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

Stephen N. Freund John C. Mitchell
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
{freunds, mitchell}@cs.stanford.edu
Phone: (415) 723-8634, Fax: (415) 725-4671

April 17, 1998

Abstract

In the standard Java implementation, a Java language program is compiled to Java bytecode. This bytecode may be sent across the network to another site, where it is then interpreted by the Java Virtual Machine. Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a *bytecode verifier* that performs a number of consistency checks before code is interpreted. As illustrated by previous attacks on the Java Virtual Machine, these tests, which include type correctness, are critical for system security. In order to analyze existing bytecode verifiers and to understand the properties that should be verified, we develop a precise specification of *statically-correct* Java bytecode, in the form of a type system. Our focus in this paper is a subset of the bytecode language dealing with object creation and initialization. For this subset, we prove that for every Java bytecode program that satisfies our typing constraints, every object is initialized before it is used. The type system is easily combined with a previous system developed by Stata and Abadi for bytecode subroutines. Our analysis of subroutines and object initialization reveals a previously unpublished bug in the Sun JDK bytecode verifier.

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

Contents

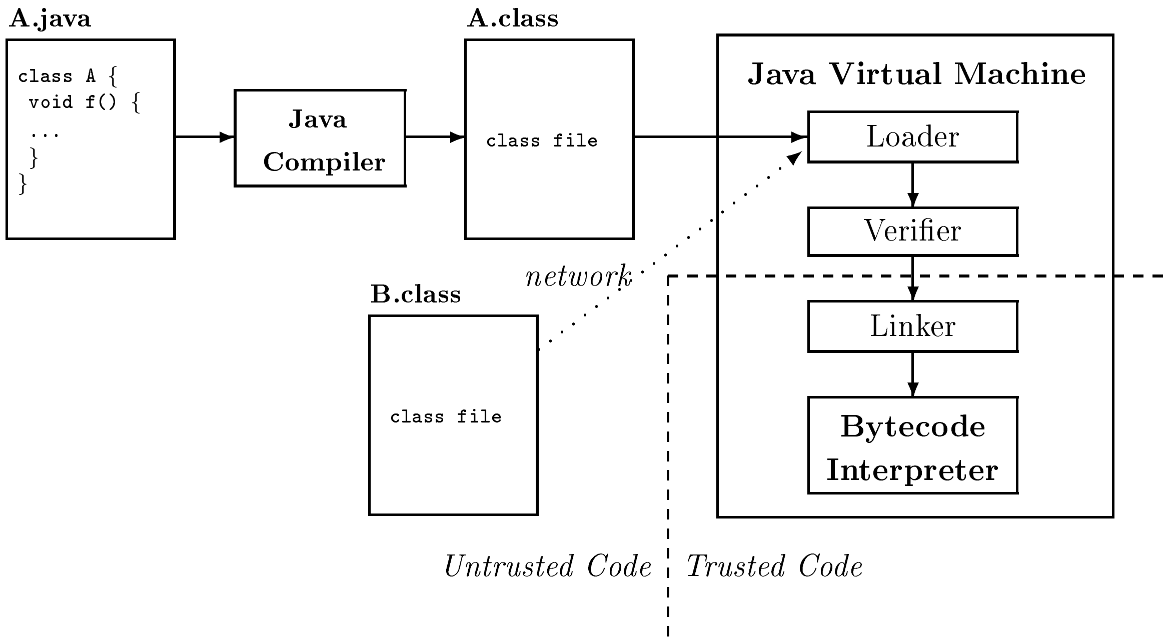

Figure 1: The Java Virtual Machine

1 Introduction

The Java programming language is a statically-typed general-purpose programming language with an implementation architecture that is designed to facilitate transmission of compiled code across a network. In the standard implementation, a Java language program is compiled to Java bytecode and this bytecode is then interpreted by the Java Virtual Machine. While many previous programming languages have been implemented using a bytecode interpreter, the Java architecture differs in that programs are commonly transmitted between users across a network in compiled form.

Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a *bytecode verifier* that performs a number of consistency checks before code is interpreted. Figure 1 shows the point at which the verifier checks a program during the compilation, transmission, and execution process. After a class file containing Java bytecodes is loaded by the Java Virtual Machine, possibly from across the network, it must pass through the bytecode verifier before being linked into the execution environment and interpreted. Thus, only trusted bytecode is used by the linker and interpreter. This protects the receiver from certain security risks and various forms of attack. The verifier rejects any program that uses an uninitialized value, jumps to invalid code, performs operations on values of the wrong type, and so on. Fairly conservative static analysis techniques are used to check these conditions, meaning that not only all faulty programs are rejected, but also many programs that would never execute an erroneous sequence of instructions. However, any bytecode program generated by a conventional compiler is accepted. The need for conservative analysis stems from the undecidability of the halting problem, as well as efficiency considerations. Specifically, since most bytecode is the result of compilation, there is very little benefit in developing complex analysis techniques to recognize patterns that do not occur.

The intermediate bytecode language, which we refer to as JVML, is a typed, machine-independent form of assembly language with some low-level instructions that reflect specific high-level Java source language constructs. For example, classes are a basic notion in JVML, 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 easily detected in the resulting bytecode. One example is the last-called first-returned property of the local subroutines. While this property will hold for every JVML program generated by compiling Java source, some effort is required to confirm this property in bytecode programs [SA98a]. Another example is the initialization of objects before use. While it is clear from the Java source language statement

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

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

Several published attacks on early forms of the Java Virtual Machine illustrate the importance of the bytecode verifier for system security. To cite one specific example, a bug in an early version of Sun's bytecode verifier allowed applets to create certain system objects which they should not have been able to create, such as class loaders [DFW96]. The problem was caused by an error in how constructors were verified and resulted in the ability to potentially compromise the security of the entire system. Clearly, problems like this give rise to the need for a correct and formal specification of the bytecode verifier. However, for a variety of reasons, there is no established formal specification; the primary specification is an informal English description that is occasionally at odds with current verifier implementations. In this paper, we develop a specification for a fragment of the bytecode language that includes object creation (allocation of memory) and initialization. This work is based on a prior study of the bytecodes for local subroutine call and return [SA98a].

In brief, we develop a specification of *statically-correct bytecode* for a fragment of JVML that includes object creation and initialization. This specification has the form of a type system, although there are several technical ways in which a type system for low-level code with jumps and type-varying use of stack locations (or registers) differs from conventional high-level type systems. We prove soundness of the type system by a traditional method using operational semantics. It follows from the soundness theorem that any bytecode program that passes the static checks will initialize every object before it is used. We examined a broad range of alternatives for specifying type systems capable of identifying that kind of error. In some cases, we found it possible to simplify our specification by being more or less conservative than current verifiers. However, we generally resisted the temptation to do so since we hoped to gain some understanding of the strength and limitations of existing verifier implementations.

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

The work described in this paper opens several promising directions. One major task, which we are currently undertaking, is to extend the specification and correctness proof to the entire Java Virtual Machine language (JVML), including the method call stack and a full object system. We also believe it will be feasible to generate an implementation of a bytecode verifier from a specification proven to be correct. This specification could be expressed in the kind of typing rules we use here, or some variant of this notation. Finally, we expect that in the long run, it will be useful to incorporate additional properties into the static analysis of Java programs. If Java is to become a popular and satisfactory general-purpose programming language, then for efficiency reasons alone, it will be necessary to replace some of the current run-time tests by conservative static analysis, perhaps reverting to run-time tests when static analysis fails.

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

2 Object Initialization

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

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

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

The first line indicates that a new `Point` object should be created and calls the `Point` constructor to initialize this object. The second line invokes a method on this object and therefore can only be allowed if the object has been initialized. Since every Java object is created by a statement like the one in the first line here, it does not seem difficult to prevent Java source language programs from invoking methods on objects that have not been initialized. While there are a few subtle situations to consider, such as when a constructor throws an exception, the issue is essentially clear cut.

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

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

The most striking difference is that memory allocation (line 1) is separated from the constructor invocation (line 4) by two lines of code. The first intervening line, `dup`, duplicates the pointer to the uninitialized object. The reason for this instruction is that a pointer to the object must be passed to the constructor. A convention of parameter passing for the stack-based architecture is that parameters to a function are popped off the stack before the function returns. Therefore, if the address were not duplicated, there would be no way for the code creating the object to access it after it is initialized. The second line, `iconst_3` pushes the constructor argument 3 onto the stack. If `p` were used again after line 5 of the bytecode program, another `dup` would have been needed prior to line 5. Depending on the number and type of constructor arguments, many different instruction sequences may appear between object allocation and initialization. For example, suppose that several new objects are passed as arguments to a constructor. In this case, it is necessary to create each of the argument objects and initialize them before passing them to the constructor. In general, the code fragment between allocation and initialization may involve substantial computation, including allocation of new objects, duplication of object pointers, and jumps to or branches from other locations in the code.

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

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

When line 5 is reached during execution, there will be two different uninitialized `Point` objects. If the bytecode verifier is to check object initialization statically, it must be able to determine which references point to the object that is initialized at line 5 and which point to the remaining uninitialized object. Otherwise, the verifier would either prevent use of an initialized object or allow use of an uninitialized one.

Sun's Java Virtual Machine Specification [LY96] describes the alias analysis used by the Sun JDK verifier. For each line of the bytecode program, some status information is recorded for every local variable and stack location. When a location points to an object that is known not to be initialized in all executions reaching this statement, the status will include not only the property *uninitialized,* but also the line number on which the uninitialized object would have been created. As references are duplicated on the stack and stored and loaded in the local variables, the analysis also duplicates these line numbers, and all references having the same line number are assumed to refer to the same object. When an object is initialized, all pointers that refer to objects created at the same line number are set to *initialized.* In other words, all references to uninitialized objects of a certain type are partitioned into equivalence classes according to what is statically known about each reference, and all references that point to uninitialized objects created on the same line are assumed to be aliases. Since aliasing is irrelevant for initialize-before-use analysis of objects that have been initialized, it is not necessary to track aliasing once a reference leads to an initialized object. This is a very simple and highly conservative form of aliasing analysis; far more sophisticated methods might be considered. However, the approach can be implemented efficiently and it is sufficiently accurate to accept bytecode produced by standard compilers.

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

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

3 JVML$_i$

This section describes the JVML$_i$ language, a subset of JVML encompassing basic constructs and object initialization. Although this language is much smaller than JVML, it is sufficient to study object initialization and formulate a sound type system encompassing the static analysis described above. The run-time environment for JVML$_i$ consists only of an operand stack and a finite set of local variables. We do not model the object heap since, as we demonstrate below, this is not necessary to study the problem of object initialization. A JVML$_i$ program will be a sequence of instructions drawn from the following list:

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

where x is a local variable name, σ is an object type, and L is an address of another instruction in the program. As a simple example, Figure 8 shows a program written in an extended form of JVML$_i$. These instructions are defined as follows:

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

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

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

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

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

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

`halt`: terminates program execution.

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

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

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

Any cases not covered by the definitions above are illegal. For example, a `pop` instruction cannot be executed if the stack is empty. Although `dup` does not appear in JVML$_i$ for simplicity, aliasing may arise by storing and loading object references from the local variables.

4 Operational and Static Semantics

4.1 Notation

This section briefly reviews the framework developed by Stata and Abadi in [SA98a] for studying JVML. A program is formally modeled as a partial function from addresses to instructions. We refer to the set of all possible instruction addresses as ADDR. Although we shall use integers to represent elements of this set, we will distinguish elements of ADDR from integers. The set of local variables accessible by a program is VAR. $\mathit{Dom}(P)$ represents the set of addresses used in program P, and $P[i]$ is the i^{th} instruction in program P. $\mathit{Dom}(P)$ will always include address 1 and is usually a range $\{1, \ldots, n\}$ for some n.

Equality on partial maps is defined as

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

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

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

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

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

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

$$([b/a]\epsilon) = \epsilon$$

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

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

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

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

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

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

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

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

We also extend values and types to sequences:

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

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 of the form $\langle pc, f, s\rangle$, which has the following meaning:

- pc is a program counter, indicating the address of the instruction that is about to be executed.
- f is a total map from $\textsc{Var}$, the set of local variables, to the values stored in the local variables in the current state.
- s is a stack of values representing the operand stack for the current state in execution.

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

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

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

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

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

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

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

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

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

Figure 2: JVML$_i$ operational semantics.

to indicate that a program P in state $\langle pc,\ f,\ s\rangle$ can move to state $\langle pc',\ f',\ s'\rangle$ in one step. The complete one-step operational semantics for JVML$_i$ is shown in Figure 2. In that figure, n is any integer, v is any value, L and j are any addresses, and x is any local variable. These operational semantic rules, with the exception of those added to study object initialization, are discussed in detail in [SA98a]. The rules have been designed so that a step cannot be made from an illegal state, such as `pop` when there is an empty stack.

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

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

This will allow the virtual machine to pick any value that is currently not used by the program. The only values being used are those which appear on the operand stack or in the local variables. 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. Also, the new value is required to be unused. This allows the program to distinguish between different objects of the same type after they have been initialized, but this fact is not necessarily needed to study the properties addressed by this paper.

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

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

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

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

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

Figure 3: Static semantics.

4.4 Static Semantics

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

$$F, S \vdash P,$$

where F is a map from $\textsc{Addr}$ to functions mapping local variables to types, and S is a map from $\textsc{Addr}$ to stack types such that S_i is the type of the operand stack at location i of the program. As described in [SA98a], elements in a map over $\textsc{Addr}$ are accessed as F_i instead of $F[i]$. Thus, $F_i[y]$ is the type of local variable y at line i of a program. The judgment which allows us to conclude that a program P is well typed by F and S is

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

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

The static type rules for (*new*) and (*use*) are straightforward. The (*new*) rule requires that the type of the object allocated by the `new` instruction is left on top of the stack. The rule for (*use*) requires that an initialized object type be on top of the stack. The (*init*) rule implements the static analysis method described in Section 2. That rule requires 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.

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

5 Soundness

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

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

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

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

The final requirement for a state to be well formed is that it has the *ConsistentInit* property. Informally, this property means that the machine state 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 this static analysis. The *ConsistentInit* property is based on a unique correspondence between uninitialized object types and run-time values.

This paragraph describes the formal definition of *ConsistentInit*, which appears in Figure 4. For each possible uninitialized object type, the (*cons init*) rule guarantees that there is some value corresponding to every occurrence of that type in the local variables and on the stack. More precisely, for every uninitialized object type $\hat{\tau}$, there is a $\hat{b}$, a value of type $\hat{\tau}$, such that every occurrence of $\hat{\tau}$ in the static types for the local variables is matched by $\hat{b}$ in the current state. This condition is line 1 of the (*corr*) rule. The stack case is covered by line 2, which defines the correspondence between values and uninitialized object types inductively.

The proof of Theorem 1 is by case analysis on all possible instructions at $P[pc]$. The proof of this theorem and those that follow appear in Appendix A. A complementary theorem is that a step can always be made from a well-formed state, unless the program has reached a `halt` instruction. This progress theorem can be stated as:

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

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

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

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

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

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

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

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

Figure 4: The *ConsistentInit* judgement.

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

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

If a program executing in our machine model attempts to perform an operation leading to a type error, it would get stuck since those operations are not defined by our operational semantics. If we prove that well-typed programs only get stuck when a `halt` instruction is reached, then we know that those programs will not attempt to perform any illegal operations. Thus, this theorem implies a form of correctness for our static analysis by showing that no erroneous programs are accepted.

6 Extensions

Several extensions to the JVML$_i$ framework and proofs described in the previous sections have been studied. As mentioned previously, 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.

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

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

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

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

Figure 5: Rules checking that a super class constructor will always be called prior to reaching a `halt` instruction.

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$_i$.

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

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

JVML$_c$ programs are sequences of instructions containing any instructions from JVML$_i$ plus one new instruction, `super` φ. This instruction represents calling a constructor of the parent class of class φ (or a different constructor of the current class). To model the initial state of a constructor invocation for class φ, a JVML$_c$ program is assumed to begin in a state in which local variable 0 contains an uninitialized reference. This corresponds to the argument of the constructor. Prior to halting, the program must call `super` on that object reference. This represents calling the super class constructor.

For simplicity, the rest of this section assumes that we are describing a constructor for object type φ, for any φ in T. We will use $\hat{\varphi}_0$ as the type of the object that is 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, machine state in the operational semantics is augmented with a fourth element, z, which indicates whether or not a super class constructor has been called on the object that is being initialized. The rules for all instructions other than `super` φ do not affect z, and are derived directly from the rules in Figure 2. For example, the rule for `inc` is:

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

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

For `super`, the operational semantics rule is:

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

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

$$(\mathit{super})\ \frac{\begin{array}{c} P[i] = \texttt{super}\ \sigma \\ \sigma \in T \\ F_{i+1} = [\sigma/\hat{\sigma}_0]F_i \\ S_i = \hat{\sigma}_0 \cdot \alpha \\ S_{i+1} = [\sigma/\hat{\sigma}_0]\alpha \\ i+1 \in \mathit{Dom}(P) \end{array}}{F, S, i \vdash P} \qquad (\mathit{wt\ constructor})\ \frac{\begin{array}{c} F_1 = F_{\text{TOP}} \\ S_1 = \epsilon \\ Z_1 = \mathit{false} \\ \varphi \in T \\ \forall i \in \mathit{Dom}(P).\ F, S, i \vdash P \\ \forall i \in \mathit{Dom}(P).\ Z, i \vdash P \text{ constructs } \varphi \end{array}}{F, S \vdash P \text{ constructs } \varphi}$$

The (*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 5. 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. The existence of unreachable code may cause more than one value of Z to conform to the rules in Figure 5. To make Z unique for any given program, we assume that, for program P, there is a unique canonical form Z_P. Thus, $Z_{P,i}$ will be a unique value for instruction i.

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

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

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

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

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

Figure 6: Operational semantics for `jsr` and `ret`.

The main difference in the proof of Theorem 4, in comparison with Theorem 3, is that the corresponding one-step soundness theorem requires an additional invariant. The invariant states that when program P is in state $\langle pc,\ f,\ s,\ z\rangle$, $z = Z_{P,pc}$. The proof of this theorem appears in 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 [SA98a, LY96]. Subroutines are used to compile the `finally` clauses of exception handlers in the Java language. Subroutines share the same activation record as the method which uses them, and they can be called from different locations in the same method, enabling all locations where `finally` code must be executed to jump to a single subroutine containing that code. The flexibility of this mechanism makes bytecode verification difficult for two main reasons:

- Subroutines are polymorphic over local variables which they do not use.
- Subroutines may call other subroutines, as long as a call stack discipline is preserved. In other words, the most recently called subroutine must be the first one to return. This is a slight simplification of the rules for subroutines defined in [LY96], which do allow a subroutine to return more than one level up in the implicit subroutine call stack in certain cases, but does match the definitions presented in [SA98a].

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

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

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

The operational semantics and type rules for these instructions are shown in Figure 6 and Figure 7. These rules are based on the rules used by Stata and Abadi [SA98a]. The meaning of $R_{P,i} = \{L\}$ in (*ret*) is defined in their paper and basically means that instruction i is an instruction belonging to the subroutine starting at address L. All other rules are the same as those for JVML$_i$.

The main issue which must be addressed in the type rules for `jsr` and `ret` is the *Consistent-Init* 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.

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

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

Figure 7: Type rules for `jsr` and `ret`.

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

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

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

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

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

These lines 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 for type rules to accept more programs, but these programs could not have been created by a compiler for any valid Java program.

The main soundness theorem, Theorem 3, has been proved for JVML$_s$, and for JVML$_c$ with subroutines, by combining the proof of JVML$_i$ soundness with the work of Stata and Abadi. These proofs appear in Appendix B.2.

6.3 Other Basic Types and Instructions

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

Of the 200 bytecode instructions in JVML, all but approximately 40 fall into this category and may be added to JVML$_i$ without trouble, although a full presentation of the operational and type rules for these instructions is beyond the scope of this paper. With these additions, and the methods described in the previous subsections, the JVML$_i$ framework can be extended to cover the whole bytecode language, except for a full object system 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.

```
1:   jsr 9            9:   store 0
2:   store 1          10:  new P
3:   jsr 9            11:  ret 0
4:   store 2
5:   load 2
6:   init P
7:   load 1
8:   use P
```

Figure 8: A program that uses an uninitialized object but is accepted by Sun's verifier.

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. A simple program exhibiting the incorrect behavior is shown in Figure 8. Line 8 of the program uses an uninitialized object, but this code is accepted by this specific implementation of the verifier. Basically, the program is able to allocate two different uninitialized objects on the same line of code without initializing either one, violating the *ConsistentInit* invariant. The program accomplishes this by allocating space for the first new object inside the subroutine and then storing the reference to that object in a local variable over which the subroutine is polymorphic before calling it again. After initializing only one of the objects, it can use either one.

The bug can be attributed to the verifier not placing any restrictions on the presence of uninitialized objects at calls to `jsr` L or `ret` x. The checks made by Sun's verifier are analogous to the (*jsr*) and (*ret*) rule in Figure 7 as they originally appeared in [SA98a], 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 demonstrates that even a fairly abstract model of the bytecode language is extremely useful at examining the complex relationships between different parts of the language, such as uninitialized objects and subroutines.

7.2 The Corrected Sun Verifier

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

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

- The static types assigned to uninitialized objects pass into constructors, i.e. any value whose type is of the form $\hat{\sigma}_0$ in our framework, are treated differently from other uninitialized objects types in the Sun verifier. Values with these types may still be used after being present at a call to or an exit from a subroutine. Also, the superclass constructor may be called anywhere, including inside a subroutine.

Treating the uninitialized object types for constructor arguments differently than other uninitialized types allows the verifier to accept programs where a subroutine must be called prior to invoking the super class constructor. Since the Java language specification requires the superclass constructor to be called prior to the start of any code protected by an exception handler, this flexibility is not required to correctly check valid Java programs, and it makes 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.

The differences in the two verification techniques would only become apparent in handwritten bytecodes using uninitialized object types in unusual ways. Since our method, while slightly more restrictive, makes both verification and our soundness proofs much simpler, we believe that our method is reasonable.

8 Related Work

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

In addition to the framework developed by Stata and Abadi [SA98a] and used in this paper, there are several different strategies being developed to describe the JVML type system and bytecode verification formally. The most closely related work is [Qia97], 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 8. After announcing our discovery of Sun's bug, a revised version of Qian's paper containing rules more similar to our rules was released.

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

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

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

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

9 Conclusions and Future Work

Given the need to guarantee type safety for mobile Java code, developing correct type checking and analysis techniques for JVML is crucial. However, there is no existing specification which fully captures how Java bytecodes must be type checked. We have built on the previous work of Stata and Abadi to develop such a specification by formulating a sound type system for a fairly complex subset of JVML which covers both subroutines and object initialization. This is one step towards developing a sound type system of the whole bytecode language. Once this type system for JVML is complete, we can describe a formal specification of the verifier and better understand what safety and security guarantees can be made by it.

Although our model is still rather abstract, it has already proved effective as a foundation for examining both JVML and existing bytecode verifiers. Even without a complete object model or notion of an object heap, we have been able to study initialization and the interaction between it and subroutines. In fact, a previously unpublished bug in Sun's verifier implementation was found as a result of the analysis performed while studying the soundness proofs for this paper.

While the study to date has examined the most complex areas of JVML, there are still several important issues to address in more detail, including issues of scale and adding a full object system to our model. The methods described in Section 6 allow most variants of simple instructions to be added in a standard, straightforward way, and we are also examining methods to factor JVML into a complete, yet minimal, set of instructions. In addition, the Java object system has been studied and discussed in other contexts [AG96, Sym97, DE97, Qia97], and these previous results can be used as a basis for objects in our JVML model. Currently, we are in the process of finishing a soundness proof for a language encompassing all of the issues presented in this paper plus objects, interfaces, classes, 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 intend to develop a synthesis method for generating an executable bytecode verifier from our type rules. This will provide a way to generate verifiers guaranteed to check our type rules correctly. In addition, our work may serve as a basis for adding new static checks to the verifier to examine both more complicated safety properties and help eliminate some of the currently necessary run-time checks. For example, we may eventually be able to eliminate some run-time checks for array bounds and pointer casts, or statically check that certain locking conventions are used in a concurrent JVML model.

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.

A JVML$_i$ Soundness

A.1 Useful Lemmas

This section will state and prove some lemmas used in the rest of the appendix. 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 then on the contents of a specific local variable.

Lemma 1

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

Proof Assume that all the hypotheses of the implication are satisfied for some F_i, S_i, f, s, $\hat{\tau}$, $\hat{b}$. Since *ConsistentInit*$(F_i, \hat{\tau} \cdot S_i, f, \hat{b} \cdot s)$, there is some $\hat{c}$ such that *Corresponds*$(F_i, \hat{\tau} \cdot S_i, f, \hat{b} \cdot s, \hat{c}, \hat{\tau})$. We proceed to show that $\hat{c} = \hat{b}$ by contradiction. Suppose $\hat{c} \neq \hat{b}$. *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*). However, this rule cannot be applied if $\hat{c} \neq \hat{b}$. Thus, we cannot conclude *StackCorresponds*$(\hat{\tau} \cdot S_i, \hat{b} \cdot s, \hat{c}, \hat{\tau})$, violating our assumption that $\hat{c}$ and $\hat{\tau}$ correspond, and our assumption must be incorrect. 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

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

Proof Assume that all the hypotheses of the implication are satisfied for some F_i, S_i, f, s, $\hat{b}$. 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 proceed to show that $\hat{c} = f[x]$ by contradiction. Suppose $\hat{c} \neq f[x]$. Then a contradiction exists because we could not have concluded that *Corresponds*$(F_i, S_i, f, s, \hat{c}, F_i[x])$ since there exists a local variable in the domain of F_i which has type $F_i[x]$ but not value $\hat{c}$. Therefore, our assumption must be wrong and $\hat{c} = f[x]$. □

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

Lemma 3

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

Proof Assume that the hypotheses of the implication are satisfied. Since we assumed *Corresponds*$(F_i, \tau \cdot S_i, f, v \cdot s, \hat{b}, \hat{\tau})$, we know that:

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

Also, *StackCorresponds*$(\tau \cdot S_i, v \cdot s, \hat{b}, \hat{\tau})$. If this is true, we must be able to conclude this 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, we may state the same notion for *ConsistentInit*.

Lemma 4

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

Proof Assume that the hypotheses of the implication are 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 and 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, has shown in the next to lemma.

Lemma 5

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

Proof The proof is by induction on the length of α_1. If $|\alpha_1| = 0$, then $\alpha_1 = \epsilon$ and $s_1 = \epsilon$, making $\alpha_1 \bullet \alpha_2 = \alpha_2$ and $s_1 \bullet s_2 = s_2$. Given these equalities, *ConsistentInit*(F_i, α_2, f, s_2) follows from the assumption *ConsistentInit*$(F_i, \alpha_1 \bullet \alpha_2, f, s_1 \bullet s_2)$.

For the inductive case, assume that the implication holds for any α_1 such that $|\alpha_1| = n$. Suppose $|\alpha_1| = n + 1$. In this case, $\alpha_1 = \tau \cdot \alpha_1'$, where $|\alpha_1'| = n$, and $\alpha_1 \bullet \alpha_2 = (\tau \cdot \alpha_1') \bullet \alpha_2 = \tau \cdot (\alpha_1' \bullet \alpha_2)$. Likewise, since $s_1 : \alpha_1$, $s_1 \bullet s_2 = v \cdot (s_1' \bullet s_2)$ for some $v : \tau$ and $s_1' : \alpha_1'$. The conclusion that *ConsistentInit*$(F_i, \alpha_1' \bullet \alpha_2, f, s_1' \bullet s_2)$ is reached by applying Lemma 4, and the inductive hypothesis then allows us to conclude *ConsistentInit*(F_i, α_2, f, s_2). □

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

Lemma 6

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

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

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

Also, *StackCorresponds*$(S_i, s, \hat{b}, \hat{\tau})$ must be true. Given that $\tau \neq \hat{\tau}$, *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

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

Proof Assume that the hypotheses of the implication are satisfied. For any $\hat{\tau} \in \hat{T}$, choose $\hat{b}$ such that *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). 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

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

Proof The proof is by induction on the length of α_1. Assume that the hypotheses of the implication are satisfied. If $|\alpha_1| = 0$, then $\alpha_1 = \epsilon$ and $s_1 = \epsilon$, making $\alpha_1 \bullet \alpha_2 = \alpha_2$ and $s_1 \bullet s_2 = s_2$. Given these equalities, *ConsistentInit*$(F_i, \alpha_1 \bullet \alpha_2, f, s_1 \bullet s_2)$ follows from the assumption *ConsistentInit*(F_i, α_2, f, s_2).

For the inductive case, assume that the implication holds for any α_1 such that $|\alpha_1| = n$. Suppose $|\alpha_1| = n + 1$. In this case, $\alpha_1 = \tau \cdot \alpha_1'$, where $|\alpha_1'| = n$, and $\alpha_1 \bullet \alpha_2 = (\tau \cdot \alpha_1') \bullet \alpha_2 = \tau \cdot (\alpha_1' \bullet \alpha_2)$. Likewise, since $s_1 : \alpha_1$, $s_1 \bullet s_2 = v \cdot (s_1' \bullet s_2)$ for some $v : \tau$ and $s_1' : \alpha_1'$. By the inductive hypothesis, *ConsistentInit*$(F_i, \alpha_1' \bullet \alpha_2, f, s_1' \bullet s_2)$. Since $\tau \notin \hat{T}$ is guaranteed by the assumption about α_1, Lemma 7 can be applied to conclude *ConsistentInit*$(F_i, \tau \cdot (\alpha_1' \bullet \alpha_2), f, v \cdot (s_1' \bullet s_2))$, and this is the same as *ConsistentInit*$(F_i, \alpha_1 \bullet \alpha_2, f, s_1 \bullet s_2)$. □

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

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

Proof Assume that the hypotheses of the implication 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 that $\forall y \in \mathit{Dom}(F_i[x \mapsto \hat{\tau}])$,

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

Before proving this, note that $\mathit{Dom}(F_i) = \mathit{Dom}(F_i[x \mapsto \hat{\tau}])$. There are two cases to consider for each y:

- $x \neq y$: 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, equation (3) must be true for this choice of y.
- $x = y$: In this case, $(F_i[x \mapsto \hat{\tau}])[y] = \hat{\tau}$ and $(f[x \mapsto \hat{b}])[y] = \hat{b}$. Thus, (3) is satisfied 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 between that type and some other value.

Lemma 10

$$\begin{array}{l} \forall F_i, S_i, f, s, v, \tau, \hat{b}, \hat{\tau}, x. \\ \quad x \in \mathit{Dom}(F_i) \\ \quad \wedge\ \hat{b} : \hat{\tau} \\ \quad \wedge\ \tau \neq \hat{\tau} \\ \quad \wedge\ \mathit{Corresponds}(F_i, S_i, f, s, \hat{b}, \hat{\tau}) \\ \Rightarrow \mathit{Corresponds}(F_i[x \mapsto \tau], S_i, f[x \mapsto v], s, \hat{b}, \hat{\tau}) \end{array}$$

Proof Assume that the hypotheses of the implication 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 that for all $y \in \mathit{Dom}(F_i[x \mapsto \tau])$,

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

Before proving this, note that $\mathit{Dom}(F_i) = \mathit{Dom}(F_i[x \mapsto \tau])$. There are two cases to consider for each y:

- $x \neq y$: 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, equation (4) must be true for this choice of y.
- $x = y$: 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 concern substitutions. 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

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

Proof Assume that all the hypotheses of the implication are satisfied. We prove the conclusions by induction on the proof 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 proof:

- (*sc 0*): If this is the case, $s = \epsilon$ and $S_i = \epsilon$. The conclusions follow trivially.
- (*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 also know that $s' : S'_i$ and *StackCorresponds*$(S'_i, s', \hat{a}, \hat{\sigma})$ must be true, allowing us to conclude that

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

  and

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

  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 equation (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 half 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 equations (7) and (8) above, this 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.
- (*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:

- $v \neq \hat{a}$: 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. This means that $[a/\hat{a}](v\cdot s') : [\sigma/\hat{\sigma}](\hat{\sigma}\cdot S_i)$ is true. 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 half of the conclusion follows.
- $v = \hat{a}$: 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 $[\sigma/\hat{\sigma}]\text{TOP} = \text{TOP}$, we may conclude that $[a/\hat{a}]v : [\sigma/\hat{\sigma}]\tau$ is true. The assertion that $[a/\hat{a}]s : [\sigma/\hat{\sigma}]S_i$ 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.

Lemma 12

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

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

- $F_i[y] = \hat{\sigma}$: 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.

- $F_i[y] \neq \hat{\sigma}$: First, we know that $[\sigma/\hat{\sigma}](F_i[y]) = F_i[y]$. In this case, there are two possibilities for $f[y]$:
  - $f[y] \neq \hat{a}$: In this case, $[a/\hat{a}](f[y]) = f[y]$, and given that $f[y] : F_i[y]$ and $F_i[y] \neq \hat{\sigma}$, the conclusions are satisfied.
  - $f[y] = \hat{a}$: In this case, $F_i[y] = \text{TOP}$ since the only valid types for $\hat{a}$ are $\hat{\sigma}$ and TOP. Since $a : \text{TOP}$ is also true, $[a/\hat{a}](f[y]) : \text{TOP}$, making the first clause of the conclusion true. Since $\hat{\sigma} \neq \text{TOP}$, 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

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

Proof We show that *StackCorresponds*$([\sigma/\hat{\sigma}]S_i, [a/\hat{a}]s, \hat{b}, \hat{\tau})$ is true by induction on the proof of *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:

- (*sc 0*): If this is the case, $s = \epsilon$ and $S_i = \epsilon$, and the conclusion follows easily.
- (*sc 1*): Assume $S_i = \hat{\tau} \cdot S_i'$ and $s = \hat{b} \cdot s'$ where *StackCorresponds*$([\sigma/\hat{\sigma}]S_i', [a/\hat{a}]s', \hat{b}, \hat{\tau})$ is true by the inductive hypothesis. *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 *StackCorresponds*$([\sigma/\hat{\sigma}]S_i, [a/\hat{a}]s, \hat{b}, \hat{\tau})$.
- (*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, *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 $\sigma \neq \tau$ and $\hat{\tau} \neq \tau$. Therefore, by rule (*sc 2*), *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 *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 required 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 easily reason about which instructions preserve the global invariants in the common case. For example, the following property describes behavior easily proved to guarantee that the stack is well typed after the instruction is executed.

Property 1 *For some instruction* I*, we state that* I **preserves** *StackType according to the following relation:*

I **preserves** *StackType if*

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

The following instructions have this property: `inc`, `pop`, `push 0`, `store` 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:

- `inc`: By the operational semantics, $pc' = pc + 1$, $s = n \cdot s''$, and $s' = (n+1) \cdot s''$ for some s''. By (*inc*), $S_{pc} = S_{pc'} = \textsc{Int} \cdot \alpha$ for some α. Choose $s_1 = n$, $s_2 = s''$, $s_3 = n+1$, and $\alpha_1 = \textsc{Int}$, $\alpha_2 = \alpha$, $\alpha_3 = \textsc{Int}$. 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 $n \cdot s'' : \textsc{Int} \cdot \alpha$ and $s'' : \alpha$, meaning $s_2 : \alpha_2$.

- `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*), $S_{pc} = \textsc{Int} \cdot S_{pc'}$. Choose $s_1 = n$, $s_2 = s''$, $s_3 = \epsilon$, $\alpha_1 = \textsc{Int}$, $\alpha_2 = S_{pc'}$, and $\alpha_3 = \epsilon$. We know that $s_1 : \alpha_1$ since $n : \textsc{Int}$. By the assumption that $s : S_{pc}$, we conclude that $n \cdot s'' : \textsc{Int} \cdot S_{pc'}$ and $s'' : S_{pc'}$, meaning that $s_2 : \alpha_2$. The type judgment $\epsilon : \epsilon$ implies that $s_3 : \alpha_3$.

$\square$

With this property, we may now prove part of Theorem 1, grouping all instructions exhibiting Property 1 into a single case.

Lemma 14 *Given P, F, and S such that $F, S \vdash P$:*

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

Proof Assume that the hypotheses of the implication 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:

- P[pc] **preserves** *StackType*: By Property 1, 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, $s' : S_{pc'}$.

- $P[pc] =$ `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 $\forall y \in \textsc{Var}.\ f[y] : F_{pc}[y]$, we know that $f[x] : F_{pc}[x]$. In addition, since $s : S_{pc}$, $f[x] \cdot s : F_{pc}[x] \cdot S_{pc}$ must be true, allowing us to conclude that $s' : S_{pc'}$.

- $P[pc] =$ `init` σ: 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 $\mathit{ConsistentInit}(F_{pc}, F_{pc}, f, s)$ imply that $\mathit{Corresponds}(F_{pc}, F_{pc}, f, s, \hat{a}, \hat{\sigma}_j)$ by Lemma 1. This means that $\mathit{StackCorresponds}(S_{pc}, s, \hat{a}, \hat{\sigma}_j)$, and in order to have proved this,

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

  must be true. Also, σ and $\hat{\sigma}_j$ must be different types since the first is an initialized object type and the second is an uninitialized object type. Also, $s'' : \alpha$, $a : \sigma$ and $\hat{a} : \hat{\sigma}_j$. Thus, we may apply Lemma 11 to (9) to conclude $s' : S_{pc'}$.

□

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

Property 2 *For some instruction* I*, we state that* I **preserves** *VariableType according to the following relation:*

$$\begin{array}{l}
\text{I } \textbf{preserves } \textit{VariableType if} \\
\quad \forall P, F, S, pc, f, s, pc', f', s'. \\
\qquad F, S \vdash P \\
\qquad \wedge\ P \vdash \langle pc,\ f,\ s\rangle \to \langle pc',\ f',\ s'\rangle \\
\qquad \wedge\ P[pc] = \text{I} \\
\quad \Rightarrow f' = f \\
\qquad \wedge\ F_{pc'} = F_{pc}
\end{array}$$

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:

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

□

Lemma 15 *Given P, F, and S such that $F, S \vdash P$:*

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

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

- P[pc] **preserves** *VariableType*: By Property 2, $f' = f$ and $F_{pc'} = F_{pc}$, and we assumed $\forall y \in \textsc{Var}.\ f[y] : F_{pc}[y]$. By substitution using the two equalities, $\forall y \in \textsc{Var}.\ f'[y] : F_{pc'}[y]$.
- $P[pc] =$ `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 $y \in \textsc{Var}$:
  - $y \neq x$: 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.
  - $y \neq x$: $f'[x] = v$ and $F_{pc'}[x] = \tau$. Since $s : S_{pc}$, we know that $v : \tau$. Thus, $f'[x] : F_{pc'}[x]$.

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

- $P[pc]$ = `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, $\mathit{Corresponds}(F_{pc}, S_{pc}, f, s, \hat{a}, \hat{\sigma}_j)$ follows from Lemma 1, meaning that $F_{pc}[y] = \hat{\sigma}_j$ implies $f[y] = \hat{a}$ for all $y \in \textsc{Var}$. Using these facts, Lemma 12 may be applied to conclude $([a/\hat{a}]f)[y] : ([\sigma/\hat{\sigma}_j]F[pc])[y]$ for all $y \in \textsc{Var}$, and the conclusion is satisfied.

$\square$

Property 3 is more complex than the previous properties. The behavior 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.

Property 3 *For some instruction* I*, we state that* I **preserves** *ConsistentInit according to the following relation:*

I **preserves** *ConsistentInit if*

$$\begin{array}{l}
\forall P, F, S, pc, f, s, pc', f', s'. \\
\quad F, S \vdash P \\
\quad \wedge\ P \vdash \langle pc,\ f,\ s\rangle \to \langle pc',\ f',\ s'\rangle \\
\quad \wedge\ s : S_{pc} \\
\quad \wedge\ \forall y \in \textsc{Var}.\ f[y] : F_{pc}[y] \\
\quad \wedge\ \mathit{ConsistentInit}(F_{pc}, S_{pc}, f, s) \\
\quad \wedge\ P[pc] = \mathrm{I} \\
\Rightarrow \exists s_1 : \alpha_1, s_2 : \alpha_2, s_3 : \alpha_3. \\
\quad s = s_1 \bullet s_2 \\
\quad \wedge\ s' = s_3 \bullet s_2 \\
\quad \wedge\ S_{pc} = \alpha_1 \bullet \alpha_2 \\
\quad \wedge\ S_{pc'} = \alpha_3 \bullet \alpha_2 \\
\quad \wedge\ \forall y \in \mathit{Dom}(\alpha_3).\ \alpha_3[y] \notin \hat{T} \\
\quad \wedge\ f' = f \\
\quad \wedge\ F_{pc'} = F_{pc}
\end{array}$$

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:

- `inc`: By the operational semantics, we know that $pc' = pc + 1$, $f = f'$, $s = n \cdot s''$ for some s'', and $s' = (n + 1) \cdot s''$. By (*inc*), $S_{pc} = S_{pc'} = \textsc{Int} \cdot \alpha$ for some α and $F_{pc} = F_{pc'}$. Choose $s_1 = n$, $s_2 = s''$, $s_3 = n+1$, and $\alpha_1 = \textsc{Int}$, $\alpha_2 = a$, $\alpha_3 = \textsc{Int}$. 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 $n \cdot s'' : \textsc{Int} \cdot \alpha$ and $s'' : \alpha$, meaning $s_2 : \alpha_2$. Finally, $\textsc{Int} \notin \hat{T}$, implying that $\forall y \in \mathit{Dom}(\alpha_3).\ \alpha_3[y] \notin \hat{T}$.

- `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 $S_{pc} = \textsc{Int} \cdot S_{pc'}$ and $F_{pc} = F_{pc'}$. Choose

$s_1 = n$, $s_2 = s''$, $s_3 = \epsilon$, $\alpha_1 = \textsc{Int}$, $\alpha_2 = S_{pc'}$, and $\alpha_3 = \epsilon$. $s_1 : \alpha_1$ since $n : \textsc{Int}$. By the assumption that $s : S_{pc}$, we conclude that $n \cdot s'' : \textsc{Int} \cdot S_{pc'}$ 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.

□

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

Lemma 16 *Given P, F, and S such that $F, S \vdash P$:*

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

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 Property 3. We know that $P[pc]$ is not a `halt` instruction.

- P[pc] **preserves** *ConsistentInit*: By Property 3, we may choose $s_1 : \alpha_1$, $s_2 : \alpha_2$, and $s_3 : \alpha_3$ such that all the conditions listed in Property 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.

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

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

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

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

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

    Thus, we know the following is true:

    $$\forall y \in \mathit{Dom}(F_{pc'}).\ F_{pc'}[y] = \hat{\sigma}_j \implies f'[y] = \hat{a} \tag{11}$$

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

- $\hat{\tau} \neq \hat{\sigma}_j$: 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 $\hat{b}$. From this, we know that the following two equations are true:

$$\forall y \in Dom(F_{pc}).\ F_{pc}[y] = \hat{\tau} \implies f[y] = \hat{b} \tag{12}$$

$$StackCorresponds(S_{pc}, s, \hat{b}, \hat{\tau}) \tag{13}$$

Since $F_{pc'} = F_{pc}$ and $f' = f$,

$$\forall y \in Dom(F_{pc'}).\ F_{pc'}[y] = \hat{\tau} \implies f'[y] = \hat{b}$$

is also true. Given that $\hat{\sigma}_j \neq \hat{\tau}$,

$$StackCorresponds(\hat{\sigma}_j \cdot S_{pc}, \hat{a} \cdot s, \hat{b}, \hat{\tau})$$

follows by rule (*sc 2*) applied to (13), and *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$ follows by (*corr*).

- $P[pc] =$ `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 true cases for each $\hat{\tau}$:

  - $\hat{\tau} = \hat{\sigma}_j$: Choose $\hat{b} = \hat{a}$. First, note that $\sigma \neq \hat{\sigma}_j$ is true since σ is an initialized object type and $\hat{\sigma}_j$ is an uninitialized object type. By Lemma 11 and Lemma 12,

  $$\forall y \in Dom([\sigma/\hat{\sigma}_j]F_{pc}).\ ([\sigma/\hat{\sigma}_j]F_{pc})[y] = \hat{\sigma}_j \implies ([a/\hat{a}]f)[y] = \hat{a}$$

  $$StackCorresponds([\sigma/\hat{\sigma}_j]S_{pc}, [a/\hat{a}]s, \hat{a}, \hat{\sigma}_j)$$

  are true. 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*).

  - $\hat{\tau} \neq \hat{\sigma}_j$: 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,

  $$\forall y \in Dom([\sigma/\hat{\sigma}_j]F_{pc}).\ ([\sigma/\hat{\sigma}_j]F_{pc})[y] = \hat{\tau} \implies ([a/\hat{a}]f)[y] = \hat{b} \tag{14}$$

  must be true. Suppose it were not. There would exist a $y \in Dom([\sigma/\hat{\sigma}_j]F_{pc})$ such that $([\sigma/\hat{\sigma}_j]F_{pc})[y] = \hat{\tau}$ and $([a/\hat{a}]f)[y] \neq \hat{b}$. However, in this case, $([\sigma/\hat{\sigma}_j]F_{pc})[y] = F_{pc}[y]$ since $\hat{\tau} \neq \hat{\sigma}_j$. Also, $f[y]$ does not equal $\hat{b}$ because if it did, then $([a/\hat{a}]f)[y]$ would be $\hat{b}$, violating the statement that $([a/\hat{a}]f)[y] \neq \hat{b}$. Therefore $f[y] \neq \hat{b}$ and $F_{pc}[y] = \hat{\tau}$. However, since y is also in $Dom(F_{pc})$, this contradicts the assumption that *Corresponds*$(F_{pc}, S_{pc}, f, s, \hat{b}, \hat{\tau})$. Therefore, no such y exists and (14) must hold.

We also know that

$$\mathit{StackCorresponds}([\sigma/\hat{\sigma}_j]\alpha, [a/\hat{a}]s'', \hat{b}, \hat{\tau}) \tag{15}$$

is true by Lemma 13. Therefore, $\mathit{Corresponds}(F_{pc'}, S_{pc'}, f', s', \hat{a}, \hat{\sigma}_j)$ is true by rule (*corr*) applied to (14) and (15).

- $P[pc] = \texttt{store}\ x$: 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 $\hat{\tau}$:
  - $\tau \neq \hat{\tau}$: 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 $x \in \mathit{Dom}(F_{pc})$, 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

    $$\mathit{Corresponds}(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$$
  - $\tau = \hat{\tau}$: 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.
- $P[pc] = \texttt{load}\ x$: By the operational semantics, $pc' = pc + 1$, $s' = f[x] \cdot s$, and $f' = f$. By rule (*store*), $S_{pc'} = F_{pc}[x] \cdot S_{pc}$ 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 $\hat{\tau}$:
  - $F_{pc}[x] \neq \hat{\tau}$: Choose $\hat{b}$ such that *Corresponds*$(F_{pc}, S_{pc}, f, s, \hat{b}, \hat{\tau})$. From this, we know that *StackCorresponds*$(S_{pc}, s, \hat{b}, \hat{\tau})$. Therefore, *StackCorresponds*$(F_{pc}[x] \cdot S_{pc}, f[x] \cdot s, \hat{b}, \hat{\tau})$ follows directly from rule (*sc 2*), and *Corresponds*$(F_{pc'}, S_{pc'}, f', s', \hat{b}, \hat{\tau})$ is true.
  - $F_{pc}[x] = \hat{\tau}$: 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 *StackCorresponds*$(S_{pc}, s, f[x], F_{pc}[x])$ 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 property is trivial for JVML$_i$, we include both for symmetry with the previous three invariants and also because it will be useful in the proofs of extensions of JVML$_i$.

Property 4 *For some instruction* I*, we state that* I **preserves** *ProgramDomain according to the following relation:*

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

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:

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

$\square$

Lemma 17 *Given P, F, and S such that $F, S \vdash P$:*

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

Proof We are guaranteed that P[pc] **preserves** *ProgramDomain* since all instructions in JVML$_i$ exhibit is property. Thus, this lemma follows directly from Property 4. $\square$

We may now prove Theorem 1:

Restatement of Theorem 1 *Given P, F, and S such that $F, S \vdash P$:*

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

Proof This theorem follows directly from Lemmas 14,15,16, and 17. $\square$

A.3 Progress

The proofs in this section 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 $F, S \vdash P$:*

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

Proof Assume that all the hypotheses of the implication 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.

- $P[pc] = \texttt{push 0}$: Choose $s' = 0 \cdot s$, $f' = f$, and $pc' = pc + 1$.
- $P[pc] = \texttt{inc}$: Since we assumed $s : S_{pc}$, and $S_{pc} = \textsc{Int} \cdot S_{pc+1}$ follows from (*inc*), $s = n \cdot s''$ for some n and s''. Therefore, choosing $s' = (n+1) \cdot s''$, $f' = f$, and $pc' = pc + 1$ will allow progress to be made.
- $P[pc] = \texttt{pop}$: Since $s : S_{pc}$, and $S_{pc} = \tau \cdot \alpha$ follows from (*pop*), $s = v \cdot s''$ for some v and s''. Therefore, choose $s' = s''$, $f' = f$, and $pc' = pc + 1$.
- $P[pc] = \texttt{if}\ L$: Since we assumed $s : S_{pc}$ and $S_{pc} = \textsc{Int} \cdot S_{pc+1} = S_L$ follows from (*if*), $s = n \cdot s''$ 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.
- $P[pc] = \texttt{store}\ x$: 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 $pc' = pc + 1$.
- $P[pc] = \texttt{load}\ x$: Choose $s' = f[x] \cdot s$, $f' = f$, and $pc' = pc + 1$.
- $P[pc] = \texttt{new}\ \sigma$: $A^{\hat{\sigma}_{pc}}$ contains an infinite number values, meaning that there is at least one value $\hat{a}$ such that $\mathit{Unused}(\hat{a}, f, s)$. Choose $s' = \hat{a} \cdot s$, $f' = f$, and $pc' = pc + 1$.
- $P[pc] = \texttt{init}\ \sigma$: Since $s : S_{pc}$, and $S_{pc} = \hat{\sigma}_j \cdot S_{pc+1}$ for some j from rule (*init*), $s = \hat{a} \cdot s''$ for some $\hat{a} \in A^{\hat{\sigma}_j}$ and s''. Also, A^{σ} contains an infinite number values, meaning that there is at least one value a such that $\mathit{Unused}(a, f, s)$. Therefore, choose $s' = [a/\hat{a}]s''$, $f' = [a/\hat{a}]f$, and $pc' = pc + 1$.
- $P[pc] = \texttt{use}\ \sigma$: Since $s : S_{pc}$ and $S_{pc} = \sigma \cdot S_{pc+1}$ follows from (*use*), $s = a \cdot s''$ 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 18 *Given P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f_0, f, s. \\ \qquad P \vdash \langle 1,\, f_0,\, \epsilon\rangle \rightarrow^* \langle pc,\, f,\, s\rangle \\ \quad \Rightarrow s : S_{pc} \\ \qquad \wedge\ \forall y \in \textsc{Var}.\ f[y] : F_{pc}[y] \\ \qquad \wedge\ \mathit{ConsistentInit}(F_{pc}, S_{pc}, f, s) \\ \qquad \wedge\ pc \in \mathit{Dom}(P) \end{array}$$

Proof The proof of this 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 follows 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

$$P \vdash \langle 1,\, f_0,\, \epsilon\rangle \rightarrow^* \langle pc',\, f',\, s'\rangle \rightarrow \langle pc,\, f,\, s\rangle$$

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

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

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

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

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

Restatement of Theorem 3 *Given P, F, and S such that $F, S \vdash P$:*

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

Proof Assume that all the hypotheses of the implication are satisfied. We first prove the first clause of the conclusion. Suppose $P \vdash \langle 1,\, f_0,\, \epsilon\rangle \rightarrow^* \langle pc,\, f,\, s\rangle$ and $P[pc] \neq \texttt{halt}$ but no further step can be taken by the program. By Lemma 18, the following are true:

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

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 a 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 18 to the assumptions of the implication. $\square$

B Soundness of Extensions

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 equivalents of z in the run-time state and Z_P:

Theorem 5 (Constructor One-step Soundness) *Given P, F, and S such that $F, S \vdash P$:*

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

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:

Property 5 *For some instruction* I*, we state that* I **preserves** *Constructor according to the following relation:*

I **preserves** *Constructor if*

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

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

$$z' = Z_{P,pc'} \tag{16}$$

can be proved by case analysis on $P[pc]$. If P$[pc]$ **preserves** *Constructor* then (16) follows from Property 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, equation (16) holds for all possible instructions, and Theorem 5 is true. $\Box$

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 $\hat{a}_\varphi : \hat{\varphi}_0$:*

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

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 $P[pc] = \texttt{halt}$. $\Box$

B.2 JVML$_s$

This section outlines the soundness proof for JVML$_i$ with subroutines. We refer the reader to the extended version of [SA98b] 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 Section 5 of [SA98a], 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 6. This part uses a simulation between the structured and stackless semantics. Once this is complete, 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 9 shows the structured operational semantics for JVML$_s$. The machine state has a fourth component, ρ, representing the subroutine call 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 and Figure 7. The judgment to conclude that a program is well typed is:

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

The map R is described in Figure 10 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 a function that assigns types to local variables taking into account the subroutine call

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

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

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

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

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

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

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

Figure 9: JVML$_s$ structured operational semantics.

stack. The type $\mathcal{F}(F, pc, \rho)[x]$ is defined by the rules:

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

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

As we will see below, as long as ρ satisfies some well-formedness conditions, $\mathcal{F}(F, pc, \rho)$ will be defined on all local variables. This definition matches the definition given in [SA98a]. We also need the *WFCallStack* judgment, which ensures that a subroutine call stack is well-formed:

(*wf 0*)
$$\frac{\mathit{Dom}(F_{pc}) = \textsc{Var}}{\mathit{WFCallStack}(P, F, pc, \epsilon)}$$

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

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

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

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

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

Figure 10: Rules labeling instructions with subroutines

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

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

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 $F, S \vdash_s P$:*

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

Proof Sketch The full details of this proof are beyond the scope of this paper, and only a sketch of the actual proof is presented. We proceed by briefly justifying each of the five invariants listed in the theorem:

- $s' : S_{pc'}$: This follows the proofs similar to Lemma 14, where that lemma and Property 1 are augmented with ρ.

- $pc' \in Dom(P)$: This follows from a proof similar to Lemma 17 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 [SA98b] 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:

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

  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:

  - $y \in Dom(F_{pc})$: 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 $\tau' = F_{pc'}[y]$.
  - $y \notin Dom(F_{pc})$: In this case, $\mathcal{F}(F, pc', \rho')[y] = \mathcal{F}(F, pc, \rho)[y]$. Also, from our assumption that $ConsistentInitWithSub(P, F, S, pc, f, s, p)$, 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 $a : \text{TOP}$, this case still holds. Thus, choose τ' such that $\mathcal{F}(F, pc, \rho)[y] = \tau'$.

- $ConsistentInitWithSub(P, F, S, pc', f', s', p')$: If we know that $\rho' = \rho$ and $Dom(F_{pc'}) = Dom(F_{pc})$, both clauses may be proved to be true 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 that instruction. This takes care of all cases except `jsr` L and `ret` x.

  For `jsr` L, we know that for all $y \in \text{VAR} \setminus Dom(F_{pc})$, $\mathcal{F}(F, pc, \rho)[y]$ is not in $\hat{T}$ by *ConsistentInitWithSub*(P, F, S, pc, f, s, p). Also, we know that that $\forall y \in Dom(F_{pc})$, $F_{pc}[y] \notin \hat{T}$ by (*jsr*). That rule also guarantees that:

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

Since there are no uninitialized object types present in F_L and S_L, it is clear that *Consistent-Init*$(F_{pc'}, S_{pc'}, f', s')$ is true. Also, we know that $\mathcal{F}(F, L, (pc+1)\cdot\rho)[y]$ will be the same as $\mathcal{F}(F, pc+1, \rho)[y]$ for all $y \in \text{VAR} \setminus \mathit{Dom}(F_L)$ from (*tt 1*), and from the statements above, we know that $\mathcal{F}(F, pc+1, \rho)[y]$ is not a type in $\hat{T}$, making the second half of *ConsistentInitWith-Sub*$(P, F, S, pc', f', s', p')$ 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 $\mathit{Dom}(F_{pc'})$ but not in $\mathit{Dom}(F_{pc})$ 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, $\forall y \in \mathit{Dom}(F_{pc'})$. $F_{pc'}[y] \notin \hat{T}$ 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*$(P, F, S, pc', f', s', p')$ 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 Property 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.

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

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

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

To show that the soundness theorems proved for JVML$_i$ also apply to JVML$_i$ with `iadd`, its suffices to prove that the four properties from Appendix A apply to `iadd`, and also 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 Property 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 $\text{INT} \notin \hat{T}$ 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 $P[pc] =$ `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 $s' = (n_1 + n_2) \cdot s''$.

References

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

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

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

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

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

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

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

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

[Qia97] Zhenyu Qian. A Formal Specification of Java(tm) Virtual Machine Instructions (draft). available from http://www.informatik.uni-bremen.de/~qian/abs-fsjvm.html, November 1997.

[SA98a] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.

[SA98b] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines (extended version). submitted for publication, January 1998.

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

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

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

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