June 11, 1998

SRC Research Report

158

A Type System for Java Bytecode Subroutines

Raymie Stata and Martín Abadi

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

A Type System for Java Bytecode Subroutines

Raymie Stata and Martín Abadi

June 11, 1998

This paper will appear in the ACM Transactions on Programming Languages and Systems. A preliminary version appeared in the Proceedings of the 25th ACM Symposium on Principles of Programming Languages.

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 which is 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.

Contents

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.

For example, the current implementation of the library contains private methods that access privileged resources without performing access control checks. This implementation depends on the verifier preventing untrusted code from calling those private methods. But if the verifier were broken, an attacker might be allowed to treat an instance of a library class as an instance of a class defined by the attacker. In this situation, the class defined by the attacker could be used as a back door. At runtime, most VMs use dispatch-table indices to name methods. Successful invocation of a method depends only on this index and on passing the right number and type of arguments to the method. If the verifier allowed the attacker to treat an instance of the library class as an instance of the back-door class, then the attacker could call the instance's methods, including the private ones that do not perform access control checks.

As this example shows, a successful attack requires more than a broken verifier that accepts rogue JVML code. If, in our example, a VM had some runtime mechanism for preventing the use of an instance of an unexpected class, then the attack

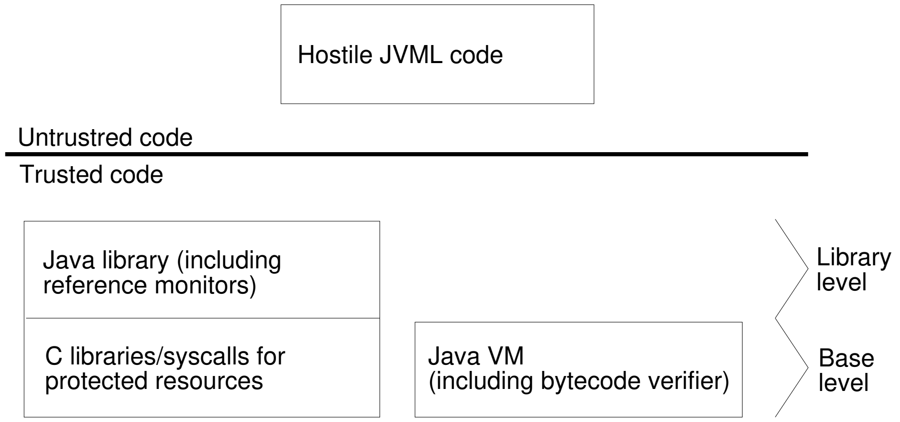

Figure 1: The Java VM and security

would fail. However, this mechanism might be expensive. Thus, the purpose of the verifier is to permit implementations of JVML that are fast without sacrificing safety. The primary customer of the verifier is the rest of the VM implementation: instruction interpretation code, JIT-compiler routines, and other code that assumes that illegal input code is filtered out by the verifier.

Given its importance for security, current descriptions of the verifier are deficient. The official specification of the verifier is a prose description [LY96]. 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 example

for Java **try-finally** statements. JVML subroutines are subsequences of the larger sequence of bytecode instructions that make up a method's body. 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: the verifier should ensure that `ret` is used in a well-structured manner, and it should support 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 and proves 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. Several appendices contain proofs.

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 one of 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 private 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 and returns the integer at the top of the stack. The second private 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. 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

```
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 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                 // 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

```
    // Assume variable 0 contains pointer to this
01  iconst_1       // Push 1
02  istore_1       // Initialize variable 1 to 1
03  jsr 13         // Call subroutine
04  aload_0        // Push this
05  iconst_0       // Push 0
06  putfield x     // Set this.x to 0
07  iconst_1       // Push 1
08  istore_0       // Set variable 0 to 1
09  iconst_0       // Push 0
10  istore_1       // Set variable 1 to 0
11  jsr 13         // Call subroutine
12  return         // End of method
    // Top of subroutine
13  iload_1        // Push variable 1
14  iconst_0       // Push 0
15  if_icmpeq 18   // Goto 18 if variable 1 equals 0
16  astore_2       // Save return address in variable 2
17  ret 2          // Return to return address in variable 2
18  pop            // Throw out current return address
19  ret 2          // Return to old address saved in variable 2
```

Figure 3: Illegal JVML code

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. For example, consider the illegal JVML method body in Figure 3. This method body has two calls to the subroutine that starts at line 13, which is polymorphic over variable `0`. The first time this subroutine is called, it stores the return address into variable `2` and returns. The second time this subroutine is called, it returns to the old return address stored away in variable `2`.

This causes control to return to line 4, at which point the code assumes that variable `0` contains a pointer. However, between the first and second calls to the subroutine, an integer has been stored into variable `0`. Thus, the code in Figure 3 will dereference an integer.

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 $Dom(P)$ for the domain of P (its set of addresses); $P[i]$ is defined only for $i \in Dom(P)$. We assume that $1 \in Dom(P)$ 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. (In JVML, these names are actually natural numbers, but we do not require this.) 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 VAR, and L ranges over 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, $Dom(g)$ is the domain of g; for $x \in Dom(g)$, $g[x]$ 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 $y \in Dom(g)$:

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

That is, $g[x \mapsto v]$ has the same value as g at all points except x, where $g[x \mapsto v]$ has value v. We define equality on maps as follows:

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

That is, $f = g$ exactly when f and g have the same domains and have equal values at all points.

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

We also use strings. The constant ϵ denotes the empty string. If s is a string, then $v \cdot s$ denotes the string obtained by prepending v to s. Sometimes we treat strings as maps from indices to integers. Then $Dom(s)$ is the set of indices of s, and $s[i]$ is the ith element of s from the right. Concatenation adds a new index for the new element but does not change the indices of the old elements; that is:

$$\forall s, i, n.\ i \in Dom(s) \Rightarrow i \in Dom(n \cdot s) \wedge (n \cdot s)[i] = s[i]$$

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 ϵ represents the empty stack.

Figure 4 contains a small-step operational semantics for all instructions other than `jsr` and `ret`. This semantics relies on the judgment

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

which means that program P can, in one step, go from state $\langle pc,\ f,\ s\rangle$ to state $\langle pc',\ f',\ s'\rangle$. By definition, the judgment $P \vdash \langle pc,\ f,\ s\rangle \to \langle pc',\ f',\ s'\rangle$ holds only when P is a program and $\langle pc,\ f,\ s\rangle$ and $\langle pc',\ f',\ s'\rangle$ are states. In Figure 4 (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.

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

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

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

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

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

4.2 Static semantics

Our static semantics employs the following typing rules for values:

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

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:

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

where T is a value type and α is a string of value types. The empty stack is typed by the empty string; a stack with n values is typed by a string with n types, each type correctly typing the corresponding value in the string of values.

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

$$F, S \vdash P$$

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

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

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

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

Figure 5: Static semantics without `jsr` and `ret`

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. (Hence the size of S_i gives the number of values in the operand stack.) For notational convenience, the vectors F and S are defined on all of $\textsc{Addr}$ 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 $\textsc{Top}$ and ϵ is (as usual) the empty string. The first two hypotheses are initial conditions; the third is a local judgment applied to each program point. Figure 5 has rules for the local judgment $F, S, i \vdash P$.

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.

4.3 One-step soundness theorem

The rules above are sound in that any step from a well-typed state leads to another well-typed state:

Theorem 1 *For all P, F, and S such that $F, S \vdash P$:*

$$
\begin{array}{l}
\forall pc, f, s, pc', s', f'. \\
\qquad \wedge \;\; P \vdash \langle pc,\, f,\, s\rangle \rightarrow \langle pc',\, f',\, s'\rangle \\
\qquad \wedge \;\; \forall y.\; f[y] : F_{pc}[y] \\
\qquad \wedge \;\; s : S_{pc} \\
\quad\; \Rightarrow s' : S_{pc'} \wedge \forall y.\; f'[y] : F_{pc'}[y]
\end{array}
$$

We do not prove this theorem, but we do prove similar theorems for programs including `jsr` and `ret`. (See the appendices.)

5 Structured semantics

This section shows how to handle `jsr` and `ret` with 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. The next section shows how to ensure that subroutines are used in a well-structured manner in the absence of an explicit call stack.

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 ρ of addresses. Thus, the state of an execution is now a four-tuple $\langle pc,\, f,\, s,\, \rho\rangle$.

Figure 6 defines the structured dynamic semantics of JVML0. The rules of Figure 6 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

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

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

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

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

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

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

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

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

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

Figure 6: Structured dynamic semantics

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 judgment:

$$F, S \vdash_s P$$

This judgment is defined by the rule:

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

The new, auxiliary judgment

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

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 7 contains rules for labeling subroutines. These rules do not permit subroutines to have multiple entry points, but they do permit multiple exits.

For some programs, more than one R may satisfy both $R_1 = \{\}$ and the constraints of Figure 7 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 judgment

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

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 5. Figure 8 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 $Dom(F_i)$.

Figure 9 gives an example of polymorphism. (Simpler examples could be found if the typing rules allowed a more liberal use of TOP.) The code, in the rightmost

$$\frac{P[i] \in \{\mathtt{inc}, \mathtt{pop}, \mathtt{push0}, \mathtt{load}\ x, \mathtt{store}\ x\} \qquad R_{i+1} = R_i}{R, i \vdash P \text{ labeled}}$$

$$\frac{P[i] = \mathtt{if}\ L \qquad R_{i+1} = R_L = R_i}{R, i \vdash P \text{ labeled}}$$

$$\frac{P[i] = \mathtt{jsr}\ L \qquad R_{i+1} = R_i \qquad R_L = \{L\}}{R, i \vdash P \text{ labeled}}$$

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

Figure 7: Rules labeling instructions with subroutines

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

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

Figure 8: Structured static semantics for jsr and ret

$F_i[0]$	$F_i[1]$	S_i	i	$P[i]$
Top	Top	ϵ	01	`push0`
Top	Top	Int $\cdot\, \epsilon$	02	`store 1`
Top	Int	ϵ	03	`jsr 6`
`(ret-from 6)`	Int	ϵ	04	`jsr 11`
`(ret-from 11)`	`(ret-from 15)`	Int $\cdot\, \epsilon$	05	`halt`
Top	Int	`(ret-from 6)` $\cdot\, \epsilon$	06	`store 0`
`(ret-from 6)`	Int	ϵ	07	`load 1`
`(ret-from 6)`	Int	Int $\cdot\, \epsilon$	08	`jsr 15`
`(ret-from 6)`	`(ret-from 15)`	Int $\cdot\, \epsilon$	09	`store 1`
`(ret-from 6)`	Int	ϵ	10	`ret 0`
`(ret-from 6)`	Int	`(ret-from 11)` $\cdot\, \epsilon$	11	`store 0`
`(ret-from 11)`	Int	ϵ	12	`load 1`
`(ret-from 11)`	Int	Int $\cdot\, \epsilon$	13	`jsr 15`
`(ret-from 11)`	`(ret-from 15)`	Int $\cdot\, \epsilon$	14	`ret 0`
undefined	Int	`(ret-from 15)` $\cdot$ Int $\cdot\, \epsilon$	15	`store 1`
undefined	`(ret-from 15)`	Int $\cdot\, \epsilon$	16	`inc`
undefined	`(ret-from 15)`	Int $\cdot\, \epsilon$	17	`ret 1`

Figure 9: Example of typing

column of Figure 9, computes the number 2 and leaves it at the top of the stack. The code contains a main body and three subroutines, which we have separated with horizontal lines. The subroutine that starts at line 6 increments the value in variable `0`. The subroutine that starts at line 11 also increments the value in variable `0` but leaves its result on the stack. These subroutines are implemented in terms of the one that starts at line 15, which increments the value at the top of the stack. This last subroutine is polymorphic over variable `0`.

5.3 One-step soundness theorem

To state the one-step soundness theorem for the structured semantics, two definitions are needed.

In the previous section, where programs do not have subroutines, the type of local variable x at program point pc is denoted by $F_{pc}[x]$. However, for program points inside subroutines, this definition is not quite adequate. For $x \in Dom(F_{pc})$, the type of x is still $F_{pc}[x]$. For other x, the type of x depends on where execution has come from. A subroutine is polymorphic over local variables that are not in $Dom(F_{pc})$: different call sites of the subroutine may have values of different types in those local variables. Therefore, we construct an assignment of types to local variables that takes into account where execution has come from. The expression:

$$\mathcal{F}(F, pc, \rho)[x]$$

denotes the type assigned to local variable x when execution reaches point pc with a subroutine call stack ρ. The type $\mathcal{F}(F, pc, \rho)[x]$ is defined by the rules:

$$\frac{x \in \mathit{Dom}(F_{pc})}{\mathcal{F}(F, pc, \rho)[x] = F_{pc}[x]} \qquad \texttt{(tt0)}$$

$$\frac{\begin{array}{c} x \notin \mathit{Dom}(F_{pc}) \\ \mathcal{F}(F, p, \rho)[x] = T \end{array}}{\mathcal{F}(F, pc, p \cdot \rho)[x] = T} \qquad \texttt{(tt1)}$$

Note that $\mathcal{F}(F, pc, \rho)[x]$ is not defined when x is not in $\mathit{Dom}(F_{pc})$ or any of the $\mathit{Dom}(F_p)$ for the p's in ρ. However, the theorems and lemmas in the following subsections consider only situations in which $\mathcal{F}$ is defined.

The second auxiliary definition for our theorem concerns an invariant on the structure of the subroutine call stack. The theorem of the previous section says that, when a program takes a step from a well-typed state, it reaches a well-typed state. With subroutines, we need to assume that the subroutine call stack of the starting state is well-formed. For example, if the address p is in the call stack, then it must be the case that the instruction at $p - 1$ is a `jsr` instruction. The new theorem says that if the program takes a step from a well-typed state with a well-formed subroutine call stack, then it reaches a well-typed state with a well-formed subroutine call stack. The following judgment expresses what we mean by a well-formed subroutine call stack:

$$\mathit{WFCallStack}(P, F, pc, \rho)$$

This judgment is defined by the rules:

$$\frac{\mathit{Dom}(F_{pc}) = \textsc{Var}}{\mathit{WFCallStack}(P, F, pc, \epsilon)} \qquad \texttt{(wf0)}$$

$$\frac{\begin{array}{c} P[p-1] = \texttt{jsr } L \\ L \in R_{P,pc} \\ \mathit{Dom}(F_{pc}) \subseteq \mathit{Dom}(F_p) \\ \mathit{WFCallStack}(P, F, p, \rho) \end{array}}{\mathit{WFCallStack}(P, F, pc, p \cdot \rho)} \qquad \texttt{(wf1)}$$

Informally, a subroutine call stack ρ is well-formed when:

- given a return address p in ρ, the instruction just before p is a `jsr` instruction that calls the routine from which p returns;
- the current program counter pc and all return addresses in ρ have the correct subroutine label associated with them;
- the return address at the bottom of the call stack is for code that can access all local variables;
- any variable that can be read and written by a callee can also be read and written by its caller.

With these two definitions, we can state a one-step soundness theorem for the structured semantics. The proof of this theorem is in Appendix A.

Theorem 2 *For all P, F, and S such that $F, S \vdash_s P$:*

$$
\begin{array}{l}
\forall pc, f, s, \rho, pc', f', s', \rho'. \\
\quad \wedge\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc',\ f',\ s',\ \rho'\rangle \\
\quad \wedge\ \mathit{WFCallStack}(P, F, pc, \rho) \\
\quad \wedge\ \forall y.\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\
\quad \wedge\ s : S_{pc} \\
\Rightarrow \\
\quad \wedge\ \mathit{WFCallStack}(P, F, pc', \rho') \\
\quad \wedge\ \forall y.\ \exists T'.\ \mathcal{F}(F, pc', \rho')[y] = T' \wedge f'[y] : T' \\
\quad \wedge\ s' : S_{pc'}
\end{array}
$$

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.

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

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

Figure 10: Stackless dynamic semantics for `jsr` and `ret`

6.1 Dynamic semantics

The stackless dynamic semantics consists of the rules of Figure 4 plus the rules for `jsr` and `ret` of Figure 10. 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. The `ret` x instruction extracts an address from local variable x; if x does not contain an address, then the program gets stuck (because $\langle f[x],\ f,\ s\rangle$ is not a state).

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:

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

The new, auxiliary judgment

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

constrains C to be an approximation of the subroutine call stack. Each element of C is a string of subroutine labels. 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 11 contains the rules for this new judgment; note that a subsequence is not necessarily a consecutive substring. Figure 12 shows an application of the rules

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

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

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

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

Figure 11: Rules approximating the subroutine call stack at each instruction

C_i	i	$P[i]$
ϵ	01	`push0`
ϵ	02	`store 1`
ϵ	03	`jsr 6`
ϵ	04	`jsr 11`
ϵ	05	`halt`
$6 \cdot \epsilon$	06	`store 0`
$6 \cdot \epsilon$	07	`load 1`
$6 \cdot \epsilon$	08	`jsr 15`
$6 \cdot \epsilon$	09	`store 1`
$6 \cdot \epsilon$	10	`ret 0`
$11 \cdot \epsilon$	11	`store 0`
$11 \cdot \epsilon$	12	`load 1`
$11 \cdot \epsilon$	13	`jsr 15`
$11 \cdot \epsilon$	14	`ret 0`
$15 \cdot 11 \cdot 6 \cdot \epsilon$	15	`store 1`
$15 \cdot 11 \cdot 6 \cdot \epsilon$	16	`inc`
$15 \cdot 11 \cdot 6 \cdot \epsilon$	17	`ret 1`

Figure 12: Example of C labeling

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

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

Figure 13: Stackless static semantics for `jsr` and `ret`

of Figure 11 to the code from Figure 9. In this example, the order of 6 and 11 could be reversed in C_{15}, C_{16}, and C_{17}.

As with R, more than one C may satisfy both $C_1 = \epsilon$ and the constraints in Figure 11. 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 11 also satisfy the constraints in Figure 7; we define R_P from C_P as follows:

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

On the other hand, programs with recursive subroutines may satisfy the constraints in Figure 7 but never those in Figure 11.

Figure 13 contains the rules that define $F, S, i \vdash P$ for `jsr` and `ret`; rules for other instructions are in Figure 5. 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`:

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

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`:

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

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. As an example, we may simply reuse the one of Figure 9, since the typing given there remains valid in the stackless static semantics.

6.3 One-step soundness theorem

The one-step soundness theorem for the stackless semantics requires some auxiliary definitions. We develop those definitions top-down, after stating the theorem. The proof of the theorem is in Appendix B.

Theorem 3 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l}
\forall pc, f, s, \rho, pc', f', s'. \\
\qquad \wedge\ \ P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle \\
\qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\
\qquad \wedge\ \ \forall y.\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\
\qquad \wedge\ \ s : S_{pc} \\
\quad \Rightarrow \exists \rho'. \\
\qquad \wedge\ \mathit{Consistent}(P, F, S, pc', f', s', \rho') \\
\qquad \wedge\ \ \forall y.\ \exists T'.\ \mathcal{F}(F, pc', \rho')[y] = T' \wedge f'[y] : T' \\
\qquad \wedge\ \ s' : S_{pc'}
\end{array}$$

In this theorem, the judgment *Consistent* performs several functions:

- Like *WFCallStack* in the one-step soundness theorem for the structured semantics, *Consistent* implies an invariant on the structure of the subroutine call stack.

- In the stackless semantics, the subroutine call stack is not explicit in the dynamic state. The judgment *Consistent* relates an implicit subroutine call stack (ρ or ρ') to the state of an execution.

- The implicit subroutine call stacks of the stackless semantics are closely related to the explicit subroutine call stacks of the structured semantics. In particular, the following implication holds:

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s'. \\ \qquad \wedge\ \ P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle \\ \qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\ \quad \Rightarrow \exists \rho'.\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc',\ f',\ s',\ \rho'\rangle \end{array}$$

  Thus, *Consistent* helps us relate the structured and the stackless semantics.

The judgment *Consistent* is the conjunction of four other judgments:

$$\begin{array}{l} \mathit{Consistent}(P, F, S, pc, f, s, \rho) \equiv \\ \qquad \wedge\ \ \mathit{WFCallStack}(P, F, pc, \rho) \\ \qquad \wedge\ \ \mathit{WFCallStack2}(P, pc, \Lambda_P(\rho), \rho) \\ \qquad \wedge\ \ \mathit{GoodStackRets}(S_{pc}, s, \Lambda_P(\rho), \rho) \\ \qquad \wedge\ \ \mathit{GoodFrameRets}(F, pc, \{\}, f, \Lambda_P(\rho), \rho) \end{array}$$

The first judgment is the judgment *WFCallStack* from the previous section. The next judgment adds more constraints on the subroutine call stack. The last two judgments constrain the values of return addresses in s (the operand stack) and f (local variables) to match those found in ρ (the implicit subroutine call stack).

The auxiliary function Λ returns the sequence of caller labels associated with a sequence of return addresses. It is a partial function defined by two rules:

$$\frac{\textit{(Empty hypothesis)}}{\Lambda_P(\epsilon) = \epsilon} \tag{10}$$

$$\frac{\begin{array}{c} P[p-1] = \texttt{jsr}\ L \\ \Lambda_P(\rho') = \lambda' \end{array}}{\Lambda_P(p \cdot \rho') = L \cdot \lambda'} \tag{11}$$

According to this definition, $\Lambda_P(\rho)$ is the string of labels of subroutines that have been called in ρ. The function $\Lambda_P(\rho)$ is used only when P is well-typed and *WFCallStack* holds, in which case $\Lambda_P(\rho)$ is always defined.

The extensions of *WFCallStack* made by *WFCallStack2* are defined by two rules:

$$\frac{C_{P,pc} = \epsilon}{\mathit{WFCallStack2}(P, pc, \epsilon, \epsilon)} \qquad \texttt{(wf20)}$$

$$\frac{\begin{array}{c} L \notin \lambda' \\ L \cdot \lambda' \text{ is a subsequence of } C_{P,pc} \\ \mathit{WFCallStack2}(P, p, \lambda', \rho') \end{array}}{\mathit{WFCallStack2}(P, pc, L \cdot \lambda', p \cdot \rho')} \qquad \texttt{(wf21)}$$

These rules ensure that no recursive calls have occurred. They also ensure that the dynamic subroutine call stack is consistent with the static approximation of the subroutine call stack.

The judgment *GoodStackRets* constrains the values of return addresses in the operand stack:

$$\frac{\forall j, L.\ j \in \mathit{Dom}(\alpha) \wedge \alpha[j] = \texttt{(ret-from } L\texttt{)} \Rightarrow \mathit{GoodRet}(L, s[j], \lambda, \rho)}{\mathit{GoodStackRets}(\alpha, s, \lambda, \rho)} \qquad \texttt{(gsr0)}$$

$$\frac{\textit{(Empty hypothesis)}}{\mathit{GoodRet}(K, p, \epsilon, \epsilon)} \qquad \texttt{(gr0)}$$

$$\frac{\textit{(Empty hypothesis)}}{\mathit{GoodRet}(K, p, K \cdot \lambda, p \cdot \rho)} \qquad \texttt{(gr1)}$$

$$\frac{\begin{array}{c} K' \neq K \\ \mathit{GoodRet}(K, p, \lambda', \rho') \end{array}}{\mathit{GoodRet}(K, p, K' \cdot \lambda', p' \cdot \rho')} \qquad \texttt{(gr2)}$$

In these definitions, we introduce *GoodRet* as an auxiliary judgment. These definitions say that, when a subroutine L has been called but has not returned, the values of return addresses for L found in s (the operand stack) agree with the return address for L in ρ (the implicit subroutine call stack).

Similarly to the judgment *GoodStackRets*, the judgment *GoodFrameRets* constrains the values of return addresses in local variables:

$$\frac{\textit{(Empty hypothesis)}}{\mathit{GoodFrameRets}(F, pc, \mu, f, \epsilon, \epsilon)} \qquad \texttt{(gfr0)}$$

$$\frac{\begin{array}{c} \forall y, L.\ \left(\begin{array}{l} \wedge\, y \in (\mathit{Dom}(F_{pc}) \backslash \mu) \\ \wedge\, F_{pc}[y] = \texttt{(ret-from } L\texttt{)} \end{array} \right) \Rightarrow \mathit{GoodRet}(L, f[y], K \cdot \lambda', p \cdot \rho') \\ \mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc}), f, \lambda', \rho') \end{array}}{\mathit{GoodFrameRets}(F, pc, \mu, f, K \cdot \lambda', p \cdot \rho')} \qquad \texttt{(gfr1)}$$

Because subroutines may not be able to read and write all variables, *GoodFrameRets* cannot constrain all variables simultaneously in the way that *GoodStackRets* constrains all elements of the operand stack. Instead, the rules for *GoodFrameRets* must work inductively on the subroutine call stack, looking at the return addresses available to each subroutine in turn.

7 Main soundness theorem

Our main soundness theorem is:

Theorem 4 *For all P, F, and S such that $F, S \vdash P$:*

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

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.

The proof of the main soundness theorem is in Appendix C.

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. As discussed below, Freund and Mitchell are currently extending our rules to a large subset of 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.
- According to Sun's rules, "each instruction keeps track of the list of `jsr` targets needed to reach that instruction" [LY96, p. 135]. Using this information, Sun's rules allow a `ret` to return not only from the most recent call but also from calls further up the subroutine call stack. Adding this flexibility to our rules should not be difficult, but it would complicate the structured dynamic semantics and would require additional information in the static semantics.

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 that remove 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]. Here 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, Nv98, 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 is mounting; several approaches are currently being pursued [FM98, Gol97, HT98, Qia98, Sar97]. Goldberg, Qian, and Saraswat all develop other formal frameworks for bytecode verification, basing them on constraints and dataflow analysis; their work is rather broad and not focused on subroutines. Hagiya and Tozawa generalize our rules for subroutines. Building on our type system, Freund and Mitchell study object initialization and its problematic interaction with subroutines; work is under way on a subset of JVML that includes objects, classes, constructors, interfaces, and exceptions.

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 in scaling up to the full JVML, such as handling objects and concurrency. 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.

Appendix

A Proof of soundness for the structured semantics

First, we prove that a step of execution preserves *WFCallStack*. Next, we prove some lemmas about types. Finally, we prove the soundness theorem for the structured semantics (Theorem 2), first by showing that well-typing of the stack is preserved and then by showing that well-typing of local variables is preserved.

A.1 Preservation of *WFCallStack*

The following lemma states certain insensitivities of *WFCallStack* to the exact value of *pc*:

Lemma 1 *For all P, F, and S such that $F, S \vdash_s P$:*

$$\begin{array}{l} \forall pc, pc', \rho. \\ \qquad \wedge \;\; \mathit{WFCallStack}(P, F, pc, \rho) \\ \qquad \wedge \;\; R_{P,pc'} = R_{P,pc} \\ \qquad \wedge \;\; \mathit{Dom}(F_{pc'}) = \mathit{Dom}(F_{pc}) \\ \quad \Rightarrow \mathit{WFCallStack}(P, F, pc', \rho) \end{array}$$

Proof Assume that P, F, S, pc, pc', and ρ satisfy the hypotheses of the lemma. We do a case split on ρ:

1. **Case:** $\rho = \epsilon$. In this case, the assumption $\mathit{WFCallStack}(P, F, pc, \rho)$ must be true by rule (`wf0`), so $\mathit{Dom}(F_{pc}) = \textsc{Var}$. Given this and the assumption that $\mathit{Dom}(F_{pc'}) = \mathit{Dom}(F_{pc})$, it follows that $\mathit{Dom}(F_{pc'}) = \textsc{Var}$. Thus, by rule (`wf0`), we can conclude that $\mathit{WFCallStack}(P, F, pc', \rho)$.

2. **Case:** $\rho \neq \epsilon$. In this case, we know that $\rho = p \cdot \rho'$ for some p and ρ'. Also, the assumption $\mathit{WFCallStack}(P, F, pc, \rho)$ must be true by rule (`wf1`), so, for some L, we know that

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

   Substituting $R_{P,pc'}$ for $R_{P,pc}$ and $\mathit{Dom}(F_{pc'})$ for $\mathit{Dom}(F_{pc})$, we have all we need to establish $\mathit{WFCallStack}(P, F, pc', \rho)$ by rule (`wf1`).

$\square$

Now we show that the structured dynamic semantics preserves $\mathit{WFCallStack}$:

Restatement of part of Theorem 2 *For all P, F, and S such that $F, S \vdash_s P$:*

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s', \rho'. \\ \qquad \wedge \;\; P \vdash_s \langle pc,\ f,\ s,\ \rho \rangle \to \langle pc',\ f',\ s',\ \rho' \rangle \\ \qquad \wedge \;\; \mathit{WFCallStack}(P, F, pc, \rho) \\ \quad \Rightarrow \mathit{WFCallStack}(P, F, pc', \rho') \end{array}$$

Proof Assume that P, F, S, pc, f, s, ρ, pc', f', s', and ρ' satisfy the hypotheses of the lemma. The assumption that a step of execution is possible starting from $\langle pc,\ f,\ s,\ \rho \rangle$ implies that $P[pc]$ is defined; it also implies that $P[pc] \neq \texttt{halt}$. We do a case split on the instruction possible at $P[pc]$, proceeding in three cases:

1. The first case is instructions that do not affect the subroutine call stack: `inc`, `if`, `load`, `store`, `pop`, and `push0`. For these instructions, their structured dynamic semantics implies that $\rho' = \rho$ and their structured static semantics implies that $R_{P,pc} = R_{P,pc'}$ and $Dom(F_{pc}) = Dom(F_{pc'})$. Given these equalities, we can conclude $WFCallStack(P, F, pc', \rho')$ by Lemma 1.

2. The second case is $P[pc] = \texttt{jsr}\ K$ for some K. In this case, the structured dynamic semantics implies that $\rho' = (pc + 1) \cdot \rho$, so we want to show that:

$$WFCallStack(P, F, K, (pc + 1) \cdot \rho)$$

   To prove this judgment using rule (`wf1`), it suffices to show:

   (a) that $P[(pc + 1) - 1] = \texttt{jsr}\ K$, which we are assuming;
   (b) that $K \in R_{P,K}$, which is true because $R_{P,K} = \{K\}$ by the structured static semantics of `jsr` (specifically, line 3 of the `jsr` rule in Figure 7);
   (c) that $Dom(F_K) \subseteq Dom(F_{(pc+1)})$, which also follows from the structured static semantics of `jsr` K (specifically, lines 2 and 3 of the `jsr` rule in Figure 8);
   (d) that $WFCallStack(P, F, pc + 1, \rho)$. We are assuming that

$$WFCallStack(P, F, pc, \rho)$$

   From the structured static semantics of $\texttt{jsr}\ K$ we know that $R_{P,(pc+1)} = R_{P,pc}$ (line 2, `jsr` rule, Figure 7) and $Dom(F_{(pc+1)}) = Dom(F_{pc})$ (line 2, `jsr` rule, Figure 8). Under these conditions,

$$WFCallStack(P, F, pc + 1, \rho)$$

   follows from Lemma 1.

3. The third case is $P[pc] = \texttt{ret}\ x$ for some x. The assumption that a step of execution is possible starting from $\langle pc,\ f,\ s,\ \rho\rangle$ implies that ρ is not empty. If ρ is not empty, then by the structured dynamic semantics of `ret` we know that $\rho = pc' \cdot \rho'$. Given the assumption that

$$WFCallStack(P, F, pc, \rho)$$

   by substitution we can conclude that

$$WFCallStack(P, F, pc, pc' \cdot \rho')$$

   This can be true only by rule (`wf1`), so

$$WFCallStack(P, F, pc', \rho')$$

   must be true.

$\square$

A.2 Lemmas about types

A few lemmas about types are needed. The first lemma helps us reason about the types of locations at program points that dynamically follow a `ret` instruction:

Lemma 2 *For all P, F, and S such that $F, S \vdash_s P$:*

$$\begin{array}{l} \forall pc, f, s, p, \rho', x. \\ \quad \wedge \; \mathit{WFCallStack}(P, F, pc, p \cdot \rho') \\ \quad \wedge \; P[pc] = \texttt{ret}\ x \\ \Rightarrow \\ \quad \wedge \; \forall y \in \mathit{Dom}(F_{pc}).\ F_p[y] = F_{pc}[y] \\ \quad \wedge \; S_p = S_{pc} \end{array}$$

Proof Assume that P, F, S, pc, f, s, p, ρ', and x satisfy the hypotheses of the lemma. The assumption $\mathit{WFCallStack}(P, F, pc, p \cdot \rho')$ must hold by rule (`wf1`), so there exists an L such that $L \in R_{P,pc}$ and $P[p-1] = \texttt{jsr}\ L$. Given such an L, the conjuncts of the conclusion are instantiations of the quantified term found in the structured static semantics of `ret` (Figure 8). □

The following lemma states an insensitivity of $\mathcal{F}$ to the exact value of pc:

Lemma 3

$$\begin{array}{l} \forall F, \rho, y, T, pc, pc'. \\ \quad \wedge \; \mathit{Dom}(F_{pc'}) = \mathit{Dom}(F_{pc}) \\ \quad \wedge \; y \in \mathit{Dom}(F_{pc}) \Rightarrow F_{pc'}[y] = F_{pc}[y] \\ \quad \wedge \; \mathcal{F}(F, pc, \rho)[y] = T \\ \Rightarrow \mathcal{F}(F, pc', \rho)[y] = T \end{array}$$

Proof Assume that F, ρ, y, T, pc, and pc' satisfy the hypotheses of the lemma.

If $y \in \mathit{Dom}(F_{pc})$, then the assumption that $\mathcal{F}(F, pc, \rho)[y] = T$ must be true by rule (`tt0`), so we can conclude that $T = F_{pc}[y]$. Given $y \in \mathit{Dom}(F_{pc})$, by assumption $F_{pc'}[y] = F_{pc}[y]$, thus $F_{pc'}[y] = T$. Therefore, by rule (`tt0`)

$$\mathcal{F}(F, pc', \rho)[y] = T$$

If $y \notin \mathit{Dom}(F_{pc})$, then $\mathcal{F}(F, pc, \rho)[y] = T$ must be true by rule (`tt1`), so we can conclude, for some p and ρ', that $\rho = p \cdot \rho'$ and that

$$\mathcal{F}(F, p, \rho')[y] = T \tag{1}$$

Given the assumptions that $y \notin \mathit{Dom}(F_{pc})$ and $\mathit{Dom}(F_{pc}) = \mathit{Dom}(F_{pc'})$, it follows that $y \notin \mathit{Dom}(F_{pc'})$. Given this and (1),

$$\mathcal{F}(F, pc', \rho)[y] = T$$

follows from rule (`tt1`). □

The next two lemmas say that $\mathcal{F}$ does not change as a result of executing a `jsr` or `ret` instruction. The lemma for `jsr` is:

Lemma 4 *For all P, F, and S such that $F, S \vdash_s P$:*

$$\begin{array}{l} \forall pc, f, \rho, L, y, T. \\ \quad\quad \wedge\ P[pc] = \texttt{jsr}\ L \\ \quad\quad \wedge\ \mathcal{F}(F, pc, \rho)[y] = T \\ \quad \Rightarrow \mathcal{F}(F, L, (pc+1) \cdot \rho)[y] = T \end{array}$$

Proof Assume that P, F, S, pc, f, ρ, L, y, and T satisfy the hypotheses of the lemma. We do a case split on y:

- **Case:** $y \in Dom(F_L)$, that is, y is one of the variables accessible in the subroutine being called. By the structured static semantics of `jsr` (line 3 of `jsr` rule in Figure 8), $Dom(F_L) \subseteq Dom(F_{pc})$, so y is an element of $Dom(F_{pc})$ as well. Given that $y \in Dom(F_{pc})$, the assumption that

  $$\mathcal{F}(F, pc, \rho)[y] = T$$

  could be true only by rule (`tt0`), so we know that $T = F_{pc}[y]$. From $y \in Dom(F_L)$ and the structured static semantics of `jsr` (Figure 8, line 5) it follows that $F_L[y] = F_{pc}[y]$, so $T = F_L[y]$. From $y \in Dom(F_L)$ and $T = F_L[y]$, it follows from rule (`tt0`) that

  $$\mathcal{F}(F, L, (pc+1) \cdot \rho)[y] = T$$

- **Case:** $y \notin Dom(F_L)$. We know that

  $$\begin{array}{c} Dom(F_{pc}) = Dom(F_{pc+1}) \\ y \in Dom(F_{pc}) \Rightarrow F_{pc}[y] = F_{pc+1}[y] \\ \mathcal{F}(F, pc, \rho)[y] = T \end{array}$$

  The first two lines follow from the structured static semantics of `jsr` (Figure 8, lines 2 and 4); the last we are assuming. From Lemma 3 it follows that

  $$\mathcal{F}(F, pc+1, \rho)[y] = T$$

  Since $y \notin Dom(F_L)$, it follows from rule (`tt1`) that

  $$\mathcal{F}(F, L, (pc+1) \cdot \rho)[y] = T$$

$\square$

The next lemma says for `ret` what the previous lemma says for `jsr`. Here, however, we need to assume that we have a well-formed subroutine call stack.

Lemma 5 *For all P, F, and S such that $F, S \vdash_s P$:*

$$\begin{array}{l} \forall pc, f, p, \rho', x, y, T. \\ \quad \wedge \;\; \mathit{WFCallStack}(P, F, pc, p \cdot \rho') \\ \quad \wedge \;\; P[pc] = \texttt{ret}\ x \\ \quad \wedge \;\; \mathcal{F}(F, pc, p \cdot \rho')[y] = T \\ \Rightarrow \mathcal{F}(F, p, \rho')[y] = T \end{array}$$

Proof Assume that P, F, S, pc, f, p, ρ', x, y, and T satisfy the hypotheses of the lemma. We do a case split on y:

- **Case:** $y \in \mathit{Dom}(F_{pc})$. Given the *WFCallStack* assumption plus the assumptions that $P[pc] = \texttt{ret}\ x$ and $y \in \mathit{Dom}(F_{pc})$, we can conclude that $F_p[y] = F_{pc}[y]$ by Lemma 2. Given that $y \in \mathit{Dom}(F_{pc})$, the assumption that

  $$\mathcal{F}(F, pc, p \cdot \rho')[y] = T$$

  must hold by rule (`tt0`), so $T = F_{pc}[y]$ and thus $T = F_p[y]$. The *WFCallStack* assumption could be true only by rule (`wf1`), so $\mathit{Dom}(F_{pc}) \subseteq \mathit{Dom}(F_p)$ and $y \in \mathit{Dom}(F_p)$. Given $T = F_p[y]$ and $y \in \mathit{Dom}(F_p)$:

  $$\mathcal{F}(F, p, \rho')[y] = T$$

  follows from rule (`tt0`).

- **Case:** $y \notin \mathit{Dom}(F_{pc})$. Given $y \notin \mathit{Dom}(F_{pc})$, the assumption that

  $$\mathcal{F}(F, pc, p \cdot \rho')[y] = T$$

  could be true only by rule (`tt1`), so we can conclude

  $$\mathcal{F}(F, p, \rho')[y] = T$$

□

The final lemma says that $\mathcal{F}$ is defined when *WFCallStack* holds:

Lemma 6 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall y, \rho, pc. \\ \quad\quad \mathit{WFCallStack}(P, F, pc, \rho) \\ \quad \Rightarrow \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \end{array}$$

Proof This lemma holds because *WFCallStack* ensures that $Dom(F_p) = \textsc{Var}$ for some p in ρ, so we are guaranteed that rule (`tt0`) will apply for at least this p.

More formally, assume an arbitrary y in $\textsc{Var}$ and P, F, S such that $F, S \vdash P$. We proceed by induction on ρ:

- In the base case, $\rho = \epsilon$. Assume pc such that $\mathit{WFCallStack}(P, F, pc, \rho)$. Because $\rho = \epsilon$, this *WFCallStack* assumption could be true only by rule (`wf0`), so $Dom(F_{pc}) = \textsc{Var}$ and thus $y \in Dom(F_{pc})$. Therefore, by rule (`tt0`), $\mathcal{F}(F, pc, \rho)[y] = F_{pc}[y]$.
- In the inductive step, $\rho = p \cdot \rho'$ for some p and ρ'. Assume pc such that $\mathit{WFCallStack}(P, F, pc, \rho)$.

  If $y \in Dom(F_{pc})$, then we can conclude $\mathcal{F}(F, pc, \rho)[y] = F_{pc}[y]$ by rule (`tt0`).

  If $y \notin Dom(F_{pc})$, then we need to use the induction hypothesis:

  $$\forall q.\ \mathit{WFCallStack}(P, F, q, \rho') \Rightarrow \exists T.\ \mathcal{F}(F, q, \rho')[y] = T$$

  Because ρ is not empty, the assumption $\mathit{WFCallStack}(P, F, pc, \rho)$ could be true only by rule (`wf1`), so $\mathit{WFCallStack}(P, F, p, \rho')$. This matches the antecedent of the induction hypothesis, so we can conclude that $\mathcal{F}(F, p, \rho')[y] = T$ for some T. Thus, by rule (`tt1`), we can conclude that $\mathcal{F}(F, pc, \rho)[y] = T$.

$\square$

A.3 Preservation of stack typing

We now turn our attention back to the soundness theorem. This subsection shows that the typing of the operand stack is preserved; the next shows that the typing of variables is preserved.

Restatement of part of Theorem 2 *For all P, F, and S such that $F, S \vdash_s P$:*

$$\begin{array}{l}
\forall pc, f, s, \rho, pc', f', s', \rho'. \\
\qquad \wedge\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc',\ f',\ s',\ \rho'\rangle \\
\qquad \wedge\ \mathit{WFCallStack}(P, F, pc, \rho) \\
\qquad \wedge\ \forall y.\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\
\qquad \wedge\ s : S_{pc} \\
\quad \Rightarrow s' : S_{pc'}
\end{array}$$

Proof Assume that P, F, S, pc, f, s, ρ, pc', f', s', and ρ' satisfy the hypotheses of the lemma. The assumption that a step of execution is possible starting from $\langle pc,\ f,\ s,\ \rho\rangle$ implies that $P[pc]$ is defined; it also implies that $P[pc] \neq$ `halt`. We do a case split on the instruction possible at $P[pc]$:

1. The first case is `ret`. We know from the structured dynamic semantics of `ret` that $s' = s$ and $\rho = pc' \cdot \rho'$. Given the latter equality and the *WFCallStack* assumption, $S_{pc'} = S_{pc}$ follows from Lemma 2. Given $S_{pc'} = S_{pc}$, $s' = s$, and the assumption that $s : S_{pc}$, it follows that $s' : S_{pc'}$.

2. The second case is `inc`, `pop`, `push0`, `if`, `jsr`, and `store`.

   Consider `inc`, for example. We know from the structured dynamic semantics of `inc` that $s = n \cdot s''$ for some integer n and value string s'', and that $s' = (n+1) \cdot s''$. We know from the structured static semantics of `inc` that

   $$S_{pc} = S_{pc'} = \text{INT} \cdot \alpha$$

   for some type string α. Given $s : S_{pc}$, $s = n \cdot s''$, and $S_{pc} = \text{INT} \cdot \alpha$, it follows that $s'' : \alpha$. Given $s'' : \alpha$, $s' = (n+1) \cdot s''$, and $S_{pc'} = \text{INT} \cdot \alpha$, it follows that $s' : S_{pc'}$.

   As another example, consider `store` x. We know from the structured dynamic semantics of `store` that $s = v \cdot s'$ for some v, and we know from the structured static semantics that $S_{pc} = T \cdot Spc'$ for some T. Given these equations and the assumption that $s : S_{pc}$, it follows that $s' : Spc'$.

   The proofs for the other instructions in this category are along similar lines.

3. The last case is `load` x. In this case, we use the assumptions to infer that $f[x] : T$ for some T such that $\mathcal{F}(F, pc, \rho)[x] = T$. The structured static semantics for `load` says that $x \in Dom(F_{pc})$, so $\mathcal{F}(F, pc, \rho)[x] = T$ must be true by rule (`tt0`), so that $T = F_{pc}[x]$. We know from the structured static semantics of `load` x that $S_{pc'} = F_{pc}[x] \cdot S_{pc}$, so $S_{pc'} = T \cdot S_{pc}$. We know from the structured dynamic semantics of `load` x that $s' = f[x] \cdot s$. Given $f[x] : T$, $s : S_{pc}$, $s' = f[x] \cdot s$, and $S_{pc'} = T \cdot S_{pc}$, it follows that $s' : S_{pc'}$.

$\square$

A.4 Preservation of local variable typing

Restatement of part of Theorem 2 *For all P, F, and S such that $F, S \vdash_s P$:*

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s', \rho', y. \\ \quad \wedge\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \to \langle pc',\ f',\ s',\ \rho'\rangle \\ \quad \wedge\ \mathit{WFCallStack}(P, F, pc, \rho) \\ \quad \wedge\ s : S_{pc} \\ \quad \wedge\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\ \Rightarrow \exists T'.\ \mathcal{F}(F, pc', \rho')[y] = T' \wedge f'[y] : T' \end{array}$$

This statement is stronger than that in Section 5.3 in that the quantifier for y has been moved to the top level.

Proof Assume that P, F, S, pc, f, s, ρ, pc', f', s', ρ', and y satisfy the hypotheses of the lemma. Let T be a type such that $\mathcal{F}(F, pc, \rho)[y] = T$ and $f[y] : T$; by assumption, such a T exists. The assumption that a step of execution is possible starting from $\langle pc, f, s, \rho\rangle$ implies that $P[pc]$ is defined; it also implies that $P[pc] \neq \texttt{halt}$. We do a case split on the instruction possible at $P[pc]$:

1. The first case is instructions that do not modify y or ρ: `inc`, `if`, `load`, `pop`, `push0`, and also `store` x for $x \neq y$. For these instructions, it follows from their structured static semantics that $Dom(F_{pc'}) = Dom(F_{pc})$ and that $F_{pc'}[y] = F_{pc}[y]$ when $y \in Dom(F_{pc})$. It follows from their structured dynamic semantics that $\rho' = \rho$. Thus, by Lemma 3, $\mathcal{F}(F, pc', \rho')[y] = T$. What remains to be shown is $f'[y] : T$. It follows from the structured dynamic semantics of these instructions that $f[y] = f'[y]$; from $f[y] : T$ and $f[y] = f'[y]$ it follows that $f'[y] : T$.

2. The next case is `store` y. By the structured static semantics of `store` y, $y \in Dom(F_{pc'})$. Thus, by rule (`tt0`), $\mathcal{F}(F, pc', \rho')[y] = F_{pc'}[y]$. What remains to be shown is $f'[y] : F_{pc'}[y]$. The structured static semantics for `store` y says that $S_{pc} = F_{pc'}[y] \cdot S_{pc'}$; the structured dynamic semantics for `store` y says that $s = f[y] \cdot S_{pc'}$. Given these equations and the assumption $s : S_{pc}$, it follows that $f'[y] : F_{pc'}[y]$.

3. The last case is `jsr` L and `ret` x. We can conclude that $\mathcal{F}(F, pc', \rho')[y] = T$ by Lemma 4 when $P[i]$ is `jsr` L and by Lemma 5 when $P[i]$ is `ret` x. What remains to be shown is $f'[y] : T$. From the structured dynamic semantics of these instructions we know that $f' = f$. Given $f[y] : T$ and $f[y] = f'[y]$, it follows that $f'[y] : T$.

$\square$

B Proof of soundness for the stackless semantics

This section proceeds roughly top-down, first stating a proposition and two lemmas, next using these to prove the soundness theorem for the stackless semantics, then proving the two lemmas.

If F and S are a typing for P in the stackless semantics, then they are a typing for P in the structured semantics:

Proposition 1 *For all P, F, S, if $F, S \vdash P$, then $F, S \vdash_s P$.*

The proofs below implicitly use this proposition when they apply lemmas and theorems that have $F, S \vdash_s P$ in their hypotheses.

The first lemma establishes a simulation between the structured semantics and the stackless semantics:

Lemma 7 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s'. \\ \qquad \wedge\ P \vdash \langle pc,\ f,\ s\rangle \to \langle pc',\ f',\ s'\rangle \\ \qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\ \quad \Rightarrow \exists \rho'.\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \to \langle pc',\ f',\ s',\ \rho'\rangle \end{array}$$

The second lemma states that *Consistent* is preserved by the structured dynamic semantics:

Lemma 8 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s', \rho'. \\ \qquad \wedge\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \to \langle pc',\ f',\ s',\ \rho'\rangle \\ \qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\ \quad \Rightarrow \mathit{Consistent}(P, F, S', pc', f', s', \rho') \end{array}$$

Given these two lemmas and the soundness theorem for the structured semantics, the soundness theorem of the stackless semantics follows directly.

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

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s'. \\ \qquad \wedge\ P \vdash \langle pc,\ f,\ s\rangle \to \langle pc',\ f',\ s'\rangle \\ \qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\ \qquad \wedge\ \forall y.\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\ \qquad \wedge\ s : S_{pc} \\ \quad \Rightarrow \exists \rho'. \\ \qquad \wedge\ \mathit{Consistent}(P, F, S, pc', f', s', \rho') \\ \qquad \wedge\ \forall y.\ \exists T'.\ \mathcal{F}(F, pc', \rho')[y] = T' \wedge f'[y] : T' \\ \qquad \wedge\ s' : S_{pc'} \end{array}$$

Proof Assume that P, F, S, pc, f, s, ρ, pc', f', and s' satisfy the hypotheses of the theorem. Use Lemma 7 to pick ρ' such that

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

The first conjunct of the conclusion follows from our selection for ρ' and Lemma 8. The last two conjuncts follow from our selection for ρ' and Theorem 2. $\square$

B.1 Analogous steps

Restatement of Lemma 7 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s'. \\ \quad \wedge\ P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle \\ \quad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\ \Rightarrow \exists \rho'.\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc',\ f',\ s',\ \rho'\rangle \end{array}$$

Proof Assume that P, F, S, pc, f, s, ρ, pc', f', and s' satisfy the hypotheses of the lemma. The assumption that a step of execution is possible starting from $\langle pc,\ f,\ s\rangle$ implies that $P[pc]$ is defined; it also implies that $P[pc] \neq$ `halt`. We do a case split on the instruction possible at $P[pc]$:

- For all instructions except `jsr` and `ret`, pick $\rho' = \rho$. A comparison of the structured and stackless dynamics in Figure 10 and Figure 6 shows that the conclusion of our lemma follows directly from our assumptions.
- For `jsr` L, pick $\rho' = (pc + 1) \cdot \rho$. Again, a comparison of Figures 10 and 6 shows that the conclusion of our lemma follows from our assumptions in this case as well.
- The case for the `ret` x instruction is the only complicated one.

  First, we show (by contradiction) that our assumptions imply that ρ has at least one element. If ρ had no elements, then the *WFCallStack2* part of the *Consistent* assumption would have to be true by rule (`wf20`). This would require that $C_{P,pc}$ be empty, but the stackless static semantics for `ret` x (line 2, `ret` rule, Figure 13) together with the relationship between R_P and C_P imply that $C_{P,pc}$ must have at least one element. Thus, *WFCallStack2* cannot hold by rule (`wf20`). Instead, it must hold by rule (`wf21`), so ρ must have at least one element.

  Therefore, $\rho = p \cdot \rho''$ for some p and ρ''. We let $\rho' = \rho''$. To prove that this selection of ρ' satisfies the conclusion, we need to show that:

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

  In order to establish this judgment, we must show that $pc' = p$. By the stackless dynamic semantics of `ret` x, $pc' = f[x]$, so this reduces to showing that $f[x] = p$.

  Because ρ has at least one element, the *WFCallStack* assumption must be true by rule (`wf1`). From this we can infer that, for some L, $P[p - 1] =$ `jsr` L and $L \in R_{P,pc}$.

– Given $P[p-1] = \texttt{jsr}\ L$ and $\rho = p \cdot \rho'$, we know by rule (`l1`) that $\Lambda_P(\rho) = L \cdot \lambda'$ for some λ'. (As mentioned in Section 6.3, $\Lambda_P(\rho)$ is always defined for well-typed programs when *WFCallStack* holds.)
– Given that $L \in R_{P,pc}$, we know by the stackless static semantics of $\texttt{ret}\ x$ that $F_{pc}[x] = \texttt{(ret-from}\ L\texttt{)}$ (line 4, `ret` rule, Figure 13). We also know from the stackless static semantics of $\texttt{ret}\ x$ that $x \in \mathit{Dom}(F_{pc})$ (line 3).

Because ρ is not empty, the *GoodFrameRets* component of the *Consistent* assumption must be true by rule (`gfr1`). Given this, $x \in \mathit{Dom}(F_{pc})$, and

$$F_{pc}[x] = \texttt{(ret-from}\ L\texttt{)}$$

it must be the case that

$$\mathit{GoodRet}(L, f[x], \Lambda_P(\rho), p \cdot \rho')$$

We know that $\Lambda_P(\rho) = L \cdot \lambda'$, so by substitution:

$$\mathit{GoodRet}(L, f[x], L \cdot \lambda', p \cdot \rho')$$

This could be true only by rule (`gr1`), so we can conclude that $f[x] = p$.

□

B.2 Preservation of *Consistent*

Expanding the definition of *Consistent*, we want to prove the following lemma:

Restatement of Lemma 8 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l}
\forall pc, f, s, \rho, pc', f', s', \rho'. \\
\quad \wedge\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc',\ f',\ s',\ \rho'\rangle \\
\quad \wedge\ \mathit{WFCallStack}(P, F, pc, \rho) \\
\quad \wedge\ \mathit{WFCallStack2}(P, pc, \Lambda_P(\rho), \rho) \\
\quad \wedge\ \mathit{GoodStackRets}(S_{pc}, s, \Lambda_P(\rho), \rho) \\
\quad \wedge\ \mathit{GoodFrameRets}(F, pc, \{\}, f, \Lambda_P(\rho), \rho) \\
\Rightarrow \\
\quad \wedge\ \mathit{WFCallStack}(P, F, pc', \rho') \\
\quad \wedge\ \mathit{WFCallStack2}(P, pc', \Lambda_P(\rho'), \rho') \\
\quad \wedge\ \mathit{GoodStackRets}(S_{pc'}, s', \Lambda_P(\rho'), \rho') \\
\quad \wedge\ \mathit{GoodFrameRets}(F, pc', \{\}, f', \Lambda_P(\rho), \rho')
\end{array}$$

We prove this by assuming the hypotheses and proving separately each conjunct of the conclusion. We prove the first conjunct (preservation of *WFCallStack*) in Section A.1. We prove the remaining conjuncts below after we state and prove some miscellaneous lemmas.

B.2.1 Miscellaneous lemmas

The following lemma describes how a program step can change $\Lambda_P(\rho)$:

Lemma 9 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s', \rho'. \\ \quad\quad \wedge\ \ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc',\ f',\ s',\ \rho'\rangle \\ \quad\quad \wedge\ \ \mathit{WFCallStack}(P, F, pc, \rho) \\ \quad \Rightarrow \\ \quad\quad \wedge\ \ \forall L.\ (P[pc] = \texttt{jsr}\ L\ \ \Rightarrow \Lambda_P(\rho') = L \cdot \Lambda_P(\rho)) \\ \quad\quad \wedge\ \ \forall x.\ (P[pc] = \texttt{ret}\ x\ \ \Rightarrow \exists L.\ L \cdot \Lambda_P(\rho') = \Lambda_P(\rho)) \end{array}$$

Proof Assume that P, F, S, pc, f, s, ρ, pc', f', s', and ρ' satisfy the hypotheses of the lemma. The assumption that a step of execution is possible starting from $\langle pc,\ f,\ s,\ \rho\rangle$ implies that $P[pc]$ is defined; it also implies that $P[pc] \neq \texttt{halt}$. We do a case split on the instruction possible at $P[pc]$:

- $\texttt{jsr}\ L$: From the structured dynamic semantics of $\texttt{jsr}\ L$, $\rho' = (pc + 1) \cdot \rho$. By assumption, $P[pc] = \texttt{jsr}\ L$. Thus, by rule (`l1`), $\Lambda_P(\rho') = L \cdot \Lambda_P(\rho)$.
- $\texttt{ret}\ x$: From the structured dynamic semantics of $\texttt{ret}\ x$, $\rho = pc' \cdot \rho'$. Thus, the *WFCallStack* assumption must be true by rule (`wf1`), so $P[pc' - 1] = \texttt{jsr}\ L$ for some L. Thus, by rule (`l1`), $\Lambda_P(\rho) = L \cdot \Lambda_P(\rho')$.

□

Lemma 10 *For all P, pc, λ, and ρ:*

$$\mathit{WFCallStack2}(P, pc, \lambda, \rho) \Rightarrow \rho \textit{ and } \lambda \textit{ have the same length}$$

This can be proven by induction on the derivation of $\mathit{WFCallStack2}(P, pc, \lambda, \rho)$.

Lemma 11 *For all K, p, λ, ρ:*

$$\begin{array}{l} K \notin \lambda\ \wedge\ \lambda \textit{ and } \rho \textit{ have the same length} \\ \Rightarrow \mathit{GoodRet}(K, p, \lambda, \rho) \end{array}$$

This lemma can be proven by induction on λ using rule (`gr0`) in the base case and rule (`gr2`) in the inductive step.

The following lemma states an insensitivity of *GoodRet*:

Lemma 12

$$\begin{aligned}
&\forall P, F, pc, p, \rho', L, \lambda', v, K.\\
&\qquad \wedge\ \ \mathit{WFCallStack2}(P, pc, L \cdot \lambda', p \cdot \rho')\\
&\qquad \wedge\ \ \mathit{GoodRet}(K, v, L \cdot \lambda', p \cdot \rho')\\
&\quad \Rightarrow \mathit{GoodRet}(K, v, \lambda', \rho')
\end{aligned}$$

Proof Assume P, F, pc, p, ρ', L, λ', v, and K satisfy the hypotheses of our lemma. The *WFCallStack2* assumption implies that ρ' and λ' have the same length (by Lemma 10). We proceed with a case split on ρ':

- When ρ' is empty, so is λ', thus $\mathit{GoodRet}(K, v, \lambda', \rho')$ follows from rule (`gr0`).
- When ρ' is not empty, we proceed with a case split on K. When $K \neq L$, the *GoodRet* assumption must be true by rule (`gr2`), so

  $$\mathit{GoodRet}(K, v, \lambda', \rho')$$

  The next case is $K = L$. Because ρ' is not empty, the *WFCallStack2* assumption must be true by rule (`wf21`), so $L \notin \lambda'$ and thus $K \notin \lambda'$. Given this and the fact that λ' and ρ' have the same length, we can conclude by Lemma 11 that

  $$\mathit{GoodRet}(K, v, \lambda', \rho')$$

□

The next lemma is used to prove the following lemma:

Lemma 13

$$\begin{aligned}
&\forall \rho, P, F, pc, \mu, f, f', x, v.\\
&\qquad \wedge\ \ \mathit{WFCallStack}(P, F, pc, \rho)\\
&\qquad \wedge\ \ \mu \subseteq \mathit{Dom}(F_{pc})\\
&\qquad \wedge\ \ x \in \mu\\
&\qquad \wedge\ \ f' = f[x \mapsto v]\\
&\qquad \wedge\ \ \mathit{GoodFrameRets}(F, pc, \mu, f, \Lambda_P(\rho), \rho)\\
&\quad \Rightarrow \mathit{GoodFrameRets}(F, pc, \mu, f', \Lambda_P(\rho), \rho)
\end{aligned}$$

Proof We proceed by induction on ρ:

- In the base case, $\rho = \epsilon$. Pick arbitrary P, F, pc, μ, f, f', x, and v. In this case, $\mathit{GoodFrameRets}(F, pc', \mu, f', \Lambda_P(\rho), \rho)$ follows from rule (`gfr0`).

- In the inductive step, $\rho = p \cdot \rho'$ for some p and ρ'. Pick arbitrary P, F, pc, μ, f, f', x, and v. Assume all hypotheses of Lemma 13 hold. The *GoodFrameRets* assumption could be true only by rule (`gfr1`), so

$$\forall y, L. \left(\begin{array}{l} \wedge\ y \in (\mathit{Dom}(F_{pc}) \backslash \mu) \\ \wedge\ F_{pc}[y] = \texttt{(ret-from } L\texttt{)} \end{array} \right) \Rightarrow \mathit{GoodRet}(L, f[y], \Lambda_P(\rho), \rho) \quad (2)$$

and also

$$\mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc}), f, \Lambda_P(\rho'), \rho') \quad (3)$$

First, we want to show that

$$\forall y, L. \left(\begin{array}{l} \wedge\ y \in (\mathit{Dom}(F_{pc}) \backslash \mu) \\ \wedge\ F_{pc}[y] = \texttt{(ret-from } L\texttt{)} \end{array} \right) \Rightarrow \mathit{GoodRet}(L, f'[y], \Lambda_P(\rho), \rho) \quad (4)$$

Assume y and L such that y is in $\mathit{Dom}(F_{pc}) \backslash \mu$ and $F_{pc}[y] = \texttt{(ret-from } L\texttt{)}$. Because $x \in \mu$, it must be that $y \neq x$, so by our assumption relating f to f', $f[y] = f'[y]$. Given this equality, $\mathit{GoodRet}(L, f'[y], \Lambda_P(\rho), \rho)$ follows from (2).

Next, we want to show that

$$\mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc}), f', \Lambda_P(\rho'), \rho') \quad (5)$$

We prove this using the induction hypothesis, which we instantiate as follows:

$$\begin{array}{ll} & \wedge\ \ \mathit{WFCallStack}(P, F, p, \rho') \\ & \wedge\ \ \mathit{Dom}(F_{pc}) \subseteq \mathit{Dom}(F_p) \\ & \wedge\ \ x \in \mathit{Dom}(F_{pc}) \\ & \wedge\ \ f' = f[x \mapsto v] \\ & \wedge\ \ \mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc}), f, \Lambda_P(\rho'), \rho') \\ \Rightarrow & \mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc}), f', \Lambda_P(\rho'), \rho') \end{array}$$

The first two antecedents of this instantiation of the induction hypothesis follow from the fact that the *WFCallStack* assumption must be true by rule (`wf1`). The next antecedent follows from the assumptions that $x \in \mu$ and $\mu \subseteq \mathit{Dom}(F_{pc})$. The fourth antecedent we are assuming, and the last is (3). Thus, we can use the induction hypothesis to conclude (5).

Given (4) and (5),

$$\mathit{GoodFrameRets}(F, pc, \mu, f', \Lambda_P(\rho), \rho)$$

follows from rule (`gfr1`).

□

The final lemma in this set states an insensitivity of *GoodFrameRets*:

Lemma 14

$$
\begin{array}{l}
\forall P, F, pc, f, \rho, pc', f', x, v, T. \\
\quad \wedge\ \ WFCallStack(P, F, pc, \rho) \\
\quad \wedge\ \ x \in Dom(F_{pc}) \\
\quad \wedge\ \ f' = f[x \mapsto v] \\
\quad \wedge\ \ F_{pc'} = F_{pc}[x \mapsto T] \\
\quad \wedge\ \ \forall K.\, T = \texttt{(ret-from } K\texttt{)} \Rightarrow GoodRet(K, v, \Lambda_P(\rho), \rho) \\
\quad \wedge\ \ GoodFrameRets(F, pc, \{\}, f, \Lambda_P(\rho), \rho) \\
\Rightarrow GoodFrameRets(F, pc', \{\}, f', \Lambda_P(\rho), \rho)
\end{array}
$$

Proof Pick arbitrary P, F, pc, f, ρ, pc', f', x, v, and T. We proceed by a case split on ρ:

- The first case is $\rho = \epsilon$. In this case, $GoodFrameRets(F, pc', \{\}, f', \Lambda_P(\rho), \rho)$ follows from rule (`gfr0`).

- The second case is $\rho = p \cdot \rho'$ for some p and ρ'. Assume all hypotheses of Lemma 14 hold. The *GoodFrameRets* assumption could be true only by rule (`gfr1`), so

$$\forall y, L. \left(\begin{array}{l} \wedge\ y \in (Dom(F_{pc})) \\ \wedge\ F_{pc}[y] = \texttt{(ret-from } L\texttt{)} \end{array} \right) \Rightarrow GoodRet(L, f[y], \Lambda_P(\rho), \rho) \tag{6}$$

and also

$$GoodFrameRets(F, p, Dom(F_{pc}), f, \Lambda_P(\rho'), \rho') \tag{7}$$

First, we want to show that

$$\forall y, L. \left(\begin{array}{l} \wedge\ y \in (Dom(F_{pc'})) \\ \wedge\ F_{pc'}[y] = \texttt{(ret-from } L\texttt{)} \end{array} \right) \Rightarrow GoodRet(L, f'[y], \Lambda_P(\rho), \rho) \tag{8}$$

By assumption, $Dom(F_{pc'}) = Dom(F_{pc})$, so (8) is equivalent to

$$\forall y, L. \left(\begin{array}{l} \wedge\ y \in (Dom(F_{pc})) \\ \wedge\ F_{pc'}[y] = \texttt{(ret-from } L\texttt{)} \end{array} \right) \Rightarrow GoodRet(L, f'[y], \Lambda_P(\rho), \rho)$$

Assume y and L such that y is in $Dom(F_{pc})$ and $F_{pc'}[y] = \texttt{(ret-from } L\texttt{)}$. For $y \neq x$, $GoodRet(L, f'[y], \Lambda_P(\rho), \rho)$ follows from (6) and our assumptions relating f to f' and F_{pc} to $F_{pc'}$. For $y = x$, $GoodRet(L, f'[y], \Lambda_P(\rho), \rho)$ follows from the assumption

$$\forall K.\, T = \texttt{(ret-from } K\texttt{)} \Rightarrow GoodRet(K, v, \Lambda_P(\rho), \rho)$$

Next, we want to show that

$$\mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc'}), f', \Lambda_P(\rho'), \rho') \tag{9}$$

By assumption, $\mathit{Dom}(F_{pc}) = \mathit{Dom}(F_{pc'})$, so (9) is equivalent to

$$\mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc}), f', \Lambda_P(\rho'), \rho')$$

We prove this using Lemma 13. Our assumptions imply the following conclusions:

$$\begin{array}{l} \mathit{WFCallStack}(P, F, p, \rho') \\ \mathit{Dom}(F_{pc}) \subseteq \mathit{Dom}(F_p) \\ x \in \mathit{Dom}(F_{pc}) \\ f' = f[x \mapsto v] \\ \mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc}), f, \Lambda_P(\rho'), \rho') \end{array}$$

The first two conclusions follow from the fact that ρ is not empty and thus the *WFCallStack* assumption can only be true by rule (`wf1`). The next two we are assuming directly. The last conclusion is (7). These conclusions are the antecedents of Lemma 13, so (9) follows.

Given (8) and (9),

$$\mathit{GoodFrameRets}(F, pc', \{\}, f', \Lambda_P(\rho), \rho)$$

follows from rule (`gfr1`).

□

B.2.2 Preservation of *WFCallStack2*

Restatement of Part of Lemma 8 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s', \rho'. \\ \quad\quad \wedge\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc',\ f',\ s',\ \rho'\rangle \\ \quad\quad \wedge\ \mathit{WFCallStack}(P, F, pc, \rho) \\ \quad\quad \wedge\ \mathit{WFCallStack2}(P, pc, \Lambda_P(\rho), \rho) \\ \quad \Rightarrow \mathit{WFCallStack2}(P, pc', \Lambda_P(\rho'), \rho') \end{array}$$

This restatement omits unneeded conjuncts in the hypotheses.

Proof Assume that P, F, S, pc, f, s, ρ, pc', f', s', and ρ' satisfy the hypotheses of the lemma. The assumption that a step of execution is possible starting from $\langle pc,\ f,\ s,\ \rho\rangle$ implies that $P[pc]$ is defined; it also implies that $P[pc] \neq$ `halt`. We do a case split on the instruction possible at $P[pc]$:

- For all instructions except `jsr` and `ret`, we have $\rho' = \rho$ (by the structured dynamic semantics), so $\Lambda_P(\rho') = \Lambda_P(\rho)$ (by substitution). Also, by the stackless static semantics of these instructions, $C_{P,pc'} = C_{P,pc}$. Given these equations,

  $$\mathit{WFCallStack2}(P, pc', \Lambda_P(\rho'), \rho')$$

  follows from the *WFCallStack2* assumption, whether that assumption is true by rule (`wf20`) or rule (`wf21`).

- In the case of `jsr` L, we have $L = pc'$ (by the structured dynamic semantics). We first prove that the *WFCallStack2* assumption implies that

  $$\Lambda_P(\rho) \text{ is a subsequence of } C_{P,pc} \tag{10}$$

  If $\rho \neq \epsilon$, then *WFCallStack2* must hold by rule (`wf21`), so (10) must be true in this case. If $\rho = \epsilon$, then $\Lambda_P(\rho) = \epsilon$ (by rule (`l0`)) and $C_{P,pc} = \epsilon$ (because *WFCallStack2* must hold by rule (`wf20`)), so (10) must be true in this case as well.

  We use rule (`wf21`) to show that $\mathit{WFCallStack2}(P, pc', \Lambda_P(\rho'), \rho')$ is true. According to this rule, it suffices to establish the following three intermediate results:

  1. $L \notin \Lambda_P(\rho)$. This follows from (10) and $L \notin C_{P,pc}$, which follows from the stackless static semantics of `jsr` L (line 2, `jsr` rule, Figure 11).
  2. $\Lambda_P(\rho')$ is a subsequence of $C_{P,pc'}$. From Lemma 9 we know that $\Lambda_P(\rho') = L \cdot \Lambda_P(\rho)$. Together with (10), this implies that

     $$\Lambda_P(\rho') \text{ is a subsequence of } L \cdot C_{P,pc} \tag{11}$$

     From the stackless static semantics of `jsr` L, we know that

     $$L \cdot C_{P,pc} \text{ is a subsequence of } C_{P,pc'} \tag{12}$$

     (lines 4 and 5, `jsr` rule, Figure 11). Given (11) and (12), we can conclude that

     $$\Lambda_P(\rho') \text{ is a subsequence of } C_{P,pc'}$$

  3. $\mathit{WFCallStack2}(P, pc+1, \Lambda_P(\rho), \rho)$. We prove this in two cases, the first when the *WFCallStack2* assumption is true by rule (`wf20`), the second when the *WFCallStack2* assumption is true by rule (`wf21`). In both of these cases we have $C_{P,pc+1} = C_{P,pc}$, which follows from the stackless static semantics of `jsr` L (line 3, `jsr` rule, Figure 11).

If the *WFCallStack2* assumption is true by rule (`wf20`), then ρ, $\Lambda_P(\rho)$, and $C_{P,pc}$ must be empty. Given $C_{P,pc+1} = C_{P,pc}$, $C_{P,pc+1}$ must be empty too. Thus,

$$\mathit{WFCallStack2}(P, pc + 1, \Lambda_P(\rho), \rho)$$

follows from rule (`wf20`).

If the *WFCallStack2* assumption is true by rule (`wf21`), then there exist p, ρ'', K, and λ'' such that

$$\begin{gathered} \rho = p \cdot \rho'' \\ \Lambda_P(\rho) = K \cdot \lambda'' \\ K \notin \lambda'' \\ K \cdot \lambda'' \text{ is a subsequence of } C_{P,pc} \\ \mathit{WFCallStack2}(P, p, \lambda'', \rho'') \end{gathered}$$

Because $C_{P,pc+1} = C_{P,pc}$, $K \cdot \lambda''$ is a subsequence of $C_{P,pc+1}$ as well. Since $K \notin \lambda''$, $K \cdot \lambda''$ is a subsequence of $C_{P,pc+1}$, and $\mathit{WFCallStack2}(P, p, \lambda'', \rho'')$,

$$\mathit{WFCallStack2}(P, pc + 1, K \cdot \lambda'', p \cdot \rho'')$$

follows from rule (`wf21`). Since $\rho = p \cdot \rho''$ and $\Lambda_P(\rho) = K \cdot \lambda''$, we obtain

$$\mathit{WFCallStack2}(P, pc + 1, \Lambda_P(\rho), \rho)$$

- In the case of `ret` x, we know from Lemma 9 and from the structured dynamic semantics that

$$\begin{gathered} \Lambda_P(\rho) = L \cdot \Lambda_P(\rho') \\ \rho = pc' \cdot \rho' \end{gathered}$$

for some L. Applying these equations to the *WFCallStack2* assumption, we can conclude that

$$\mathit{WFCallStack2}(P, pc, L \cdot \Lambda_P(\rho'), pc' \cdot \rho')$$

This judgment can be true only by rule (`wf21`), so we can conclude

$$\mathit{WFCallStack2}(P, pc', \Lambda_P(\rho'), \rho')$$

□

B.2.3 Preservation of *GoodStackRets*

Restatement of Part of Lemma 8 *For all P, F, and S such that $F, S \vdash P$:*

$$
\begin{array}{l}
\forall pc, f, s, \rho, pc', f', s', \rho'. \\
\quad \wedge\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc',\ f',\ s',\ \rho'\rangle \\
\quad \wedge\ \mathit{WFCallStack}(P, F, pc, \rho) \\
\quad \wedge\ \mathit{WFCallStack2}(P, pc, \Lambda_P(\rho), \rho) \\
\quad \wedge\ \mathit{GoodFrameRets}(F, pc, \{\}, f, \Lambda_P(\rho), \rho) \\
\quad \wedge\ \forall j, L. \left(\begin{array}{l} \wedge\ j \in \mathit{Dom}(S_{pc}) \\ \wedge\ S_{pc}[j] = \texttt{(ret-from}\ L\texttt{)} \end{array}\right) \Rightarrow \mathit{GoodRet}(L, s[j], \Lambda_P(\rho), \rho) \\
\Rightarrow \forall j, L. \left(\begin{array}{l} \wedge\ j \in \mathit{Dom}(S_{pc'}) \\ \wedge\ S_{pc'}[j] = \texttt{(ret-from}\ L\texttt{)} \end{array}\right) \Rightarrow \mathit{GoodRet}(L, s'[j], \Lambda_P(\rho'), \rho')
\end{array}
$$

Again, this restatement omits unneeded conjuncts in the hypotheses. Also, for convenience, this restatement expands the definition of *GoodStackRets*.

Proof Assume that P, F, S, pc, f, s, ρ, pc', f', s', and ρ' satisfy the hypotheses of the lemma. The assumption that a step of execution is possible starting from $\langle pc,\ f,\ s,\ \rho\rangle$ implies that $P[pc]$ is defined; it also implies that $P[pc] \neq \texttt{halt}$. We do a case split on the instruction possible at $P[pc]$:

- First we look at all instructions except `jsr` and `ret`. For these instructions, $\rho = \rho'$ by the structured dynamic semantics of the individual instructions; also, by substitution, $\Lambda_P(\rho) = \Lambda_P(\rho')$ as well.

  Assume j and L such that $j \in \mathit{Dom}(S_{pc'})$ and $S_{pc'}[j] = \texttt{(ret-from}\ L\texttt{)}$; we need to show that
  $$\mathit{GoodRet}(L, s'[j], \Lambda_P(\rho'), \rho')$$
  This follows from rule (`gr0`) when $\rho' = \epsilon$. Assume that $\rho' \neq \epsilon$ and proceed with a case split on j:

  - The first case is stack slots that are unchanged, that is, stack slots that are not written by the instruction but rather are carried over. The exact set is going to vary by instruction, but values of j in this set will satisfy the following:
    $$j \in \mathit{Dom}(Spc)\ \wedge\ S_{pc'}[j] = S_{pc}[j]\ \wedge\ s'[j] = s[j]$$
    For these stack elements, if $S_{pc'}[j]$ equals $\texttt{(ret-from}\ L\texttt{)}$, then $S_{pc}[j]$ also equals $\texttt{(ret-from}\ L\texttt{)}$. This implies $\mathit{GoodRet}(L, s[j], \Lambda_P(\rho), \rho)$, which by substitution means $\mathit{GoodRet}(L, s'[j], \Lambda_P(\rho'), \rho')$.

– The second case is stack slots that are written by the instruction. It turns out that this case applies only to `load` for the following reasons:
  * The instructions `pop`, `if`, and `store` do not write the stack, so this second case does not apply to these instructions.
  * For `inc` and `push0`, if j is the element written (that is, the top element of s'), we know from the stackless static semantics that the type $S_{pc'}[j]$ is INT, which violates our assumption that $S_{pc'}[j]$ equals `(ret-from` L`)`, so this second case does not apply to these instructions either.

  For `load` x, we know from the stackless static semantics of `load` x that $x \in \mathit{Dom}(F_{pc})$ and $F_{pc}[x] = S_{pc'}[j]$. Given the assumption that $\rho' \neq \epsilon$, the *GoodFrameRets* assumption must be true by rule (`gfr1`). The antecedents of this rule and the assumptions that $x \in \mathit{Dom}(F_{pc})$ and $S_{pc'}[j] = F_{pc}[x] =$ `(ret-from` L`)` imply that

$$\mathit{GoodRet}(L, f[x], \Lambda_P(\rho), \rho)$$

  By the structured dynamic semantics of `load` x, $s'[j] = f[x]$ and $\rho' = \rho$, so by substitution

$$\mathit{GoodRet}(L, s'[j], \Lambda_P(\rho'), \rho')$$

- For `jsr` and `ret`, assume j and L such that $j \in \mathit{Dom}(S_{pc'})$ and $S_{pc'}[j] =$ `(ret-from` L`)`; we need to show that

$$\mathit{GoodRet}(L, s'[j], \Lambda_P(\rho'), \rho') \tag{13}$$

  But first we prove a helpful fact. For both `jsr` and `ret`, when $j \in \mathit{Dom}(S_{pc})$ we can assume that $S_{pc}[j] = S_{pc'}[j]$. For `jsr`, this follows from the stackless static semantics (line 6, `jsr` rule, Figure 13). For `ret`, this follows from Lemma 2; to apply Lemma 2, we need that $\rho = pc' \cdot \rho'$, which follows from the structured dynamic semantics of `ret`.

  Given $S_{pc}[j] = S_{pc'}[j]$ for $j \in \mathit{Dom}(S_{pc})$, $S_{pc'}[j] =$ `(ret-from` L`)`, and the *GoodStackRets* assumption, we can conclude:

$$j \in \mathit{Dom}(S_{pc}) \Rightarrow \mathit{GoodRet}(L, s[j], \Lambda_P(\rho), \rho)$$

  Given this and the fact that $s[j] = s'[j]$ for $j \in \mathit{Dom}(S_{pc})$ (which follows from the structured dynamic semantics of `jsr` and `ret`), we can conclude:

$$j \in \mathit{Dom}(S_{pc}) \Rightarrow \mathit{GoodRet}(L, s'[j], \Lambda_P(\rho), \rho) \tag{14}$$

  Now we return to proving (13) for the cases of interest:

– In the case of `jsr` K, we do a case split on j:

1. **Case:** $j \notin Dom(S_{pc})$, that is, j is the index of the top of the stack. In this case we know from the stackless static semantics of `jsr` K that $K = L$ (line 6, `jsr` rule, Figure 13), we know from the structured dynamic semantics that $s'[j] = pc + 1$ and $\rho' = (pc + 1) \cdot \rho$, and we know by Lemma 9 that $\Lambda_P(\rho') = L \cdot \Lambda_P(\rho)$. Given these equations, (13) follows by rule (`gr1`).
2. **Case:** $j \in Dom(S_{pc})$. In this case, we know from the stackless static semantics of `jsr` K that $K \neq L$ (line 7, `jsr` rule, Figure 13). We know from the structured dynamic semantics that $\rho' = (pc + 1) \cdot \rho$, and we know from Lemma 9 that $\Lambda_P(\rho') = K \cdot \Lambda_P(\rho)$. Given these equations and (14), (13) follows from rule (`gr2`).

– In the case of `ret`, we know from Lemma 9 and the structured dynamic semantics that:

$$\Lambda_P(\rho) = K \cdot \Lambda_P(\rho')$$
$$\rho = pc' \cdot \rho'$$

for some K. Given these equations, we prove (13) by cases on L:

1. **Case:** $K \neq L$. By the stackless static semantics, $Dom(S_{pc'})$ is equal to $Dom(S_{pc})$, so $j \in Dom(S_{pc})$. Given this, the conclusion of (14) must be true. Given $\Lambda_P(\rho) = L \cdot \Lambda_P(\rho')$ and $K \neq L$, the conclusion of (14) can be true only by rule (`gr2`), so we can conclude

$$GoodRet(K, s'[j], \Lambda_P(\rho'), \rho')$$

2. **Case:** $K = L$. From the *WFCallStack2* assumption and Lemma 10 we know that $\Lambda_P(\rho)$ and ρ have the same length; given this, $\Lambda_P(\rho')$ and ρ' must also have the same length. The assumption that a step of execution is possible starting from $\langle pc, f, s, \rho\rangle$ implies that ρ is not empty, thus the *WFCallStack2* assumption must be true by rule (`wf21`), so $L \notin \Lambda_P(\rho')$. Thus, (13) follows from Lemma 11.

□

B.2.4 Preservation of *GoodFrameRets*

Restatement of Part of Lemma 8 *For all P, F, and S such that $F, S \vdash P$:*

$$
\begin{array}{l}
\forall pc, f, s, \rho, pc', f', s', \rho'. \\
\qquad \wedge\ P \vdash_s \langle pc,\ f,\ s,\ \rho\rangle \rightarrow \langle pc',\ f',\ s',\ \rho'\rangle \\
\qquad \wedge\ \mathit{WFCallStack}(P, F, pc, \rho) \\
\qquad \wedge\ \mathit{WFCallStack2}(P, pc, \Lambda_P(\rho), \rho) \\
\qquad \wedge\ \mathit{GoodStackRets}(S_{pc}, s, \Lambda_P(\rho), \rho) \\
\qquad \wedge\ \mathit{GoodFrameRets}(F, pc, \{\}, f, \Lambda_P(\rho), \rho) \\
\quad \Rightarrow \mathit{GoodFrameRets}(F, pc', \{\}, f', \Lambda_P(\rho'), \rho')
\end{array}
$$

This time, we need all conjuncts in the hypotheses.

Proof Assume that P, F, S, pc, f, s, ρ, pc', f', s', and ρ' satisfy the hypotheses of the lemma. The assumption that a step of execution is possible starting from $\langle pc,\ f,\ s,\ \rho\rangle$ implies that $P[pc]$ is defined; it also implies that $P[pc] \neq$ `halt`.

When $\rho' = \epsilon$, we have $\Lambda_P(\rho') = \epsilon$ by rule (`l0`). Thus, we can conclude

$$\mathit{GoodFrameRets}(F, pc', \{\}, f', \Lambda_P(\rho'), \rho')$$

by rule (`gfr0`).

When ρ' is not empty, we do a case split on the instruction possible at $P[pc]$:

- The first case is `inc`, `pop`, `push0`, `if`, and `load` x for any x. It follows from the stackless static semantics of these instructions that $\mathit{Dom}(F_{pc}) = \mathit{Dom}(F_{pc'})$ and, for all y in $\mathit{Dom}(F_{pc})$, $F_{pc}[y] = F_{pc'}[y]$. It follows from the structured dynamic semantics of these instructions that $f = f'$ and $\rho = \rho'$.

  Because ρ is not empty, we can assume that $\rho' = p \cdot \rho''$ for some p and ρ''. The *GoodFrameRets* assumption must be true by rule (`gfr1`), so we can conclude that

  $$
  \begin{array}{l}
  \wedge\ \forall y, L. \begin{pmatrix} \wedge\ y \in \mathit{Dom}(F_{pc}) \\ \wedge\ F_{pc}[y] = (\texttt{ret-from}\ L) \end{pmatrix} \Rightarrow \mathit{GoodRet}(L, f[y], \Lambda_P(\rho), \rho) \\
  \wedge\ \mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc}), f, \Lambda_P(\rho''), \rho'')
  \end{array}
  $$

  Using the equalities $f = f'$, $\rho = \rho'$, $\mathit{Dom}(F_{pc}) = \mathit{Dom}(F_{pc'})$ and, for y in $\mathit{Dom}(F_{pc})$, $F_{pc}[y] = F_{pc'}[y]$, we can conclude:

  $$
  \begin{array}{l}
  \wedge\ \forall y, L. \begin{pmatrix} \wedge\ y \in \mathit{Dom}(F_{pc'}) \\ \wedge\ F_{pc'}[y] = (\texttt{ret-from}\ L) \end{pmatrix} \Rightarrow \mathit{GoodRet}(L, f'[y], \Lambda_P(\rho'), \rho') \\
  \wedge\ \mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc'}), f', \Lambda_P(\rho''), \rho'')
  \end{array}
  $$

From this we can conclude

$$GoodFrameRets(F, pc', \{\}, f', \Lambda_P(\rho'), \rho')$$

by rule (`gfr1`).

- The next case is `store` x. Let v and T satisfy $s = v \cdot s'$ and $S_{pc} = T \cdot S_{pc'}$. We know these exist by the structured dynamic semantics and stackless static semantics of `store` x, respectively. We want to apply Lemma 14 to conclude:

  $$GoodFrameRets(F, pc', \{\}, f', \Lambda_P(\rho), \rho)$$

  To do so, we must discharge the antecedents of Lemma 14. The first antecedent, *WFCallStack*, we are assuming. The second and third antecedents follow from the stackless static semantics and the structured dynamic semantics of `store` x, respectively. The fourth antecedent follows from the stackless static semantics. The fifth and sixth antecedents follow from the *GoodStackRets* and *GoodFrameRets* assumptions, respectively. Thus, we can indeed apply Lemma 14 to conclude

  $$GoodFrameRets(F, pc', \{\}, f', \Lambda_P(\rho), \rho)$$

  The structured dynamic semantics of `store` x implies that $\rho = \rho'$. Substituting, we get

  $$GoodFrameRets(F, pc', \{\}, f', \Lambda_P(\rho'), \rho')$$

- The next case is `jsr` K. From Lemma 9 and the structured dynamic semantics of `jsr` K we know that

  $$\begin{gathered}\Lambda_P(\rho') = K \cdot \Lambda_P(\rho)\\ \rho' = (pc+1) \cdot \rho\\ f' = f\\ pc' = K\end{gathered}$$

  From the stackless static semantics we know that

  $$\begin{gathered}Dom(F_K) \subseteq Dom(F_{pc})\\ Dom(F_{pc+1}) = Dom(F_{pc})\end{gathered}$$

  We want to prove:

  $$GoodFrameRets(F, pc', \{\}, f', \Lambda_P(\rho'), \rho')$$

  Substituting, this is equivalent to

  $$GoodFrameRets(F, K, \{\}, f, K \cdot \Lambda_P(\rho), (pc+1) \cdot \rho)$$

To establish this via rule ($\texttt{gfr1}$), it suffices to show that

$$\forall y, L. \begin{pmatrix} \wedge\ y \in \mathit{Dom}(F_K) \\ \wedge\ F_K[y] = \texttt{(ret-from } L\texttt{)} \end{pmatrix} \\ \Rightarrow \mathit{GoodRet}(L, f[y], K \cdot \Lambda_P(\rho), (pc+1) \cdot \rho) \tag{15}$$

and also that

$$\mathit{GoodFrameRets}(F, pc+1, \mathit{Dom}(F_K), f, \Lambda_P(\rho), \rho) \tag{16}$$

To prove (15), we assume y and L such that y is in $\mathit{Dom}(F_K)$ and $F_K[y] =$ $\texttt{(ret-from } L\texttt{)}$ and show that

$$\mathit{GoodRet}(L, f[y], K \cdot \Lambda_P(\rho), (pc+1) \cdot \rho) \tag{17}$$

To show this, first we show that

$$\mathit{GoodRet}(L, f[y], \Lambda_P(\rho), \rho) \tag{18}$$

If ρ is empty, then (18) follows from rule ($\texttt{gr0}$). If ρ is not empty, then the *GoodFrameRets* assumption must hold by rule ($\texttt{gfr1}$). Given the first line of rule ($\texttt{gfr1}$) and the fact that $y \in \mathit{Dom}(F_K)$ and $F_K[y] = \texttt{(ret-from } L\texttt{)}$, (18) follows. We know $K \neq L$ from the stackless static semantics of $\texttt{jsr}\ K$ (line 8, Figure 13). Given this and (18), we can conclude (17) by rule ($\texttt{gr2}$).

Next, we need to prove (16). If ρ is empty, then (16) follows from rule ($\texttt{gfr0}$). Otherwise, we assume that $\rho = p \cdot \rho''$ for some p and ρ''. Given $\rho = p \cdot \rho''$, to establish (16) via rule ($\texttt{gfr1}$), it suffices to show that

$$\forall y, L. \begin{pmatrix} \wedge\ y \in (\mathit{Dom}(F_{pc+1}) \backslash \mathit{Dom}(F_K)) \\ \wedge\ F_{pc+1}[y] = \texttt{(ret-from } L\texttt{)} \end{pmatrix} \Rightarrow \mathit{GoodRet}(L, f[y], \Lambda_P(\rho), \rho) \tag{19}$$

and also that

$$\mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc+1}), f, \Lambda_P(\rho''), \rho'') \tag{20}$$

Because $\rho \neq \epsilon$, the *GoodFrameRets* assumption must be true by rule ($\texttt{gfr1}$), so we can conclude that

$$\forall y, L. \begin{pmatrix} \wedge\ y \in \mathit{Dom}(F_{pc}) \\ \wedge\ F_{pc}[y] = \texttt{(ret-from } L\texttt{)} \end{pmatrix} \Rightarrow \mathit{GoodRet}(L, f[y], \Lambda_P(\rho), \rho) \tag{21}$$

and also that

$$\mathit{GoodFrameRets}(F, p, \mathit{Dom}(F_{pc}), f, \Lambda_P(\rho''), \rho'') \tag{22}$$

To prove (20), we simply observe that, because $Dom(F_{pc+1}) = Dom(F_{pc})$, (22) is equivalent to (20).

To prove (19), we assume y and L such that y is in $Dom(F_{pc+1}) \backslash Dom(F_K)$ and $F_{pc+1}[y]$ equals `(ret-from` L`)` and show that

$$GoodRet(L, f[y], \Lambda_P(\rho), \rho) \tag{23}$$

From the stackless static semantics of `jsr` K (line 4, `jsr` rule, Figure 13), because y is in $Dom(F_{pc+1}) \backslash Dom(F_K)$, we can conclude that $F_{pc}[y]$ equals $F_{pc+1}[y]$ which in turn equals `(ret-from` L`)`. Also, given $Dom(F_{pc+1}) = Dom(F_{pc})$, we can conclude that $y \in Dom(F_{pc})$. Given (21), $y \in Dom(F_{pc})$, and $F_{pc}[y] =$ `(ret-from` L`)`, we can conclude (23).

- The last case is `ret` x. From Lemma 9 and the structured dynamic semantics of `ret` x we know that, for some K:

$$\begin{gathered} \Lambda_P(\rho) = K \cdot \Lambda_P(\rho') \\ \rho = pc' \cdot \rho' \\ f' = f \end{gathered}$$

We want to prove:

$$GoodFrameRets(F, pc', \{\}, f', \Lambda_P(\rho'), \rho')$$

If ρ' is empty, then this follows from rule (`gfr0`).

Otherwise, we have that $\rho' = p \cdot \rho''$ for some p and ρ''. We are assuming that

$$GoodFrameRets(F, pc, \{\}, f, \Lambda_P(\rho), \rho)$$

Applying the equations above, this is equivalent to

$$GoodFrameRets(F, pc, \{\}, f', K \cdot \Lambda_P(\rho'), pc' \cdot \rho')$$

This must be true by rule (`gfr1`), so we can conclude that

$$\begin{array}{l} \forall y, L. \begin{pmatrix} \wedge\ y \in Dom(F_{pc}) \\ \wedge\ F_{pc}[y] = \texttt{(ret-from } L\texttt{)} \end{pmatrix} \\ \qquad \Rightarrow GoodRet(L, f'[y], K \cdot \Lambda_P(\rho'), pc' \cdot \rho') \end{array} \tag{24}$$

and also that

$$GoodFrameRets(F, pc', Dom(F_{pc}), f', \Lambda_P(\rho'), \rho') \tag{25}$$

Because $\rho' = p \cdot \rho''$, (25) must be true by rule (`gfr1`), so

$$\forall y, L. \begin{pmatrix} \wedge\ y \in (Dom(F_{pc'}) \backslash Dom(F_{pc})) \\ \wedge\ F_{pc'}[y] = \texttt{(ret-from}\ L\texttt{)} \end{pmatrix} \Rightarrow \mathit{GoodRet}(L, f'[y], \Lambda_P(\rho'), \rho') \tag{26}$$

and also

$$\mathit{GoodFrameRets}(F, p, Dom(F_{pc'}), f', \Lambda_P(\rho''), \rho'') \tag{27}$$

Because $\rho \neq \epsilon$, the *WFCallStack* assumption must be true by rule (`wf1`), so $P[pc' - 1] = \texttt{jsr}\ M$ for M such that $R_{P,pc} = \{M\}$. Given this, by the stackless static semantics of `ret` x (lines 2 and 5, `ret` rule, Figure 13), we know that, for y in $Dom(F_{pc})$, $F_{pc'}[y] = F_{pc}[y]$. Combining this with (24), we get

$$\forall y, L. \begin{pmatrix} \wedge\ y \in Dom(F_{pc}) \\ \wedge\ F_{pc'}[y] = \texttt{(ret-from}\ L\texttt{)} \end{pmatrix}$$
$$\Rightarrow \mathit{GoodRet}(L, f'[y], K \cdot \Lambda_P(\rho'), pc' \cdot \rho')$$

Using Lemma 12 and the *WFCallStack2* assumption, we can conclude

$$\forall y, L. \begin{pmatrix} \wedge\ y \in Dom(F_{pc}) \\ \wedge\ F_{pc'}[y] = \texttt{(ret-from}\ L\texttt{)} \end{pmatrix} \Rightarrow \mathit{GoodRet}(L, f'[y], \Lambda_P(\rho'), \rho') \tag{28}$$

Combining (26) and (28), we can conclude

$$\forall y, L. \begin{pmatrix} \wedge\ y \in Dom(F_{pc'}) \\ \wedge\ F_{pc'}[y] = \texttt{(ret-from}\ L\texttt{)} \end{pmatrix} \Rightarrow \mathit{GoodRet}(L, f'[y], \Lambda_P(\rho'), \rho') \tag{29}$$

Given (29) and (27), we can use rule (`gfr1`) to conclude

$$\mathit{GoodFrameRets}(F, pc', \{\}, f', \Lambda_P(\rho'), \rho')$$

□

C Proof of main soundness theorem

To prove Theorem 4, our main soundness theorem, we use two lemmas. This section first states these lemmas, proves the main soundness theorem, then proves the two lemmas.

The first lemma says that well-typed programs get "stuck" only at `halt` instructions:

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

$$\begin{array}{l} \forall pc, f, s, \rho. \\ \qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\ \qquad \wedge\ \forall y.\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\ \qquad \wedge\ s : S_{pc} \\ \qquad \wedge\ pc \in \mathit{Dom}(P) \\ \qquad \wedge\ P[pc] \neq \mathtt{halt} \\ \quad \Rightarrow \exists pc', f', s'.\ P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle \end{array}$$

The next lemma takes invariants stated about individual steps of execution and extends them over multi-step execution sequences:

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

$$\begin{array}{l} \forall pc, f_0, f, s. \\ \qquad P \vdash \langle 1,\ f_0,\ \epsilon\rangle \rightarrow^* \langle pc,\ f,\ s\rangle \\ \quad \Rightarrow \exists \rho. \\ \qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\ \qquad \wedge\ \forall y.\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\ \qquad \wedge\ s : S_{pc} \\ \qquad \wedge\ pc \in \mathit{Dom}(P) \end{array}$$

We use these lemmas to prove the main soundness theorem:

Restatement of main soundness theorem *For all P, F, and S such that $F, S \vdash P$:*

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

Proof Assume that P, F, S, pc, f, and s satisfy the hypotheses of this theorem. Under this assumption, the second conjunct of the conclusion ($s : S_{pc}$) follows directly from Lemma 16.

We prove the first conjunct of the conclusion by contradiction. Assume that $P[pc] \neq \mathtt{halt}$. Our previous assumptions about P, F, S, pc, f, and s imply the hypotheses of Lemma 16, so we can conclude that there exists ρ such that P, F, S, pc, f, s, and ρ satisfy all hypotheses of Lemma 15. However, the conclusion of Lemma 15 contradicts our assumption that

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

Thus, we are forced to conclude that $P[pc] = \mathtt{halt}$. $\square$

C.1 Making progress

Lemma 15 says that a well-typed program does not get stuck unless it hits a `halt` instruction:

Restatement of Lemma 15 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l} \forall pc, f, s, \rho. \\ \qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\ \qquad \wedge\ \forall y.\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\ \qquad \wedge\ s : S_{pc} \\ \qquad \wedge\ pc \in \mathit{Dom}(P) \\ \qquad \wedge\ P[pc] \neq \mathtt{halt} \\ \quad \Rightarrow \exists pc', f', s'.\ P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle \end{array}$$

Proof Assume that P, F, S, pc, f, s, and ρ satisfy the hypotheses of the lemma. We proceed with a case split on the instruction at $P[pc]$, constructing pc', f', and s' satisfying the stackless dynamic semantics of the instruction at $P[pc]$.

- For `load` x, `push0`, and `jsr` L, progress can always be made. For `load` x and `push0`, $s' = s$, $f' = f$, and $pc' = pc + 1$. For `jsr` L, $s' = (L + 1) \cdot s$, $f' = f$, and $pc' = pc + 1$.

- For `inc`, `if` L, `pop`, and `store` x, progress can be made when the stack has at least one value of the appropriate type in it. The assumption that $s : S_{pc}$ plus the static constraints on S_{pc} for each instruction ensure that the stack does indeed have an appropriately typed value on top. For all instructions, assume $s = v \cdot s''$ for some v and s''. For `inc`, `if` L, and `pop`, we take $f' = f$; for `store` x, we take $f' = f[x \mapsto v]$. For `if` L, `pop`, and `store` x, we take $s' = s''$. For `inc`, v must be some integer n, and we take $s' = (n+1) \cdot s''$. For `inc`, `pop`, and `store` x, $pc' = pc + 1$. For `if` L, v must be some integer n, and we take $pc' = pc + 1$ if n is zero and we take $pc' = L$ for other values of n.

- For `ret` x, progress can be made in states where $f[x]$ is an address. The assumption that
$$\exists T.\ \mathcal{F}(F, pc, \rho)[x] = T \wedge f[x] : T$$
and the stackless static semantics of `ret` x imply that $f[x]$ does contain an address. Thus, progress is possible, and we take $pc' = f[x]$, $f' = f$, and $s' = s$.

□

C.2 Chained soundness theorem

Before proving Lemma 16, we state and prove one more invariant about individual steps of execution:

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

$$\begin{array}{l} \forall pc, f, s, \rho, pc', f', s'. \\ \qquad \wedge\ P \vdash \langle pc,\ f,\ s\rangle \rightarrow \langle pc',\ f',\ s'\rangle \\ \qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\ \quad \Rightarrow pc' \in \mathit{Dom}(P) \end{array}$$

Proof Assume that P, F, S, pc, f, s, and ρ satisfy the hypotheses of the lemma. The assumption that a step of execution is possible starting from $\langle pc,\ f,\ s\rangle$ implies that $P[pc]$ is defined; it also implies that $P[pc] \neq$ `halt`. We do a case split on the instruction possible at $P[pc]$:

- For `pop`, `push0`, `inc`, `load`, and `store`, $pc' = pc + 1$, and $pc + 1$ is constrained by the stackless static semantics of these instructions to be in $\mathit{Dom}(P)$.

- For `if` L, pc' equals either $pc + 1$ or L, depending on which way the branch goes. Both $pc + 1$ and L are constrained by the stackless static semantics of `if` L to be in $\mathit{Dom}(P)$.

- For `jsr` L, $pc' = L$, and L is constrained by the stackless static semantics of `jsr` L to be in $\mathit{Dom}(P)$.

- For `ret` x, we first show that our assumptions imply that ρ has at least one element. If ρ had no elements, then the *WFCallStack2* part of the *Consistent* assumption would have to be true by rule (`wf20`). This would require that $C_{P,pc}$ be empty, but the stackless static semantics for `ret` x implies that $C_{P,pc}$ must have at least one element. Thus, *WFCallStack2* cannot hold by rule (`wf20`). Instead, it must hold by rule (`wf21`), so ρ must have at least one element.

  Let $\rho = p \cdot \rho'$ for some p and ρ'. Using Lemma 7, we have that $pc' = p$. The *WFCallStack* part of the *Consistent* assumption must be true by rule (`wf1`), so $P[p - 1] =$ `jsr` L for some L. From the stackless static semantics of `jsr` L we have that $(i+1) \in \mathit{Dom}(P)$ for all i such that $P[i] =$ `jsr` L. Thus, substituting $p - 1$ for i, we can conclude that p, and thus pc', is in $\mathit{Dom}(P)$.

□

Lemma 16 applies Lemma 17 and Theorem 3 to multi-step executions starting from the initial state:

Restatement of Lemma 16 *For all P, F, and S such that $F, S \vdash P$:*

$$\begin{array}{l}
\forall pc, f_0, f, s. \\
\quad P \vdash \langle 1,\ f_0,\ \epsilon\rangle \rightarrow^* \langle pc,\ f,\ s\rangle \\
\quad \Rightarrow \exists \rho. \\
\qquad \wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\
\qquad \wedge\ \ \forall y.\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\
\qquad \wedge\ \ s : S_{pc} \\
\qquad \wedge\ pc \in \mathit{Dom}(P)
\end{array}$$

Proof Assume P, F, and S such that $F, S \vdash P$ and proceed by induction on the number of execution steps in $P \vdash \langle 1,\ f_0,\ \epsilon\rangle \rightarrow^* \langle pc,\ f,\ s\rangle$:

- In the base case, $\langle pc,\ f,\ s\rangle$ equals $\langle 1,\ f_0,\ \epsilon\rangle$, that is, no steps of execution are taken. In Section 3, it is assumed that $P[1]$ is defined for all programs, so we can conclude $pc \in \mathit{Dom}(P)$. We pick $\rho = \epsilon$ and observe that:

$$\begin{array}{l}
\wedge\ \mathit{Consistent}(P, F, S, 1, f_0, \epsilon, \epsilon) \\
\wedge\ \forall y.\ \exists T.\ \mathcal{F}(F, 1, \epsilon)[y] = T \wedge f_0[y] : T \\
\wedge\ \epsilon : S_1
\end{array}$$

  Given the values of F_1, S_1, and $C_{P,1}$ it is not hard to check this.

- In the inductive step, let $\langle pc_n,\ f_n,\ s_n\rangle$ be a state such that

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

  By induction, we know that there exists a ρ_n such that:

$$\begin{array}{l}
\wedge\ \mathit{Consistent}(P, F, S, pc_n, f_n, s_n, \rho_n) \\
\wedge\ \forall y.\ \exists T.\ \mathcal{F}(F, pc_n, \rho_n)[y] = T \wedge f_n[y] : T \\
\wedge\ s_n : S_{pc_n} \\
\wedge\ pc_n \in \mathit{Dom}(P)
\end{array}$$

  Our assumptions and these conjuncts satisfy the hypotheses of Theorem 3 and Lemma 17, so we can conclude that

$$\begin{array}{l}
\wedge\ \mathit{Consistent}(P, F, S, pc, f, s, \rho) \\
\wedge\ \forall y.\ \exists T.\ \mathcal{F}(F, pc, \rho)[y] = T \wedge f[y] : T \\
\wedge\ s : S_{pc} \\
\wedge\ pc \in \mathit{Dom}(P)
\end{array}$$

$\square$

Acknowledgements

We thank Luca Cardelli, Drew Dean, Sophia Drossopoulou, Stephen Freund, Cynthia Hibbard, Mark Lillibridge, Greg Morrisett, George Necula, Frank Yellin, and anonymous referees 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 1997.

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

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. Technical Report STAN-CS-TN-98-62, Department of Computer Science, Stanford University, April 1998. To appear in *Proceedings of OOPSLA '98*.

[Gol97] Allen Goldberg. A specification of Java loading and bytecode verification. To appear in *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, 1998; Web page at `http://www.kestrel.edu/~goldberg/Bytecode.html`, 1997.

[HT98] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. On the Web at `http://nicosia.is.s.u-tokyo.ac.jp/members/hagiya.html`; a preliminary version appeared in SIG-Notes, PRO-17-3, Information Processing Society of Japan, pages 13–18, 1998.

[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, 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.

[Nv98] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe—definitely. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 161–170, January 1998.

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

[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). 1997.

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