by the Java Virtual Machine. The intermediate bytecode language, which we refer to as JVML, is a typed, machine-independent form of assembly language with some low-level instructions that reflect specific high-level Java source language constructs. For example, classes are a basic notion in JVML. Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a *bytecode verifier* that performs a number of consistency checks before code is interpreted. This protects the receiver from certain security risks and various forms of attack.

In this paper, we develop a specification for a fragment of the bytecode language that includes object creation (allocation of memory) and initialization. This work is based on a prior study of the bytecodes for local subroutine

call and return [2]. We prove soundness of the type system by a traditional method using operational semantics. It follows from the soundness theorem that any bytecode program that passes the static checks will initialize every object before it is used.

2 Object Initialization

As in many other object-oriented languages, the Java implementation creates new objects in two steps. The first step is to allocate space for the object. This usually requires some environment specific operation to obtain an appropriate region of memory. In the second step, user-defined code is executed to initialize the object. In Java, the user initialization code is provided by a constructor defined in the class of the object. Only after both of these steps are completed can a method be invoked on an object.

In the Java source language, allocation and initialization are combined into a single statement. This is illustrated in the following code fragment:

```
Point p = new Point(3);
p.Print();
```

Since every Java object is created by a statement like the one in the first line here, it does not seem difficult to prevent Java source language programs from invoking methods on objects that have not been initialized.

It is much more difficult to recognize initialization-before-use in bytecode. This can be seen by looking at the five lines of bytecode that are produced by compiling the two lines of source code above:

```
1:  new #1 <Class Point>
2:  dup
3:  iconst_3
4:  invokespecial #4 <Method Point(int)>
5:  invokevirtual #5 <Method void Print()>
```

The most striking difference is that memory allocation (line 1) is separated from the constructor invocation (line 4) by two lines of code. The first intervening line, `dup`, duplicates the pointer to the uninitialized object. This line is needed due to the calling convention of the Java Virtual Machine. The second line, `iconst_3`, pushes the constructor argument 3 onto the stack.

Since pointers may be duplicated, as above, and there may be more than one uninitialized object present at any time, some form of aliasing analysis must be used. Sun's Java Virtual Machine Specification [1] describes the alias analysis used by the Sun's JDK verifier. For each line of the bytecode program, some status information is recorded for every local variable and stack location. When a location points to an object that is not known to be initialized in all executions reaching this statement, the status will include not only the property *uninitialized,* but also the line number on which the uninitialized object would have been created. As references are duplicated on the stack

and stored and loaded in the local variables, the analysis also duplicates these line numbers. All references having the same line number are assumed to refer to the same object. When an object is initialized, all pointers that refer to objects created at the same line number are set to *initialized.* In other words, all references to objects of a certain type are partitioned into equivalence classes according to what is statically known about each reference. Since aliasing is irrelevant for objects that have been initialized, it is not necessary to track aliasing once a reference leads to an initialized object.

Since our approach is type based, the status information associated with each reference for the alias analysis is recorded as part of its type.

3 JVML$_i$

This section describes the JVML$_i$ language, a subset of JVML encompassing basic constructs and object initialization. The run-time environment for JVML$_i$ consists only of an operand stack and a finite set of local variables. A JVML$_i$ program will be a map from ADDR to instructions drawn from the following list:

$$\begin{aligned} \textit{instruction} ::= \ & \texttt{push 0} \mid \texttt{inc} \mid \texttt{pop} \\ & \mid \texttt{if } L \\ & \mid \texttt{store } x \mid \texttt{load } x \\ & \mid \texttt{new } \sigma \mid \texttt{init } \sigma \mid \texttt{use } \sigma \\ & \mid \texttt{halt} \end{aligned}$$

where x is a local variable name, σ is an object type, and L is an address of another instruction in the program. We refer the reader to Stata and Abadi for a description of those instructions not defined below:

`new` σ**:** allocates a new, uninitialized object of type σ.

`init` σ**:** initializes a previously uninitialized object of type σ. This represents calling the constructor of an object.

`use` σ**:** performs an operation on an initialized object of type σ. This corresponds to several operations in JVML, including method invocation (`invokevirtual`), accessing an instance field (`putfield`/`getfield`), etc.

4 Operational and Static Semantics

4.1 Values and Types

JVML$_i$ types are generated by the grammar:

$$\tau ::= \textsc{Int} \mid \sigma \mid \hat{\sigma}_i \mid \textsc{Top}$$

where $\sigma \in T$, the set of valid object types. The type $\hat{\sigma}_i$ will be given to values resulting from allocating an object of type σ on line i of a program. The type INT will be used for the type of integers. One final type is TOP. Any value of

$$\frac{P[pc] = \texttt{new}\ \sigma \quad \hat{a} \in A^{\hat{\sigma}_{pc}},\ \mathit{Unused}(\hat{a}, f, s)}{P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc+1,\ f,\ \hat{a}\cdot s\rangle} \qquad \frac{P[pc] = \texttt{init}\ \sigma \quad \hat{a} \in A^{\hat{\sigma}_j} \quad a \in A^{\sigma},\ \mathit{Unused}(a, f, s)}{P \vdash \langle pc,\ f,\ \hat{a}\cdot s\rangle \rightarrow \langle pc+1,\ [a/\hat{a}]f,\ [a/\hat{a}]s\rangle}$$

$$\frac{P[pc] = \texttt{use}\ \sigma \quad a \in A^{\sigma}}{P \vdash \langle pc,\ f,\ a\cdot s\rangle \rightarrow \langle pc+1,\ f,\ s\rangle}$$

Fig. 1. JVML$_i$ operational semantics.

any type also has type TOP. This type will represent unusable values in our static analysis.

Each object and uninitialized object type has a corresponding infinite set of values which can be distinguished from values of any other type. For any object type σ, this set of values is A^{σ}. Likewise, there is a set of values $A^{\hat{\sigma}_i}$ for all uninitialized object types $\hat{\sigma}_i$. The basic type rules for values are:

$$\frac{v \text{ is a value}}{v : \textsc{Top}} \qquad \frac{n \text{ is an integer}}{n : \textsc{Int}} \qquad \frac{a \in A^{\tau}}{a : \tau} \qquad \frac{a \in A^{\hat{\tau}_i}}{a : \hat{\tau}_i}$$

where $\tau \in T$ and $i \in \textsc{Addr}$. We also extend values and types to sequences:

$$\frac{}{\epsilon : \epsilon} \qquad \frac{a : \tau \quad s : \alpha}{a\cdot s : \tau\cdot\alpha}$$

4.2 Operational Semantics

The bytecode interpreter for JVML$_i$ instructions is modeled using the standard framework of operational semantics. Each instruction is characterized by a transformation of machine states, where a machine state is a tuple of the form $\langle pc,\ f,\ s\rangle$, which has the following meaning:

- pc is a program counter, indicating the address of the instruction that is about to be executed.
- f is a total map from VAR, the set of local variables, to values.
- s is a stack of values representing the operand stack.

Each bytecode instruction yields one or more rules in the operational semantics. These rules use the judgment

$$P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle$$

to indicate that a program P in state $\langle pc,\ f,\ s\rangle$ can move to state $\langle pc',\ f',\ s'\rangle$ in one step. The one-step operational semantics for the instructions we have added to study object initialization are shown in Figure 1.

$$(new)\quad \frac{\begin{array}{c} P[i] = \texttt{new}\ \sigma \\ \sigma \in T \\ F_{i+1} = F_i \\ S_{i+1} = \hat{\sigma}_i \cdot S_i \\ \hat{\sigma}_i \notin S_i \\ \forall y \in Dom(F_i).\ F_i[y] \neq \hat{\sigma}_i \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P} \qquad (init)\quad \frac{\begin{array}{c} P[i] = \texttt{init}\ \sigma \\ \sigma \in T \\ F_{i+1} = [\sigma/\hat{\sigma}_j]F_i \\ S_i = \hat{\sigma}_j \cdot \alpha,\quad j \in Dom(P) \\ S_{i+1} = [\sigma/\hat{\sigma}_j]\alpha \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P}$$

$$(use)\quad \frac{\begin{array}{c} P[i] = \texttt{use}\ \sigma \\ \sigma \in T \\ F_{i+1} = F_i \\ S_i = \sigma \cdot S_{i+1} \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P}$$

Fig. 2. Static semantics.

The rules for object initialization use the additional judgment *Unused*, defined by

$$(unused)\qquad \frac{\begin{array}{c} a \notin s \\ \forall y \in \textsc{Var}.\ f[y] \neq a \end{array}}{Unused(a, f, s)}$$

This will allow the virtual machine to pick a value that is currently not used by the program. The operational rules not presented here are discussed in [2]. These rules have been designed so that a step cannot be made from an illegal state, such as being at a `pop` statement when there is an empty stack.

When a new object is created, a currently unused value of an uninitialized object type is placed on the stack. The type of that value is determined by the object type named in the instruction and the line number of the instruction. When the value for an uninitialized object is initialized by an `init` instruction, all occurrences of that value are replaced by a value corresponding to an initialized object.

4.3 Static Semantics

A program P is well typed if there exist F and S such that

$$F, S \vdash P.$$

F is a map from ADDR to functions mapping local variables to types. As described in [2], elements in a map over ADDR are accessed as F_i instead of $F[i]$. Thus, $F_i[y]$ is the type of local variable y at line i of a program. Likewise, S is a map from ADDR to stack types such that S_i is the type of the operand stack at location i of the program. The judgment which allows us to conclude

that a program P is well typed by F and S is

$$(wt\ prog) \qquad \frac{\begin{array}{c} F_1 = F_{\text{TOP}} \\ S_1 = \epsilon \\ \forall i \in Dom(P).\ F, S, i \vdash P \end{array}}{F, S \vdash P}$$

where F_{TOP} is a function mapping all variables in VAR to TOP. The first two lines of (*wt prog*) constrain the initial conditions for the program's execution. The third line requires that each instruction in the program is well typed according to local judgments for each instruction. The rules for those instructions dealing with object initialization are presented in Figure 2.

4.4 Soundness

We prove that any well-typed program will not get stuck due to an error. This notion is captured by our soundness theorem.

Theorem 4.1 (Soundness) *Given P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f_0, f, s. \\ \quad \left(\begin{array}{l} P \vdash \langle 1,\ f_0,\ \epsilon \rangle \rightarrow^* \langle pc,\ f,\ s \rangle \\ \wedge\ \neg \exists pc', f', s'.\ P \vdash \langle pc,\ f,\ s \rangle \rightarrow \langle pc',\ f',\ s' \rangle \end{array} \right) \\ \Rightarrow P[pc] = \texttt{halt} \end{array}$$

In the initial state, the first instruction in the program is about to be executed, the operand stack is empty, and f_0 may map the local variables to any values.

5 Discussion

Given the need to guarantee type safety for mobile Java code, developing correct type checking and analysis techniques for JVML is crucial. However, there is not an existing specification which fully captures how Java bytecodes must be type checked. We have built on the previous work of Stata and Abadi to develop such a specification by formulating a sound type system for a fairly complex subset of JVML. Although our model is still rather abstract, it has already proved effective as a foundation for examining both JVML and existing bytecode verifiers. Even without a complete object model or notion of an object heap, we have been able to study initialization, and also several extensions to the work presented here. In fact, a previously unpublished bug in Sun's verifier implementation was found as a result of the analysis performed while studying the soundness proofs for JVML$_i$ extended with subroutines.

Acknowledgement

Thanks to Martín Abadi and Raymie Stata (DEC SRC) for their assistance on this project. Also, we thank Frank Yellin and Sheng Liang for several

interesting discussions.

References

[1] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[2] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.