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. This bytecode may be sent across the network to another site, where it 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. As illustrated by previous attacks on the Java Virtual Machine, these tests, which include type correctness, are critical for system security. In order to analyze existing bytecode verifiers and to understand the properties that should be verified, we develop a precise specification of *statically-correct* Java bytecode, in the form of a type system. Our focus in this paper is 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. The type system is easily combined with a previous system developed by Stata and Abadi for bytecode subroutines. Our analysis of subroutines and object initialization reveals a previously unpublished bug in the Sun JDK bytecode verifier.

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. While many previous programming languages have been implemented using a bytecode interpreter, the Java architecture differs in that programs are commonly transmitted between users across a network in compiled form.

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. Figure 1 shows the point at which the verifier checks a program during the compilation, transmission, and execution process. After a class file containing Java bytecodes is loaded by the Java Virtual Machine, it must pass through the bytecode verifier before being linked into the execution environment and interpreted. This protects the receiver from certain security risks and various forms of attack.

The verifier checks to make sure that every opcode is valid, all jumps lead to legal instructions, methods have structurally correct signatures, and that type constraints are satisfied. Conservative static analysis techniques are used to check these conditions. As a result, many programs that would never execute an erroneous instruction are rejected. However, any bytecode program generated by a conventional compiler is accepted. The need for conservative analysis stems from the undecidability of the halting problem, as well as efficiency considerations. Specifically, since most bytecode is the result of compilation, there is very little benefit in developing complex analysis techniques to recognize patterns that could be considered legal but do not occur in compiler output.

The intermediate bytecode language, which we refer to as JVML, is a typed, machine-independent form with some low-level instructions that reflect specific high-level Java source language constructs. For example, classes are a basic notion in JVML, and there is a form of "local subroutine" call and return designed to allow efficient implementation of the source language

*Supported in part by NSF grants CCR-9303099 and CCR-9629754, ONR MURI Award N00014-97-1-0505, and a NSF Graduate Research Fellowship.

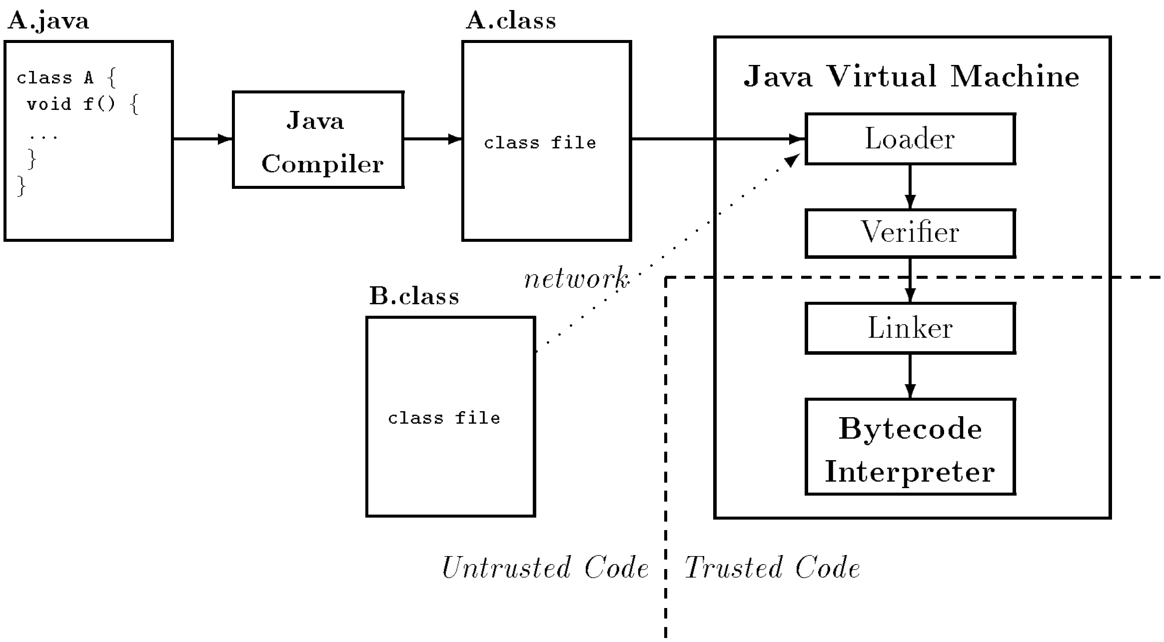

Figure 1: The Java Virtual Machine

`try-finally` construct. While some amount of type information is included in JVML to make type-checking possible, there are some high-level properties of Java source code that are not easy to detect in the resulting bytecode. One example is the last-called first-returned property of the local subroutines. While this property will hold for every JVML program generated by compiling Java source, some effort is required to confirm this property in bytecode programs [SA98].

Another example is the initialization of objects before use. While it is clear from the Java source language statement

$$\texttt{A x = new A(}\langle \textit{parameters} \rangle\texttt{)}$$

that the `A` class constructor will be called before any methods can be invoked through the pointer `x`, this is not obvious from a simple scan of the resulting JVML program. One reason is that many bytecode instructions may be needed to evaluate the parameters for the call to the constructor. In the bytecode, these will be executed after space has been allocated for the object and before the object is initialized. Another reason, discussed in more detail in Section 2, is that the structure of the Java Virtual Machine requires copying of pointers to uninitialized objects. Therefore, some form of aliasing analysis is needed to make sure that an object is initialized before it is used.

Several published attacks on early forms of the Java Virtual Machine illustrate the importance of the bytecode verifier for system security. To cite one specific example, a bug in an early version of Sun's bytecode verifier allowed applets to create certain system objects which they should not have been able to create, such as class loaders [DFW96]. The problem was caused by an error in how constructors were verified and resulted in the ability to potentially compromise the security of the entire system. Clearly, problems like this give rise to the need for a correct and formal specification of the bytecode verifier. However, for a variety of reasons, there is no established formal specification; the primary specification is an informal English description that is occasionally at odds with current verifier implementations.

Building on a prior study of the bytecodes for local subroutine call and return [SA98], this paper develops a specification of *statically-correct bytecode* for a fragment of JVML that includes object creation (allocation of memory) and initialization. This specification has the form of a type system, although there are several technical ways in which a type system for low-level code with jumps and type-varying use of stack locations (or registers) differs from conventional high-level type systems. 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. We have examined a broad range of alternatives for specifying type systems capable of identifying that kind of error. In some cases, we found it possible to simplify our specification by being more or less conservative than current verifiers. However, we generally resisted the temptation to do so since we hoped to gain some understanding of the strength and limitations of existing verifier implementations.

In addition to proving soundness for the simple language, we have structured the main lemmas and proofs so that they apply to any additional bytecode commands that satisfy certain general conditions. This makes it relatively straightforward to combine our analysis with the prior work of Abadi and Stata, showing type soundness for bytecode programs that combine object creation with subroutines. In analyzing the interaction between object creation and subroutines, we have identified a previously unpublished bug in the Sun implementation of the bytecode verifier. This bug allows a program to use an object before it has been initialized; details appear in Section 7. Our type-based framework also made it possible to evaluate various repairs to fix this error and prove correctness for a modified system.

Section 2 describes the problem of object initialization in more detail, and Section 3 presents JVML_i, the language which we formally study in this paper. The operational semantics and type system for this language is presented in Section 4. Some sound extensions to JVML_i, including subroutines, are discussed in Section 6, and Section 7 describes how this work relates to Sun's implementation. Section 8 discusses some other projects dealing with bytecode verification, and Section 9 gives directions for future work and concludes.

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 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();
```

The first line indicates that a new `Point` object should be created and calls the `Point` constructor to initialize this object. The second line invokes a method on this object and therefore can be allowed only if the object has been initialized. 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. While there are a few subtle situations to consider, such as when a constructor throws an exception, the issue is essentially clear cut.

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 preceding two lines of source code:

```
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. The reason for this instruction is that a pointer to the object must be passed to the constructor. A convention of parameter passing for the stack-based architecture is that parameters to a function are popped off the stack before the function returns. Therefore, if the address were not duplicated, there would be no way for the code creating the object to access it after it is initialized. The second line, `iconst_3` pushes the constructor argument 3 onto the stack. If `p` were used again after line 5 of the bytecode program, another `dup` would have been needed prior to line 5.

Depending on the number and type of constructor arguments, many different instruction sequences may appear between object allocation and initialization. For example, suppose that several new objects are passed as arguments to a constructor. In this case, it is necessary to create each of the argument objects and initialize them before passing them to the constructor. In general, the code fragment between allocation and initialization may involve substantial computation, including allocation of new objects, duplication of object pointers, and jumps to or branches from other locations in the code.

Since pointers may be duplicated, some form of aliasing analysis must be used. More specifically, when a constructor is called, there may be several pointers to the object that is initialized as a result, as well as pointers to other uninitialized objects. In order to verify code that uses pointers to initialized objects, it is therefore necessary to keep track of which pointers are aliases (name the same object). Some hint for this is given by the following bytecode sequence:

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

When line 5 is reached during execution, there will be two different uninitialized `Point` objects. If the bytecode verifier is to check object initialization statically, it must be able to determine which references point to the object that is initialized at line 5 and which point to the remaining uninitialized object. Otherwise, the verifier would either prevent use of an initialized object or allow use of an uninitialized one. (The bytecode program above is valid and accepted by verifiers using the static analysis described below.)

Sun's Java Virtual Machine Specification [LY96] describes the alias analysis used by the Sun 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 known not 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, and 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 uninitialized objects of a certain type are partitioned into equivalence classes according to what is statically known about each reference, and all references that point to uninitialized objects created on the same line are assumed to be aliases. This is a very simple and highly conservative form of aliasing analysis; far more sophisticated methods might be considered. However, the approach can be implemented efficiently and it is sufficiently accurate to accept bytecode produced by standard compilers.

Our specification of statically-correct Java bytecode in Section 4 uses the same form of aliasing analysis as the Sun JDK verifier. Since our approach is type based, the status information associated with each reference is recorded as part of its type.

One limitation of aliasing analysis based on line numbers is that no verifiable program can ever be able to reference two objects allocated on the same line, without first initializing at least one of them. If this situation were to occur, references would exist to two different objects from the same static aliasing-equivalence class. Unfortunately, there was an oversight in this regard in the development of the Sun verifier, which allowed such a case to exist (as of version 1.1.4). As discussed in Section 7, aliasing based on line numbers makes it problematic for a subroutine to return an uninitialized object.

3 JVML$_i$

This section describes the JVML$_i$ language, a subset of JVML encompassing basic constructs and object initialization. Although this language is much smaller than JVML, it is sufficient to study object initialization and formulate a sound type system encompassing the static analysis described above. 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 sequence of instructions drawn from the following list:

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

where x is a local variable name, σ is an object type, and L is an address of another instruction in the program. Informally, these instructions have the following effects:

`push 0`: pushes integer 0 onto the stack.

`inc`: adds one to the value on the top of the stack, if that value is an integer.

`pop`: removes the top element from the stack, provided that the stack is not empty.

`if` L: if the top element on the stack is not 0, execution jumps to instruction L. Otherwise, execution steps to the next sequential instruction. This assumes that the top element is an integer.

`store` x: removes a value from the top of the stack and stores it into local variable x.

`load` x: loads the value from local variable x and places it on the top of the stack.

`halt`: terminates program execution.

`new` σ: allocates a new, uninitialized object of type σ and places it on the stack.

`init` σ: initializes the object on top of the operand stack, which must be a previously uninitialized object obtained by a call to `new` σ. This represents calling the constructor of an object. In this model, we assume that constructors always properly initialize their argument and return. However, as described in Section 6, there are several additional properties which must be checked to verify that constructors do in fact behave correctly.

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

In each case, all explicit or implicit assumptions must be satisfied in order for the statement to be executed. For example, a `pop` instruction cannot be executed if the stack is empty. The exact conditions required to execute each instruction are specified in the definition of the operational semantics, in Section 4.3. Although `dup` does not appear in JVML$_i$ for simplicity, aliasing may arise by storing and loading object references from the local variables.

4 Operational and Static Semantics

4.1 Notation

This section briefly reviews the framework developed by Stata and Abadi in [SA98] for studying JVML. We begin with a set of instruction addresses $\textsc{Addr}$. Although we shall use integers to represent elements of this set, we will distinguish elements of $\textsc{Addr}$ from integers. A program is formally represented as a partial function from addresses to instructions. $Dom(P)$ is the set of addresses used in program P, and $P[i]$ is the i^{th} instruction in program P. $Dom(P)$ will always include address 1 and is usually a range $\{1, \ldots, n\}$ for some n.

Equality on partial maps is defined as

$$f = g \text{ iff } Dom(f) = Dom(g) \wedge \forall y \in Dom(f).\ f[y] = g[y]$$

Update and substitution operations are also defined. $\forall y \in Dom(f)$:

$$(f[x \mapsto v])[y] = \begin{cases} v & \text{if } x = y \\ f[y] & \text{otherwise} \end{cases}$$

$$([b/a]f)[y] = [b/a](f[y]) = \begin{cases} b & \text{if } f[y] = a \\ f[y] & \text{otherwise} \end{cases}$$

where a, b, and v range over the codomain of f. This notation for partial maps will be used throughout this paper.

Sequences will also be used. The empty sequence is ϵ, and $v \cdot s$ represents placing v on the front of sequence s. A sequence of one element, $v \cdot \epsilon$, will sometimes be abbreviated to v. When convenient, we shall also treat sequences as partial maps from positions to elements of the sequence. For a sequence s, $Dom(s)$ is the set of indices into s, and $s[i]$ is the i^{th} element in s from the right. Substitution is also defined on sequences, in the same manner as substitution on partial maps.

4.2 Values and Types

The types will be integers and object types. For objects, we assume there is some set T of possible object types. These types include all class names to which a program may refer. In addition, there is a set $\hat{T}$ of types for uninitialized objects. The contents of this set is defined in terms of T:

$$\hat{\sigma}_i \in \hat{T} \text{ iff } \sigma \in T \wedge i \in \textsc{Addr}$$

The type $\hat{\sigma}_i$ is used for an object of type σ allocated on line i of a program, until it has been initialized. Given these definitions, JVML$_i$ types are generated by the grammar:

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

where $\sigma \in T$ and $\hat{\sigma}_i \in \hat{T}$. The type $\textsc{Int}$ will be used for integers. We discuss the addition of other basic types in Section 6. The type $\textsc{Top}$ is the supertype of all types, with any value of any type also having type $\textsc{Top}$. This type will represent unusable values in our static analysis. In general, a type meta-variable τ may refer to any type, including any object type $\sigma \in T$ or uninitialized-object type $\hat{\sigma}_i \in \hat{T}$. In the case that a type meta-variable is known to refer to some uninitialized object type, we will write it as $\hat{\tau}$, for example.

Each object type 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 $\hat{\sigma}_i$. In our model, we only need to know one piece of information for each object, namely, whether or not it has been initialized. Therefore, drawing uninitialized and initialized object "references" from different sets is sufficient for our purposes, and we do not need to model an object store. As we will see below, this representation has some impact on how object initialization is modeled in our operational semantics. Values of the form $\hat{a}$ or $\hat{b}$ will refer to values known to be of some uninitialized object type.

The basic type rules for values are:

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

We also extend values and types to sequences:

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

4.3 Operational Semantics

The bytecode interpreter for JVML$_i$ is modeled using the standard framework of operational semantics. Each

$$\frac{P[pc] = \texttt{inc}}{P \vdash \langle pc,\, f,\, n \cdot s\rangle \rightarrow \langle pc+1,\, f,\, (n+1)\cdot s\rangle}$$

$$\frac{P[pc] = \texttt{pop}}{P \vdash \langle pc,\, f,\, v \cdot s\rangle \rightarrow \langle pc+1,\, f,\, s\rangle} \qquad \frac{P[pc] = \texttt{push 0}}{P \vdash \langle pc,\, f,\, s\rangle \rightarrow \langle pc+1,\, f,\, 0\cdot s\rangle}$$

$$\frac{P[pc] = \texttt{load } x}{P \vdash \langle pc,\, f,\, s\rangle \rightarrow \langle pc+1,\, f,\, f[x]\cdot s\rangle} \qquad \frac{P[pc] = \texttt{store } x}{P \vdash \langle pc,\, f,\, v\cdot s\rangle \rightarrow \langle pc+1,\, f[x \mapsto v],\, s\rangle}$$

$$\frac{P[pc] = \texttt{if } L}{P \vdash \langle pc,\, f,\, 0\cdot s\rangle \rightarrow \langle pc+1,\, f,\, s\rangle} \qquad \frac{\begin{array}{c}P[pc] = \texttt{if } L\\ n \neq 0\end{array}}{P \vdash \langle pc,\, f,\, n\cdot s\rangle \rightarrow \langle L,\, f,\, s\rangle}$$

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

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

Figure 2: JVML$_i$ operational semantics.

instruction is characterized by a transformation of machine states, where a machine state is a tuple $\langle pc,\, f,\, s\rangle$ with 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 the values stored in the local variables in the current state.
- s is a stack of values representing the operand stack for the current state in execution.

The machine begins execution in state $\langle 1,\, f_0,\, \epsilon\rangle$. In this state, the first instruction in the program is about to be executed, the operand stack is empty, and the local variables may contain any values. This means that f_0 may map the local variables to any values.

Each bytecode instruction has 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 complete one-step operational semantics for JVML$_i$ is shown in Figure 2. In that figure, n is any integer, v is any value, L and j are any addresses, and x is any local variable. These operational semantic rules, with the exception of those added to study object initialization, are discussed in detail in [SA98]. The rules have been designed so that a step cannot be made from an illegal state. For example, it is not possible to execute a `pop` instruction when there is an empty stack.

The rules for allocating and initializing objects need to generate object values not in use by the program. The only values in use are those which appear on the operand stack or in the local variables. Therefore, the notion of being unused may be characterized by

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

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` σ instruction, all occurrences of that value are replaced by a new value corresponding to an initialized object. In some sense, initialization may be thought of as a substitution of a new, initialized object for an uninitialized object. The new value is required to be unused. This allows the program to distinguish between different objects of the same type

$$(inc)\quad \frac{\begin{array}{c} P[i] = \texttt{inc} \\ F_{i+1} = F_i \\ S_{i+1} = S_i = \textsc{Int} \cdot \alpha \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P} \qquad (if)\quad \frac{\begin{array}{c} P[i] = \texttt{if}\ L \\ F_{i+1} = F_L = F_i \\ S_i = \textsc{Int} \cdot S_{i+1} = \textsc{Int} \cdot S_L \\ i+1 \in Dom(P) \\ L \in Dom(P) \end{array}}{F, S, i \vdash P}$$

$$(pop)\quad \frac{\begin{array}{c} P[i] = \texttt{pop} \\ F_{i+1} = F_i \\ S_i = \tau \cdot S_{i+1} \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P} \qquad (push\ 0)\quad \frac{\begin{array}{c} P[i] = \texttt{push 0} \\ F_{i+1} = F_i \\ S_{i+1} = \textsc{Int} \cdot S_i \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P}$$

$$(load)\quad \frac{\begin{array}{c} P[i] = \texttt{load}\ x \\ x \in Dom(F_i) \\ F_{i+1} = F_i \\ S_{i+1} = F_i[x] \cdot S_i \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P} \qquad (store)\quad \frac{\begin{array}{c} P[i] = \texttt{store}\ x \\ x \in Dom(F_i) \\ F_{i+1} = F_i[x \mapsto \tau] \\ S_i = \tau \cdot S_{i+1} \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P}$$

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

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

Figure 3: Static semantics.

after they have been initialized, but this fact is not necessarily needed to study the properties addressed by this paper.

4.4 Static Semantics

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

$$F, S \vdash P,$$

where F is a map from $\textsc{Addr}$ to functions mapping local variables to types, and S is a map from $\textsc{Addr}$ to stack types such that S_i is the type of the operand stack at location i of the program. As described in [SA98], elements in a map over $\textsc{Addr}$ 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.

The Java Virtual Machine Specification [LY96] describes the verifier as both computing the type information stored in F and S and checking it. However, we assume that the information stored in F and S has already been computed prior to the type checking stage. This simplifies matters since it separates the two tasks and prevents the type synthesis from complicating the static semantics. In other words, we only need to trust the implementation of the type checker, and not the implementation of the type inferencing part of the analysis. If the two stages are combined, as they are in current implementations, a bad program could be accepted due to an error in the process of computing type information. However, separating the two tasks prevents the type checker from accepting a bad program due to such an error.

The judgment which allows us to conclude that a program P is well typed by F and S is

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

where $F_{\textsc{Top}}$ is a function mapping all variables in $\textsc{Var}$ to $\textsc{Top}$. The first two lines of (*wt prog*) constrain the initial conditions for the program's execution to match the type of the values given to the initial state in the operational semantics. The third line requires that each instruction in the program is well typed according to the local judgments presented in Figure 3.

The (*new*) rule in Figure 3 requires that the type of the object allocated by the `new` instruction is left on top of the stack. Note that this rule is only applicable if the uninitialized object type about to be placed on top of the type stack does not appear anywhere in F_i or S_i. This restriction is crucial to ensure that we do not create a situation in which a running program may have two different values mapping to the same statically computed uninitialized object type.

The rule for (*use*) requires an initialized object type on top of the stack. The (*init*) rule implements the static analysis method described in Section 2. The rule specifies that all occurrences of the type on the top of the stack are replaced by an initialized type. This will change the types of all references to the object that is being initialized since all those references will be in the same static equivalence class, and, therefore, have the same type.

Figure 4 shows a JVML_i program and the type information demonstrating that it is a well-typed program according to the rules in this section.

5 Soundness

This section outlines the soundness proof for JVML_i. The main soundness theorem states that no well-typed program will cause a run-time type error. Before stating the main soundness theorem, a one-step soundness theorem is presented. One-step soundness implies that any valid transition from a well-formed state leads to another well-formed state.

Theorem 1 (One-step Soundness) *Given P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f, s, pc', f', s'. \\ \quad P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle \\ \quad \wedge\ \ s : S_{pc} \\ \quad \wedge\ \ \forall y \in \textsc{Var}.\ f[y] : F_{pc}[y] \\ \quad \wedge\ \ \mathit{ConsistentInit}(F_{pc}, S_{pc}, f, s) \\ \Rightarrow s' : S_{pc'} \\ \quad \wedge\ \ \forall y \in \textsc{Var}.\ f'[y] : F_{pc'}[y] \\ \quad \wedge\ \ \mathit{ConsistentInit}(F_{pc'}, S_{pc'}, f', s') \\ \quad \wedge\ \ pc' \in \mathit{Dom}(P) \end{array}$$

This theorem lists the four factors which dictate whether or not a state is well formed. The values on the operand stack must have the types expected by the static type rules, and the local variable contents must match the types in F. In addition, the program counter must always be in the domain of the program. This can be assumed on the left-hand side of the implication since the operational semantics guarantee that transitions can only be made if $P[pc]$ is defined. If the program counter were not in the domain of the program, no step could be made.

The final requirement for a state to be well formed is that it has the *ConsistentInit* property. Informally, this property means that the machine cannot access two different uninitialized objects created on the same line of code. As mentioned in Section 2, this invariant is critical

i	$P[i]$	$F_i[0]$	S_i
1:	`new C`	Top	ϵ
2:	`new C`	Top	$\hat{\mathtt{C}}_1 \cdot \epsilon$
3:	`store 0`	Top	$\hat{\mathtt{C}}_2 \cdot \hat{\mathtt{C}}_1 \cdot \epsilon$
4:	`load 0`	$\hat{\mathtt{C}}_2$	$\hat{\mathtt{C}}_1 \cdot \epsilon$
5:	`load 0`	$\hat{\mathtt{C}}_2$	$\hat{\mathtt{C}}_2 \cdot \hat{\mathtt{C}}_1 \cdot \epsilon$
6:	`init C`	$\hat{\mathtt{C}}_2$	$\hat{\mathtt{C}}_2 \cdot \hat{\mathtt{C}}_2 \cdot \hat{\mathtt{C}}_1 \cdot \epsilon$
7:	`use C`	`C`	$\mathtt{C} \cdot \hat{\mathtt{C}}_1 \cdot \epsilon$
8:	`halt`	`C`	$\hat{\mathtt{C}}_1 \cdot \epsilon$

Figure 4: A JVML$_i$ program and its static type information.

$$(\mathit{cons\ init}) \quad \frac{\forall \hat{\tau} \in \hat{T}.\ \exists \hat{b} : \hat{\tau}.\ \mathit{Corresponds}(F_i, S_i, f, s, \hat{b}, \hat{\tau})}{\mathit{ConsistentInit}(F_i, S_i, f, s)}$$

$$(\mathit{corr}) \quad \frac{\begin{array}{c}\forall x \in \mathit{Dom}(F_i).\ F_i[x] = \hat{\tau} \quad \Longrightarrow \quad f[x] = \hat{b} \\ \mathit{StackCorresponds}(S_i, s, \hat{b}, \hat{\tau})\end{array}}{\mathit{Corresponds}(F_i, S_i, f, s, \hat{b}, \hat{\tau})}$$

$$(\mathit{sc\ 0}) \quad \frac{}{\mathit{StackCorresponds}(\epsilon, \epsilon, \hat{b}, \hat{\tau})}$$

$$(\mathit{sc\ 1}) \quad \frac{\mathit{StackCorresponds}(S_i, s, \hat{b}, \hat{\tau})}{\mathit{StackCorresponds}(\hat{\tau} \cdot S_i, \hat{b} \cdot s, \hat{b}, \hat{\tau})}$$

$$(\mathit{sc\ 2}) \quad \frac{\begin{array}{c}\tau \neq \hat{\tau} \\ \mathit{StackCorresponds}(S_i, s, \hat{b}, \hat{\tau})\end{array}}{\mathit{StackCorresponds}(\tau \cdot S_i, v \cdot s, \hat{b}, \hat{\tau})}$$

Figure 5: The *ConsistentInit* judgement.

for the soundness of this static analysis. The *Consistent-Init* property requires a unique correspondence between uninitialized object types and run-time values.

Figure 5 presents the formal definition of *Consistent-Init*. In that figure, F_i is a map from local variables to types, S_i is a stack type. The judgment (*cons init*) is satisfied only if every uninitialized object type $\hat{\tau}$ has some value $\hat{b}$ that *Corresponds* to it. The first line of rule (*corr*) guarantees that every occurrence of $\hat{\tau}$ in the static types of the local variables is matched by $\hat{b}$ in the run-time state. The second line of that rule uses an auxiliary judgment to assert the same property about the stack type and operand stack inductively. Given this invariant, we are able to assume that when an `init` instruction is executed, all stack slots and local variables affected by the type substitution in rule (*init*) applied to that instruction contain the object that is being initialized.

The proof of Theorem 1 is by case analysis on all possible instructions at $P[pc]$. The proof of this theorem and those that follow appear in the extended version of this paper.

A complementary theorem is that a step can always be made from a well-formed state, unless the program has reached a `halt` instruction. This progress theorem can be stated as:

Theorem 2 (Progress) *Given P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f, s. \\ \quad s : S_{pc} \\ \quad \wedge\ \forall y \in \textsc{Var}.\ f[y] : F_{pc}[y] \\ \quad \wedge\ ConsistentInit(F_{pc}, S_{pc}, f, s) \\ \quad \wedge\ pc \in Dom(P) \\ \quad \wedge\ P[pc] \neq \texttt{halt} \\ \Rightarrow \exists pc', f', s'.\ P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle \end{array}$$

Theorem 1 and Theorem 2 can be used to prove inductively that a program beginning in a valid initial state will always be in a well-formed state, regardless of how many steps are made. In addition, a program will never get stuck unless it reaches a `halt` in instruction. When it does reach a `halt` instruction, the stack will have the correct type, which is important since the return value for a program, or method in the full JVML, is returned as the top value on the stack. The following theorem captures this soundness property:

Theorem 3 (Soundness) *Given P, F, and S such that $F, S \vdash P$:*

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

If a program executing in our machine model attempts to perform an operation leading to a type error, such as using an uninitialized object, it would get stuck since those operations are not defined by our operational semantics. By proving that well-typed programs only get stuck when a `halt` instruction is reached, we know that well-typed programs will not attempt to perform any illegal operations. Thus, this theorem implies that our static analysis is correct by showing that no erroneous programs are accepted. Therefore, no accepted program uses an uninitialized object.

One technical point of interest is the asymmetry of the checks in rule (*corr*). That rule requires that all locations sharing type $\hat{\tau}$ contain the same value $\hat{b}$, but it does not require that all occurrences of $\hat{b}$ map to the type $\hat{\tau}$ in the static type information. We do not need to check the other direction because the rule is used only in the hypothesis of rule (*cons init*), where the condition on the existential quantification of $\hat{b}$ requires that $\hat{b} : \hat{\tau}$. Therefore, $\hat{b} \in A^{\hat{\tau}}$, and the only types which we may assign to $\hat{b}$ are $\hat{\tau}$ and Top.

This allows us to assume that, as long as the stack and local variables are well typed when rule (*cons init*) is used, any occurrences of $\hat{b}$ are matched by either $\hat{\tau}$ or Top. Thus, with the exception of occurrences of Top, the correspondence between $\hat{b}$ and $\hat{\tau}$ holds in both directions. The situation for Top introduces a special case in the proofs but does not affect soundness, and the asymmetric checks are sufficient to prove the soundness of the system. If we were to change our model so that object values could potentially have more than one uninitialized object type, i.e. all uninitialized object references are drawn from a single set, then we would need to check both directions for the correspondence and explicitly deal with the special case for Top in the (*corr*) judgment.

6 Extensions

Several extensions to the JVML_i framework described in the previous sections have been studied. First, there are additional static checks which must be performed on constructors in order to guarantee that they do properly initialize objects. Section 6.1 presents JVML_c, an extension of JVML_i modeling constructors. Another extension, JVML_s, combining object initialization and subroutines, is described in Section 6.2. Section 6.3 shows how any of these languages may be easily extended with other basic operations and primitive types. The combination of these features yields a sound type system covering the most complex pieces of the JVML language.

6.1 $\mathbf{JVML}_c$

The typing rules in Section 4 are adequate to check code which creates, initializes, and uses objects, assuming that calls to `init` σ do in fact properly initialize objects. However, since initialization is performed by user-defined constructors, the verifier must check that these constructors do correctly initialize objects when called. This section studies verification of JVML constructors using JVML$_c$, an extension of JVML$_i$.

The rules for checking constructors are defined in [LY96] and can be summarized by three basic points:

- When a constructor is invoked, local variable 0 contains a reference to the object that is being initialized.
- A constructor must apply either a different constructor of the same class or a constructor from the parent class to the object that is being initialized before the constructor exits. For simplicity, we may refer to either of these actions as invoking the super class constructor.
- The only deviation from this requirement is for constructors of class `Object`. Since, by the Java language definition, `Object` is the only class without a superclass, constructors for `Object` need not call any other constructor. This one special case has not been modeled by our rules, but would be trivial to add.

Note that these rules do not imply that constructors will always return. For example, they do not prevent non-termination due to an infinite loop within a constructor. A more interesting case is when two constructors from the same class call each other recursively and, therefore, never fully construct an object passed to them. While programs potentially exhibiting this behavior could be detected by intra-procedural analysis, this type of analysis falls outside of the bounds of the current verifier.

JVML$_c$ programs are sequences of instructions containing any instructions from JVML$_i$ plus one new instruction, `super` σ. This instruction represents calling a constructor of the parent class of class σ (or a different constructor of the class σ).

For simplicity, the rest of this section assumes that we are describing a constructor for object type φ, for some φ in T. To model the initial state of a constructor invocation for class φ, a JVML$_c$ program is assumed to begin in a state in which local variable 0 contains an uninitialized reference. This corresponds to the argument of the constructor. Prior to halting, the program must call `super` φ on that object reference. As mentioned above, this represents calling the super class constructor.

We will use $\hat{\varphi}_0$ as the type of the object stored in local variable 0 at the start of execution. The value in local variable 0 must be drawn from the set $A^{\hat{\varphi}_0}$. We now assume $\textsc{Addr}$ includes 0, although 0 will not be in the domain of any program. Also, machine state in the operational semantics is augmented with a fourth element, z, which indicates whether or not a super class constructor has been called on the object that is being initialized. The rules for all instructions other than `super` do not affect z, and are derived directly from the rules in Figure 2. For example, the rule for `inc` is:

$$\frac{P[pc] = \mathtt{inc}}{P \vdash_c \langle pc,\, f,\, n \cdot s,\, z\rangle \rightarrow \langle pc+1,\, f,\, (n+1)\cdot s,\, z\rangle}$$

As demonstrated in Theorem 4 below, the initial state for execution of a constructor for φ is $\langle 1,\, f_0[0 \mapsto \hat{a}_\varphi],\, \epsilon,\, \mathit{false}\rangle$ where $\hat{a}_\varphi \in A^{\hat{\varphi}_0}$.

For `super`, the operational semantics rule is:

$$\frac{\begin{array}{c} P[pc] = \mathtt{super}\ \sigma \\ \hat{a} \in A^{\hat{\sigma}_0} \\ a \in A^{\sigma},\ \mathit{Unused}(a, f, s) \end{array}}{P \vdash_c \langle pc,\, f,\, \hat{a}\cdot s,\, z\rangle \rightarrow \langle pc+1,\, [a/\hat{a}]f,\, [a/\hat{a}]s,\, \mathit{true}\rangle}$$

The typing rule for `super` is very similar to the rule for `init`, and is shown below with the judgment for determining whether a program is a valid constructor for objects of type φ. All the other typing rules are the same as those appearing in Figure 3.

$$(\mathit{super}) \qquad \frac{\begin{array}{c} P[i] = \mathtt{super}\ \sigma \\ F_{i+1} = [\sigma/\hat{\sigma}_0]F_i \\ S_i = \hat{\sigma}_0 \cdot \alpha \\ S_{i+1} = [\sigma/\hat{\sigma}_0]\alpha \\ i+1 \in \mathit{Dom}(P) \end{array}}{F, S, i \vdash P}$$

$$(\mathit{wt\ cst}) \quad \frac{\begin{array}{c} F_1 = F_{\mathrm{TOP}}[0 \mapsto \hat{\varphi}_0] \\ S_1 = \epsilon \\ Z_1 = \mathit{false} \\ \varphi \in T \\ \forall i \in \mathit{Dom}(P).\ F, S, i \vdash P \\ \forall i \in \mathit{Dom}(P).\ Z, i \vdash P \text{ constructs } \varphi \end{array}}{F, S \vdash P \text{ constructs } \varphi}$$

The $(\mathit{wt\ cst})$ rule is analogous to $(\mathit{wt\ prog})$ from Section 4. However, this rule places an additional restriction on the structure of well-typed programs. The judgment

$$Z, i \vdash P \text{ constructs } \varphi$$

is a local judgment which gives Z_i the value *true* or *false* depending on whether or not all possible execution sequences reaching instruction i would have called `super` φ or not. The local judgments are defined in Figure 6. As seen by those rules, one can only conclude that

$$\frac{\begin{array}{c} P[i] \in \{\texttt{inc}, \texttt{pop}, \texttt{push 0}, \texttt{load } x, \texttt{store } x, \texttt{new } \sigma, \texttt{init } \sigma, \texttt{use } \sigma\} \\ Z_{i+1} = Z_i \end{array}}{Z, i \vdash P \text{ constructs } \varphi}$$

$$\frac{\begin{array}{c} P[i] = \texttt{if } L \\ Z_{i+1} = Z_L = Z_i \end{array}}{Z, i \vdash P \text{ constructs } \varphi}$$

$$\frac{\begin{array}{c} P[i] = \texttt{super } \varphi \\ Z_{i+1} = \mathit{true} \end{array}}{Z, i \vdash P \text{ constructs } \varphi}$$

$$\frac{\begin{array}{c} P[i] = \texttt{halt} \\ Z_i = \mathit{true} \end{array}}{Z, i \vdash P \text{ constructs } \varphi}$$

Figure 6: Rules checking that a super class constructor will always be called prior to reaching a `halt` instruction.

a program is a valid constructor for φ if every path to each `halt` instruction has called `super` φ. These judgments also reject any programs which call `super` for a class other than φ. The existence of unreachable code may cause more than one value of Z to conform to the rules in Figure 6. To make Z unique for any given program, we assume that, for program P, there is a unique canonical form Z_P. Thus, $Z_{P,i}$ will be a unique value for instruction i.

The main soundness theorem for constructors includes a guarantee that constructors do call `super` on the uninitialized object:

Theorem 4 (Constructor Soundness) *Given* P, F, S, φ, *and* $\hat{a}_\varphi$ *such that* $F, S \vdash P$ constructs φ *and* $\hat{a}_\varphi : \hat{\varphi}_0$ *:*

$$\begin{array}{l} \forall pc, f_0, f, s, z. \\ \quad P \vdash_c \langle 1,\, f_0[0 \mapsto \hat{a}_\varphi],\, \epsilon,\, \mathit{false}\rangle \to^* \langle pc,\, f,\, s,\, z\rangle \\ \quad \wedge\ \neg\exists pc', f', s', z'. \\ \qquad\qquad P \vdash_c \langle pc,\, f,\, s,\, z\rangle \to \langle pc',\, f',\, s',\, z'\rangle \\ \Rightarrow P[pc] = \texttt{halt} \\ \quad \wedge\ z = \mathit{true} \end{array}$$

The main difference in the proof of Theorem 4, in comparison with Theorem 3, is that the corresponding one-step soundness theorem requires an additional invariant. The invariant states that when program P is in state $\langle pc,\, f,\, s,\, z\rangle$, $z = Z_{P,pc}$. The proof of this theorem appears in the extended version of this paper.

This analysis for constructors is combined with the analysis of normal methods in a more complete JVML model currently being developed.

6.2 JVML$_s$

The JVML bytecodes for subroutines have also been added to JVML$_i$ and are presented in another extended language, JVML$_s$. While this section will not go into all the details of subroutines, detailed discussions of bytecode subroutines can be found in several other works [SA98, LY96]. Subroutines are used to compile the `finally` clauses of exception handlers in the Java language. Subroutines share the same activation record as the method which uses them, and they can be called from different locations in the same method, enabling all locations where `finally` code must be executed to jump to a single subroutine containing that code. The flexibility of this mechanism makes bytecode verification difficult for two main reasons:

- Subroutines are polymorphic over local variables which they do not use.
- Subroutines may call other subroutines, as long as a call stack discipline is preserved. In other words, the most recently called subroutine must be the first one to return. This is a slight simplification of the rules for subroutines defined in [LY96], which do allow a subroutine to return more than one level up in the implicit subroutine call stack in certain cases, but does match the definitions presented in [SA98].

JVML$_s$ programs contain the same set of instructions as JVML$_i$ programs and, also, the following:

`jsr` L: jumps to instruction L, and pushes the return address onto the stack. The return address is the instruction immediately after the `jsr` instruction.

$$\frac{P[pc] = \texttt{jsr}\ L}{P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle L,\ f,\ (pc+1)\cdot s\rangle}$$

$$\frac{P[pc] = \texttt{ret}\ x}{P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle f[x],\ f,\ s\rangle}$$

Figure 7: Operational semantics for `jsr` and `ret`.

$$(jsr)\quad \frac{\begin{array}{c} P[i] = \texttt{jsr}\ L \\ Dom(F_{i+1}) = Dom(F_i) \\ Dom(F_L) \subseteq Dom(F_i) \\ \forall y \in Dom(F_i).\ F_i[y] \notin \hat{T} \\ \forall y \in Dom(S_i).\ S_i[y] \notin \hat{T} \\ \forall y \in Dom(F_i)\backslash Dom(F_L).\ F_{i+1}[y] = F_i[y] \\ \forall y \in Dom(F_L).\ F_L[y] = F_i[y] \\ S_L = (\texttt{ret-from}\ L)\cdot S_i \\ (\texttt{ret-from}\ L) \notin S_i \\ \forall y \in Dom(F_L).\ F_L[y] \neq (\texttt{ret-from}\ L) \\ i+1 \in Dom(P) \\ L \in Dom(P) \end{array}}{F, S, i \vdash P}$$

$$(ret)\quad \frac{\begin{array}{c} P[i] = \texttt{ret}\ x \\ R_{P,i} = \{L\} \\ x \in Dom(F_i) \\ F_i[x] = (\texttt{ret-from}\ L) \\ \forall y \in Dom(F_i).\ F_i[y] \notin \hat{T} \\ \forall y \in Dom(S_i).\ S_i[y] \notin \hat{T} \\ \forall j.\ P[j] = \texttt{jsr}\ L \Rightarrow \left(\begin{array}{l} \forall y \in Dom(F_i).\ F_{j+1}[y] = F_i[y] \\ \wedge\ S_{j+1} = S_i \end{array}\right) \end{array}}{F, S, i \vdash P}$$

Figure 8: Type rules for `jsr` and `ret`.

`ret` x: jumps to the instruction address stored in local variable x.

The operational semantics and typing rules for these instructions are shown in Figure 7 and Figure 8. These rules are based on the rules used by Stata and Abadi [SA98]. The type `(ret-from` L`)` is introduced to indicate the type of an address to which subroutine L may return. The meaning of $R_{P,i} = \{L\}$ in (*ret*) is defined in their paper and basically means that instruction i is an instruction belonging to the subroutine starting at address L. All other rules are the same as those for JVML$_i$.

The main issue concerning initialization which must be addressed in the typing rules for `jsr` and `ret` is the preservation of the *ConsistentInit* invariant. A type loophole could be created by allowing a subroutine and the caller of that subroutine to exchange references to uninitialized objects in certain situations. An example of this behavior is described in Section 7.

When subroutines are used to compile `finally` blocks by a Java compiler, uninitialized object references will never be passed into or out of a subroutine. The Java language prevents a program from splitting allocation and initialization of an object between code inside and outside of a `finally` clause since both are part of the same Java operation, as described in Section 2. Either both steps occur outside of the subroutine, or both steps occur inside the subroutine. We restrict programs not to have uninitialized objects accessible when calling or returning from a subroutine. For (*ret*), the following two lines are added. These prevent the subroutine from allocating a new object without initializing it:

$$\forall y \in Dom(F_i).\ F_i[y] \notin \hat{T}$$
$$\forall y \in Dom(S_i).\ S_i[y] \notin \hat{T}$$

Similar lines are added to (*jsr*). The discussion of the interaction between subroutines and uninitialized objects in the Java Virtual Machine specification is vague and inconsistent with current implementations, but the rules we have developed seem to fit the general strategy described in the specification.

This is certainly not the only way to prevent subroutines and object initialization from causing problems. For example, slightly less restrictive rules could be added to (*jsr*):

$$\forall y \in Dom(F_L).\ F_L[y] \notin \hat{T}$$
$$\forall y \in Dom(S_i).\ S_i[y] \notin \hat{T}$$

These conditions still allow uninitialized objects to be present when a subroutine is called, but those objects cannot be touched since they are stored in local variables which are not accessed in the body of the subroutine. This would allow the typing rules to accept more programs, but these programs could not have been created by a compiler from any valid Java program.

The main soundness theorem, Theorem 3, has been proved for JVML$_s$, and for JVML$_c$ with subroutines, by combining the proof of JVML$_i$ soundness with the work of Stata and Abadi. These proofs appear in the extended version of this paper.

6.3 Other Basic Types and Instructions

Many JVML instructions are variants of operations for different basic types. For example, there are four `add` instructions corresponding to addition on values of type INT, FLOAT, LONG, and DOUBLE. Likewise, many other simple operations have several different forms. These instructions and other basic types can be added to JVML$_i$, or any of the extended languages, easily. These instructions do not complicate any of the soundness proofs since they only operate on basic types and do not interfere with object initialization or subroutine analysis.

The only tricky case is that LONG and DOUBLE values take up two local variables or two stack slots since they are stored as two-word values. Although this requires an additional check in the rules for `load` and `store` to prevent the program from accessing a partially over-written two-word value, this does not pose any serious difficulty.

Of the 200 bytecode instructions in JVML, all but approximately 40 fall into this category and may be added to JVML$_i$ without trouble, although a full presentation of the operational and type rules for these instructions is beyond the scope of this paper. With these additions, and the methods described in the previous subsections, the JVML$_i$ framework can be extended to cover the whole bytecode language, except for a full object system, exceptions, arrays, and concurrency. Considering objects and classes requires the addition of an object heap and a method call stack, as well as a typing environment containing class declarations. We are currently developing an extended system covering all these topics except concurrency.

7 The Sun Verifier

This section describes the relationship between the rules we have developed for object initialization and subroutines and the rules implicitly used to verify programs in Sun's implementation. We first describe a mistake we have found in Sun's rules and then compare their corrected rules with our rules for JVML$_s$.

```
1:   jsr 10        // jump to subroutine
2:   store 1       // store uninitialized object
3:   jsr 10        // jump to subroutine
4:   store 2       // store another uninitialized object
5:   load 2        // load one of them
6:   init P        // initialize it
7:   load 1        // load the other
8:   use P         // use uninitialized object!!!
9:   halt

10:  store 0       // store return address
11:  new P         // allocate new object
12:  ret 0         // return from subroutine
```

Figure 9: A program that uses an uninitialized object but is accepted by Sun's verifier.

7.1 The Sun JDK 1.1.4 Verifier

As a direct result of the insight gained by carrying out the soundness proof for JVML$_s$, a previously unpublished bug was discovered in Sun's JDK 1.1.4 implementation of the bytecode verifier. A simple program exhibiting the incorrect behavior is shown in Figure 9. Line 8 of the program uses an uninitialized object, but this code is accepted by this specific implementation of the verifier. Basically, the program is able to allocate two different uninitialized objects on the same line of code without initializing either one, violating the *ConsistentInit* invariant. The program accomplishes this by allocating space for the first new object inside the subroutine and then storing the reference to that object in a local variable over which the subroutine is polymorphic before calling it again. After initializing only one of the objects, it can use either one.

The bug can be attributed to the verifier not placing any restrictions on the presence of uninitialized objects at calls to `jsr` L or `ret` x. The checks made by Sun's verifier are analogous to the (*jsr*) and (*ret*) rule in Figure 8 as they originally appeared in [SA98], without the additions described in the previous section. Removing these lines allows subroutines to return uninitialized objects to the caller and to store uninitialized values across subroutine calls, which clearly leads to problems.

Although this bug does not immediately create any security loopholes in the Sun Java Virtual Machine, it does demonstrate the need for a more formal specification of the verifier. It also shows that even a fairly abstract model of the bytecode language is extremely useful at examining the complex relationships between different parts of the language, such as initialization and subroutines.

7.2 The Corrected Sun Verifier

After describing this bug to the Sun development team, they have taken steps to repair their verifier implementation. While they did not use the exact rules we have presented in this paper, they have changed their implementation to close the potential type loophole. This section briefly describes the difference between their approach and ours. The Sun implementation may be summarized as follows [Lia97]:

- Uninitialized objects may appear anywhere in the local variables or on the operand stack at `jsr` L or `ret` x instructions, but they can not be used after the instruction has executed. In other words, their static type is made TOP in the post-instruction state. This difference does not affect the ability of either Sun's rules or our rules to accept code created for valid Java language programs.

- The static types assigned to uninitialized objects passed into constructors are treated differently from other uninitialized object types in the Sun verifier. Values with these types may still be used after being present at a call to or an exit from a subroutine. Also, the superclass constructor may be called anywhere, including inside a subroutine.

Treating the uninitialized object types for constructor arguments differently than other uninitialized types allows the verifier to accept programs where a subroutine must be called prior to invoking the super class constructor. This is demonstrated by Figure 10. That figure shows a constructor for class `C`, as well as a rough translation of it into JVML$_c$ with subroutines (we ignore the code required for the exception handler).

The bytecode translation of the constructor will be rejected by our analysis because control jumps to a subroutine when a local variable contains the uninitialized

```
class C extends Object {
  C() {
    int i;
    try {
      i = 0;
    } finally {
    }
    super();
  }
  ...
}
```

```
1:  push 0   // put 0 on stack
2:  store 1  // store it in ``i''
3:  jsr 7    // jump to subroutine
4:  load 0   // load constructor arg.
5:  super C  // call superclass cnstr.
6:  halt

7:  store 2  // store return address
8:  ret 2    // return from subroutine
```

Figure 10: A constructor which will always call a subroutine before invoking the super class constructor, and its translation into JVML_c with subroutines (ignoring the exception handler). Note that this is not a valid Java program.

object passed into the constructor. It is accepted by the Sun verifier due to their special treatment of the type of the uninitialized object passed into the constructor. However, the Java language specification requires that the superclass constructor be called prior to the start of any code protected by an exception handler. Therefore, the Java program in Figure 10 is not valid. The added flexibility of their method is not required to verify valid Java programs, but it does make the analysis much more difficult. In fact, several published attacks, including the one described in Section 1, may be attributed to errors in this part of the verifier. Other verifiers, such as the Microsoft verifier, currently reject the bytecode translation of this class.

In summary, the differences in the two verification techniques would only become apparent in handwritten bytecodes using uninitialized object types in unusual ways, and both systems are sufficient to type check translations of valid Java programs. Since our method, while slightly more restrictive, makes both verification and our soundness proofs much simpler, we believe that our method is reasonable.

8 Related Work

There are several other projects currently examining bytecode verification and the creation of correct bytecode verifiers. This section describes some of these projects, as well as related work in contexts other than Java. There have also been many studies of the Java language type system [Sym97, DE97, NvO98], but we will mostly focus on bytecode level projects. Although the other studies are certainly useful, and closely related to this work in some respects, they do not address the unique way in which the bytecode language is used and the special structures in JVML.

In addition to the framework developed by Stata and Abadi [SA98] and used in this paper, there are other strategies being developed to describe the JVML type system and bytecode verification formally. The most closely related work is [Qia98], which presents a static type system for a larger fragment of JVML than is presented here. While that system uses the same general approach as we do, we have attempted to present a simpler type system by abstracting away some of the unnecessary details left in Qian's framework, such as different forms of name resolution in the constant pool and varying instruction lengths. Also, our model of subroutines, based on the work of Stata and Abadi, is very different. The rules for object initialization used in the original version of Qian's paper were similar to Sun's faulty rules, and they incorrectly accepted the program in Figure 9. After announcing our discovery of Sun's bug, a revised version of Qian's paper containing rules more similar to our rules was released.

Hagiya and Tozawa present a type system for the fragment of JVML concerning subroutines [HT98]. We are currently examining ways in which ideas from that type system may used to eliminate some of the simplifications to the subroutine mechanism in the work of Stata and Abadi.

Another approach using concurrent constraint programming is also being developed [Sar97]. This approach is based on transforming a JVML program into a concurrent constraint program. While this approach must also deal with the difficulties in analyzing subroutines and object initialization statically, it remains to be seen whether it will yield a better framework for studying JVML, and whether the results can be easily translated into a verifier specification.

Other avenues toward a formal description of the verifier, including model checking [PV98] and data flow analysis techniques [Gol98], are also currently being pursued.

A completely different approach has been taken by Cohen, who is developing a formal execution model for JVML which does not require bytecode verification [Coh97]. Instead, safety checks are built into the interpreter. Although these run-time checks make the performance of his defensive JVM too slow to use in practice, this method is useful for studying JVML execution and understanding the checks required to safely execute a program.

The Kimera project has developed a more experimental method to determine the correctness of existing bytecode verifiers [SMB97]. After implementing a verifier from scratch, programs with randomly inserted errors were fed into that verifier, as well as several commercially produced verifiers. Any differences among implementations meant a potential flaw. While this approach is fairly good at tracking down certain classes of implementation mistakes and is effective from a software engineering perspective, it does not lead to the same concise, formal model like some of the other approaches, including the approach presented in this paper. It also may not find JVML specification errors or more complex bugs, such as the one described in Section 7.

Other recent work has studied type systems for low-level languages other than JVML. These studies include the TIL intermediate languages for ML [TMC+96], and the more recent work on typed assembly language [MCGW98]. The studies touch on some of the same issues as this study, and the type system for typed assembly language does contain a distinction between types for initialized and uninitialized values. However, these languages do not contain some of the constructs found in JVML, and they do not require aspects of the static analysis required for JVML, such as the alias analysis required for object initialization.

9 Conclusions and Future Work

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 no 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 which covers both subroutines and object initialization. This is one step towards developing a sound type system for the whole bytecode language. Once this type system for JVML is complete, we can describe a formal specification of the verifier and better understand what safety and security guarantees can be made by it.

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 the interaction between it and subroutines. 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 this paper.

The work described in this paper opens several promising directions. One major task, which we are currently undertaking, is to extend the specification and correctness proof to the entire JVML, including the method call stack and a full object system. The methods described in Section 6 allow most variants of simple instructions to be added in a standard, straightforward way, and we are also examining methods to factor JVML into a complete, yet minimal, set of instructions. In addition, the Java object system has been studied and discussed in other contexts [AG96, Sym97, DE97, Qia98], and these previous results can be used as a basis for objects in our JVML model. We are in the process of finishing a soundness proof for a language encompassing the JVML elements presented in this paper plus objects, interfaces, classes, arrays, and exceptions. Other issues that have not been addressed to date are concurrency and dynamic loading, both of which are key concepts in the Java Virtual Machine.

We also believe it will be feasible to generate an implementation of a bytecode verifier from a specification proven to be correct. This specification could be expressed in the kind of typing rules we use here, or some variant of this notation.

Finally, we expect that in the long run, it will be useful to incorporate additional properties into the static analysis of Java programs. If Java is to become a popular and satisfactory general-purpose programming language, then for efficiency reasons alone, it will be necessary to replace some of the current run-time tests by conservative static analysis, perhaps reverting to run-time tests when static analysis fails. For example, we may be able to eliminate some run-time checks for array bounds and pointer casts. Other safety properties, such as the use of certain locking conventions in a concurrent JVML model, could also be added to our static analysis.

Acknowledgments: Thanks to Martín Abadi and Raymie Stata (DEC SRC) for their assistance on this project. We thank Li Gong for his encouragement, and Frank Yellin and Sheng Liang of JavaSoft for several useful discussions.

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[Coh97] Rich Cohen. Defensive Java Virtual Machine Version 0.5 alpha Release. Available from http://www.cli.com/software/djvm/index.html, November 1997.

[DE97] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. In *European Conference On Object Oriented Programming*, pages 389–418, 1997.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to netscape and beyond. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 190–200, 1996.

[Gol98] Allen Goldberg. A specification of java loading and bytecode verification. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, 1998. Available from http://www.kestrel.edu/~goldberg.

[HT98] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. Available from http://nicosia.is.s.u-tokyo.ac.jp/members/hagiya.html. A preliminary version appeared in SIG-Notes, PRO-17-3, Information Processing Society of Japan, 1998.

[Lia97] Sheng Liang. personal communication, November 1997.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. From system F to typed assembly language. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.

[NvO98] Tobias Nipkow and David von Oheimb. Java$_{light}$ is Type-Safe - Definitely. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.

[PV98] Joachim Posegga and Harald Vogt. Byte code verification for java smart cards based on model checking. In *5th European Symposium on Research in Computer Security (ESORICS)*, Louvain-la-Neuve, Belgium, 1998. Springer LNCS.

[Qia98] Zhenyu Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java.* Springer-Verlag, 1998.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.

[Sar97] Vijay Saraswat. The Java bytecode verification problem. Available from http://www.research.att.com/~vj, November 1997.

[SMB97] Emin Gün Sirer, Sean McDirmid, and Brian Bershad. Kimera: A Java system architecture. Available from http://kimera.cs.washington.edu, November 1997.

[Sym97] Don Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory Technical Report, 1997.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. *ACM SIGPLAN Notices*, 31(5):181–192, May 1996.

Sun, Sun Microsystems, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.