# A Type System for Java Bytecode Subroutines

Raymie Stata and Martín Abadi
Digital Equipment Corporation
Systems Research Center

## Abstract

Java is typically compiled into an intermediate language, JVML, that is interpreted by the Java Virtual Machine. Because mobile JVML code is not always trusted, a bytecode verifier enforces static constraints that prevent various dynamic errors. Given the importance of the bytecode verifier for security, its current descriptions are inadequate. This paper proposes using typing rules to describe the bytecode verifier because they are more precise than prose, clearer than code, and easier to reason about than either.

JVML has a subroutine construct used for the compilation of Java's **try-finally** statement. Subroutines are a major source of complexity for the bytecode verifier because they are not obviously last-in/first-out and because they require a kind of polymorphism. Focusing on subroutines, we isolate an interesting, small subset of JVML. We give typing rules for this subset and prove their correctness. Our type system constitutes a sound basis for bytecode verification and a rational reconstruction of a delicate part of Sun's bytecode verifier.

## 1 Bytecode verification and typing rules

The Java language is typically compiled into an intermediate language that is interpreted by the Java Virtual Machine (VM) [LY96]. This intermediate language, which we call JVML, is an object-oriented language similar to Java. Its features include packages, classes with single inheritance, and interfaces with multiple inheritance. However, unlike method bodies in Java, method bodies in JVML are sequences of bytecode instructions. These instructions are fairly high-level but, compared to the structured statements used in Java, they are more compact and easier to interpret.

JVML code is often shipped across networks to Java VMs embedded in web browsers and other applications. Mobile JVML code is not always trusted by the VM that receives it. Therefore, a bytecode verifier enforces static constraints on mobile JVML code. These constraints rule out type errors (such as dereferencing an integer), access control violations (such as accessing a private method from outside its class), object initialization failures (such as accessing a newly allocated object before its constructor has been called), and other dynamic errors.

Figure 1 illustrates how bytecode verification fits into the larger picture of Java security. The figure represents trusted and untrusted code. At the base of the trusted code is the Java VM itself—including the bytecode verifier—plus the operating system, which provides access to privileged resources. On top of this base layer is the Java library, which provides controlled access to those privileged resources. Java security depends on the VM correctly interpreting JVML code, which in turn depends on the verifier rejecting illegal JVML code. If the verifier were broken but the rest of the VM assumed it was correct, then JVML code could behave in ways not anticipated by the Java library, circumventing the library's access controls.

Given its importance for security, current descriptions of the verifier are deficient. The official specification of the verifier is a 20-page, prose description. Although good by the standards of prose, this description is ambiguous, imprecise, and hard to reason about. In addition to this specification, Sun distributes what could be considered a reference implementation of the verifier. As a description, this implementation is precise, but it is hard to understand and, like the prose description, is hard to reason about. Furthermore, the implementation disagrees with the prose description.

This paper proposes using typing rules to describe the verifier. Typing rules are more precise than prose, easier to understand than code, and they can be manipulated formally. Such rules would give implementors of the verifier a systematic framework on which to base their code, increasing confidence in its correctness. Such rules would also give implementors of the rest of the VM an unambiguous statement of what they can and cannot assume about legal JVML code.

From a typing perspective, JVML is interesting in at least two respects:

- In JVML, a location can hold different types of values at different program points. This flexibility allows locations to be reused aggressively, allowing interpreters to save space. Thus, JVML contrasts with most typed languages, in which a location has only one type throughout its scope.
- JVML has *subroutines* to help compilers generate compact code for Java **try-finally** statements. JVML subroutines are subsequences of the larger sequence of bytecode instructions that make up a method's body.

To appear in the Proceedings of the 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998.

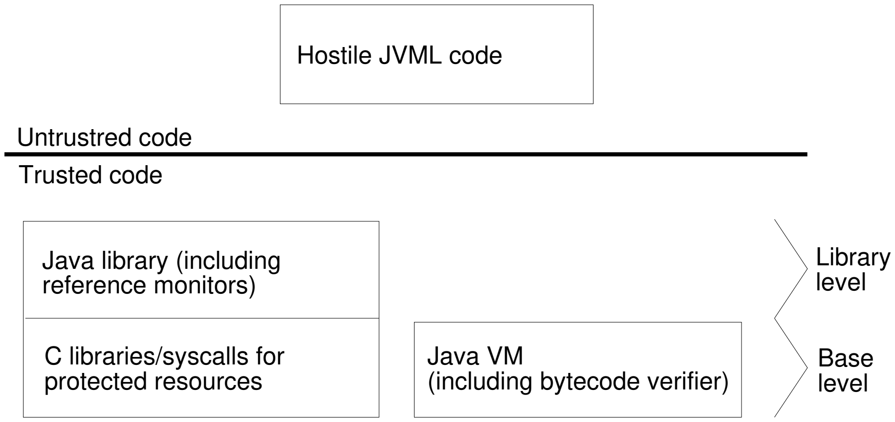


Figure 1: The Java VM and security

The JVML instruction `jsr` jumps to the start of one of these subsequences, and the JVML instruction `ret` returns from one. Subroutines introduce two significant challenges to the design of the bytecode verifier: ensuring that `ret` is used in a well-structured manner, and supporting a certain kind of polymorphism. We describe these challenges in more detail in Section 2.

This paper addresses these typing problems. It defines a small subset of JVML, called JVML0, presents a type system and a dynamic semantics for JVML0, and proves the soundness of the type system with respect to the dynamic semantics. JVML0 includes only 9 instructions, and ignores objects and several other features of JVML. This restricted scope allows us to focus on the challenges introduced by subroutines. Thus, our type system provides a precise, sound approach to bytecode verification, and a rational reconstruction of a delicate part of Sun's bytecode verifier.

We present the type system in three stages:

1. The first stage is a simplified version for a subset of JVML0 that excludes `jsr` and `ret`. This simplified version provides an introduction to our notation and approach, and illustrates how we give different types to locations at different program points.

2. The next stage considers all of JVML0 but uses a structured semantics for `jsr` and `ret`. This structured semantics has an explicit subroutine call stack for ensuring that subroutines are called on a last-in, first-out basis. In the context of this structured semantics, we show how to achieve the polymorphism desired for subroutines.

3. The last stage uses a stackless semantics for `jsr` and `ret` in which return addresses are stored in random-access memory. The stackless semantics is closer to Sun's. It admits more efficient implementations, but it does not dynamically enforce a last-in, first-out discipline on calls to subroutines. Because such a discipline is needed for type safety, we show how to enforce it statically.

The next section describes JVML subroutines in more detail and summarizes our type system. Section 3 gives the syntax and an informal semantics for JVML0. Sections 4–6 present our type system in the three stages outlined above. Section 7 states the main soundness theorem for our type system. Sections 8 and 9 discuss related work, including Sun's bytecode verifier. Section 8 also considers how our type system could be extended to the full JVML. Section 10 concludes. Most formal claims and all proofs are omitted in this summary.

## 2 Overview of JVML subroutines and our type system

JVML subroutines are subsequences of a method's larger sequence of instructions; they behave like miniature procedures within a method body. Subroutines are used for compiling Java's **try**-**finally** statement.

Consider, for example, the Java method named `bar` at the top of Figure 2. The **try** body can terminate in three ways: immediately when `i` does not equal 3, with an exception raised by the call to `foo`, or with an execution of the **return** statement. In all cases, the compiler must guarantee that the code in the **finally** block is executed when the **try** body terminates. Instead of replicating the **finally** code at each escape point, the compiler can put the **finally** code in a JVML subroutine. Compiled code for an escape from a **try** body executes a `jsr` to the subroutine containing the **finally** code.

Figure 2 illustrates the use of subroutines for compiling **try**-**finally** statements. It contains a possible result of compiling the method `bar` into JVML, putting the **finally** code in a subroutine in lines 13–16.

Figure 2 also introduces some of JVML's runtime structures. JVML bytecode instructions read and write three memory regions. The first region is an object heap shared by all method activations; the heap does not play a part in the example code of Figure 2. The other two regions are private to each activation of a method. The first of these regions is the *operand stack*, which is intended to be used on a short-term basis in the evaluation of expressions. For example, the instruction `iconst_3` pushes the integer constant 3 onto this stack, while `ireturn` terminates the current method returning the integer at the top of the stack. The second region is a set of locations known as *local variables*, which are intended to be used on a longer-term basis to hold values across expressions and statements (but not across method activations). Local variables are not operated on directly.

```
int bar(int i) {
    try {
        if (i == 3) return this.foo();
    } finally {
        this.ladida();
    }
    return i;
}
```

```
01 iload_1               // Push i
02 iconst_3              // Push 3
03 if_icmpne 10          // Goto 10 if i does not equal 3
   // Then case of if statement
04 aload_0               // Push this
05 invokevirtual foo     // Call this.foo
06 istore_2              // Save result of this.foo()
07 jsr 13                // Do finally block before returning
08 iload_2               // Recall result from this.foo()
09 ireturn               // Return result of this.foo()
   // Else case of if statement
10 jsr 13                // Do finally block before leaving try
   // Return statement following try statement
11 iload_1               // Push i
12 ireturn               // Return i
   // finally block
13 astore_3              // Save return address in local variable 3
14 aload_0               // Push this
15 invokevirtual ladida  // Call this.ladida()
16 ret 3                 // Return to address saved on line 13
   // Exception handler for try body
17 astore_2              // Save exception
18 jsr 13                // this.foo raised exception:  do finally block
19 aload_2               // Recall exception
20 athrow                // Rethrow exception
   // Exception handler for finally body
21 athrow                // Rethrow exception
```

Exception table (maps regions of code to their exception handlers):

| Region | Target |
|---|---|
| 1–12 | 17 |
| 13–16 | 21 |

Figure 2: Example compilation of **try-finally** into JVML

Rather, values in local variables are pushed onto the stack and values from the stack are popped into local variables via the `load` and `store` instructions respectively. For example, the instruction `aload_0` pushes the object reference in local variable `0` onto the operand stack, while the instruction `istore_2` pops the top value off the operand stack and saves it in local variable `2`.

Subroutines pose two challenges to the design of a type system for JVML:

- *Polymorphism.* Subroutines are polymorphic over the types of the locations they do not touch. For example, consider how variable `2` is used in Figure 2. At the `jsr` on line 7, variable `2` contains an integer and is assumed to contain an integer when the subroutine returns. At the `jsr` on line 18, variable `2` contains a pointer to an exception object and is assumed to contain a pointer to an exception object when the subroutine returns. Inside a subroutine, the type of a location such as variable `2` can depend on the call site of the subroutine. (Subroutines are not parametric over the types of the locations they touch; the polymorphism of JVML is thus weaker than that of ML.)

- *Last-in, first-out behavior.* In most languages, when a return statement in procedure $P$ is executed, the dynamic semantics guarantees that control will return to the point from which $P$ was most recently called. The same is not true of JVML. The `ret` instruction takes a variable as a parameter and jumps to whatever address that variable contains. This semantics means that, unless adequate precautions are taken, the `ret` instruction can transfer control to almost anywhere. Using `ret` to jump to arbitrary places in a program is inimical to static typing, especially in the presence of polymorphism.

Our type system allows polymorphic subroutines and enforces last-in, first-out behavior. It consists of rules that relate a program (a sequence of bytecode instructions) to static information about types and subroutines. This information maps each memory location of the VM to a type at each program point, identifies the instructions that make up subroutines, indicates the variables over which subroutines are polymorphic, and gives static approximations to the dynamic subroutine call stack.

Our type system guarantees the following properties for well-typed programs:

- *Type safety.* An instruction will never be given an operand stack with too few values in it, or with values of the wrong type.

- *Program counter safety.* Execution will not jump to undefined addresses.

- *Bounded operand stack.* The size of the operand stack will never grow beyond a static bound.

## 3 Syntax and informal semantics of JVML0

In JVML0, our restricted version of JVML, a program is a sequence of instructions:

$$P ::= \mathit{instruction}^*$$

We treat programs as partial maps from addresses to instructions. We write $\textsc{Addr}$ for the set of all addresses. Addresses are very much like positive integers, and we use the constant 1 and the function $+$ on addresses. However, to provide more structure to our semantics, we treat numbers and addresses as separate sets. When $P$ is a program, we write $\mathit{Dom}(P)$ for the domain of $P$ (its set of addresses). We assume that $1 \in \mathit{Dom}(P)$ for every program $P$.

In JVML0, there are no classes, methods, or objects. There is no object heap, but there is an operand stack and a set of local variables. We write $\textsc{Var}$ for the set of names of local variables. Local variables and the operand stack both contain *values*. A value is either an integer or an address.

JVML0 has only 9 instructions:

$$\begin{array}{rl} \mathit{instruction} ::= & \texttt{inc} \mid \texttt{pop} \mid \texttt{push0} \\ & \mid \texttt{load}\ x \mid \texttt{store}\ x \\ & \mid \texttt{if}\ L \\ & \mid \texttt{jsr}\ L \mid \texttt{ret}\ x \\ & \mid \texttt{halt} \end{array}$$

where $x$ ranges over $\textsc{Var}$ and $L$ ranges over $\textsc{Addr}$. Informally, these instructions behave as follows:

- The `inc` instruction increments the value at the top of the operand stack if that value is an integer. The `pop` instruction pops the top off the operand stack. The `push0` instruction pushes the integer 0 onto the operand stack.

- The `load` $x$ instruction pushes the current value of local variable $x$ onto the operand stack. The `store` $x$ instruction pops the top value off the operand stack and stores it into local variable $x$.

- The `if` $L$ instruction pops the top value off the operand stack and either falls through when that value is the integer 0 or jumps to $L$ otherwise.

- At address $p$, the `jsr` $L$ instruction jumps to address $L$ and pushes return address $p + 1$ onto the operand stack. The `ret` $x$ instruction jumps to the address stored in $x$.

- The `halt` instruction halts execution.

## 4 Semantics without subroutines

This section introduces our approach and some notation. It presents static and dynamic semantics for the subset of JVML0 that excludes `jsr` and `ret`.

We use (partial) maps extensively throughout this paper. When $g$ is a map, $\mathit{Dom}(g)$ is the domain of $g$; for $x \in \mathit{Dom}(g)$, $g[x]$ is the value of $g$ at $x$, and $g[x \mapsto v]$ is the map with the same domain as $g$ defined by the following equation, for all $y \in \mathit{Dom}(g)$:

$$(g[x \mapsto v])[y] = \begin{cases} g[y] & x \neq y \\ v & x = y \end{cases}$$

We define equality on maps as follows:

$$f = g \ \equiv\ \mathit{Dom}(f) = \mathit{Dom}(g) \wedge \forall x \in \mathit{Dom}(f).\ f[x] = g[x]$$

We often use maps with domain $\textsc{Addr}$. We call those maps vectors. When $F$ is a vector and $i$ is an address, we may write $F_i$ instead of $F[i]$.

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

$$\frac{P[pc] = \texttt{pop}}{P \vdash \langle pc,\ f,\ v \cdot s\rangle \rightarrow \langle pc+1,\ f,\ s\rangle} \qquad \frac{P[pc] = \texttt{push0}}{P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc+1,\ f,\ 0 \cdot s\rangle}$$

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

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

Figure 3: Dynamic semantics without `jsr` and `ret`

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

$$\frac{\begin{array}{c} P[i] = \texttt{pop} \\ F_{i+1} = F_i \\ S_i = T \cdot S_{i+1} \\ i+1 \in \mathit{Dom}(P) \end{array}}{F, S, i \vdash P} \qquad \frac{\begin{array}{c} P[i] = \texttt{push0} \\ F_{i+1} = F_i \\ S_{i+1} = \textsc{Int} \cdot S_i \\ i+1 \in \mathit{Dom}(P) \end{array}}{F, S, i \vdash P}$$

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

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

Figure 4: Static semantics without `jsr` and `ret`

We also use strings. The constant $\epsilon$ denotes the empty string. If $s$ is a string, then $v \cdot s$ denotes the string obtained by prepending $v$ to $s$.

### 4.1 Dynamic semantics

We model a state of an execution as a tuple $\langle pc, f, s\rangle$, where $pc$ is the program counter, $f$ is the current state of local variables, and $s$ is the current state of the operand stack.

- The program counter $pc$ is an address, that is, an element of ADDR.
- The current state of local variables $f$ is a total map from VAR to the set of values.
- The current state of the stack $s$ is a string of values.

All executions start from states of the form $\langle 1, f, \epsilon\rangle$, where $f$ is arbitrary and where $\epsilon$ represents the empty stack.

Figure 3 contains a small-step operational semantics for all instructions other than `jsr` and `ret`. This semantics relies on the judgement

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

which means that program $P$ can take one step from state $\langle pc, f, s\rangle$ to state $\langle pc', f', s'\rangle$. In Figure 3 (and in the rest of this paper), $n$ matches only integers while $v$ matches any value. Thus, for example, the pattern "$n \cdot s$" represents a non-empty stack whose top element is an integer. Note that there is no rule for `halt`—execution stops when a `halt` instruction is reached. Execution may also stop as the result of a dynamic error, for example attempting to execute a `pop` instruction with an empty stack.

### 4.2 Static semantics

Our static semantics employs the following typing rules for values:

$$\frac{v \text{ is a value}}{v : \textsc{Top}} \qquad \frac{n \text{ is an integer}}{n : \textsc{Int}} \qquad \frac{K, L \text{ are addresses}}{K : \texttt{(ret-from } L\texttt{)}}$$

The type TOP includes all values. The type INT is the type of integers. Types of the form `(ret-from` $L$`)` include all addresses. However, the typing rules for programs of Sections 5 and 6 make a more restricted use of address types, preserving strong invariants. As the syntax `(ret-from` $L$`)` suggests, we use $L$ as the name for the subroutine that starts at address $L$, and use `(ret-from` $L$`)` as the type of return addresses generated when $L$ is called. Collectively, we refer to the types TOP, INT, and `(ret-from` $L$`)` as value types.

Types are extended to stacks as follows:

$$\frac{\textit{(Empty hypothesis)}}{\epsilon : \epsilon} \qquad \frac{v : T \quad s : \alpha}{v \cdot s : T \cdot \alpha}$$

where $T$ is a value type and $\alpha$ is a string of value types.

A program is well-typed if there exists a vector $F$ of maps from variables to types and a vector $S$ of strings of types satisfying the judgement:

$$F, S \vdash P$$

The vectors $F$ and $S$ contain static information about the local variables and operand stack, respectively. The map $F_i$ assigns types to local variables at program point $i$. The string $S_i$ gives the types of the values in the operand stack at program point $i$. For notational convenience, the vectors $F$ and $S$ are defined on all of ADDR even though $P$ is not; $F_j$ and $S_j$ are dummy values for out-of-bounds $j$.

We have one rule for proving $F, S \vdash P$:

$$\frac{\begin{array}{c} F_1 = \mathcal{E} \\ S_1 = \epsilon \\ \forall i \in \mathit{Dom}(P).\ F, S, i \vdash P \end{array}}{F, S \vdash P}$$

where $\mathcal{E}$ is the map that maps all variables to TOP and $\epsilon$ is (as usual) the empty string. The first two hypotheses are initial conditions; the third is a local judgement applied to each program point. Figure 4 has rules for the local judgement $F, S, i \vdash P$. These rules constrain $F$ and $S$ at point $i$ by referring to $F_j$ and $S_j$ for all points $j$ that are control-flow successors of $i$.

## 5 Structured semantics

This section shows how to handle `jsr` and `ret`, achieving the kind of polymorphism described in Section 2. To isolate the problem of polymorphism from the problem of ensuring that subroutines are used in a well-structured manner, this section presents what we call the structured semantics for JVML0. This semantics is structured in that it defines the semantics of `jsr` and `ret` in terms of an explicit subroutine call stack. This section shows how to achieve polymorphism in the context of the structured semantics.

### 5.1 Dynamic semantics

In the structured semantics, we augment the state of an execution to include a subroutine call stack. This call stack holds the return addresses of subroutines that have been called but have not yet returned. We model this call stack as a string $\rho$ of addresses. Thus, the state of an execution is now a four-tuple $\langle pc, f, s, \rho\rangle$.

Figure 5 defines the structured dynamic semantics of JVML0. The rules of Figure 5 use the subroutine call stack to communicate return addresses from `jsr` instructions to the corresponding `ret` instructions. Although `ret` takes an operand $x$, the structured dynamic semantics ignores this operand; similarly, the structured dynamic semantics of `jsr` pushes the return address onto the operand stack as well as onto the subroutine call stack. These definitions enable us to transfer the properties of the structured semantics of this section to the stackless semantics of the next section.

### 5.2 Static semantics

The structured static semantics relies on a new typing judgement:

$$F, S \vdash_s P$$

This judgement is defined by the rule:

$$\frac{\begin{array}{c} F_1 = \mathcal{E} \\ S_1 = \epsilon \\ R_1 = \{\} \\ \forall i \in \mathit{Dom}(P).\ R, i \vdash P \text{ labeled} \\ \forall i \in \mathit{Dom}(P).\ F, S, i \vdash_s P \end{array}}{F, S \vdash_s P}$$

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

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

$$\frac{P[pc] = \texttt{push0}}{P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc+1,\ f,\ 0 \cdot s,\ \rho\rangle}$$

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

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

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

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

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

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

Figure 5: Structured dynamic semantics

$$\frac{\begin{array}{c} P[i] \in \{\texttt{inc}, \texttt{pop}, \texttt{push0}, \texttt{load}\ x, \texttt{store}\ x\} \\ R_{i+1} = R_i \end{array}}{R, i \vdash P \text{ labeled}}$$

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

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

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

Figure 6: Rules labeling instructions with subroutines

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

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

Figure 7: Structured static semantics for `jsr` and `ret`

The new, auxiliary judgement

$$R, i \vdash P \text{ labeled}$$

is used to define what it means to be "inside" a subroutine. Unlike in most languages, where procedures are demarcated syntactically, in JVML0 and JVML the instructions making up a subroutine are identified by constraints in the static semantics. For the instruction at address $i$, $R_i$ is a subroutine label that identifies the subroutine to which the instruction belongs. These labels take the form of either the empty set or a singleton set consisting of an address. If an instruction's label is the empty set, then the instruction belongs to the top level of the program. If an instruction's label is the singleton set $\{L\}$, then the instruction belongs to the subroutine that starts at address $L$. Figure 6 contains rules for labeling subroutines. These rules do not permit subroutines to have multiple entry points, but they permit multiple exits.

For some programs, more than one $R$ may satisfy both $R_1 = \{\}$ and the constraints of Figure 6 because the labeling of unreachable code is not unique. It is convenient to assume a canonical $R$ for each program $P$, when one exists. (The particular choice of $R$ does not matter.) We write $R_P$ for this canonical $R$, and $R_{P,i}$ for the value of $R_P$ at address $i$.

Much as in Section 5, the judgement

$$F, S, i \vdash_s P$$

imposes local constraints near program point $i$. For the instructions considered in Section 5 (that is, for all instructions but `jsr` and `ret`), the rules are the same as in Figure 4. Figure 7 contains rules for `jsr` and `ret`.

The elements of $F$ need not be defined on all variables. For an address $i$ inside a subroutine, the domain of $F_i$ includes only the variables that can be read and written inside that subroutine. The subroutine is polymorphic over variables outside $Dom(F_i)$.

## 6 Stackless semantics

The remaining problem is to eliminate the explicit subroutine call stack of the previous section, using instead the operand stack and local variables to communicate return addresses from a `jsr` to a `ret`. As discussed in Section 2, when the semantics of `jsr` and `ret` are defined in terms of the operand stack and local variables, uncontrolled use of `ret` combined with the polymorphism of subroutines is inimical to type safety. This section presents a static semantics that rules out problematic uses of `ret`. This static semantics restricts programs to operate as if an explicit subroutine call stack like the one from the previous section were present. In fact, the soundness argument for the stackless semantics relies on a simulation between the structured semantics and the stackless semantics.

The static and dynamic semantics described in this section are closest to typical implementations of JVML. Thus, we consider these the official semantics of JVML0.

### 6.1 Dynamic semantics

The stackless dynamic semantics consists of the rules of Figure 3 plus the rules for `jsr` and `ret` of Figure 8. The `jsr` instruction pushes the return address onto the operand stack. To use this return address, a program must first pop it into a local variable and then reference that local variable in a `ret` instruction.

### 6.2 Static semantics

To define the stackless static semantics, we revise the rule for $F, S \vdash P$ of Section 4. The new rule is:

$$\frac{\begin{array}{c} F_1 = \mathcal{E} \\ S_1 = \epsilon \\ C_1 = \epsilon \\ \forall i \in Dom(P).\ C, i \vdash P \text{ strongly labeled} \\ \forall i \in Dom(P).\ F, S, i \vdash P \end{array}}{F, S \vdash P}$$

The new, auxiliary judgement

$$C, i \vdash P \text{ strongly labeled}$$

constrains $C$ to be an approximation of the subroutine call stack. Each element of $C$ is a string of subroutine labels.

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

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

Figure 8: Stackless dynamic semantics for `jsr` and `ret`

$$\frac{\begin{array}{c} P[i] \in \{\texttt{inc}, \texttt{pop}, \texttt{push0}, \texttt{load}\ x, \texttt{store}\ x\} \\ C_{i+1} = C_i \end{array}}{C, i \vdash P \text{ strongly labeled}}$$

$$\frac{\begin{array}{c} P[i] = \texttt{if}\ L \\ C_{i+1} = C_L = C_i \end{array}}{C, i \vdash P \text{ strongly labeled}}$$

$$\frac{\begin{array}{c} P[i] = \texttt{jsr}\ L \\ L \notin C_i \\ C_{i+1} = C_i \\ C_L = L \cdot c \\ C_i \text{ is a subsequence of } c \end{array}}{C, i \vdash P \text{ strongly labeled}}$$

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

Figure 9: Rules approximating the subroutine call stack at each instruction

01 `jsr 4` // $C_1 = \epsilon$
02 `jsr 7` // $C_2 = \epsilon$
03 `halt` // $C_3 = \epsilon$
04 `store 1` // $C_4 = 4 \cdot \epsilon$
05 `jsr 10` // $C_5 = 4 \cdot \epsilon$
06 `ret 1` // $C_6 = 4 \cdot \epsilon$
07 `store 2` // $C_7 = 7 \cdot \epsilon$
08 `jsr 10` // $C_8 = 7 \cdot \epsilon$
09 `ret 2` // $C_9 = 7 \cdot \epsilon$
10 `store 3` // $C_{10} = 10 \cdot 4 \cdot 7 \cdot \epsilon$
11 `ret 3` // $C_{11} = 10 \cdot 4 \cdot 7 \cdot \epsilon$

Figure 10: Example of $C$ labeling

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

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

Figure 11: Stackless static semantics for `jsr` and `ret`

For each address $i$, the string $C_i$ is a linearization of the subroutine call graph to $i$. Of course, such a linearization of the subroutine call graph exists only when the call graph is acyclic, that is, when subroutines do not recurse. (We believe that we can prove our theorems while allowing recursion, but disallowing recursion simplifies our proofs and agrees with Sun's specification [LY96, p. 124].) Figure 9 contains the rules for this new judgement, and Figure 10 gives an example; in this example, the order 4 and 7 could be reversed in $C_{10}$ and $C_{11}$.

As with $R$, more than one $C$ may satisfy both $C_1 = \epsilon$ and the constraints in Figure 9. We assume a canonical $C$ for each program $P$, when one exists. We write $C_P$ for this canonical $C$, and $C_{P,i}$ for the value of $C_P$ at address $i$. Programs that satisfy the constraints in Figure 9 also satisfy the constraints in Figure 6; we define $R_P$ from $C_P$ as follows:

$$R_{P,i} = \begin{cases} \{\} & \text{when } C_{P,i} = \epsilon \\ \{L\} & \text{when } C_{P,i} = L \cdot c \text{ for some } c \end{cases}$$

Figure 11 contains the rules that define $F, S, i \vdash P$ for `jsr` and `ret`; rules for other instructions are in Figure 4. The rule for `jsr` $L$ assigns the type `(ret-from` $L$`)` to the return address pushed onto the operand stack. This type will propagate to any location into which this return address is stored, and it is checked by the following hypotheses in the rule for `ret`:

$$\begin{array}{c} x \in Dom(F_i) \\ F_i[x] = (\texttt{ret-from}\ L) \end{array}$$

Typing return addresses helps ensure that the return address used by a subroutine $L$ is a return address for $L$, not for some other subroutine. By itself, ensuring that the return address used by a subroutine $L$ is a return address for $L$ does not guarantee last-in, first-out behavior. One also has to ensure that the only return address for $L$ available inside $L$ is the most recent return address, not one tucked away during a previous invocation. This is achieved by the following hypotheses in the rule for `jsr`:

$$\begin{array}{c} (\texttt{ret-from}\ L) \notin S_i \\ \forall y \in Dom(F_L).\ F_L[y] \neq (\texttt{ret-from}\ L) \end{array}$$

These hypotheses guarantee that the only value of type `(ret-from` $L$`)` available inside $L$ is the most recent value of this type pushed by the `jsr` instruction. (These hypotheses might be redundant for reachable code; we include them because our rules apply also to unreachable code.) Except for the lines discussed above, the rules for `jsr` and `ret` are the same as those of the structured static semantics.

## 7 Soundness

Our main soundness theorem is:

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

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

This theorem says that if a computation stops, then it stops because it has reached a `halt` instruction, not because the program counter has gone out of bounds or because a precondition of an instruction does not hold. This theorem also says that the operand stack is well-typed when a computation stops. This last condition is important because, when a JVML method returns, its return value is on the operand stack.

One of the main lemmas in the proof of this theorem establishes a correspondence between the stackless semantics and the structured semantics. The lemma relies on the definition of a relation between states of the stackless semantics and subroutine call stacks. The lemma says that if $F, S \vdash P$ and

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

then, for all $\rho$ related to $\langle pc, f, s\rangle$, there exists $\rho'$ related to $\langle pc', f', s'\rangle$ such that

$$P \vdash_s \langle pc, f, s, \rho\rangle \to \langle pc', f', s', \rho'\rangle$$

This lemma enables us to obtain properties of the stackless semantics from properties of the simpler structured semantics.

The full version of this paper contains additional results, for example implying a bound on the size of the operand stack. It also contains complete proofs.

## 8 Sun's rules

Sun has published two descriptions of the bytecode verifier, a prose specification and a reference implementation. This section compares our rules with both of these descriptions.

### 8.1 Scope

While our rules simply check static information, Sun's bytecode verifier infers that information. Inference may be important in practice, but only checking is crucial for type safety (and for security). It is therefore reasonable to study checking apart from inference.

JVML has around 200 instructions, while JVML0 has only 9. A rigorous treatment of most of the remaining JVML instructions should pose only minor problems. In particular, many of these instructions are for well understood, arithmetic operations; small difficulties may arise because of their exceptions and other idiosyncrasies. The other instructions (around 20) concern objects and concurrency. Their rigorous treatment would require significant additions to our semantics—for example, a model of the heap. Fortunately, some of these additions are well understood in the context of higher-level, typed languages. Stephen Freund (in collaboration with John Mitchell and with us) is currently extending our rules to the full JVML.

### 8.2 Technical differences

Our rules differ from Sun's reference implementation in the handling of recursive subroutines. Sun's specification disallows recursive subroutines, as do our rules, but Sun's reference implementation allows recursion in certain cases. We believe that recursion is sound in the sense that it does not introduce security holes. However, recursion is an unnecessary complication since it is not useful for compiling Java. Therefore, we believe that the specification should continue to disallow recursion and that the reference implementation should be corrected.

Our rules deviate from Sun's specification and reference implementation in a few respects.

- Sun's rules forbid `load` $x$ when $x$ is uninitialized or holds a return address. Our rules are more general without compromising soundness.
- Sun's rules allow at most one `ret` instruction per subroutine, while our rules allow an arbitrary number.
- Our rules allow `ret` to return only from the most recent call, while Sun's rules allow `ret` to return from calls further up the subroutine call stack. Adding this flexibility to our rules would complicate the structured semantics, but it should not be difficult.

Finally, our rules differ from Sun's reference implementation on a couple of other points. Sun's specification is ambiguous on these points and, therefore, does not provide guidance.

- Sun's reference implementation does not constrain unreachable code. Our rules put constraints on all code. Changing our rules to ignore unreachable code would not require fundamental changes.
- When it comes to identifying what subroutine an instruction belongs to, our rules are more restrictive than the rules implicit in Sun's reference implementation. The flexibility of Sun's reference implementation is important for compiling **finally** clauses that can throw exceptions. Changing our rules to capture Sun's approach would not be difficult, but changing our soundness proof to support this approach may be.

## 9 Other related work

In addition to Sun's, there exist several implementations of the bytecode verifier. Only recently has there been any systematic attempt to understand all these implementations. In particular, the Kimera project has tested several implementations, pointing out some mistakes and discrepancies [SMB97]. We take a complementary approach, based on rigorous reasoning rather than on testing. Both rigorous reasoning and testing may affect our confidence in bytecode verification. While testing does not provide an adequate replacement for precise specifications and proofs, it is a cost-effective way to find certain flaws and oddities.

More broadly, there have been several other implementations of the Java VM. Of particular interest is a partial implementation developed at Computational Logic, Inc. [Coh97]. This implementation is defensive, in the sense that it includes strong (and expensive) dynamic checks, removing the need for bytecode verification. The implementation is written in a formal language, and is intended as a model rather than for production use. Ultimately, one may hope to prove that the defensive implementation is equivalent to an aggressive implementation plus a sound bytecode verifier (perhaps one based on our rules).

There have also been typed intermediate languages other than JVML. Several have been developed for ML and Haskell [TIC97]. We discuss the TIL intermediate languages [Mor95, MTC+96] as representative examples. The TIL intermediate languages provide static guarantees similar to those of JVML. Although these languages have sophisticated type systems, they do not include an analogue to JVML subroutines; instead, they include constructs as high-level as Java's **try-finally** statement. Therefore, the main problems addressed in this paper do not arise in the context of TIL.

Finally, the literature contains many proofs of type soundness for higher-level languages, and in particular proofs for a fragment of Java [DE97, Sym97]. Those proofs have not had to deal with JVML peculiarities (in particular, with subroutines); nevertheless, their techniques may be helpful in extending our work to the full JVML.

In summary, there has not been much work closely related to ours. We do not find this surprising, given that the handling of subroutines is one of the most original parts of the bytecode verifier; it was not derived from prior papers or systems [Yel97]. However, interest in the formal treatment of bytecode verification seems to be mounting; several approaches are currently being pursued [Qia97, Sar97].

## 10 Conclusions

The bytecode verifier is an important part of the Java VM; through static checks, it helps reconcile safety with efficiency. Common descriptions of the bytecode verifier are ambiguous and contradictory. This paper suggests the use of a type system as an alternative to those descriptions. It explores the viability of this suggestion by developing a sound type system for a subset of JVML. This subset, despite its small size, is interesting because it includes JVML subroutines, a source of substantial difficulty in the design of a type system.

Our results so far support the hypothesis that a type system is a good way to describe the bytecode verifier. Significant problems remain, such as handling objects and concurrency, and scaling up to the full JVML. However, we believe that these problems will be no harder than those posed by subroutines, and that a complete type system for JVML could be both tractable and useful.

## Acknowledgements

We thank Luca Cardelli, Drew Dean, Sophia Drossopoulou, Stephen Freund, Mark Lillibridge, Greg Morrisett, George Necula, and Frank Yellin for useful information and suggestions.

## References

[Coh97] Richard M. Cohen. Defensive Java Virtual Machine version 0.5 alpha release. Web pages at `http://www.cli.com/`, May 13, 1997.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Java is type safe—probably. In *Proceedings of ECOOP'97*, pages 389–418, June 1997.

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

[Mor95] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995.

[MTC+96] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, 1996.

[Qia97] Zhenyu Qian. A formal specification of Java(tm) Virtual Machine instructions (draft). Web page at `http://www.informatik.uni-bremen.de/~qian/abs-fsjvm.html`, 1997.

[Sar97] Vijay Saraswat. The Java bytecode verification problem. Web page at `http://www.research.att.com/~vj/main.html`, 1997.

[SMB97] Emin Gün Sirer, Sean McDirmid, and Brian Bershad. Kimera: A Java system security architecture. Web pages at `http://kimera.cs.washington.edu/`, 1997.

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

[TIC97] ACM SIGPLAN Workshop on Types in Compilation (TIC97). June 1997.

[Yel97] Frank Yellin. Private communication. March 1997.