A Type System for Object Initialization in the Java Bytecode Language

STEPHEN N. FREUND and JOHN C. MITCHELL
Stanford University

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 executed 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 run. These checks include type correctness and, as illustrated by previous attacks on the Java Virtual Machine, 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 article 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 Our analysis of subroutines and object initialization reveals a previously unpublished bug in the Sun JDK bytecode verifier.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.1 [**Theory of Computation**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification*; F.3.3 [**Theory of Computation**]: Studies of Program Constructs—*type structure*

General Terms: Languages, Verification

Additional Key Words and Phrases: Bytecode languages, Java, object initialization, type checking

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 a Java bytecode program, and this program is then interpreted by the Java Virtual

The authors were supported in part by NSF grants CCR-9303099 and CCR-9629754, and ONR MURI Award N00014-97-1-0505. S. Freund received additional support through an NSF Graduate Research Fellowship. A preliminary version of this article appeared at OOPSLA '98.

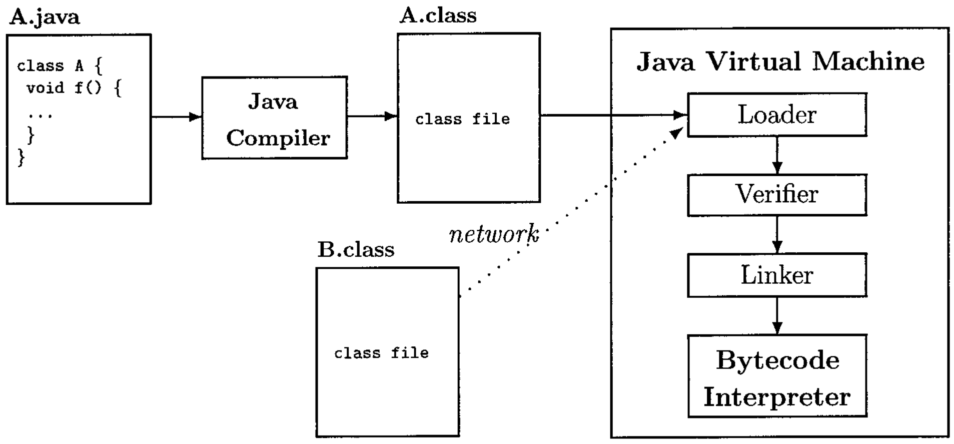

Fig. 1. The Java Virtual Machine.

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 programs 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 executed. 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 every opcode is valid, that all jumps lead to legal instructions, that methods have structurally correct signatures, and that type constraints are satisfied. Conservative static analysis techniques are used to check these conditions. The need for conservative analysis stems from the undecidability of the halting problem, as well as efficiency considerations. 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. Since most bytecode programs are 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 `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 program. 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 [Stata and Abadi 1999].

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 object reference `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 a bytecode program, these instructions 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 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 various implementations 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 [Dean et al. 1997]. 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 [Stata and Abadi 1999], this article 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 type systems for high-level languages. 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. One of these trade-offs is discussed in Section 6.

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 instructions 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 ways 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 article. The operational semantics and type system for this language are presented in Section 4. Some sound extensions to our type system, 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, as 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.

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. As a convention of the stack-based virtual machine architecture, 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 references to 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 [Lindholm and Yellin 1996] 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 in 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 to two different objects belonging to the same equivalence class would exist. 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. Sun corrected this bug in the verifier after we reported the problem to them.

3. JVML$_i$

This section describes the JVML$_i$ language, an idealized subset of JVML including basic constructs and object initialization. Although this language is much smaller than JVML and simplified in certain ways, 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 a program counter, 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:

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

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 1 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. The top element on the stack must be 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 instruction 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. The `use` instruction is an abstraction of several operations in JVML, including method invocation (`invokevirtual`) and accessing an instance field (`putfield`/`getfield`).

Errors occur when these instructions are executed with the machine in an invalid state. 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 operational semantics presented 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 [1999] for studying JVML. We begin with a set of instruction addresses ADDR. Although we shall write elements of this set as positive integers, we will distinguish elements of ADDR from integers. A program P is formally represented by a partial function from addresses to instructions, where $Dom(P)$ is the set of addresses in program P, and $P[i]$ is the instruction at index i 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 as follows for any partial map f:

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

where $y \in Dom(f)$, and a, b, and v are in the codomain of f.

For sequences, we write ϵ for the empty sequence and write $v \cdot s$ to place v on the front of sequence s. A sequence of one element, $v \cdot \epsilon$, will sometimes be abbreviated to v. Appending one sequence to another is written as $s_1 \bullet s_2$. This operation can be defined by the two equations $\epsilon \bullet s = s$ and $(v \cdot s_1) \bullet s_2 = v \cdot (s_1 \bullet s_2)$. One final operation on sequences is substitution:

$$\begin{aligned} ([b/a]\epsilon) &= \epsilon \\ [b/a](v \cdot s) = ([b/a]v) \cdot ([b/a]s) &= \begin{cases} b \cdot ([b/a]s) & \text{if } v = a \\ v \cdot ([b/a]s) & \text{otherwise} \end{cases} \end{aligned}$$

where a, b, and v are of the same kind as what is being stored in sequence s.

4.2 Values and Types

The types include integers and object types. For objects, there is a 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 depends on T:

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

The type $\hat{\sigma}_i$ is used for an object of type σ allocated on line i of a program, that it has been initialized. Using this notation, JVML$_i$ types are generated by the grammar

$$\tau ::= \text{INT} \mid \sigma \mid \hat{\sigma}_i \mid \text{TOP}$$

where $\sigma \in T$ and $\hat{\sigma}_i \in \hat{T}$. The type INT will be used for integers. We discuss the addition of other basic types in Section 6. The type TOP is the supertype of all types, with any value of any type also having type TOP. This type will represent unusable values in our static analysis. In general, a type metavariable τ may refer to any type, including INT, TOP, any object type $\sigma \in T$, or uninitialized-object type $\hat{\sigma}_i \in \hat{T}$. In the case that a type metavariable 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 τ has a corresponding infinite set of values, A^{τ}, different from values of any other type. 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

$$\frac{v \text{ is a value}}{v : \text{TOP}} \quad \frac{n \text{ is an integer}}{n : \text{INT}} \quad \frac{a \in A^{\tau},\ \tau \in T \cup \hat{T}}{a : \tau}.$$

We also extend values and types to sequences:

$$\frac{}{\epsilon : \epsilon} \quad \frac{a : \tau \quad s : \alpha}{a \cdot s : \tau \cdot \alpha} \quad \frac{s_1 : \alpha_1 \quad s_2 : \alpha_2}{s_1 \bullet s_2 : \alpha_1 \bullet \alpha_2}$$

4.3 Operational Semantics

The bytecode interpreter for JVML$_i$ 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 $\langle pc, f, s\rangle$ with the following meaning:

—pc is a program counter indicating the address of the instruction about to be executed.

—f is a total map from a set VAR of local variables to the values stored in the local variables in the current state. The set VAR is a finite set of local variable names, which are drawn from the set of integers.

—s is a stack of values representing the operand stack for the current state in execution.

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

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

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

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

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

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

Fig. 2. JVML$_i$ operational semantics.

The machine begins execution in state $\langle 1,\ f_0,\ \epsilon\rangle$ where f_0 may map the local variables to any values. In this state, (1) the first instruction in the program is about to be executed, (2) the operand stack is empty, and (3) the local variables may contain any values. If method invocation were included in our model, the local variables would be initialized to contain the parameter values.

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

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

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 rules, with the exception of those added to study object initialization, are discussed in detail in Stata and Abadi [1999]. 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 or to load a value from a variable not in VAR.

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 is defined by

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

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 also required to be unused, allowing the program to distinguish between different objects of the same type after they have been initialized. The `use` instruction generates a run-time error when it is applied to an uninitialized object reference.

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 partial map from ADDR to functions mapping local variables to types, and S is a partial map from ADDR to stack types such that S_i is the type of the operand stack at location i of the program. Following Stata and Abadi [1999], application of the partial map F to address $i \in$ ADDR is written 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 [Lindholm and Yellin 1996] describes the verifier as both computing the type information stored in F and S and checking it. However, we assume that the information given in F and S has already been computed prior to the type-checking stage. This simplifies matters, since it separates synthesis from checking and prevents the type synthesis from complicating the static semantics.

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

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

where F_{TOP} is a function mapping all variables in VAR to TOP. The first two lines of (*wt prog*) set the initial conditions for program execution to the types of the values stored in the variables and on the stack in the initial

$$(inc)\quad \frac{P[i] = \texttt{inc} \quad F_{i+1} = F_i \quad S_{i+1} = S_i = \text{INT} \cdot \alpha \quad i+1 \in Dom(P)}{F, S, i \vdash P}$$

$$(if)\quad \frac{P[i] = \texttt{if}\ L \quad F_{i+1} = F_L = F_i \quad S_i = \text{INT} \cdot S_{i+1} = \text{INT} \cdot S_L \quad i+1 \in Dom(P) \quad L \in Dom(P)}{F, S, i \vdash P}$$

$$(pop)\quad \frac{P[i] = \texttt{pop} \quad F_{i+1} = F_i \quad S_i = \tau \cdot S_{i+1} \quad i+1 \in Dom(P)}{F, S, i \vdash P}$$

$$(push\ 0)\quad \frac{P[i] = \texttt{push 0} \quad F_{i+1} = F_i \quad S_{i+1} = \text{INT} \cdot S_i \quad i+1 \in Dom(P)}{F, S, i \vdash P}$$

$$(load)\quad \frac{P[i] = \texttt{load}\ x \quad x \in Dom(F_i) \quad F_{i+1} = F_i \quad S_{i+1} = F_i[x] \cdot S_i \quad i+1 \in Dom(P)}{F, S, i \vdash P}$$

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

$$(halt)\quad \frac{P[i] = \texttt{halt}}{F, S, i \vdash P}$$

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

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

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

Fig. 3. JVML$_i$ static semantics.

state. The third line requires that each instruction in the program is well typed according to the local judgments presented in Figure 3. Most of these rules are relatively straightforward.

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 applicable only 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.

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

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

The rule for (*use*) requires an initialized object type on top of the stack. The (*init*) rule is the key to 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. This is proved using a one-step soundness theorem. One-step soundness means that any valid transition from a well-formed state leads to another well-formed state. The four factors indicating whether or not a state is well formed are described in more detail below.

THEOREM 1 (ONE-STEP SOUNDNESS). *Given P, F, and S such that $F, S \vdash P$,*

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall pc, f, s, pc', f', s'.} \\
& P \vdash \langle pc, f, s\rangle \to \langle pc', f', s'\rangle \\
& \wedge\ s : S_{pc} \\
& \wedge\ \forall y \in \text{VAR}.\ f[y] : F_{pc}[y] \\
& \wedge\ ConsistentInit(F_{pc}, S_{pc}, f, s) \\
\Rightarrow & s' : S_{pc'} \\
& \wedge\ \forall y \in \text{VAR}.\ f'[y] : F_{pc'}[y] \\
& \wedge\ ConsistentInit(F_{pc'}, S_{pc'}, f', s') \\
& \wedge\ pc' \in Dom(P)
\end{array}
$$

The first condition that a well-formed state must satisfy is that the values on the operand stack must have the types expected by the static type rules, written $s : S_{pc}$. Likewise, the line $\forall y \in$ VAR. $f[y] : F_{pc}[y]$

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

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

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

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

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

Fig. 5. The *ConsistentInit* judgment.

requires that the local variable contents match the types in F. In addition, the program counter must always be in the domain of the program, written $pc \in \mathit{Dom}(P)$. This can be assumed on the left-hand side of the implication, since the operational semantics guarantees that transitions can be made only if $P[pc]$ is defined. If the program counter were not in the domain of the program, no step could be taken.

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

Figure 5 presents the formal definition of *ConsistentInit*. In that figure, F_i is a map from local variables to types, and 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. The stack correspondence is defined inductively to match the way in which most instructions manipulate the stack. 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 $P[pc]$ and appears in Appendix A. Theorem 1 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.

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 follows:

THEOREM 2 (PROGRESS). *Given P, F, and S such that $F, S \vdash P$,*

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

By combining Theorem 1 with Theorem 2, we may prove that a program will never get stuck unless it reaches a `halt` instruction. When it does reach a `halt` instruction, the stack will have the correct type. Showing that the program ends with a valid stack is important because the return value for a program, or method in the full JVML, is returned as the top value on the stack. The main soundness theorem states these properties:

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. In particular, 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, e.g., 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*) rule.

6. EXTENSIONS

We have studied extensions to the JVML$_i$ framework described in the previous sections. 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 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$_i$. 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).

The rules for checking constructors are defined in Lindholm and Yellin [1996] and can be summarized by three basic points:

(1) When a constructor is invoked, local variable 0 contains a reference to the object that is being initialized.
(2) 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.
(3) The only deviation from the second 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 nontermination 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 interprocedural analysis, this type of analysis falls outside of the bounds of the current verifier, which was designed to check only one method at a time.

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 begins in a state in which local variable 0 contains an uninitialized reference, the argument of the constructor. Prior to halting, the program must call `super` φ on that object reference. This instruction represents a call to the superclass constructor or a different constructor the current class.

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 ADDR includes 0, although 0 will not be in the domain of any program. Also, the machine state in the operational semantics is augmented with a fourth element, z, which indicates whether or not a superclass 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

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

For `super`, the operational semantics rule is

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

As stated 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 $\hat{a}_{\varphi} \in A^{\hat{\varphi}_0}$.

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 in Figure 3.

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

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

The (*wt constructor*) rule is analogous to (*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 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 stored local variable 0 at the beginning of execution:

THEOREM 4 (CONSTRUCTOR SOUNDNESS). *Given P, F, S, φ, and $\hat{a}_\varphi$ such that $F, S \vdash P$ constructs φ and $\hat{a}_\varphi : \hat{\varphi}_0$:*

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

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

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

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

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

Fig. 6. Rules checking that a superclass constructor will always be called prior to reaching a `halt` instruction.

The main difference between the proof of Theorem 4 and the proof of Theorem 3 is that the corresponding one-step soundness theorem requires an additional invariant. The invariant states, that, when program P is in state $\langle pc, f, s, z\rangle$, we have $z = Z_{P,pc}$. The proof of this theorem appears in Appendix B.1.

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 [Hagiya and Tozawa 1998; Lindholm and Yellin 1996; O'Callahan 1999; Sarta and Abadi 1999]. 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:

(1) Subroutines are polymorphic over local variables which they do not use.
(2) Subroutines may call other subroutines, as long as a last-in first-out ordering is preserved. In other words, the most recently called subroutine must be the first one to return.

The second condition is a slight simplification of the rules for subroutines defined in Lindholm and Yellin [1996], which do allow a subroutine to

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

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

Fig. 7. Operations semantics for `jsr` and `ret`.

return more than one level up in the implicit subroutine call stack in certain cases, but does match the definitions presented in Sarti and Abadi [1999]. A comparison between our rules and the Sun rules appears in Section 7.

JVML$_s$ programs contain the same set of instructions as JVML$_i$ programs and 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.
`ret` x:	jumps to the instruction address stored in local variable x.

The operational semantics and typing rules for these instructions are shown in Figures 7 and 8. These rules are based on the rules used by Stata and Abadi [1999]. 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 article 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:

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

The same lines are added to (*jsr*). The discussion of the interaction between subroutines and uninitialized objects in the Java Virtual Machine specifi-

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

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

Fig. 8. Type rules for jsr and ret.

cation is vague and inconsistent with current implementations, but the rules we have developed seem to fit the general strategy described in the specification.

This solution 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 are not likely to be created by reasonable Java compilers.

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 Appendix B.2.

6.3 Other Basic Types and Instructions

Of the 200 bytecode instructions in JVML, all but approximately 40 are variants of simple operations for different basic types. For example, there are four `add` instructions corresponding to addition on values of type INT, FLOAT, LONG, and DOUBLE. 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. An example showing how these simple instructions can be added to our framework appears in the Appendix.

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 overwritten two-word value, it does not pose any serious difficulty.

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. To add concurrency would require incorporating the Java Virtual Machine shared-memory model, which has been studied elsewhere in Knapp et al. [1998].

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$_s$.

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 (the current verifier at the time we discovered the problem). 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.

```
 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
```

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

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 Stata and Abadi [1999], 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 among different parts of the language, such as initialization and subroutines. In fact, the bug was uncovered by simply constructing test programs reflecting tricky cases in the soundness proof.

7.2 The Corrected Sun Verifier

After learning about this bug, the Sun development team has taken steps to repair their verifier implementation. While they did not use the exact rules we have presented in this article, 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 (personal communication, Sheng Liang, Nov. 1997):

—Uninitialized objects may appear anywhere in the local variables or on the operand stack at `jsr` L or `ret` x instructions, but they cannot be used after the instruction has executed. In other words, their static type is made TOP in the postinstruction 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.

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

```
    // try block body
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

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

Fig. 10. A constructor which will always call a subroutine before invoking the superclass constructor, and its translation into JVML$_c$ with subroutines (ignoring the exception handler). Note that this is not a valid Java program.

—The static types assigned to uninitialized objects passed into constructors, i.e., any value whose type is of the form $\hat{\sigma}_0$ in our framework, 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 superclass 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 object passed into the constructor. It is accepted by the Sun verifier due to the 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 the development team's 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 bytecode programs 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 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 [Drossopoulou et al. 1999; Nipkow and von Oheimb 1998; Syme 1997], but we will emphasize bytecode-level projects. Although the other studies are certainly useful, and closely related to this work in some respects, they do not address the way in which the bytecode language is used and the special structures in JVML.

In addition to the framework developed by Stata and Abadi [1999] and used in this article, there are other strategies being developed to describe the JVML type system and bytecode verification formally. The most closely related work is Qian [1998], 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 [1998] present a type system for the fragment of JVML concerning subroutines. We are currently examining ways in which ideas from that type system may be used to eliminate some of the simplifications to the subroutine mechanism in the work of Stata and Abadi. Several recent projects have departed from Sun's original specification and have developed significantly different static semantics for JVML subroutines. O'Callahan [1999] presents a system based on ideas from the TAL type system of Morrisett et al. [1998]. Other work has borrowed ideas from the type system of Haskell [Jones 1998; Yelland 1999].

Another approach using concurrent constraint programming was also proposed [Saraswat 1997]. 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 [Posegga and Vogt 1998] and data flow analysis techniques [Goldberg 1998], are also currently being pursued.

A completely different approach was taken by Cohen [1997], who developed a formal execution model for JVML which does not require bytecode verification. 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 [Sirer et al. 1997]. 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 article. 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 [Tarditi et al. 1996], and the more recent work on typed assembly language [Morrisett et al. 1998]. 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 toward 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.

The work described in this article 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 [Arnold and Gosling 1996; Drossopoulou et al. 1999; Qian 1998; Syme 1997], 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 article 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 in the long run, that 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.

APPENDIX A. JVML$_i$ SOUNDNESS

This appendix presents the soundness proof for the JVML$_i$ type system. Appendix A.1 proves some useful lemmas used in Appendices A.2, A.3, and A.4, which contain the proofs of Theorems 1, 2, and 3. Appendix B extends these proofs to show the soundness of the JVML$_c$ and JVML$_s$ type systems.

A.1 Preliminary Lemmas

We begin with two lemmas that conclude the correspondence between specific values and types based, first, on the contents of the top of the stack, and, second, on the contents of a specific local variable.

LEMMA 1.

$$\begin{aligned}
&\forall F_i,\ S_i,\ f,\ s,\ \hat{\tau},\ \hat{b}.\\
&\quad \hat{\tau} \in \hat{T}\\
&\quad \wedge\ ConsistentInit(F_i,\ \hat{\tau} \cdot S_i,\ f,\ \hat{b} \cdot s)\\
&\Rightarrow\ Corresponds(F_i,\ \hat{\tau} \cdot S_i,\ f,\ \hat{b} \cdot s,\ \hat{b},\ \hat{\tau})
\end{aligned}$$

PROOF. Assume that all the hypotheses are satisfied for some F_i, S_i, f, s, $\hat{\tau}$, and $\hat{b}$. Since *ConsistentInit*(F_i, $\hat{\tau} \cdot S_i$, f, $\hat{b}$, s), there is some $\hat{c}$ such that *Corresponds*(F_i, $\hat{\tau} \cdot S_i$, f, $\hat{b} \cdot s$, $\hat{c}$, $\hat{\tau}$). *StackCorresponds*($\hat{\tau} \cdot S_i$, $\hat{b} \cdot s$, $\hat{c}$, $\hat{\tau}$) must be true to have concluded *Corresponds*(F_i, $\hat{\tau} \cdot S_i$, f, $\hat{b} \cdot s$, $\hat{c}$, $\hat{\tau}$), and the only way by which we could have concluded this is using rule (*sc 1*). Therefore $\hat{b} = \hat{c}$, and *Corresponds*$(F_i, \hat{\tau} \cdot S_i, f, \hat{b} \cdot s, \hat{b}, \hat{\tau})$. □

LEMMA 2.

$$\begin{aligned} &\forall F_i, S_i, f, s, x. \\ &\qquad F_i[x] \in \hat{T} \\ &\qquad \wedge\ ConsistentInit(F_i, S_i, f, s) \\ &\quad \Rightarrow\ Corresponds(F_i, S_i, f, s, f[x], F_i[x]) \end{aligned}$$

PROOF. Assume that all the hypotheses are satisfied for some F_i, S_i, f, s, and $\hat{b}$. Since *ConsistentInit*(F_i, S_i, f, s), there is some $\hat{c}$ such that *Corresponds*$(F_i, S_i, f, s, \hat{c}, F_i[x])$. We know that $\hat{c} = f[x]$, or else we could not have derived the above correspondence. □

The next three lemmas show that *Corresponds* and *ConsistentInit* are preserved when values are popped off the stack. We first show that *Corresponds* is preserved for a single pop.

LEMMA 3.

$$\begin{aligned} &\forall F_i, S_i, f, s, v, \tau, \hat{b}, \hat{\tau}. \\ &\qquad Corresponds(F_i, \tau \cdot S_i, f, v \cdot s, \hat{b}, \hat{\tau}) \\ &\quad \Rightarrow\ Corresponds(F_i, S_i, f, s, \hat{b}, \hat{\tau}) \end{aligned}$$

PROOF. Assume that the hypothesis is satisfied. Since we assumed *Corresponds*$(F_i, \tau \cdot S_i, f, v \cdot s, \hat{b}, \hat{\tau})$, we know that

$$\forall x \in Dom(F_i).\ F_i[x] = \hat{\tau} \Rightarrow f[x] = \hat{b}. \tag{1}$$

Also, *StackCorresponds*$(\tau \cdot S_i, v \cdot s, \hat{b}, \hat{\tau})$. If this is true, we must be able to conclude it by either (*sc 1*) or (*sc 2*). In both cases, *StackCorresponds*$(S_i, s, \hat{b}, \hat{\tau})$ must be true. From this and (1), *Corresponds*$(F_i, S_i, f, s, \hat{b}, \hat{\tau})$ follows from rule (*corr*). □

Using this lemma, we may prove the same property for *ConsistentInit*.

LEMMA 4.

$$\begin{aligned} &\forall F_i, S_i, f, s, v, \tau. \\ &\qquad ConsistentInit(F_i, \tau \cdot S_i, f, v \cdot s) \\ &\quad \Rightarrow\ ConsistentInit(F_i, S_i, f, s) \end{aligned}$$

PROOF. Assume that the hypothesis is satisfied for some choice of F_i, S_i, f, s, v, τ. For any $\hat{\tau}$, choose $\hat{b}$ such that *Corresponds*$(F_i, \tau \cdot S_i, f, v \cdot s, \hat{b}, \hat{\tau})$. Such a $\hat{b}$ exists by our assumption that *ConsistentInit*$(F_i, \tau \cdot S_i, f, v \cdot s)$. For this choice of $\hat{b}$ and $\hat{\tau}$, *Corresponds*$(F_i, S_i, f, s, \hat{b}, \hat{\tau})$ follows from Lemma 3. Since a $\hat{b}$ may be chosen for every $\hat{\tau}$ in this way, *ConsistentInit*(F_i, S_i, f, s) follows. □

The previous lemma may be generalized to popping any number values off the stack, as shown below.

LEMMA 5.

$$\begin{array}{l}\forall F_i,\ \alpha_1,\ \alpha_2,\ f,\ s_1,\ s_2. \\ \quad s_1 : \alpha_1 \\ \quad \wedge\ ConsistentInit(F_i,\ \alpha_1 \bullet \alpha_2,\ f,\ s_1 \bullet s_2) \\ \Rightarrow\ ConsistentInit(F_i,\ \alpha_2,\ f,\ s_2)\end{array}$$

PROOF. The proof is by induction on the length of α_1. □

In a fashion similar to the previous three lemmas, we also prove that pushing any number of values does not affect *Corresponds* or *ConsistentInit*, as long as the new values are not uninitialized objects. As before, we start with *Corresponds*:

LEMMA 6.

$$\begin{array}{l}\forall F_i,\ S_i,\ f,\ s,\ v,\ \tau,\ \hat{b},\ \hat{\tau}. \\ \quad \tau \neq \hat{\tau} \\ \quad \wedge\ Corresponds(F_i,\ S_i,\ f,\ s,\ \hat{b},\ \hat{\tau}) \\ \Rightarrow\ Corresponds(F_i,\ \tau \cdot S_i,\ f,\ v \cdot s,\ \hat{b},\ \hat{\tau})\end{array}$$

PROOF. Assume that the hypotheses are satisfied. Since we assumed *Corresponds*$(F_i,\ S_i,\ f,\ s,\ \hat{b},\ \hat{\tau})$, the following implication holds:

$$\forall x \in Dom(F_i).\ F_i[x] = \hat{\tau} \Rightarrow f[x] = \hat{b} \tag{2}$$

Also, *StackCorresponds*$(S_i,\ s,\ \hat{b},\ \hat{\tau})$ must be true. Given that $\tau \neq \hat{\tau}$, it is clear that *StackCorresponds*$(\tau \cdot S_i,\ v \cdot s,\ \hat{b},\ \hat{\tau})$ follows from rule (*sc 2*). Using this and (2), *Corresponds*$(F_i,\ \tau \cdot S_i, f,\ v \cdot s,\ \hat{b},\ \hat{\tau})$ follows by (*corr*). □

LEMMA 7.

$$\begin{array}{l}\forall F_i,\ S_i,\ f,\ s,\ v,\ \tau. \\ \quad \tau \notin \hat{T} \\ \quad \wedge\ ConsistentInit(F_i,\ S_i,\ f,\ s) \\ \Rightarrow\ ConsistentInit(F_i,\ \tau \cdot S_i,\ f,\ v \cdot s)\end{array}$$

PROOF. Assume the hypotheses are satisfied. For any $\hat{\tau} \in \hat{T}$, choose $\hat{b}$ such that we have *Corresponds*$(F_i,\ S_i,\ f,\ s,\ \hat{b},\ \hat{\tau})$. Such a $\hat{b}$ exists by our assumption that *ConsistentInit*$(F_i,\ S_i,\ f,\ s)$. Also, for this choice of $\hat{b}$ and $\hat{\tau}$, we prove *Corresponds*$(F_i,\ \tau \cdot S_i,\ f,\ v \cdot s,\ \hat{b},\ \hat{\tau})$ using Lemma 6. Since $\tau \notin \hat{T}$, we know that $\tau \neq \hat{\tau}$. All other conditions of Lemma 6 are satisfied, implying *Corresponds*$(F_i,\ \tau \cdot S_i,\ f,\ v \cdot s,\ \hat{b},\ \hat{\tau})$. Since a $\hat{b}$ can be chosen in this way for all $\hat{\tau}$, we conclude *ConsistentInit*$(F_i,\ \tau \cdot S_i,\ f,\ v \cdot s)$ by (*cons init*). □

LEMMA 8.

$$\begin{array}{l} \forall F_i,\ \alpha_1,\ \alpha_2,\ f,\ s_1,\ s_2. \\ \qquad s_1 : \alpha_1 \\ \qquad \wedge\ \forall y \in Dom(\alpha_1).\ \alpha_1[y] \notin \hat{T} \\ \qquad \wedge\ ConsistentInit(F_i,\ \alpha_2,\ f,\ s_2) \\ \quad \Rightarrow\ ConsistentInit(F_i,\ \alpha_1 \bullet \alpha_2,\ f,\ s_1 \bullet s_2) \end{array}$$

PROOF. The proof is by induction on the length of α_1. □

The next lemma shows that a value known to correspond to a certain uninitialized object type may be stored in a local variable without breaking the correspondence.

LEMMA 9.

$$\begin{array}{l} \forall F_i,\ S_i,\ f,\ s,\ \hat{b},\ \hat{\tau},\ x. \\ \qquad Corresponds(F_i,\ S_i,\ f,\ s,\ \hat{b},\ \hat{\tau}) \\ \quad \Rightarrow\ Corresponds(F_i[x \mapsto \hat{\tau}],\ S_i,\ f[x \mapsto \hat{b}],\ s,\ \hat{b},\ \hat{\tau}) \end{array}$$

PROOF. Assuming $Corresponds(F_i,\ S_i,\ f,\ s,\ \hat{b},\ \hat{\tau})$, we know that

$$StackCorresponds(S_i,\ s,\ \hat{b},\ \hat{\tau}).$$

In order to use (*corr*) to prove the conclusion of this lemma, we must also show that $\forall y \in Dom(F_i[x \mapsto \hat{\tau}])$,

$$(F_i[x \mapsto \hat{\tau}])[y] = \hat{\tau} \Rightarrow (f[x \mapsto \hat{b}])[y] = \hat{b}. \tag{3}$$

There are two cases to consider for each y:

Case 1: $x \neq y$. In this case, $(F_i[x \mapsto \hat{\tau}])[y] = F_i[y]$. Likewise, $(f[x \mapsto \hat{b}])[y] = f[y]$. Since $Corresponds(F_i,\ S_i,\ f,\ s,\ \hat{b},\ \hat{\tau})$ is true, line (3) must be true for this choice of y.

Case 2: $x = y$. In this case, $(F_i[x \mapsto \hat{\tau}])[y] = \hat{\tau}$ and $(f[x \mapsto \hat{b}])[y] = \hat{b}$. Thus, (3) holds when $x = y$.

Thus, the conditions for (*corr*) are satisfied, and we may conclude that the lemma holds. □

Similarly, a value known not to be an uninitialized object of a certain type may be stored in a local variable without breaking any known correspondence, as shown below.

LEMMA 10.

$$\begin{aligned}
\forall F_i,\ & S_i, f, s, v, \tau, \hat{b}, \hat{\tau}, x. \\
& \tau \neq \hat{\tau} \\
& \wedge\ \mathit{Corresponds}(F_i, S_i, f, s, \hat{b}, \hat{\tau}) \\
\Rightarrow\ & \mathit{Corresponds}(F_i[x \mapsto \tau], S_i, f[x \mapsto v], s, \hat{b}, \hat{\tau})
\end{aligned}$$

PROOF. Assume that the hypotheses are satisfied. Since we assumed *Corresponds*(F_i, S_i, f, s, $\hat{b}$, $\hat{\tau}$), we know that

$$\mathit{StackCorresponds}(S_i, s, \hat{b}, \hat{\tau}).$$

In order to use (*corr*) to prove the conclusion of this lemma, we must also show, for all $y \in \mathit{Dom}(F_i[x \mapsto \tau])$, that

$$(F_i[x \mapsto \tau])[y] = \hat{\tau} \Rightarrow (f[x \mapsto v])[y] = \hat{b}. \tag{4}$$

There are two cases to consider for each y:

Case 1: $x \neq y$. In this case, $(F_i[x \mapsto \tau])[y] = F_i[y]$. Likewise, $(f[x \mapsto v])[y] = f[y]$. Since *Corresponds*($F_i$, S_i, f, s, $\hat{b}$, $\hat{\tau}$) is true, line (4) must be true for this choice of y.

Case 2: $x = y$. In this case, $(F_i[x \mapsto \tau])[y] \neq \hat{\tau}$, and (4) is satisfied when $x = y$.

Thus, the conditions for (*corr*) are satisfied, and we may conclude that the lemma holds. □

The next two lemmas in this section are concerned with substitution. The first shows that substitution of an initialized object type for an uninitialized object type, and an initialized object for the corresponding uninitialized object, preserves the stack type. Also, the correspondence between the uninitialized object type and value on the stack is preserved by the substitution.

LEMMA 11.

$$\begin{aligned}
\forall S_i,\ & s, a : \sigma, \hat{a} : \hat{\sigma}. \\
& s : S_i \\
& \wedge\ \mathit{StackCorresponds}(S_i, s, \hat{a}, \hat{\sigma}) \\
& \wedge\ \sigma \neq \hat{\sigma} \\
& \wedge\ \hat{\sigma} \in \hat{T} \\
\Rightarrow\ & [a/\hat{a}]s : [\sigma/\hat{\sigma}]S_i \\
& \wedge\ \mathit{StackCorresponds}([\sigma/\hat{\sigma}]S_i, [a/\hat{a}]s, \hat{a}, \hat{\sigma})
\end{aligned}$$

PROOF. Assume that all the hypotheses are satisfied. We prove the conclusions by induction on the derivation of *StackCorresponds*$(S_i, s, \hat{a}, \hat{\sigma})$. There is one base case and two inductive cases to consider, depending on which judgment is used in the final step of the derivation:

Case 1: (*sc 0*). If this is the case, $s = \epsilon$ and $S_i = \epsilon$. The conclusions follow trivially.

Case 2: (*sc 1*). In this case, $s = \hat{a} \cdot s'$ and $S_i = \hat{\sigma} \cdot S'_i$ for some s' and S'_i. We begin by showing that $[a/\hat{a}](\hat{a} \cdot s') : [\sigma/\hat{\sigma}](\hat{\sigma} \cdot S'_i)$. Since $s' : S'_i$ and *StackCorresponds*$(S'_i, s', \hat{a}, \hat{\sigma})$, we may conclude that

$$[a/\hat{a}]s' : [\sigma/\hat{\sigma}]S'_i \tag{5}$$

and

$$StackCorresponds([\sigma/\hat{\sigma}]S'_i, [a/\hat{a}]s', \hat{a}, \hat{\sigma}) \tag{6}$$

by the inductive hypothesis. It is also clear that the following two equations hold:

$$[a/\hat{a}]\hat{a} = a \tag{7}$$

$$[\sigma/\hat{\sigma}]\hat{\sigma} = \sigma \tag{8}$$

These allow us to conclude that $[a/\hat{a}]\hat{a} : [\sigma/\hat{\sigma}]\hat{\sigma}$. Combining this fact and line (5), we know that $([a/\hat{a}]\hat{a}) \cdot ([a/\hat{a}]s') : ([\sigma/\hat{\sigma}]\hat{\sigma}) \cdot ([\sigma/\hat{\sigma}]S'_i)$, and $[a/\hat{a}](\hat{a} \cdot s') : [\sigma/\hat{\sigma}](\hat{\sigma} \cdot S'_i)$ follows. Thus, the first clause of the conclusion is satisfied.

StackCorresponds$(\sigma \cdot [\sigma/\hat{\sigma}]S'_i, a \cdot [a/\hat{a}]s', \hat{a}, \hat{\sigma})$ follows by (*sc 2*) and (6), plus the fact that $\sigma \neq \hat{\sigma}$. Using Eqs. (7) and (8) above, this correspondence can be rewritten as *StackCorresponds*$([\sigma/\hat{\sigma}](\hat{\sigma} \cdot S'_i), [a/\hat{a}](\hat{a} \cdot s'), \hat{a}, \hat{\sigma})$ using the distributive nature of substitution over sequences.

Case 3: (*sc 2*). In this case, $s = v \cdot s'$ and $S_i = \tau \cdot S'_i$ for some s' and S'_i where $\tau \neq \hat{\sigma}$. Since $s : S_i$, we know that $v : \tau$. We proceed as in the previous case to conclude that $[a/\hat{a}]s' : [\sigma/\hat{\sigma}]S'_i$ and *StackCorresponds*$([\sigma/\hat{\sigma}]S'_i, [a/\hat{a}]s', \hat{a}, \hat{\sigma})$ by the inductive hypothesis. There are two cases for v:

(1) $v \neq \hat{a}$: Since we also know that $\tau \neq \hat{\sigma}$, we may conclude that $[a/\hat{a}]v : [\sigma/\hat{\sigma}]\tau$ and $([a/\hat{a}]v) \cdot ([a/\hat{a}]s') : ([\sigma/\hat{\sigma}]\tau) \cdot ([\sigma/\hat{\sigma}]S'_i)$ are true. The previous type assignment may be rewritten as $[a/\hat{a}](v \cdot s') : [\sigma/\hat{\sigma}](\hat{\sigma} \cdot S_i)$. In addition, *StackCorresponds*$(([\sigma/\hat{\sigma}]\tau) \cdot ([\sigma/\hat{\sigma}]S'_i), ([a/\hat{a}]v) \cdot ([a/\hat{a}]s'), \hat{a}, \hat{\sigma})$ follows from rule (*sc 2*), since $[\sigma/\hat{\sigma}]\tau \neq \hat{\sigma}$. This may be rewritten as *StackCorresponds*$([\sigma/\hat{\sigma}](\tau \cdot S'_i), [a/\hat{a}](v \cdot s'), \hat{a}, \hat{\sigma})$, and the second clause of the conclusion follows.

(2) $v = \hat{a}$: In this case, τ must be TOP, since the only valid types for $\hat{a}$ are TOP and $\hat{\sigma}$. The latter is ruled out because we used (*sc 2*) as the final step in the proof of *StackCorresponds*$(\tau \cdot S'_i,\ v \cdot s',\ \hat{a},\ \hat{\sigma})$. Since any value has type TOP and $[\sigma/\hat{\sigma}]\mathrm{TOP} = \mathrm{TOP}$, we may conclude that $[a/\hat{a}]v : [\sigma/\hat{\sigma}]\tau$ is true. The assertion $[a/\hat{a}]s : [\sigma/\hat{\sigma}]S_i$ follows directly from this, as above. *StackCorresponds*$(([\sigma/\hat{\sigma}]\tau) \cdot ([\sigma/\hat{\sigma}]S'_i),\ ([a/\hat{a}]v) \cdot ([a/\hat{a}]s'),\ \hat{a},\ \hat{\sigma})$ follows from rule (*sc 2*), since $[\sigma/\hat{\sigma}]\tau \neq \hat{\sigma}$, and *StackCorresponds*$([\sigma/\hat{\sigma}](\tau \cdot S'_i),\ [a/\hat{a}](v \cdot s'),\ \hat{a},\ \hat{\sigma})$ is true. □

The next lemma is analogous to the previous for a specific local variable y. In other words, replacing an uninitialized object value and the corresponding types with an initialized object value and initialized object type does not affect the correspondence or whether the local variable y has the correct type.

LEMMA 12.

$$\begin{array}{l} \forall F_i,\ a:\sigma,\ \hat{a}:\hat{\sigma},\ y. \\ \qquad y \in Dom(F_i) \\ \qquad \wedge\ f[y]:F_i[y] \\ \qquad \wedge\ F_i[y] = \hat{\sigma} \Rightarrow f[y] = \hat{a} \\ \qquad \wedge\ \sigma \neq \hat{\sigma} \\ \quad \Rightarrow\ ([a/\hat{a}]f)[y] : ([\sigma/\hat{\sigma}]F_i)[y] \\ \qquad \wedge\ ([\sigma/\hat{\sigma}]F_i)[y] = \hat{\sigma} \Rightarrow ([a/\hat{a}]f)[y] = \hat{a} \end{array}$$

PROOF. Assume that the hypotheses are satisfied. There are two cases for $F_i[y]$:

Case 1: $F_i[y] = \hat{\sigma}$. Thus, $f[y] = \hat{a}$. Also, we know that $[a/\hat{a}](f[y]) = a$ and $[\sigma/\hat{\sigma}](F_i[y]) = \sigma$. Since $a:\sigma$ by our assumptions, the first clause of the conclusion is true. Since $\sigma \neq \hat{\sigma}$, the second clause is also true.

Case 2: $F_i[y] \neq \hat{\sigma}$. First, we know that $([\sigma/\hat{\sigma}]F_i)[y] = F_i[y]$. In addition, one of the assumptions of the implication is that $f[y]:F_i[y]$. In this case, there are two possibilities for $f[y]$:

(1) $f[y] \neq \hat{a}$: In this case, $[a/\hat{a}](f[y]) = f[y]$, and since $F_i[y] \neq \hat{\sigma}$, the conclusions are satisfied.
(2) $f[y] = \hat{a}$: In this case, $F_i[y] = \mathrm{TOP}$, since the only valid types for $\hat{a}$ are $\hat{\sigma}$ and TOP. Since $a:\mathrm{TOP}$ is also true, $[a/\hat{a}](f[y]):\mathrm{TOP}$, making the first clause of the conclusion true. Since $\hat{\sigma} \neq \mathrm{TOP}$, the second clause of the conclusion also follows. □

The next lemma shows that initializing an object of one uninitialized object type does not affect the correspondence between other uninitialized object types and values.

LEMMA 13.

$$\begin{aligned}
\forall S_i,\ & s,\ a:\sigma,\ \hat{a}:\hat{\sigma},\ \hat{b}:\hat{\tau}. \\
& s:S_i \\
& \wedge\ \mathit{StackCorresponds}(S_i,\ s,\ \hat{b},\ \hat{\tau}) \\
& \wedge\ \hat{\tau} \neq \sigma \\
& \wedge\ \hat{\tau} \neq \hat{\sigma} \\
\Rightarrow\ & \mathit{StackCorresponds}([\sigma/\hat{\sigma}]S_i,\ [a/\hat{a}]s,\ \hat{b},\ \hat{\tau})
\end{aligned}$$

PROOF. We show that $\mathit{StackCorresponds}([\sigma/\hat{\sigma}]S_i,\ [a/\hat{a}]s,\ \hat{b},\ \hat{\tau})$ is true by induction on the proof of $\mathit{StackCorresponds}(S_i,\ s,\ \hat{b},\ \hat{\tau})$. There is one base case and two inductive cases to consider, depending on which judgment is used in the final step of the proof:

Case 1: (*sc 0*). If this is the case, $s = \epsilon$ and $S_i = \epsilon$, and the conclusion follows easily.

Case 2: (*sc 1*). Assume $S_i = \hat{\tau} \cdot S'_i$ and $s = \hat{b} \cdot s'$ where $\mathit{StackCorresponds}([\sigma/\hat{\sigma}]S'_i,\ [a/\hat{a}]s',\ \hat{b},\ \hat{\tau})$ is true by the inductive hypothesis. $\mathit{StackCorresponds}(\hat{\tau} \cdot [\sigma/\hat{\sigma}]S'_i,\ \hat{b} \cdot [a/\hat{a}]s',\ \hat{b},\ \hat{\tau})$ follows by rule (*sc 1*), and since $\hat{\tau} \neq \hat{\sigma}$ and $\hat{b} \neq \hat{a}$, we may rewrite this as $\mathit{StackCorresponds}([\sigma/\hat{\sigma}]S_i,\ [a/\hat{a}]s,\ \hat{b},\ \hat{\tau})$.

Case 3: (*sc 2*). Assume $S_i = \tau \cdot S'_i$ and $s = v \cdot s'$ where $v:\tau$ and $\hat{\tau} \neq \tau$. By the inductive hypothesis, $\mathit{StackCorresponds}([\sigma/\hat{\sigma}]S'_i,\ [a/\hat{a}]s',\ \hat{b},\ \hat{\tau})$ is true. Also, $[\sigma/\hat{\sigma}]\tau \neq \hat{\tau}$, since $\sigma \neq \hat{\tau}$ and $\hat{\tau} \neq \tau$. Therefore, by rule (*sc 2*), $\mathit{StackCorresponds}(([\sigma/\hat{\sigma}]\tau) \cdot ([\sigma/\hat{\sigma}]S'_i),\ ([a/\hat{a}]v) \cdot ([a/\hat{a}]s'),\ \hat{b},\ \hat{\tau})$ is true. This can be rewritten as $\mathit{StackCorresponds}([\sigma/\hat{\sigma}]S_i,\ [a/\hat{a}]s,\ \hat{b},\ \hat{\tau})$. □

A.2 One-Step Soundness

In order to prove Theorem 1, we prove that each of the four invariants is preserved by a program step. For each invariant, we first state a general property of instruction behavior, based on the operational and static semantics, that guarantees the invariant will not be violated by any instruction exhibiting the property. These properties will allow us to reason about which instructions preserve the global invariants in the common case easily. For example, the following property describes behavior easily proved to guarantee that the stack is well typed after the instruction is executed.

Definition 1. For some instruction I, I *preserves StackType* if

$$\begin{array}{l}\forall P, F, S, pc, f, s, pc', f', s'.\\ \quad F, S \vdash P\\ \quad \wedge\ P \vdash \langle pc, f, s\rangle \rightarrow \langle pc', f', s'\rangle\\ \quad \wedge\ s : S_{pc}\\ \quad \wedge\ P[pc] = \mathrm{I}\\ \Rightarrow \exists s_1 : \alpha_1,\ s_2 : \alpha_2,\ s_3 : \alpha_3.\\ \quad s = s_1 \bullet s_2\\ \quad \wedge\ s' = s_3 \bullet s_2\\ \quad \wedge\ S_{pc} = \alpha_1 \bullet \alpha_2\\ \quad \wedge\ S_{pc'} = \alpha_3 \bullet \alpha_2\end{array}$$

Intuitively, this property splits the stack into a segment popped by the instruction, a segment pushed by the instruction, and a part that stays the same. Each of these three parts are well typed by the corresponding pieces of type information from S. The conclusion of the implication may be simplified to $s : S_{pc} \wedge s' : S_{pc'}$ by taking s_2 and α_2 to be ϵ. However, the way we have presented it enables us to separate the elements pushed and popped from the stack from the part of the stack untouched by the common stack operations.

LEMMA 14. *The following instructions have this property:* `inc`, `pop`, `push 0`, `store` x, `load` x, `new` σ, `use` σ, *and* `if` L.

PROOF. Two representative cases are shown. In each case, assume we have P, F, S, pc, f, s, pc', f', and s' which satisfy all the hypotheses:

Case 1: `inc`. By the operational semantics, $pc' = pc + 1$, $s = n \cdot s''$, and $s' = (n + 1) \cdot s''$ for some s''. By (*inc*), $S_{pc} = S_{pc'} = \mathrm{INT} \cdot \alpha$ for some α. Choose $s_1 = n$, $s_2 = s''$, $s_3 = n + 1$, and $\alpha_1 = \mathrm{INT}$, $\alpha_2 = \alpha$, $\alpha_3 = \mathrm{INT}$. Clearly, $s_1 : \alpha_1$ and $s_3 : \alpha_3$, since n and $n + 1$ are integers. By the assumption that $s : S_{pc}$, we may conclude that $n \cdot s'' : \mathrm{INT} \cdot \alpha$ and $s'' : \alpha$, meaning $s_2 : \alpha_2$.

Case 2: `if` L. By the operational semantics, $pc' \in \{pc + 1, L\}$. Also, $s = n \cdot s''$ for some integer n and stack s''. In addition, by (*if*), $S_{pc} = \mathrm{INT} \cdot S_{pc'}$. Choose $s_1 = n$, $s_2 = s''$, $s_3 = \epsilon$, $\alpha_1 = \mathrm{INT}$, $\alpha_2 = S_{pc'}$, and $\alpha_3 = \epsilon$. We know that $s_1 : \alpha_1$, since $n : \mathrm{INT}$. By the assumption that $s : S_{pc}$, we conclude that $n \cdot s'' : \mathrm{INT} \cdot S_{pc'}$ and $s'' : S_{pc'}$, meaning that $s_2 : \alpha_2$. The type judgment $\epsilon : \epsilon$ implies that $s_3 : \alpha_3$. □

With this lemma, we may now prove part of Theorem 1, grouping all instructions shown to exhibit the property presented in Definition 1 into a single case.

LEMMA 15. *Given P, F, and S such that $F, S \vdash P$,*

$$\begin{aligned}
&\forall pc, f, s, pc', f', s'. \\
&\qquad P \vdash \langle pc, f, s\rangle \rightarrow \langle pc', f', s'\rangle \\
&\qquad \wedge\ s : S_{pc} \\
&\qquad \wedge\ \forall y \in \text{VAR}.\ f[y] : F_{pc}[y] \\
&\qquad \wedge\ ConsistentInit(F_{pc}, S_{pc}, f, s) \\
&\quad \Rightarrow\ s' : S_{pc'}
\end{aligned}$$

PROOF. Assume that the hypotheses are satisfied. We proceed by examining the possible instructions at $P[pc]$, noting that $P[pc] \neq$ `halt`, since a transition is made from the current state:

Case 1: $P[pc]$ *Preserves StackType.* By Lemma 14, we may choose s_2, s_3, α_2, and α_3 such that $s' = s_3 \bullet s_2$, $S_{pc'} = \alpha_3 \bullet \alpha_2$, $s_3 : \alpha_3$, and $s_2 : \alpha_2$. Thus, $s' : S_{pc'}$.

Case 2: $P[pc] =$ `load` x. By the operational semantics, we know that $s' = f[x] \cdot s$. By rule (*load*) and the fact that $pc' = pc + 1$, we also know that $S_{pc'} = F_{pc}[x] \cdot S_{pc}$ must be true. Given the assumption that $\forall y \in$ VAR. $f[y] : F_{pc}[y]$, we know that $f[x] : F_{pc}[x]$. In addition, since $s : S_{pc}$, it follows that $f[x] \cdot s : F_{pc}[x] \cdot S_{pc}$, allowing us to conclude that $s' : S_{pc'}$.

Case 3: $P[pc] =$ `init` σ. To prove this case, we first show that *StackCorresponds*$(\alpha, s'', \hat{a}, \hat{\sigma}_j)$. We then dispatch the hypotheses of Lemma 11 to conclude that $s' : S_{pc'}$. By the operational semantics, $pc' = pc + 1$, $s = \hat{a} \cdot s''$, and $s' = [a/\hat{a}]s''$ for some s'', $\hat{a}$, and $a \in A^{\sigma}$. By rule (*init*), $S_{pc} = \hat{\sigma}_j \cdot \alpha$ and $S_{pc'} = [\sigma/\hat{\sigma}_j]\alpha$ are true. To prove $s' : S_{pc'}$, we first note that $s : S_{pc}$ and *ConsistentInit*(F_{pc}, S_{pc}, f, s) imply that *Corresponds*$(F_{pc}, S_{pc}, f, s, \hat{a}, \hat{\sigma}_j)$ by Lemma 1. This means that *StackCorresponds*$(S_{pc}, s, \hat{a}, \hat{\sigma}_j)$, and in order to have proved this,

$$StackCorresponds(\alpha, s'', \hat{a}, \hat{\sigma}_j) \tag{9}$$

must be true. We now turn our attention to the hypotheses of Lemma 11. The types σ and $\hat{\sigma}_j$ must be different because the first is an initialized object type, and the second is an uninitialized object type. Also, $s'' : \alpha$, $a : \sigma$, and $\hat{a} : \hat{\sigma}_j$. These conditions, in addition to (9), are sufficient to conclude $s' : S_{pc'}$ using Lemma 11. □

Definition 2 captures a behavior of all instructions known not to alter the local variables.

Definition 2. For some instruction I, I *preserves VariableType* if

$$\begin{array}{l} \forall P, F, S, pc, f, s, pc', f', s'. \\ \quad F, S \vdash P \\ \quad \wedge\ P \vdash \langle pc, f, s\rangle \rightarrow \langle pc', f', s'\rangle \\ \quad \wedge\ P[pc] = \mathrm{I} \\ \Rightarrow f' = f \\ \quad \wedge\ F_{pc'} = F_{pc} \end{array}$$

LEMMA 16. *The following instructions have this property:* `inc`, `pop`, `push 0`, `load` x, `new` σ, `use` σ, *and* `if` L.

PROOF. Two representative cases are shown. In each case, assume that we have P, F, S, pc, f, s, pc', f', and s' which satisfy all the hypotheses:

Case 1: `inc`. By the operational semantics, $pc' = pc + 1$ and $f' = f$. By (*inc*), $F_{pc} = F_{pc+1}$. From these pieces of information, it is clear that $F_{pc} = F_{pc'}$ is also true.

Case 2: `if` L. By the operational semantics, $pc' \in \{pc + 1, L\}$. Also, $f' = f$. In addition, by (*if*) it follows that $F_{pc} = F_{pc+1} = F_L$. Given the possible values for pc', we can conclude that $F_{pc} = F_{pc'}$. □

LEMMA 17. *Given* P, F, *and* S *such that* $F, S \vdash P$,

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

PROOF. Assume that the hypotheses are satisfied. We proceed by examining the possible instructions at $P[pc]$. Note, that, as before, $P[pc] \neq$ `halt`:

Case 1: $P[pc]$ *Preserves VariableType*. By Lemma 16, $f' = f$ and $F_{pc'} = F_{pc}$, and we assumed $\forall y \in \mathrm{VAR}.\ f[y] : F_{pc}[y]$. By substitution using the two equalities, $\forall y \in \mathrm{VAR}.\ f'[y] : F_{pc'}[y]$.

Case 2: $P[pc] =$ `store` x. From the operational and static semantics, we know that $pc' = pc + 1$, $f' = f[x \mapsto v]$, and $F_{pc'} = F_{pc}[x \mapsto \tau]$ where $s = v \cdot s'$ and $S_{pc} = \tau \cdot S_{pc'}$. There are two cases to consider to prove that $f'[y] : F_{pc'}[y]$ for all $y \in \mathrm{VAR}$:

(1) $y \neq x$: In this case, $f'[y] = f[y]$ and $F_{pc'}[y] = F_{pc}[y]$. From the hypotheses of the implication, $f'[y] : F_{pc'}[y]$ is true.
(2) $y = x$: $f'[x] = v$ and $F_{pc'}[x] = \tau$. Since $s : S_{pc}$, we know that $v : \tau$.

Therefore, $f'[x]:F_{pc'}[x]$.

Thus, $\forall y \in \text{VAR}.\ f'[y]:F_{pc'}[y]$.

Case 3: $P[pc] =$ `init` σ. In this case, we know that $pc' = pc + 1$, and the static and operational semantics imply that $f' = [a/\hat{a}]f$ and $F_{pc'} = [\sigma/\hat{\sigma}_j]F_{pc}$ where $s = \hat{a} \cdot s''$ and $S_{pc} = \hat{\sigma}_j \cdot \alpha$. Also, $\hat{\sigma}_j \in \hat{T}$ and $\sigma \in T$. In addition, $Corresponds(F_{pc}, S_{pc}, f, s, \hat{a}, \hat{\sigma}_j)$ follows from Lemma 1, meaning that $F_{pc}[y] = \hat{\sigma}_j$ implies $f[y] = \hat{a}$ for all $y \in \text{VAR}$. Since we are not considering subroutines yet, $y \in \text{VAR}$ implies that $y \in Dom(F_i)$. Using these facts, Lemma 12 may be applied to conclude $([a/\hat{a}]f)[y]:([\sigma/\hat{\sigma}_j]F[pc])[y]$ for all $y \in \text{VAR}$, and the conclusion is satisfied. We discuss how this lemma is affected by subroutines and variable polymorphism in Appendix B.2. □

Definition 3 is more complex than the previous properties. The intuition captured is that an instruction will not touch the local variables, but it may pop off any number of values from the stack and push any number of new values, as long as the new ones are not uninitialized objects. While there are many parts to the conclusion in the implication of this property, the truth of each one of these may be obtained by a simple examination of the JVML$_i$ semantics.

Definition 3. For some instruction I, I *preserves ConsistentInit if*

$$
\begin{array}{ll}
\forall P, F, S, pc, f, s, pc', f', s'. & \\
\quad F, S \vdash P & \\
\quad \wedge\ P \vdash \langle pc, f, s\rangle \rightarrow \langle pc', f', s'\rangle & \\
\quad \wedge\ s:S_{pc} & \\
\quad \wedge\ \forall y \in \text{VAR}.\ f[y]:F_{pc}[y] & \\
\quad \wedge\ ConsistentInit(F_{pc}, S_{pc}, f, s) & \\
\quad \wedge\ P[pc] = \text{I} & \\
\Rightarrow\ \exists s_1:\alpha_1,\ s_2:\alpha_2,\ s_3:\alpha_3. & \\
\quad s = s_1 \bullet s_2 & \\
\quad \wedge\ s' = s_3 \bullet s_2 & \\
\quad \wedge\ S_{pc} = \alpha_1 \bullet \alpha_2 & \\
\quad \wedge\ S_{pc'} = \alpha_3 \bullet \alpha_2 & \\
\quad \wedge\ \forall y \in Dom(\alpha_3) \cdot \alpha_3[y]\ \notin\ \hat{T} & \\
\quad \wedge\ f' = f & \\
\quad \wedge\ F_{pc'} = F_{pc} &
\end{array}
$$

To simplify the presentation of this definition, we allow elements of a sequence to be accessed as array elements. Therefore, consider $\alpha_3[y]$ to refer to the element yth from the left in sequence α_3.

LEMMA 18. *The following instructions have this property:* `inc`, `pop`, `push 0`, `use` σ, `if` L.

PROOF. Two representative cases are shown. In each case, assume that we have P, F, S, pc, f, s, pc', f', and s' which satisfy all the hypotheses:

Case 1: `inc`. By the operational semantics, we know that $pc' = pc + 1$, $f = f'$, $s = n \cdot s''$, and $s' = (n + 1) \cdot s''$ for some s''. By (*inc*), $S_{pc} = S_{pc'} = \mathrm{INT} \cdot \alpha$ for some α, and $F_{pc} = F_{pc'}$. Choose $s_1 = n$, $s_2 = s''$, $s_3 = n + 1$, $\alpha_1 = \mathrm{INT}$, $\alpha_2 = \alpha$, and $\alpha_3 = \mathrm{INT}$. Clearly, $s_1 : \alpha_1$ and $s_3 : \alpha_3$, since n and $n + 1$ are integers. By the assumption that $s : S_{pc}$, we may conclude that $n \cdot s'' : \mathrm{INT} \cdot \alpha$ and $s'' : \alpha$, meaning $s_2 : \alpha_2$. Finally, $\mathrm{INT} \notin \hat{T}$, implying that $\forall y \in Dom(\alpha_3) \cdot \alpha_3[y] \notin \hat{T}$.

Case 2: `if` L. By the operational semantics, $pc' \in \{pc + 1, L\}$ and $f = f'$. Also, $s = n \cdot s''$ for some integer n and stack s''. In addition, (*if*) implies that $S_{pc} = \mathrm{INT} \cdot S_{pc'}$ and $F_{pc} = F_{pc'}$. Choose $s_1 = n$, $s_2 = s''$, $s_3 = \epsilon$, $\alpha_1 = \mathrm{INT}$, $\alpha_2 = S_{pc'}$, and $\alpha_3 = \epsilon$. We know that $s_1 : \alpha_1$, since $n : \mathrm{INT}$. By the assumption that $s : S_{pc}$, we conclude that $n \cdot s'' : \mathrm{INT} \cdot S_{pc'}$ and $s'' : S_{pc'}$ meaning that $s_2 : \alpha_2$. The type judgment $\epsilon : \epsilon$ implies that $s_3 : \alpha_3$. Finally, $\alpha_3 = \epsilon$, so there is no uninitialized object type in α_3. □

Not all instructions exhibit this property, because some introduce new uninitialized objects or operate on elements not on the top of the stack, such as the `new` and `init` instructions, respectively.

We now show that *ConsistentInit* is preserved by all instructions.

LEMMA 19. *Given P, F, and S such that $F, S \vdash P$,*

$$\begin{aligned}
&\forall pc, f, s, pc', f', s'.\\
&\qquad P \vdash \langle pc, f, s\rangle \rightarrow \langle pc', f', s'\rangle\\
&\qquad \wedge\ s : S_{pc}\\
&\qquad \wedge\ \forall y \in \mathrm{VAR}.\ f[y] : F_{pc}[y]\\
&\qquad \wedge\ ConsistentInit(F_{pc}, S_{pc}, f, s)\\
&\quad \Rightarrow\ ConsistentInit(F_{pc'}, S_{pc'}, f', s')
\end{aligned}$$

PROOF. Assume that we have P, F, S, pc, f, s, pc', f', and s' which satisfy all the hypotheses. We proceed by case analysis on $P[pc]$, where the first case will contain all instructions that satisfy the property in Definition 3. We know that $P[pc]$ is not a `halt` instruction.

Case 1: *$P[pc]$ Preserves ConsistentInit.* By Lemma 18, we may choose $s_1 : \alpha_1$, $s_2 : \alpha_2$, and $s_3 : \alpha_3$ such that all the conditions listed in Definition 3 are satisfied. Note that this ensures $s = s_1 \bullet s_2$ and $S_{pc} = \alpha_1 \bullet \alpha_2$. Since *ConsistentInit*$(F_{pc}, S_{pc}, f, s)$, Lemma 5 proves that *ConsistentInit*$(F_{pc}, \alpha_2, f, s_2)$. Since we also know $s_3 : \alpha_3$ and no uninitialized types appear in α_3, Lemma 8 may be applied to prove that *ConsistentInit*$(F_{pc}, \alpha_3 \bullet \alpha_2, f, s_3 \bullet s_2)$. Also, since $F_{pc'} = F_{pc}$ and $f' = f$, *ConsistentInit*$(F_{pc'}, S_{pc'}, f', s')$ is true.

Case 2: $P[pc] =$ `new` σ. By the operational semantics, $s' = \hat{a} \cdot s$, and we know that $S_{pc'} = \hat{\sigma}_{pc} \cdot S_{pc}$ by rule (*init*) and the fact that $pc' = pc + 1$.

This means that $\hat{a} : \hat{\sigma}_{pc}$. For each $\hat{\tau} \in \hat{T}$, we must choose a $\hat{b} : \hat{\tau}$ such that *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$ to conclude that the lemma is true. There are two cases to consider for each $\hat{\tau}$:

(1) $\hat{\tau} = \hat{\sigma}_{pc}$: Choose $\hat{b} = \hat{a}$. Given that $\hat{\sigma}_{pc} \notin S_{pc}$ is true by the static semantics, it is obvious that *StackCorresponds*$(S_{pc}, s, \hat{a}, \hat{\sigma}_{pc})$ is true, since it may be proved using only rules (*sc 0*) and (*sc 2*). Applying rule (*sc 1*), we also know that

$$StackCorresponds(S_{pc'}, s', \hat{a}, \hat{\sigma}_{pc}) \tag{10}$$

will be true. We also know from the static semantics that $F_{pc'} = F_{pc}$ is true. In addition, we know that

$$\forall y \in Dom(F_{pc}) \cdot F_{pc}[y] \neq \hat{\sigma}_{pc}.$$

Thus, we know the following is true:

$$\forall y \in Dom(F_{pc'}) \cdot F_{pc'}[y] = \hat{\sigma}_{pc} \Rightarrow f'[y] = \hat{a} \tag{11}$$

Lines (10) and (11) imply that *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$ is true using rule (*corr*).

(2) $\hat{\tau} \neq \hat{\sigma}_{pc}$: Choose $\hat{b}$ such that *Corresponds*$(F_{pc}, S_{pc}, f, s, \hat{b}, \hat{\tau})$. By our assumption that *ConsistentInit*(F_{pc}, S_{pc}, f, s), there is such a b, and *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$ follows by Lemma 6.

Case 3: $P[pc] =$ `init` σ. By the operational semantics, $pc' = pc + 1$, $s = \hat{a} \cdot s''$, $f' = [a/\hat{a}]f$, and $s' = [a/\hat{a}]s''$ for some s'', $\hat{a}$, and $a : \sigma$. By rule (*init*), $S_{pc} = \hat{\sigma}_j \cdot \alpha$ for some α, meaning that $s'' : \alpha$ and $\hat{a} : \hat{\sigma}_j$ follow from the assumption $s : S_{pc}$. Also, we know that $S_{pc'} = [\sigma/\hat{\sigma}_j]\alpha$ and $F_{pc'} = [\sigma/\hat{\sigma}_j]F_{pc}$ from this rule. For each $\hat{\tau} \in \hat{T}$, we must find $\hat{b}$ such that *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$. If we can, then *ConsistentInit*$(F_{pc'}, S_{pc'}, f', s')$ follows by rule (*cons init*). There are two cases for each $\hat{\tau}$:

(1) $\hat{\tau} = \hat{\sigma}_j$: Choose $\hat{b} = \hat{a}$. First, note that $\sigma \neq \hat{\sigma}_j$, since σ is an initialized object type and $\hat{\sigma}_j$ is an uninitialized object type. Lemma 11 and Lemma 12 can be used to show that

$$\forall y \in Dom([\sigma/\hat{\sigma}_j]F_{pc}).\ ([\sigma/\hat{\sigma}_j]F_{pc})[y] = \hat{\sigma}_j \Rightarrow ([a/\hat{a}]f)[y] = \hat{a}$$
$$StackCorresponds([\sigma/\hat{\sigma}_j]S_{pc}, [a/\hat{a}]s'', \hat{a}, \hat{\sigma}_j).$$

These two lemmas may be applied, since the hypotheses of the implication and the facts about a, $\hat{a}$, σ, and $\hat{\sigma}_j$ satisfy the conditions for those lemmas. *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{a}, \hat{\sigma}_j)$ follows directly from these two equations using rule (*corr*).

(2) $\hat{\tau} \neq \hat{\sigma}_j$: Choose $\hat{b}$ such that *Corresponds*$(F_{pc}, S_{pc}, f, s, \hat{b}, \hat{\tau})$. By our assumption that *ConsistentInit*(F_{pc}, S_{pc}, f, s), there is such a $\hat{b}$. We

now show that

$$Corresponds(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$$

is true. First,

$$\forall y \in Dom([\sigma/\hat{\sigma}_j]F_{pc}) \cdot ([\sigma/\hat{\sigma}_j]F_{pc})[y] = \hat{\tau} \Rightarrow ([a/\hat{a}]f)[y] = \hat{b} \qquad (12)$$

must be true. Consider any y such that $([\sigma/\hat{\sigma}_j]F[pc])[y] = \hat{\tau}$. In this case, $F_{pc}[y] = \hat{\tau}$ since $\hat{\tau} \neq \hat{\sigma}_j$. In addition, the correspondence between $\hat{b}$ and $\hat{\tau}$ implies that $f[y] = \hat{b}$. Therefore, $([a/\hat{a}]f[y] \neq \hat{b}$ because $\hat{a} \neq \hat{b}$ and a is unused in f. We also know, that, given *StackCorresponds*$(\alpha, s'', \hat{b}, \hat{\tau})$ is a necessary condition for *Corresponds*$(F_{pc}, S_{pc}, f, s, \hat{b}, \hat{\tau})$,

$$StackCorresponds([\sigma/\hat{\sigma}_j]\alpha, [a/\hat{a}]s'', \hat{b}, \hat{\tau}) \qquad (13)$$

is true by Lemma 13. Therefore, *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{a}, \hat{\sigma}_j)$ is true by rule (*corr*) applied to (12) and (13).

Case 4: $P[pc] = \texttt{store}\ x$. By the operational semantics, $pc' = pc + 1$, $s = v \cdot s'$, and $f' = f[x \mapsto v]$ for some v. By rule (*store*), $S_{pc} = \tau \cdot S_{pc'}$ and $F_{pc'} = F_{pc}[x \mapsto \tau]$ for some τ. Since we assumed $s : S_{pc}$, we know that $v : \tau$. For each $\hat{\tau} \in \hat{T}$, we must find $\hat{b}$ such that *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$. If we can, then *ConsistentInit*$(F_{pc'}, S_{pc'}, f', s')$ follows by rule (*cons init*). There are two cases for each $\hat{\tau}$:

(1) $\tau \neq \hat{\tau}$: Choose $\hat{b}$ such that *Corresponds*$(F_{pc}, S_{pc}, f, s, \hat{b}, \hat{\tau})$. We are guaranteed that this will exist by the assumption that *ConsistentInit* holds. Since (*store*) ensures $x \in Dom(F_{pc})$, Lemma 10 may be applied to conclude *Corresponds*$(F_{pc}[x \mapsto \tau], S_{pc}, f[x \mapsto v], s, \hat{b}, \hat{\tau})$. Simplifying this and appealing to Lemma 3, we conclude that

$$Corresponds(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau}).$$

(2) $\tau = \hat{\tau}$: Choose $\hat{b} = v$. In this case, *Corresponds*$(F_{pc}, S_{pc}, f, s, v, \hat{\tau})$ must be true by Lemma 1. We may apply Lemma 9 to this to conclude that *Corresponds*$(F_{pc}[x \mapsto \hat{\tau}], S_{pc}, f[x \mapsto v], s, v, \hat{\tau})$. Finally, we know that *Corresponds*$(F_{pc'}, S_{pc'}, f', s', v, \hat{\tau})$ is true using Lemma 3.

Case 5: $P[pc] = \texttt{load}\ x$. By the operational semantics, $pc' = pc + 1$, $s' = f[x] \cdot s$, and $f' = f$. By rule (*load*), $S_{pc'} = F_{pc}[x] \cdot S_{pc}$ and $F_{pc'} = F_{pc}$. We know that $f[x] : F_{pc}[x]$. For each $\hat{\tau} \in \hat{T}$, we must find $\hat{b}$ such that *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$. If we can, then *ConsistentInit*$(F_{pc'}, S_{pc'}, f', s')$ follows by rule (*cons init*). There are two cases for each $\hat{\tau}$:

(1) $F_{pc}[x] \neq \hat{\tau}$: Choose $\hat{b}$ such that *Corresponds*$(F_{pc}, S_{pc}, f, s, \hat{b}, \hat{\tau})$. From this, we know that *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$ by Lemma 6.

(2) $F_{pc}[x] = \hat{\tau}$: Choose $\hat{b} = f[x]$. Since *ConsistentInit*(F_{pc}, S_{pc}, f, s), we conclude *Corresponds*$(F_{pc}, S_{pc}, f, s, f[x], F_{pc}[x])$ and *StackCorre-*

sponds$(S_{pc}, s, f[x], F_{pc}[x])$ by Lemma 2. Applying (*sc 1*) to this, we know that *StackCorresponds*$(F_{pc}[x] \cdot S_{pc}, f[x] \cdot s, f[x], F_{pc}[x])$ is true, and *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$ is true. □

Although the following definition is trivial for JVML$_i$, we include it both for symmetry with the previous three invariants and because it will be useful in the proofs of extensions of JVML$_i$.

Definition 4. For some instruction I, I *preserves ProgramDomain* if

$$\begin{array}{l} \forall P, F, S, pc, f, s, pc', f', s'. \\ \quad F, S \vdash P \\ \quad \wedge\ P \vdash \langle pc, f, s\rangle \rightarrow \langle pc', f', s'\rangle \\ \quad \wedge\ P[pc] = \mathrm{I} \\ \Rightarrow pc' \in Dom(P) \end{array}$$

LEMMA 20. *The following instructions have this property:* `inc`, `pop`, `push 0`, `load` x, `store` x, `new` σ, `use` σ, `init` σ, *and* `if` L.

PROOF. Two representative cases are shown. In each case, assume that we have P, F, S, pc, f, s, pc', f', and s' which satisfy all the hypotheses:

Case 1: `inc`. By the operational semantics, $pc' = pc + 1$. By (*inc*), $pc + 1 \in Dom(P)$.

Case 2: `if` L. By the operational semantics, $pc' \in \{pc + 1, L\}$. By (*if*), both of these possible values for pc' are in the domain of P. □

LEMMA 21. *Given* P, F, *and* S *such that* $F, S \vdash P$,

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

PROOF. We are guaranteed that $P[pc]$ *preserves ProgramDomain*, since all instructions in JVML$_i$ exhibit is property. Thus, this lemma follows directly from Lemma 20. □

We may now prove Theorem 1:

RESTATEMENT OF THEOREM 1. *Given P, F, and S such that $F, S \vdash P$,*

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

PROOF. This theorem follows directly from Lemmas 15, 17, 19, and 21. □

A.3 Progress

The proofs in this subsection and the next are based on the corresponding proofs of Stata and Abadi. The progress theorem is easily proved by showing that any instruction, except `halt`, will allow a program to take a step from a well-formed state. For each instruction, we must simply show how to construct the state to which the program can step.

RESTATEMENT OF THEOREM 2. *Given P, F, and S such that $F, S \vdash P$,*

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

PROOF. Assume that all the hypotheses are satisfied for some pc, f, and s. We proceed by case analysis on possible instructions $P[pc]$. The proof of each case will simply choose values of pc', f', and s' such that a step may be taken by the program according to the operational semantics.

Case 1: $P[pc]$ = `push 0`. Choose $s' = 0 \cdot s$, $f' = f$, and $pc' = pc + 1$.

Case 2: $P[pc]$ = `inc`. Since we assumed $s : S_{pc}$, and $S_{pc} = \text{INT} \cdot S_{pc+1}$ follows from (*inc*), we know that $s = n \cdot s''$ for some n and s''. Therefore, choosing $s' = (n + 1) \cdot s''$, $f' = f$, and $pc' = pc + 1$ will allow progress to be made.

Case 3: $P[pc]$ = `pop`. Since $s : S_{pc}$, and $S_{pc} = \tau \cdot \alpha$ follows from (*pop*), we know that $s = v \cdot s''$ for some v and s''. Therefore, choose $s' = s''$, $f' =$

f, and $pc' = pc + 1$. Also, the static semantics guarantees that $x \in Dom(F_{pc})$.

Case 4: $P[pc] = \texttt{if}\ L$. Since we assumed $s : S_{pc}$ and $S_{pc} = \text{INT} \cdot S_{pc+1}$ follows from (*if*), we know that $s = n \cdot s''$ for some n and s''. Therefore, choosing $s' = s''$, $f' = f$, and $pc' = pc + 1$ if $n = 0$, or $pc' = L$ otherwise, will allow progress to be made.

Case 5: $P[pc] = \texttt{store}\ x$. Since $s : S_{pc}$, and $S_{pc} = \tau \cdot S_{pc+1}$ follows from (*pop*), $s = v \cdot s''$ for some v and s''. Therefore, choose $s' = s''$, $f' = f[x \mapsto v]$, and $pc' = pc + 1$.

Case 6: $P[pc] = \texttt{load}\ x$. Choose $s' = f[x] \cdot s$, $f' = f$, and $pc' = pc + 1$.

Case 7: $P[pc] = \texttt{new}\ \sigma$. $A^{\hat{\sigma}_{pc}}$ contains an infinite number values. Since VAR is fine made, there is at least one value $\hat{a}$ such that $Unused(\hat{a}, f, s)$. Choose $s' = \hat{a} \cdot s$, $f' = f$, and $pc' = pc + 1$.

Case 8: $P[pc] = \texttt{init}\ \sigma$. Since $s : S_{pc}$, and $S_{pc} = \hat{\sigma}_j \cdot S_{pc+1}$ for some j follows from rule (*init*), we know that $s = \hat{a} \cdot s''$ for some $\hat{a} \in A^{\hat{\sigma}_j}$ and s''. As in the previous case, there exists for at least one value a such that $Unused(a, f, s)$. Therefore, choose $s' = [a/\hat{a}]s''$, $f' = [a/\hat{a}]f$, and $pc' = pc + 1$.

Case 9: $P[pc] = \texttt{use}\ \sigma$. Since $s : S_{pc}$ and $S_{pc} = \sigma \cdot S_{pc+1}$ follows from (*use*), we know that $s = a \cdot s''$ for some $a \in A^{\sigma}$ and s''. Therefore, choose $s' = s''$, $f' = f[x \mapsto v]$, and $pc' = pc + 1$. □

A.4 Soundness

We first extend the one-step soundness theorem to execution sequences of any length:

LEMMA 22. *Given P, F, and S such that $F, S \vdash P$,*

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall pc, f_0, f, s.} \\
 & P \vdash \langle 1, f_0, \epsilon \rangle \rightarrow^* \langle pc, f, s \rangle \\
\Rightarrow & s : S_{pc} \\
 & \wedge\ \forall y \in \text{VAR}.\ f[y] : F_{pc}[y] \\
 & \wedge\ ConsistentInit(F_{pc}, S_{pc}, f, s) \\
 & \wedge\ pc \in Dom(P)
\end{array}$$

PROOF. The proof is by induction on n, the number of execution steps. The base case is when $n = 0$. In this case, $\langle pc, f, s \rangle = \langle 1, f_0, \epsilon \rangle$. The conclusions of the implication follow from the initial machine state, the assumption that all programs have at least one line, and the constraints on S_1 and F_1 in rule (*wt prog*).

To prove the inductive step, we assume the lemma to be true for sequences of length n and prove the lemma for execution sequences of length $n + 1$. In this case, the execution sequence must be

$$P \vdash \langle 1, f_0, \epsilon \rangle \to^n \langle pc', f', s' \rangle \to \langle pc, f, s \rangle$$

for some pc', f' and s'. Since n steps were taken from $\langle 1, f_0, \epsilon \rangle$ to reach $\langle pc', f', s' \rangle$, we may apply the inductive hypothesis to conclude that

$$\begin{array}{l} s' : S_{pc'} \\ \wedge\ \forall y \in \text{VAR}.\ f'[y] : F_{pc'}[y] \\ \wedge\ ConsistentInit(F_{pc'}, S_{pc'}, f', s') \\ \wedge\ pc' \in Dom(P). \end{array}$$

Applying Theorem 1 to these four conditions and the execution step from $\langle pc', f', s' \rangle$ to $\langle pc, f, s \rangle$, we conclude

$$\begin{array}{l} s : S_{pc} \\ \wedge\ \forall y \in \text{VAR}.\ f[y] : F_{pc}[y] \\ \wedge\ ConsistentInit(F_{pc}, S_{pc}, f, s) \\ \wedge\ pc \in Dom(P). \end{array}$$

Thus, this lemma is true for execution sequences of any length. □

RESTATEMENT OF THEOREM 3. *Given P, F, and S such that $F, S \vdash P$,*

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

PROOF. Assume all the hypotheses are satisfied. We first prove the first clause of the conclusion. Suppose $P \vdash \langle 1, f_0, \epsilon \rangle \to^* \langle pc, f, s \rangle$ and $P[pc] \neq$ `halt`, but no further step can be taken by the program. By Lemma 22, the following are true:

$$\begin{array}{l} s : S_{pc} \\ \forall y \in \text{VAR}.\ f[y] : F_{pc}[y] \\ ConsistentInit(F_{pc}, S_{pc}, f, s) \\ pc \in Dom(P) \end{array}$$

However, these four assertions and the assumption that $P[pc] \neq$ `halt` mean, that, by Theorem 2, there does exist a state into which the program can step. This contradicts the assumption that the program is stuck at $\langle pc, f, s \rangle$, and we conclude that our assumption about $P[pc]$ is wrong. Thus, $P[pc] =$ `halt`.

We can conclude that the second half of the conjunction, $s : S_{pc}$, is true directly from the application of Lemma 22 to the assumptions of the implication. □

B. SOUNDNESS OF EXTENSIONS

This appendix describes how to extend the proofs from Appendix A to cover other type systems. The first two sections discuss JVML$_c$ and JVML$_s$, and the third section describes how to extend any of these systems to include additional primitive types and operations.

B.1 JVML$_c$

This section gives a brief overview of the soundness proof for JVML$_i$ with constructors. As previously described, the one-step soundness theorem must be augmented with another global invariant stating the equivalence of z in the run-time state and Z_P:

THEOREM 5 (CONSTRUCTOR ONE-STEP SOUNDNESS). *Given P, F, and S such that $F, S \vdash P$,*

$$\begin{array}{l}\forall pc, f, s, pc', f', s'.\\ \qquad P \vdash \langle pc, f, s, z\rangle \rightarrow \langle pc', f', s', z'\rangle\\ \qquad \wedge\ s : S_{pc}\\ \qquad \wedge\ \forall y \in \text{VAR}.\ f[y] : F_{pc}[y]\\ \qquad \wedge\ ConsistentInit(F_{pc}, S_{pc}, f, s)\\ \qquad \wedge\ z = Z_{P,pc}\\ \quad \Rightarrow\ s' : S_{pc'}\\ \qquad \wedge\ \forall y \in \text{VAR}.\ f'[y] : F_{pc'}[y]\\ \qquad \wedge\ ConsistentInit(F_{pc'}, S_{pc'}, f', s')\\ \qquad \wedge\ z' = Z_{P,pc'}\\ \qquad \wedge\ pc' \in Dom(P)\end{array}$$

PROOF SKETCH. With the exception of proving that this one new invariant is preserved by all instructions, proof of this theorem may be obtained with minor modifications to the proof of Theorem 1 in Appendix A. To prove that $z' = Z_{P,pc'}$, we first define a new property:

Definition 5. For some instruction I, I *preserves Constructor* if

$$\begin{array}{l}\forall P, F, S, pc, f, s, pc', f', s'.\\ \qquad F, S \vdash P\\ \qquad \wedge\ P \vdash \langle pc, f, s, z\rangle \rightarrow \langle pc', f', s', z'\rangle\\ \qquad \wedge\ P[pc] = \text{I}\\ \quad \Rightarrow\ z' = z\\ \qquad \wedge\ Z_{P,pc'} = Z_{P,pc}\end{array}$$

Note that all instructions except `super` σ guarantee this property. Given that the hypotheses of Theorem 5 are satisfied,

$$z' = Z_{P,pc'} \tag{14}$$

can be proved by case analysis on $P[pc]$. If $P[pc]$ *preserves Constructor* then (14) follows from Definition 5 and the assumption that $z = Z_{P,pc}$. For `super` σ, the only other possible instruction, $z' = \mathit{true}$ follows from the operational semantics, and $Z_{P,pc'} = \mathit{true}$ follows from the definition of Z_P and the fact that $pc' = pc + 1$. Therefore, line (14) holds for all possible instructions, and Theorem 5 is true. □

Extending the previously described progress theorem from Appendix A.3 to cover constructors is also relatively straightforward, and the constructor soundness theorem then follows:

RESTATEMENT OF THEOREM 4. *Given* P, F, S, φ, *and* $\hat{a}_\varphi$ *such that* $F, S \vdash P$ constructs φ *and* $\hat{a}_\varphi : \hat{\varphi}_0$:

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

PROOF SKETCH. The first half of the conclusion follows from reasoning similar to that the proof presented in Appendix A.4. The second half is true given the facts that z will be equal to $Z_{P,pc}$ and that the static semantics guarantees that $Z_{P,pc} = \mathit{true}$ if $P[pc] = \mathtt{halt}$. □

B.2 JVML$_s$

This section outlines the soundness proof for JVML$_i$ with subroutines. We refer the reader to the extended version of Stata and Abadi [1999] for many of the details omitted from this section. The proof sketch consists of three basic steps. We first define JVML$_i$ with subroutines in terms of a structured operational semantics based on the semantics presented in Stata and Abadi [1999, Sect. 5], and we present additional definitions needed for the proof. Next, we state and discuss the one-step soundness theorem for the structured semantics. The third step relates the structured semantics to the stackless semantics for JVML$_s$, shown previously in Figure 2 and Figure 7. This part uses a simulation between the structured and stackless semantics. Once the simulation is shown, the steps leading to the main soundness theorem for JVML$_s$ follow easily from our proofs in Appendix A and the proofs of Stata and Abadi.

Figure 11 shows the structured operational semantics for JVML$_s$. The machine state has a fourth component, ρ, representing the subroutine call

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

$$\frac{P[pc] = \texttt{pop}}{P \vdash_s \langle pc,\ f,\ v \cdot s,\ \rho\rangle \to \langle pc+1,\ f,\ s,\ \rho\rangle}$$

$$\frac{P[pc] = \texttt{push 0}}{P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \to \langle pc+1,\ f,\ 0 \cdot s,\ \rho\rangle}$$

$$\frac{P[pc] = \texttt{load}\ x}{P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \to \langle pc+1,\ f,\ f[x] \cdot s,\ \rho\rangle}$$

$$\frac{P[pc] = \texttt{store}\ x}{P \vdash_s \langle pc,\ f,\ v \cdot s,\ \rho\rangle \to \langle pc+1,\ f[x \mapsto v],\ s,\ \rho\rangle}$$

$$\frac{P[pc] = \texttt{if}\ L}{P \vdash_s \langle pc,\ f,\ 0 \cdot s,\ \rho\rangle \to \langle pc+1,\ f,\ s,\ \rho\rangle}$$

$$\frac{\begin{array}{c} P[pc] = \texttt{if}\ L \\ n \neq 0 \end{array}}{P \vdash_s \langle pc,\ f,\ n \cdot s,\ \rho\rangle \to \langle L,\ f,\ s,\ \rho\rangle}$$

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

$$\frac{P[pc] = \texttt{ret}\ x}{P \vdash_s \langle pc,\ f,\ s,\ pc' \cdot \rho\rangle \to \langle pc',\ f,\ s,\ \rho\rangle}$$

$$\frac{\begin{array}{c} P[pc] = \texttt{new}\ \sigma \\ \hat{a} \in A^{\hat{\sigma}_{pc}},\ \mathit{Unused}(\hat{a}, f, s) \end{array}}{P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \to \langle pc+1,\ f,\ \hat{a} \cdot s,\ \rho\rangle}$$

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

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

Fig. 11. JVML$_c$ structured operational semantics.

stack, which is implicit in the real virtual machine. The only instructions which change ρ are `jsr` L and `ret` x. The static type rules for the structured operational semantics are the same rules presented in Figure 3

$$\frac{\begin{array}{c}P[i] \in \{\texttt{inc}, \texttt{pop}, \texttt{push 0}, \texttt{load } x, \texttt{store } x, \texttt{new } \sigma, \texttt{init } \sigma, \texttt{use } \sigma\} \\ R_{i+1} = R_i\end{array}}{R, i \vdash P \text{ labeled}}$$

$$\frac{\begin{array}{c}P[i] = \texttt{if } L \\ R_{i+1} = R_L = R_i\end{array}}{R, i \vdash P \text{ labeled}}$$

$$\frac{\begin{array}{c}P[i] = \texttt{jsr } L \\ R_{i+1} = R_i \\ R_L = \{L\}\end{array}}{R, i \vdash P \text{ labeled}}$$

$$\frac{P[i] \in \{\texttt{halt}, \texttt{ret } x\}}{R, i \vdash P \text{ labeled}}$$

Fig. 12. Rules labeling instructions with subroutines.

and Figure 8. The judgment to conclude that a program is well typed is

$$\frac{\begin{array}{c}F_1 = F_{\text{TOP}} \\ S_1 = \epsilon \\ R_1 = \{\ \ \} \\ \forall i \in Dom(P).\ R,\ i \vdash P \text{ labeled} \\ \forall i \in Dom(P).\ F,\ S,\ i \vdash P\end{array}}{F,\ S \vdash_s P} \qquad (wt\ prog\ sub)$$

The map R is described in Figure 12 and relates a line of the program to the subroutine to which that line belongs. R_P represents the canonical R for program P.

Before stating the one-step soundness theorem, several additional definitions are needed. We first define the types of local variables taking into account the subroutine call stack. The type $\mathcal{F}(F,\ pc,\ \rho)[x]$ is defined by the following rules:

$$\frac{x \in Dom(F_{pc})}{\mathcal{F}(F,\ pc,\ \rho)[x] = F_{pc}[x]} \qquad (tt\ 0)$$

$$\frac{\begin{array}{c}x \notin Dom(F_{pc}) \\ \mathcal{F}(F,\ p,\ \rho)[x] = \tau\end{array}}{\mathcal{F}(F,\ pc,\ p \cdot \rho)[x] = \tau} \qquad (tt\ 1)$$

As we will see below, as long as ρ satisfies some well-formedness conditions, there will be some τ such that $\mathcal{F}(F,\ pc,\ \rho)[x] = \tau$ for every $x \in$ VAR. This definition matches the definition given in Stata and Abadi [1999]. We also need the *WFCallStack* judgment, which ensures that a subroutine call stack is well-formed:

$$\frac{Dom(F_{pc}) = \text{VAR}}{WFCallStack(P, F, pc, \epsilon)} \qquad (wf\ 0)$$

$$\frac{\begin{array}{c} P[p-1] = \texttt{jsr}\ L \\ L \in R_{P,pc} \\ Dom(F_{pc}) \subseteq Dom(F_p) \\ WFCallStack(P, F, p, \rho) \end{array}}{WFCallStack(P, F, pc, p \cdot \rho)} \qquad (wf\ 1)$$

In the presence of subroutines, the *ConsistentInit* invariant described in Section 5 is insufficient for guaranteeing that uninitialized objects are not used, and it must be strengthened. We define the new invariant *ConsistentInitWithSub* as follows:

$$\begin{array}{l} ConsistentInitWithSub(P, F, S, pc, f, s, p) \equiv \\ \quad ConsistentInit(F_{pc}, S_{pc}, f, s) \\ \quad \wedge\ \forall y \in \text{VAR} \backslash Dom(F_{pc}).\ \neg \exists \tau.\ \mathcal{F}(F, pc, \rho)[y] = \tau \wedge \tau \in \hat{T} \end{array}$$

The first line of the definition ensures the necessary correspondence between values and uninitialized object types, taking into account the subroutine call stack. In addition, we guarantee that uninitialized objects are not hidden in variables inaccessible to the program at the current instruction.

With these definitions, we may now state the one-step soundness theorem.

THEOREM 6 (STRUCTURED ONE-STEP SOUNDNESS). *Given P, F, and S such that $F, S \vdash_s P$,*

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s', \rho'. \\ \qquad P \vdash_s \langle pc, f, s, \rho \rangle \rightarrow \langle pc', f', s', \rho' \rangle \\ \qquad \wedge\ s : S_{pc} \\ \qquad \wedge\ \forall y \in \text{VAR}.\ \exists \tau.\ \mathcal{F}(F, pc, \rho)[y] = \tau \wedge f[y] : \tau \\ \qquad \wedge\ WFCallStack(P, F, pc, \rho) \\ \qquad \wedge\ ConsistentInitWithSub(P, F, S, pc, f, s, \rho) \\ \quad \Rightarrow\ s' : S_{pc'} \\ \qquad \wedge\ WFCallStack(P, F, pc', \rho') \\ \qquad \wedge\ \forall y \in \text{VAR}.\ \exists \tau'.\ \mathcal{F}(F, pc', \rho')[y] = \tau' \wedge f'[y] : \tau' \\ \qquad \wedge\ ConsistentInitWithSub(P, F, S, pc', f', s', \rho') \\ \qquad \wedge\ pc' \in Dom(P) \end{array}$$

PROOF SKETCH. Many of the details of this proof may be found in the work of Stata and Abadi [1999] and in Appendix A. Therefore, we proceed by briefly justifying each of the five invariants listed in the theorem and indicating the main differences between this proof and the previous ones:

—$s' : S_{pc'}$: This follows from a proof similar to the proof of Lemma 15, where that lemma and Definition 1 are augmented with ρ.

—$pc' \in Dom(P)$: This follows from a proof similar to Lemma 21 for all instructions in the language. The one complicated case, `ret` x, is proved by Stata and Abadi.

—$WFCallStack(P, F, pc', \rho')$: Appendix A.1 of Stata and Abadi [1999] proves this invariant. All instructions other than `jsr` and `ret` satisfy the conditions of Lemma 1 in that paper and preserve this invariant simply because they do not affect the subroutine call stack or the domain of visible local variables.

—$\forall y \in \text{VAR}.\ \exists \tau'.\ \mathcal{F}(F, pc', \rho')[y] = \tau' \wedge f'[y] : \tau'$: This is proved by Stata and Abadi for all instructions except those added to study object initialization. Although `new` σ and `use` σ are trivial to prove, `init` σ is fairly tricky. We cannot rely on our previous proofs, since they do not take into account the polymorphism of local variables. For this case, the following equations are derived from the operational and static semantics

$$\begin{aligned} pc' &= pc + 1 \\ \rho' &= \rho \\ s &= \hat{a} \cdot s'' \\ f' &= [a/\hat{a}]f \\ S_{pc} &= \hat{\sigma}_j \cdot \alpha \\ F_{pc'} &= [\sigma/\hat{\sigma}_j]F_{pc} \end{aligned}$$

for some $\hat{a}$, a, σ, and j. From these, we know that $Corresponds(F_{pc}, S_{pc}, f, s, \hat{a}, \hat{\sigma}_j)$ by Lemma 1. To prove that

$$\forall y \in \text{VAR}.\ \exists \tau'.\ \mathcal{F}(F, pc', \rho')[y] = \tau' \wedge f'[y] : \tau'$$

we find such a τ' for each y:

(1) $y \in Dom(F_{pc})$: We know that $\mathcal{F}(F, pc, \rho)[y] = F_{pc}[y]$. By Lemma 12, $([a/\hat{a}]f)[y]$: $([\sigma/\hat{\sigma}_j]F_{pc})[y]$ must be true, meaning that $f'[y] : F_{pc'}[y]$. Since $Dom(F_{pc'}) = Dom(F_{pc})$, we know that $\mathcal{F}(F, pc', \rho')[y] = F_{pc'}[y]$. Thus, choose $\tau' = F_{pc'}[y]$.

(2) $y \notin Dom(F_{pc})$: In this case, $\mathcal{F}(F, pc', \rho')[y] = \tau$ only if $\mathcal{F}(F, pc, \rho)[y] = \tau$. Also, from our assumption that $ConsistentInitWithSub(P, F, S, pc, f, s, \rho)$, we know that $\mathcal{F}(F, pc, \rho)[y] \neq \hat{\sigma}_j$. If $f[y] \neq \hat{a}$, then we are done, since local variable y is not affected. If $f[y] = \hat{a}$, then $\mathcal{F}(F, pc', \rho)[y] = \text{TOP}$, and since $\hat{a} : \text{TOP}$, this case still holds. Thus, choose τ' such that $\mathcal{F}(F, pc, \rho)[y] = \tau'$.

—*ConsistentInitWithSub*$(P, F, S, pc', f', s', \rho')$: If we know that $\rho' = \rho$ and $Dom(F_{pc'}) = Dom(F_{pc})$, both clauses may be proved by restricting our proofs from Appendix A.2 to consider only variables in $Dom(F_{pc})$ and noting that no variables outside of the domain of the current instruction are affected by executing an instruction. This takes care of all cases except `jsr` L and `ret` x.

—For `jsr` L, we know that

$$\forall y \in \text{VAR} \backslash Dom(F_{pc}).\ \mathscr{F}(F, pc, \rho)[y] = \tau \Rightarrow \tau \notin \hat{T} \tag{15}$$

by *ConsistentInitWithSub*$(P, F, S, pc, f, s, \rho)$. Also, we know that $\forall y \in Dom(F_{pc})$, $F_{pc}[y] \notin \hat{T}$ by (*jsr*). That rule also guarantees that

$$\begin{gathered}\forall y \in Dom(S_L).\ S_L[y] \notin \hat{T}\\ \forall y \in Dom(F_L).\ F_L[y] \notin \hat{T}\\ Dom(F_{pc+1}) = Dom(F_{pc})\\ \forall y \in Dom(F_{pc+1}) \backslash Dom(F_L).\ F_{pc+1}[y] = F_{pc}[y]\end{gathered}$$

There are no uninitialized object types present in F_L and S_L, and *ConsistentInit*(F_L, S_L, f', s') is derivable. Also, the fourth line above and (15) indicate that

$$\forall y \in Dom(F_{pc}) \backslash Dom(F_L).\ \forall \tau.\ \mathscr{F}(F, L, (pc + 1) \cdot \rho)[y] = \tau \Rightarrow \tau \notin \hat{T} \tag{16}$$

because, for all such y, $\mathscr{F}(F, L, (pc + 1) \cdot \rho)[y] = F_{pc+1}[y]$. From (*jsr*), we know that $F_{pc+1}[y]$, where we have already stated that $F_{pc}[y] \notin \hat{T}$. In addition, if $y \notin Dom(F_{pc})$, then rule (*tt 1*) shows that we can derive $\mathscr{F}(F, L, (pc + 1) \cdot \rho)[y] = \tau$ only when we can also derive $\mathscr{F}(F, pc + 1, \rho)[y] = \tau$. Given that $Dom(F_{pc+1}) = Dom(F_{pc})$, this is the same as deriving $\mathscr{F}(F, pc, \rho)[y] = \tau$. From (15), we know that such a τ is not a type in $\hat{T}$. Combining this with (16) makes the second half of *ConsistentInitWithSub*$(P, F, S, pc', f', s', \rho')$ true.

For `ret` x, we know that $\rho = pc' \cdot \rho'$. The types stored in $F_{pc'}$ are constrained in two ways. Those variables also in the domain of the subroutine from which we are returning must have the same type at pc' as at pc, and these types do not belong to $\hat{T}$, as constrained by (*ret*). Those variables in $Dom(F_{pc'})$ but not in $Dom(F_{pc})$ must also not contain uninitialized object types, since those local variables must have the same types in $F_{pc'}$ as in $F_{pc'-1}$. This is true because $P[pc' - 1] =$ `jsr` L for some L, and the (*jsr*) rule will constrain these variables to not belong to $\hat{T}$. Thus, $\forall y \in Dom(F_{pc'})$. $F_{pc'}[y] \notin \hat{T}$ is true. As before, no uninitialized object types appear in $F_{pc'}$ and $S_{pc'}$, making *ConsistentInit*$(F_{pc'}, S_{pc'}, f', s')$ trivially true. The second half of *ConsistentInitWithSub*$(P, F, S, pc', f', s', \rho')$ easily follows from the above statements as well. □

The third step for proving soundness for JVML$_i$ with subroutines is to show a simulation between the structured and stackless semantics. While we do not describe the details, much of the proof follows directly from Appendix B of Stata and Abadi, and the insight that `new` σ, `init` σ, and `use` σ neither change the subroutine call stack nor touch any value or type corresponding to a return address. Once the simulation has been shown, the main soundness theorem is easily proved.

B.3 Primitive Types and Basic Operations

One of the benefits of the proof style used in Appendix A is that it relates very simple properties of instruction execution to the preservation of the global invariants required by the one-step soundness theorem. For example, any instruction whose operational and static semantics exhibit the property in Definition 1 is guaranteed to preserve the invariant that the operand stack is well typed.

Using these properties, we can add many more instructions and basic types to JVML$_i$ with very little effort. Instead of reasoning about the global invariants directly, we may reason in terms of the much simpler properties.

For an example of this, we add the instruction `iadd`. The operational and static semantics for this instruction are

$$\frac{P[pc] = \texttt{iadd}}{P \vdash \langle pc, f, n_1 \cdot n_2 \cdot s\rangle \rightarrow \langle pc + 1, f, (n_1 + n_2) \cdot s\rangle}$$

$$\frac{\begin{array}{c} P[i] = \texttt{iadd} \\ F_{i+1} = F_i \\ S_i = \text{INT} \cdot \text{INT} \cdot \alpha \\ S_{i+1} = \text{INT} \cdot \alpha \\ i + 1 \in Dom(P) \end{array}}{F, S, i \vdash P} \qquad (iadd)$$

To show that the soundness theorems proved for JVML$_i$ also apply to JVML$_i$ with `iadd`, its suffices to prove (1) that the four properties from Appendix A apply to `iadd` and (2) that progress can always be made if that instruction is about to be executed in a well-formed state. We first prove the four properties:

—`iadd` *preserves StackType*: Assume that the hypotheses of the implication in Definition 1 are satisfied. Since $s : S_{pc}$ and $pc' = pc + 1$, we know that $s = n_1 \cdot n_2 \cdot s''$ for some s'', and integers n_1 and n_2. Choose $s_1 = n_1 \cdot n_2$, $s_2 = s''$, $s_3 = (n_1 + n_2)$, $\alpha_1 = \text{INT} \cdot \text{INT}$, $\alpha_2 = \alpha$, and $\alpha_3 = \text{INT}$.

—`iadd` *preserves VariableType*: From the operational and static semantics, we know that $pc' = pc + 1$, $f' = f$, and $F_{pc'} = F_{pc}$.

—iadd *preserves ConsistentInit*: Choose s_1, s_2, s_3, α_1, α_2, and α_3 as in the stack case. Also note that $f' = f$, $F_{pc'} = F_{pc}$, and INT $\notin \hat{T}$ follow from the operational and static semantics, given that $pc' = pc + 1$.

—iadd *preserves ProgramDomain*: From the operational semantics, $pc' = pc + 1$, and (*iadd*) ensures $i + 1 \in Dom(P)$.

To show that progress can always be made, assume that hypotheses of Theorem 2 are satisfied and that $P[pc]$ = iadd. As above, we know that $s = n_1 \cdot n_2 \cdot s''$ for some values of n_1, n_2, and s''. Choose $pc' = pc + 1$, $f' = f$, and $s' = (n_1 + n_2) \cdot s''$.

ACKNOWLEDGMENTS

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 useful discussions.

REFERENCES

ARNOLD, K. AND GOSLING, J. 1996. *The Java Programming Language.* Addison-Wesley, Reading, MA.

COHEN, R. 1997. Defensive Java virtual machine version 0.5 alpha release. Available via http//www.cli.com/software/djvm/index.html.

DROSSOPOLOU, S., EISENBACH, S., AND KHURSHID, S. 1999. Is the Java type system sound? *Theory Pract. Obj. Syst. 5*, 1, 3–24.

DEAN, D., FELTEN, D. W., WALLACH, D. S., AND BALFANZ, D. 1997. Java security: Web browsers and beyond. In *Internet Besieged: Countering Cyberspace Scofflaws*, D. E. Denning and P. J. Denning, Eds. ACM Press, New York.

GOLDBERG, A. 1998. A specification of Java loading and bytecode verification. In *Proceedings of the ACM Conference on Computer and Communication Security*. ACM, New York.

HAGIYA, M. AND TOZAWA, A. 1998. On a new method for dataflow analysis of Java virtual machine subroutines. In *Proceedings of the 5th Static Analysis Symposium*.

JONES, M. 1998. The functions of Java bytecode. In *Proceedings of the Workshop on the Formal Underpinnings of the Java Paradigm*.

KNAPP, A., CENCIARELLI, P., REUS, B., AND WIRSING, M. 1998. An event-based structural operational semantics of multi-threaded Java. In *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed. Lecture Notes in Computer Science, vol. 1523. Springer-Verlag, Berlin.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA.

MORRISETT, G., CRARY, K., GLEW, N., AND WALKER, D. 1998. From system F to typed assembly language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*. ACM, New York. Also in *ACM Trans. Program. Lang. Syst. 21*, 3 (May 1999), pp. 527–568.

NIPKOW, T. AND VON OHEIMB, D. 1998. Java$_{light}$ is type-safe—Definitely. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*. ACM, New York.

O'CALLAHAN, R. 1999. A simple, comprehensive type system for Java bytecode subroutines. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*. ACM, New York.

POSEGGA, J. AND VOGT, H. 1998. Bytecode verification for Java smart cards based on model checking. In *Proceedings of the 5th European Symposium on Research in Computer Security (ESORICS)*. Lecture Notes in Computer Science. Springer-Verlag, Berlin.

QIAN, Z. 1998. A formal specification of Java(tm) virtual machine instructions. In *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed. Lecture Notes in Computer Science, vol. 1523. Springer-Verlag, Berlin.

STATA, R. AND ABADI, M. 1999. A type system for Java bytecode subroutines. *ACM Trans. Program. Lang. Syst. 21*, 1 (Jan.), 90–137.

SARASWAT, V. 1997. The Java bytecode verification problem. Available via http://www.research.att.com/~vj.

SIRER, E. G., MCDIRMID, S., AND BERSHAD, B. 1997. Kimera: A Java system architecture. Available via http://kimera.cs.washington.edu.

SYME, D. 1997. Proving Java type soundness. Tech. Rep. 427, Univ. of Cambridge Computer Laboratory.

TARDITI, D., MORRISETT, G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. 1996. TIL: A type-directed optimizing compiler for ML. *ACM SIGPLAN Not. 31*, 5 (May), 181–192.

YELLAND, P. 1999. A compositional account of the Java Virtual Machine. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*. ACM, New York.

Received November 1998; revised July 1999; accepted July 1999