
*Journal of Automated Reasoning* **30:** 271–321, 2003.



# A Type System for the Java Bytecode Language and Verifier

STEPHEN N. FREUND[1] and JOHN C. MITCHELL[2]
[1] *Williams College*
[2] *Stanford University*

**Abstract.** The Java Virtual Machine executes bytecode programs that may have been sent from other, possibly untrusted, locations on the network. Since the transmitted code may be written by a malicious party or corrupted during network transmission, the Java Virtual Machine contains a bytecode verifier to check the code for type errors before it is run. As illustrated by reported attacks on Java run-time systems, the verifier is essential for system security. However, no formal specification of the bytecode verifier exists in the Java Virtual Machine Specification published by Sun. In this paper, we develop such a specification in the form of a type system for a subset of the bytecode language. The subset includes classes, interfaces, constructors, methods, exceptions, and bytecode subroutines. We also present a type checking algorithm and prototype bytecode verifier implementation, and we conclude by discussing other applications of this work. For example, we show how to extend our formal system to check other program properties, such as the correct use of object locks.

**Key words:** Java Virtual Machine, bytecode verification, type systems.

## 1. Introduction

The Java bytecode language, which we refer to as JVML, is the platform independent representation of compiled Java programs. In order to prevent devious applets or improperly structured code from causing security problems, the Java Virtual Machine bytecode verifier performs a number of consistency checks on bytecode before it is executed [29]. These consistency checks reject any code that may cause a type error when executed, thereby preventing bytecode programs from exploiting type errors to circumvent the Java security architecture.

Several published attacks on various implementations of the Java Virtual Machine illustrate the importance of the bytecode verifier for system security. To cite one specific example, a bug in an early version of Sun's bytecode verifier allowed applets to create certain system objects that they should not have been able to create, such as `ClassLoaders` [11]. The problem was caused by an error in how constructors were verified and resulted in the ability to potentially compromise the security of the entire system. In a previous study, we identified another error in the Sun verifier that allowed JVML code to use an object that was not properly initialized [18]. A number of other loopholes and inconsistencies have also been

reported (see, for example, [40, 12, 46, 6]). Problems like these demonstrate the need for a correct and formal specification of the bytecode verifier. When we started this work, however, the only existing specification was an informal English description that was incomplete and incorrect in some respects.

The primary contribution of this paper is a sound type system for a large fragment of JVML. We study the subset of JVML that both captures the most difficult static analysis problems in bytecode verification and brings to light the subtle interactions between features previously examined in isolation. In particular, we focus on the following JVML features:

- classes, interfaces, and objects;
- constructors and object initialization;
- virtual and interface method invocation;
- arrays;
- exceptions and subroutines; and
- integer and float primitive types.

In an earlier study, we examined object initialization and formalized the way in which a type system may prevent Java bytecode programs from using objects before they have been initialized [18]. In this work, we extend our formal system to handle bytecode subroutines and the rest of the features listed above.

Subroutines are a form of local call and return that allow for space-efficient compilation of `try-finally` structures in the Java language. Bytecode programs that use subroutines are allowed to manipulate return addresses in certain ways, and the bytecode verifier must ensure that these return addresses are used appropriately. In addition, subroutines introduce a limited form of polymorphism into the type system. Our treatment of subroutines is based on the semantics developed by Stata and Abadi [43], with modifications to be closer to the original Sun specification and to include exception handlers.

While we cover most major features of JVML, we do omit some pieces, such as additional primitive types and control structures, `final` and `static` modifiers, access levels, class initialization, concurrency, and packages. These omitted features would contribute to the complexity of our system only in the sheer number of cases that they introduce, and they do not introduce any challenging or new problems. For example, the excluded primitive types and operations on them are all similar to cases in our study, as are the missing control structures, such as the `tableswitch` instruction. Static methods share much in common with normal methods, and checks for proper use of `final` and other access modifiers are well understood and straightforward to include.

We also present a type checking algorithm for our type system and describe our experiences with a prototype verifier based on this algorithm. Our core algorithm employs dataflow analysis to construct valid types for JVML programs. Unfortunately, subroutines complicate matters by introducing polymorphism and making control flow graph construction difficult. Therefore, we divide the algorithm into

three phases that compute the control flow graph, identify uses of polymorphism, and then synthesize the type information via dataflow analysis.

Our type system and checker have several applications beyond bytecode verification. For example, we extend our system to check additional safety properties, such as ensuring that object locks are acquired and released properly by each method. We also augment the verifier to track more information through the type system to determine where run-time checks, such as `null` pointer and array bounds tests, may be eliminated.

Before proceeding, we summarize a few notational conventions used throughout this paper. Function update, written $f[x \mapsto v]$, is defined as

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

for all $y \in Dom(f)$. The related operation $[b/a]f$ changes $f$ in the following way:

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

for all $y \in Dom(f)$.

We use sequences to model stacks, as well as lists of values and types. The empty sequence is $\epsilon$, and $v \cdot s$ places $v$ at the front of sequence $s$. Appending one sequence to another is written as $s_1 \bullet s_2$, and the substitution $[b/a]s$ replaces all occurrences of $a$ in $s$ with $b$.

## 2. JVML$_f$

In this section, we informally introduce JVML$_f$, an idealized subset of JVML encompassing the features listed in the introduction. A Java compiler translates a source program into a collection of *class file*s, one for each declared class or interface. In addition to containing the bytecode instructions for methods, a class file contains the symbolic name and type information for any class, interface, method, or field mentioned in the source code. The Java Virtual Machine uses this information to verify code and resolve references. Since the binary format of class files is not easy to read or manipulate, we shall represent their contents in a more manageable form closer to the declarations in the original source program. We illustrate the compilation process and our class file representation by showing a Java program and its translation into JVML$_f$ declarations in Figure 1 and Figure 2, respectively. We define JVML$_f$ more precisely below.

The collection of declarations in Figure 2 contains the `Object` and `Throwable` Java library classes so that all referenced classes are present. We assume that these are the only two library classes and that they are present in all JVML$_f$ programs. The JVML$_f$ declaration for each class contains the set of instance fields for objects of that class, the interfaces declared to be implemented by the class, and all the methods declared in the class. Each method consists of an array of instructions and

```
interface Foo {
  int foo(int y);
}

class A extends Object
        implements Foo {
  int num;
  A(int x) {
    num = x;
  }

  int foo(int y) {
    A a;
    try {
      a = new A(y);
    } catch (Throwable e) {
      num = 2;
    }
    return 6;
  }
}
```

```
class B extends A {
  A array[];
  B(int x) {
    super(x);
    num = foo(2);
  }
}
```

*Figure 1.* Declaration of several Java classes.

a list of exception handlers, and all methods in the superclass of a class are copied into the subclass unless they are overridden. Copying methods into subclasses simplifies the bookkeeping for method lookup in the formal semantics, but it does not impact the verifier requirements or specification. The special name `<init>` is given to constructors.

The execution environment for JVML$_f$ programs consists of a stack of activation records and a heap. Each activation record contains a program counter, a local operand stack, and a set of local variables. These pieces of information are not shared between different activation records, although different activation records may contain references to the same objects in the heap. Most JVML$_f$ bytecode instructions operate on the operand stack, and the `store` and `load` instructions are used to store and load intermediate values in the local variables. Constructing or deleting activation records upon method invocation or return is left to the Java Virtual Machine.

Figure 3 contains the full JVML$_f$ instruction set, and the next few paragraphs briefly describe the interesting aspects of these instructions. In Figure 3, $v$ is an integer, real number, or the special value `null`; $x$ is a local variable; $L$ is an instruction address; and $\sigma$ and $\tau$ are a class name and valid array component type, respectively. We refer the reader to the Java Virtual Machine specification for a detailed discussion of these bytecode instructions [29].

Bytecode programs use *method references*, *interface method references*, and *field references* to identify methods, interface methods, and fields from Java language programs. These *reference*s contain three pieces of information about the method or field that they describe:

```
class Object {
  super: None
  fields: { }
  interfaces: { }
  methods:
}
```

```
class Throwable {
  super: Object
  fields: { }
  interfaces: { }
  methods:
}
```

```
interface Foo {
  interfaces: { }
  methods:
    {|Foo, foo, int → int|}I
}
```

```
class A {
  super: Object
  fields: { {|A, num, int|}F }
  interfaces: { Foo }
  methods:
    {|A, <init>, int → void|}M {
      1: load 0
      2: invokespecial {|Object, <init>, ϵ → void|}M
      3: load 0
      4: load 1
      5: putfield {|A, num, int|}F
      6: return
    }
    {|A, foo, int → int|}M {
      1: new A
      2: store 2
      3: load 2
      4: load 1
      5: invokespecial {|A, <init>, int → void|}M
      6: goto 11
      7: pop
      8: load 0
      9: push 2
      10: putfield {|A, num, int|}F
      11: push 6
      12: returnval
      Exception table:  from   to  target  type
                          1     6    7     Throwable
    }
}
```

```
class B {
  super: A
  fields: { {|A, num, int|}F, {|B, array, (Array A)|}F }
  interfaces: { Foo }
  methods:
    {|B, <init>, int → void|}M {
      1: load 0
      2: load 1
      3: invokespecial {|A, <init>, int → void|}M
      4: load 0
      5: load 0
      6: push 2
      7: invokevirtual {|A, foo, int → int|}M
      8: putfield {|A, num, int|}F
      9: return
    }
    {|B, foo, int → int|}M { /* as in superclass */ }
}
```

*Figure 2.* Translation of the code from Figure 1 into JVML$_f$.

*Instruction* ::= `push` $v$ | `pop` | `store` $x$ | `load` $x$
| `add` | `ifeq` $L$ | `goto` $L$
| `new` $\sigma$
| `invokevirtual` *Method-Ref*
| `invokeinterface` *Interface-Method-Ref*
| `invokespecial` *Method-Ref*
| `getfield` *Field-Ref*
| `putfield` *Field-Ref*
| `newarray` $\tau$ | `arraylength`
| `arrayload` $\tau$ | `arraystore` $\tau$
| `throw` | `jsr` $L$ | `ret` $x$
| `return` | `returnval`

*Figure 3.* The JVML$_f$ instruction set.

- the class or interface in which it was declared,
- the field or method name, and
- its type.

For example, the bytecode instruction `putfield` $\{\!|$`A, num, int`$|\!\}_{\mathtt{F}}$ refers to the `num` instance field with type `int` declared in class `A`. In contrast to simple names, *reference*s contain enough information to

1. check uses of methods and fields without loading the class to which they belong,
2. dynamically link class files safely, and
3. provide unique symbolic names to overloaded methods and fields (overloading is resolved at compile time in Java).

The following grammar generates *reference*s:

$$\begin{aligned} \textit{Method-Ref} &::= \{\!|\textit{Class-Name}, \textit{Label}, \textit{Method-Type}|\!\}_{\mathtt{M}} \\ \textit{Interface-Method-Ref} &::= \{\!|\textit{Interface-Name}, \textit{Label}, \textit{Method-Type}|\!\}_{\mathtt{I}} \\ \textit{Field-Ref} &::= \{\!|\textit{Class-Name}, \textit{Label}, \textit{Field-Type}|\!\}_{\mathtt{F}} \end{aligned}$$

A *Field-Type* may be `int`, `float`, any class or interface name, or an array type. A *Method-Type* is a type $\alpha \rightarrow \gamma$, where $\alpha$ is a possibly empty sequence of *Field-Type*s and $\gamma$ is the return type of the function (or `void`). Figure 4 shows the grammar to generate these types, plus several additional types and type constructors used in the static semantics but not by any JVML$_f$ program. For example, the type `Top`, the supertype of all types, will be used in the typing rules, but it cannot be mentioned in a JVML$_f$ program.

A JVML$_f$ program may raise exceptions, which are objects of type `Throwable`, in two ways. First, a program may use the `throw` instruction to raise an exception. The `throw` instruction requires that the top of the stack contain an object whose class is `Throwable`, or a subclass of it. Some JVML$_f$ instructions also generate exceptions when their arguments are not valid. For example, any instruction that performs an operation on an object will generate a run-time exception when it is applied to the `null` reference. In these cases, the virtual machine creates a new

$$
\begin{array}{rrcl}
\tau \in & \textit{Type} & ::= & \textit{Ref} \mid \textit{Prim} \mid \textit{Ret} \mid \texttt{Top} \\
& \textit{Prim} & ::= & \texttt{float} \mid \texttt{int} \\
& \textit{Ref} & ::= & \textit{Simple-Ref} \mid \textit{Uninit} \mid \textit{Array} \mid \texttt{Null} \\
& \textit{Simple-Ref} & ::= & \textit{Class-Name} \mid \textit{Interface-Name} \\
& \textit{Component} & ::= & \textit{Simple-Ref} \mid \textit{Prim} \mid \textit{Array} \\
& \textit{Array} & ::= & \texttt{(Array } \textit{Component}\texttt{)} \\
& \textit{Uninit} & ::= & \texttt{(Uninit } \textit{Class-Name } i\texttt{)} \\
& \textit{Ret} & ::= & \texttt{(Ret } L\texttt{)} \\
\\
\beta \in & \textit{Type-List} & ::= & \epsilon \mid \textit{Type} \cdot \textit{Type-List} \\
\\
\alpha \to \gamma \in & \textit{Method-Type:} & ::= & \textit{Arg-List} \to \textit{Return} \\
& \textit{Return} & ::= & \textit{Simple-Ref} \mid \textit{Array} \mid \textit{Prim} \mid \texttt{void} \\
\kappa \in & \textit{Field-Type} & ::= & \textit{Simple-Ref} \mid \textit{Array} \mid \textit{Prim} \\
& \textit{Arg-List} & ::= & \epsilon \mid \textit{Field-Type} \cdot \textit{Arg-List} \\
\\
\sigma, \varphi \in & \textit{Class-Name} & & \\
\omega \in & \textit{Interface-Name} & &
\end{array}
$$

*Figure 4.* JVML$_f$ types.

`Throwable` object for the exception. Although real implementations create objects of different classes to indicate different kinds of run-time errors, the JVML$_f$ execution model generates `Throwable` objects for all cases. Generating only a single kind of exception simplifies the JVML$_f$ dynamic semantics, but it does not fundamentally change program behavior or language expressiveness.

When a program raises an exception, the virtual machine searches for an appropriate handler in the list of handlers associated with the currently executing method. An appropriate handler is one declared to protect the current instruction and to handle exceptions of the class of the object that was thrown, or some superclass of it. If the virtual machine finds an appropriate handler, execution jumps to the first instruction of the exception handler's code. Otherwise, the virtual machine pops the top activation record and repeats the process on the new topmost activation record.

## 3. Dynamic Semantics

This section gives an overview of the formal execution model for JVML$_f$ programs. Section 3.1 describes the representation of programs as environments, Section 3.2 shows how machine states are modeled, and Section 3.3 describes the semantics of the bytecode instructions.

### 3.1. ENVIRONMENTS

The JVML$_f$ semantics, as in most semantic frameworks, models the declarations in a program as an environment. As shown in Figure 5, a JVML$_f$ environment

$$\Gamma^C : \quad \textit{Class-Name} \rightharpoonup \left\langle \begin{array}{l} \texttt{super} : \textit{Class-Name} \cup \{\textit{None}\}, \\ \texttt{interfaces} : \text{set of } \textit{Interface-Name}, \\ \texttt{fields} : \text{set of } \textit{Field-Ref} \end{array} \right\rangle$$

$$\Gamma^M : \quad \textit{Method-Ref} \rightharpoonup \left\langle \begin{array}{l} \texttt{code} : \textit{Instruction}^+, \\ \texttt{handlers} : \textit{Handler}^* \end{array} \right\rangle$$

$$\Gamma^I : \ \textit{Interface-Name} \rightharpoonup \left\langle \begin{array}{l} \texttt{interfaces} : \text{set of } \textit{Interface-Name}, \\ \texttt{methods} : \text{set of } \textit{Interface-Method-Ref} \end{array} \right\rangle$$

$$\Gamma = \Gamma^C \cup \Gamma^I \cup \Gamma^M$$

*Figure 5.* Format of a JVML$_f$ program environment.

$\Gamma$ is a partial map from class names, interface names, and *Method-Ref*s to their respective definitions.

To access information about a method $M$ declared in $\Gamma$, we write $\Gamma[M]$. Similar notation is used to access interface and class declarations. If $\Gamma[M] = \langle P,\ H\rangle$ for some *Method-Ref* $M$, then $Dom(P)$ is a range $\{1, \ldots, n\}$ of addresses from the set ADDR, and $P[i]$ is the $i$th instruction in $P$. The exception handler array $H$ is a partial map from integer indexes to exception handlers. An exception handler $\langle b,\ e,\ t,\ \sigma\rangle$ catches exceptions of type $\sigma$ that occur when the program counter is in the range $[b, e)$. Control transfers to address $t$ when an exception is caught by this handler. We use the notation $M_\Gamma[i] = I$ to indicate that $\Gamma[M] = \langle P,\ H\rangle$ and $P[i] = I$.

The dynamic and static semantic rules determine properties of a program by looking up information in its environment. For example, the inference rules to conclude that one type is a subtype of another examine the declarations in an environment to determine the program's class hierarchy. Thus, we write the subtyping judgment as $\Gamma \vdash \tau_1 <: \tau_2$ to indicate that $\tau_1$ is a subtype of $\tau_2$, given a specific environment $\Gamma$. The subtyping rules, given in Figure 6, match the rules used to model subtyping in the Java language [14, 44], with extensions to cover the JVML$_f$ specific types. The JVML$_f$ dynamic semantics uses the subtyping judgment to model run-time type tests. Figure 7 summarizes the subtyping judgment form, as well as the other major judgments presented in this paper.

### 3.2. MACHINE STATE

The JVML$_f$ virtual machine's execution state is a configuration $C = A; h$, where $A$ is a stack of activation records and $h$ is a memory heap. The activation record stack is defined as follows:

$$\begin{aligned} A &::= A' \mid \langle e\rangle_{\text{exc}} \cdot A' \\ A' &::= \langle M,\ pc,\ f,\ s,\ z\rangle \cdot A' \mid \epsilon \end{aligned}$$

$$[<:_C \text{ REFL}]\ \frac{\sigma \in \textit{Class-Name}}{\Gamma \vdash \sigma <:_C \sigma} \qquad [<:_C \text{ SUPER}]\ \frac{\Gamma \vdash \sigma_1 <:_C \sigma_2 \quad \Gamma[\sigma_2].\texttt{super} = \sigma_3}{\Gamma \vdash \sigma_1 <:_C \sigma_3} \qquad [<:_I \text{ REFL}]\ \frac{\omega \in \textit{interface-Name}}{\Gamma \vdash \omega <:_I \omega}$$

$$[<:_I \text{ SUPER}]\ \frac{\Gamma \vdash \omega_1 <:_I \omega_2 \quad \omega_2 \in \Gamma[\omega_3].\texttt{interfaces}}{\Gamma \vdash \omega_1 <:_I \omega_3} \qquad [<:_A \text{ PRIM}]\ \frac{\tau \in \textit{Prim}}{\Gamma \vdash \tau <:_A \tau} \qquad [<:_A \text{ REF}]\ \frac{\Gamma \vdash \tau_1 <:_R \tau_2}{\Gamma \vdash \tau_1 <:_A \tau_2}$$

$$[<:_R \text{ CLASS}]\ \frac{\Gamma \vdash \sigma_1 <:_C \sigma_2}{\Gamma \vdash \sigma_1 <:_R \sigma_2} \qquad [<:_R \text{ INTERFACE}]\ \frac{\Gamma \vdash \omega_1 <:_I \omega_2}{\Gamma \vdash \omega_1 <:_R \omega_2} \qquad [<:_R \text{ CLASS INT}]\ \frac{\Gamma \vdash \sigma_1 <:_C \sigma_2 \quad \omega_1 \in \Gamma[\sigma_2].\texttt{interfaces} \quad \Gamma \vdash \omega_1 <:_I \omega_2}{\Gamma \vdash \sigma_1 <:_R \omega_2}$$

$$[<:_R \texttt{Null}]\ \frac{\tau \in \textit{Simple-Ref} \cup \textit{Array} \cup \{\texttt{Null}\}}{\Gamma \vdash \texttt{Null} <:_R \tau} \qquad [<:_R \text{ ARRAY ARRAY}]\ \frac{\Gamma \vdash \tau <:_A \tau'}{\Gamma \vdash \texttt{(Array } \tau\texttt{)} <:_R \texttt{(Array } \tau'\texttt{)}} \qquad [<:_R \text{ INT } \texttt{Object}]\ \frac{\omega \in \textit{Interface-Name}}{\Gamma \vdash \omega <:_R \texttt{Object}}$$

$$[<:_R \text{ ARRAY } \texttt{Object}]\ \frac{\texttt{(Array } \tau\texttt{)} \in \textit{Array}}{\Gamma \vdash \texttt{(Array } \tau\texttt{)} <:_R \texttt{Object}} \qquad [<: \text{ REF}]\ \frac{\Gamma \vdash \tau_1 <:_R \tau_2}{\Gamma \vdash \tau_1 <: \tau_2} \qquad [<: \text{ REFL}]\ \frac{\tau \in \textit{Uninit} \cup \textit{Prim} \cup \textit{Ret}}{\Gamma \vdash \tau <: \tau}$$

$$[<: \texttt{Top}]\ \frac{\tau \in \textit{Ref} \cup \textit{Prim} \cup \textit{Ret} \cup \{\texttt{Top}\}}{\Gamma \vdash \tau <: \texttt{Top}} \qquad [<: \epsilon]\ \frac{}{\Gamma \vdash \epsilon <: \epsilon} \qquad [<: \text{ SEQ}]\ \frac{\Gamma \vdash \alpha_1 <: \alpha_2 \quad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash \tau_1 \cdot \alpha_1 <: \tau_2 \cdot \alpha_2}$$

$$[<: \text{ MAP}]\ \frac{\textit{Dom}(F_1) = \textit{Dom}(F_2) \quad \forall y \in \textit{Dom}(F_1).\ \Gamma \vdash F_1[y] <: F_2[y]}{\Gamma \vdash F_1 <: F_2}$$

*Figure 6.* JVML$_f$ subtyping rules.

Each part of the activation record $\langle M,\ pc,\ f,\ s,\ z\rangle$ has the following meaning:

- $M$: the *Method-Ref* of the method.
- $pc$: the address of the next instruction to be executed in the method's code array.
- $f$: a map from VAR, the set of local variables, to values.
- $s$: the operand stack.
- $z$: initialization information for the object being initialized in a constructor, as described below.

Records of the form $\langle e\rangle_{\text{exc}}$ are for exception handling. When a program throws an exception, the virtual machine places an activation record of the form $\langle e\rangle_{\text{exc}}$ onto the stack, where $e$ is a reference to a `Throwable` object.

| | |
|---|---|
| $\Gamma \vdash \tau_1 <: \tau_2$ | $\tau_1$ is a subtype of $\tau_2$. |
| $\Gamma \vdash C_0 \rightarrow C_1$ | A program in configuration $C_0$ evaluates to $C_1$ in a single step. |
| $\Gamma \vdash$ `wt` | $\Gamma$ is well-formed. |
| $\Gamma \vdash d$ `ty` | The definition of $d$ only refers to names of classes or interfaces defined in $\Gamma$. |
| $\Gamma \vdash d\ K$ | The declaration of $d$ is a valid $K$, where $K \in \{$`class`, `interface`, `method`$\}$. |
| $\Gamma, F, S \vdash P, H : M$ | The method body $\langle P,\ H \rangle$ conforms to *Method-Ref* $M$ given $\Gamma$, $F$, and $S$. |
| $\Gamma, F, S, i \vdash P : M$ | Instruction $i$ of $P$ is well-typed given $\Gamma$, $F$, and $S$. |
| $\Gamma, F, S \vdash h$ handles $P$ | Handler $h$ is well-typed given $\Gamma$, $F$, and $S$. |
| $\Gamma \vdash h$ `wt` | $h$ is a well-typed heap. |
| $\Gamma, h \vdash v : \tau$ | The value $v$ has type $\tau$ given environment $\Gamma$ and heap $h$. |
| $\Gamma \vdash C$ `wt` | The configuration $C$ is well-typed in environment $\Gamma$. |

*Figure 7.* Summary of the main JVML$_f$ logical judgments.

The heap contains all objects and arrays allocated by the program, and we model the heap as a function $h$ mapping locations to records. Records for objects and arrays have different forms, but all records contain tags to support run-time type tests and method dispatch. Objects of class $\sigma$ have the following form:

$$\langle\!\langle \{\!|\sigma_1,\ l_1,\ \kappa_1|\!\}_{\mathrm{F}} = v_1, \ldots, \{\!|\sigma_n,\ l_n,\ \kappa_n|\!\}_{\mathrm{F}} = v_n \rangle\!\rangle_{\sigma}$$

An object's fields will always be indexed by a *Field-Ref*. We often abbreviate this type of record as $\langle\!\langle \{\!|\sigma_i,\ l_i,\ \kappa_i|\!\}_{\mathrm{F}} = v_i \rangle\!\rangle_{\sigma}^{i \in I}$ where the subscript $i \in I$ refers to the $i$th field in the record. Arrays are stored in a similar type of record:

$$[[v_i]]^{i \in [0..n-1]}_{(\texttt{Array}\ \tau)}$$

With these definitions, a program's heap $h$ is a partial map from locations drawn from the set LOC to records:

$$\begin{array}{ll} h : \mathrm{LOC} \rightarrow & \\ \quad \langle\!\langle \{\!|\sigma_i,\ l_i,\ \kappa_i|\!\}_{\mathrm{F}} = v_i \rangle\!\rangle_{\sigma}^{i \in I} & \text{(object)} \\ \quad | \ \langle\!\langle \{\!|\sigma_i,\ l_i,\ \kappa_i|\!\}_{\mathrm{F}} = v_i \rangle\!\rangle_{\varphi \Diamond (\texttt{Uninit}\ \sigma\ j)}^{i \in I} & \text{(uninit. object)} \\ \quad | \ [[v_i]]^{i \in [0..n-1]}_{(\texttt{Array}\ \tau)} & \text{(array)} \end{array}$$

The middle form is for uninitialized objects, and it is described in Section 3.3.1. We shall use the notation $h[a].\{\!|\sigma,\ l,\ \kappa|\!\}_{\mathrm{F}}$ to access the value of field $\{\!|\sigma,\ l,\ \kappa|\!\}_{\mathrm{F}}$ from the instance at location $a$ in heap $h$, and we create a new heap with a modified value for that field using the notation $h[a.\{\!|\varphi,\ l,\ \kappa|\!\}_{\mathrm{F}} \mapsto v]$. The function *Tag* returns the tag of a heap record, or any other run-time value:

$$\mathit{Tag}(h, v) = \begin{cases} \texttt{int} & \text{if } v \text{ is an integer,} \\ \texttt{float} & \text{if } v \text{ is a float,} \\ \texttt{Null} & \text{if } v = \texttt{null,} \\ \tau & \text{if } v \in \mathrm{LOC} \text{ and } h[v] = \langle\!\langle \ldots \rangle\!\rangle_{\tau} \text{ or } h[v] = [[\ldots]]_{\tau}. \end{cases}$$

The semantics use the *Blank* function to create new records. The function is defined separately for each of the three kinds of records:

$$\begin{aligned} \mathit{Blank}(\sigma) &= \langle\!\langle \{\!|\sigma_i,\ l_i,\ \kappa_i|\!\}_{\mathtt{F}} = \mathit{Zero}(\kappa_i) \rangle\!\rangle_{\sigma}^{i\in I}, \\ \mathit{Blank}(\sigma \Diamond (\mathtt{Uninit}\ \sigma'\ j)) &= \langle\!\langle \{\!|\sigma_i,\ l_i,\ \kappa_i|\!\}_{\mathtt{F}} = \mathit{Zero}(\kappa_i) \rangle\!\rangle_{\sigma \Diamond (\mathtt{Uninit}\ \sigma'\ j)}^{i\in I}, \\ \mathit{Blank}((\mathtt{Array}\ \tau), n) &= [[v_i = \mathit{Zero}(\tau)]]_{(\mathtt{Array}\ \tau)}^{i\in[0..n-1]}, \end{aligned}$$

where $\Gamma[\sigma].\mathtt{fields} = \{\{\!|\sigma_i,\ l_i,\ \kappa_i|\!\}_{\mathtt{F}}\}^{i\in I}$. The *Zero* function computes default values for each type:

$$\mathit{Zero}(\tau) = \begin{cases} 0 & \text{if } \tau = \mathtt{int}, \\ 0.0 & \text{if } \tau = \mathtt{float}, \\ \mathtt{null} & \text{if } \tau \in \mathit{Ref}. \end{cases}$$

### 3.3. OPERATIONAL SEMANTICS

We model the behavior of JVML$_f$ instructions as transitions on machine states in the standard framework of operational semantics. The semantic rules capture all possible behaviors of JVML$_f$ programs, and we present the rules in Figures 8–10. Each row in these tables describes the conditions under which a program represented by $\Gamma$ can move from configuration $C_0$ to configuration $C_1$. The first column indicates the instruction form captured by the rule. If the instruction about to be executed matches that form and all conditions in the second column are satisfied, then a step may be made from a configuration matching the pattern in the third column to a configuration matching the pattern in the last column.

For example, Figure 8 contains the rule for the `getfield` instruction, which pops an object reference off the stack and pushes the value stored in the specified field of that object. The rule indicates that the program represented by $\Gamma$ may move from configuration $C_0$ to $C_1$ in a single step if $M_\Gamma[pc] = \mathtt{getfield}\{\!|\varphi,\ l,\ \kappa|\!\}_{\mathtt{F}}$, and $C_0$ and $C_1$ match the patterns in the table. In this rule, and all others, if we apply a function $f$ to an argument $x$, we have the implicit requirement that $x \in \mathit{Dom}(f)$. We describe several other representative rules from these tables.

The rule for `arraystore` verifies that the array reference is valid, that the index is in bounds, and that the type of the value being stored is a subtype of the type of elements stored in the array. This last check is required because of a potential type loophole caused by the covariant subtyping of arrays.

The rule for `invokevirtual` uses the run-time tag of the receiver to construct the *Method-Ref* for the new activation record when a method is called. The notation $f[1..n \mapsto v_n \cdot v_{n-1} \cdots v_1 \cdot \epsilon]$ is an abbreviation for

$$f[1 \mapsto v_1][2 \mapsto v_2] \cdots [n \mapsto v_n].$$

The function $f_0$ maps the local variables to arbitrary values. In contrast to the Java Virtual Machine Specification, the receiver and arguments are left on the

$\Gamma \vdash C_0 \to C_1$

| $M_\Gamma[pc]$ | Cond | $C_0$ | $C_1$ |
|---|---|---|---|
| `push` $v$ | | $\langle M,\ pc,\ f,\ s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ v \cdot s,\ z\rangle \cdot A; h$ |
| `pop` | | $\langle M,\ pc,\ f,\ v \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ s,\ z\rangle \cdot A\ ; h$ |
| `add` | $Tag(h, v_1) = Tag(h, v_2) = \tau$ | $\langle M,\ pc,\ f,\ v_1 \cdot v_2 \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ (v_1 +_\tau v_2) \cdot s,\ z\rangle \cdot A; h$ |
| `load` $x$ | | $\langle M,\ pc,\ f,\ s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ f[x] \cdot s,\ z\rangle \cdot A; h$ |
| `store` $x$ | | $\langle M,\ pc,\ f,\ v \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f[x \mapsto v],\ s,\ z\rangle \cdot A; h$ |
| `ifeq` $L$ | $v_1 \neq v_2$ | $\langle M,\ pc,\ f,\ v_1 \cdot v_2 \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ s,\ z\rangle \cdot A; h$ |
| `ifeq` $L$ | $v_1 = v_2$ | $\langle M,\ pc,\ f,\ v_1 \cdot v_2 \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ L,\ f,\ s,\ z\rangle \cdot A; h$ |
| `goto` $L$ | | $\langle M,\ pc,\ f,\ s,\ z\rangle \cdot A; h$ | $\langle M,\ L,\ f,\ s,\ z\rangle \cdot A; h$ |
| `getfield` $\{\!\|\varphi,\ l,\ \kappa\|\!\}_{\mathtt{F}}$ | $\Gamma \vdash Tag(h, b) <: \varphi$ | $\langle M,\ pc,\ f,\ b \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ h[b].\{\!\|\varphi,\ l,\ \kappa\|\!\}_{\mathtt{F}} \cdot s,\ z\rangle \cdot A; h$ |
| `putfield` $\{\!\|\varphi,\ l,\ \kappa\|\!\}_{\mathtt{F}}$ | $\Gamma \vdash Tag(h, b) <: \varphi \qquad \Gamma \vdash Tag(h, v) <: \kappa$ | $\langle M,\ pc,\ f,\ v \cdot b \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ s,\ z\rangle \cdot A; h[b.\{\!\|\varphi,\ l,\ \kappa\|\!\}_{\mathtt{F}} \mapsto v]$ |
| `newarray` $\tau$ | $b \notin Dom(h)$ | $\langle M,\ pc,\ f,\ n \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ b \cdot s,\ z\rangle \cdot A;$ $h[b \mapsto Blank(\texttt{(Array}\ \tau\texttt{)}, n)]$ |
| `arraylength` | $h[b] = [[v_0, \ldots, v_{n-1}]]_{\texttt{(Array}\ \tau\texttt{)}}$ | $\langle M,\ pc,\ f,\ b \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ n \cdot s,\ z\rangle \cdot A; h$ |
| `arrayload` $\tau$ | $h[b] = [[v_0, \ldots, v_{n-1}]]_{\texttt{(Array}\ \tau'\texttt{)}}$ $\Gamma \vdash \tau' <: \tau \qquad 0 \leq k < n$ | $\langle M,\ pc,\ f,\ k \cdot b \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ v_k \cdot s,\ z\rangle \cdot A; h$ |
| `arraystore` $\tau$ | $h[b] = [[v_0, \ldots, v_{n-1}]]_{\texttt{(Array}\ \tau'\texttt{)}}$ $\Gamma \vdash \tau' <: \tau \qquad \Gamma \vdash Tag(h, v') <: \tau'$ $0 \leq k < n$ | $\langle M,\ pc,\ f,\ v' \cdot k \cdot b \cdot s,\ z\rangle \cdot$ $A; h$ | $\langle M,\ pc+1,\ f,\ s,\ z\rangle \cdot$ $A; h[b \mapsto [[v_0, \ldots, v_{k-1}, v',$ $v_{k+1}, \ldots, v_{n-1}]]_{\texttt{(Array}\ \tau'\texttt{)}}]$ |
| `jsr` $L$ | | $\langle M,\ pc,\ f,\ s,\ z\rangle \cdot A; h$ | $\langle M,\ L,\ f,\ (pc+1) \cdot s,\ z\rangle \cdot A; h$ |
| `ret` $x$ | | $\langle M,\ pc,\ f,\ s,\ z\rangle \cdot A; h$ | $\langle M,\ f[x],\ f,\ s,\ z\rangle \cdot A; h$ |
| `new` $\sigma$ | $b \notin Dom(h)$ | $\langle M,\ pc,\ f,\ s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ f,\ a \cdot s,\ z\rangle \cdot A;$ $h[b \mapsto Blank(\sigma \lozenge \texttt{(Uninit}\ \sigma\ \ pc\texttt{)})]$ |
| `invokespecial` $N$ | $N = \{\!\|\texttt{Object, <init>, } \epsilon \to \texttt{void}\|\!\}_{\mathtt{M}}$ $Tag(h, b) = \varphi \lozenge \texttt{(Uninit}\ \sigma\ \ 0\texttt{)}$ $c \notin Dom(h)$ $\Gamma[\sigma]\texttt{.super} = \texttt{Object}$ | $\langle M,\ pc,\ f,\ b \cdot s,\ \langle b,\ \texttt{null}\rangle\rangle \cdot A; h$ | $\langle M,\ pc+1,\ [c/b]f,\ [c/b]s,\ \langle b,\ c\rangle\rangle \cdot A;$ $h[c \mapsto Blank(\varphi)]$ |
| `invokespecial` $N$ | $N = \{\!\|\texttt{Object, <init>, } \epsilon \to \texttt{void}\|\!\}_{\mathtt{M}}$ $Tag(h, b) = \texttt{Object} \lozenge \texttt{(Uninit Object}\ j\texttt{)}$ $c \notin Dom(h) \qquad j \neq 0$ | $\langle M,\ pc,\ f,\ b \cdot s,\ z\rangle \cdot A; h$ | $\langle M,\ pc+1,\ [c/b]f,\ [c/b]s,\ z\rangle \cdot A;$ $h[c \mapsto Blank(\texttt{Object})]$ |

*Figure 8.* Operational semantics, part 1.

$\Gamma \vdash C_0 \to C_1$

| $M_\Gamma[pc]$ | Cond | $C_0$ | $C_1$ |
|---|---|---|---|
| invokevirtual $\{\!\|\varphi,\ m',\ \alpha \to \gamma\|\!\}_{\mathtt{M}}$ | $Tag(h,b) = \sigma \qquad \|\alpha\| = \|s_1\|$<br>$\Gamma \vdash \sigma <: \varphi \qquad N = \{\!\|\sigma,\ m',\ \alpha \to \gamma\|\!\}_{\mathtt{M}}$ | $\langle M,\ pc,\ f,\ s_1 \bullet (b \cdot s),\ z\rangle \cdot A;\ h$ | $\langle N,\ 1,\ f_0[0 \mapsto b, 1..\|\alpha\| \mapsto s_1],\ \epsilon,\ \emptyset\rangle \cdot$<br>$\langle M,\ pc,\ f,\ s_1 \bullet (b \cdot s),\ z\rangle \cdot A;\ h$ |
| invokeinterface $\{\!\|\omega,\ m',\ \alpha \to \gamma\|\!\}_{\mathtt{I}}$ | $Tag(h,b) = \sigma \qquad \|\alpha\| = \|s_1\|$<br>$\Gamma \vdash \sigma <: \varphi \qquad N = \{\!\|\sigma,\ m',\ \alpha \to \gamma\|\!\}_{\mathtt{M}}$ | $\langle M,\ pc,\ f,\ s_1 \bullet (b \cdot s),\ z\rangle \cdot A;\ h$ | $\langle N,\ 1,\ f_0[0 \mapsto b, 1..\|\alpha\| \mapsto s_1],\ \epsilon,\ \emptyset\rangle \cdot$<br>$\langle M,\ pc,\ f,\ s_1 \bullet (b \cdot s),\ z\rangle \cdot A;\ h$ |
| invokespecial $\{\!\|\varphi,\ \texttt{<init>},\ \alpha \to \texttt{void}\|\!\}_{\mathtt{M}}$ | $\varphi \neq \texttt{Object}$<br>$Tag(h,a) = \varphi_0 \Diamond (\texttt{Uninit}\ \ \varphi'\ \ j)$<br>$b \notin Dom(h) \qquad \|\alpha\| = \|s_1\|$<br>$\varphi = \varphi' \vee (\Gamma[\varphi'].\texttt{super} = \varphi \wedge j = 0)$ | $\langle M,\ pc,\ f,\ s_1 \bullet (a \cdot s),\ z\rangle \cdot A;\ h$ | $\langle N,\ 1,\ f_0[0 \mapsto b, 1..\|\alpha\| \mapsto s_1],\ \epsilon,\ \langle b,\ \texttt{null}\rangle\rangle \cdot$<br>$\langle M,\ pc,\ f,\ s_1 \bullet (a \cdot s),\ z\rangle \cdot A;$<br>$h[b \mapsto Blank(\varphi_0 \Diamond (\texttt{Uninit}\ \ \varphi\ \ 0))]$ |
| return | $m \neq \texttt{<init>} \qquad \|\alpha_m\| = \|s_1\|$<br>$M = \{\!\|\sigma_m,\ m,\ \alpha_m \to \texttt{void}\|\!\}_{\mathtt{M}}$ | $\langle M,\ pc,\ f,\ s,\ z\rangle \cdot$<br>$\langle M',\ pc',\ f',\ s_1 \bullet (b \cdot s'),\ z'\rangle \cdot A;\ h$ | $\langle M',\ pc'+1,\ f',\ s',\ z'\rangle \cdot A;\ h$ |
| return | $m \neq \texttt{<init>}$<br>$M = \{\!\|\sigma_m,\ m,\ \epsilon \to \texttt{void}\|\!\}_{\mathtt{M}}$ | $\langle M,\ pc,\ f,\ s,\ z\rangle \cdot \epsilon;\ h$ | $\epsilon;\ h$ |
| returnval | $m \neq \texttt{<init>} \qquad \|\alpha_m\| = \|s_1\|$<br>$M = \{\!\|\sigma_m,\ m,\ \alpha_m \to \gamma_m\|\!\}_{\mathtt{M}}$<br>$\gamma_m \neq \texttt{void}$ | $\langle M,\ pc,\ f,\ v \cdot s,\ z\rangle \cdot$<br>$\langle M',\ pc',\ f',\ s_1 \bullet (b \cdot s'),\ z'\rangle \cdot A;\ h$ | $\langle M',\ pc'+1,\ f',\ v \cdot s',\ z'\rangle \cdot A;\ h$ |
| return | $M = \{\!\|\sigma,\ \texttt{<init>},\ \alpha \to \texttt{void}\|\!\}_{\mathtt{M}}$<br>$\|\alpha\| = \|s_1\| \qquad c \neq \texttt{null}$ | $\langle M,\ pc,\ f,\ s,\ \langle b,\ c\rangle\rangle \cdot \langle M',\ pc',\ f',$<br>$s_1 \bullet (a \cdot s'),\ \langle a,\ \texttt{null}\rangle\rangle \cdot A;\ h$ | $\langle M',\ pc'+1,\ [c/a]f',\ [c/a]s',\ \langle a,\ c\rangle\rangle \cdot$<br>$A;\ h$ |
| return | $M = \{\!\|\sigma,\ \texttt{<init>},\ \alpha \to \texttt{void}\|\!\}_{\mathtt{M}}$<br>$\|\alpha\| = \|s_1\| \qquad c \neq \texttt{null}$<br>$z' \neq \langle a,\ \texttt{null}\rangle$ | $\langle M,\ pc,\ f,\ s,\ \langle b,\ c\rangle\rangle \cdot$<br>$\langle M',\ pc',\ f',\ s_1 \bullet (a \cdot s'),\ z'\rangle \cdot A;\ h$ | $\langle M',\ pc'+1,\ [c/a]f',\ [c/a]s',\ z'\rangle \cdot A;\ h$ |
| throw | $\Gamma \vdash Tag(h,b) <: \texttt{Throwable}$ | $\langle M,\ pc,\ f,\ b \cdot s,\ z\rangle \cdot A;\ h$ | $\langle b\rangle_{\text{exc}} \cdot \langle M,\ pc,\ f,\ b \cdot s,\ z\rangle \cdot A;\ h$ |

$\Gamma \vdash C_0 \to C_1$

| Cond | $C_0$ | $C_1$ |
|---|---|---|
| $Tag(h,b) = \sigma$<br>$CorrectHandler(\Gamma, M, pc, \sigma) = 0$ | $\langle b\rangle_{\text{exc}} \cdot \langle M,\ pc,\ f,\ s,\ z\rangle \cdot A;\ h$ | $\langle b\rangle_{\text{exc}} \cdot A;\ h$ |
| $Tag(h,b) = \sigma$<br>$CorrectHandler(\Gamma, M, pc, \sigma) = t$<br>$t \neq 0$ | $\langle b\rangle_{\text{exc}} \cdot \langle M,\ pc,\ f,\ s,\ z\rangle \cdot A;\ h$ | $\langle M,\ t,\ f,\ b \cdot \epsilon,\ z\rangle \cdot A;\ h$ |
| | $\langle b\rangle_{\text{exc}} \cdot \epsilon;\ h$ | $\epsilon;\ h$ |

*Figure 9.* Operational semantics, part 2.

$\Gamma \vdash C_0 \to C_1$

| $M_\Gamma[pc]$ | Cond | $C_0$ | $C_1$ |
|---|---|---|---|
| `getfield` $\{\!\|\varphi, l, \kappa\|\!\}_{\mathrm{F}}$ | $b = \mathtt{null}$<br>$e \notin Dom(h)$ | $\langle M, pc, f, b \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, b \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `putfield` $\{\!\|\varphi, l, \tau\|\!\}_{\mathrm{F}}$ | $b = \mathtt{null}$<br>$e \notin Dom(h)$ | $\langle M, pc, f, v \cdot b \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, v \cdot b \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `newarray` $\tau$ | $n < 0$<br>$e \notin Dom(h)$ | $\langle M, pc, f, n \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, n \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `arraylength` | $b = \mathtt{null}$<br>$e \notin Dom(h)$ | $\langle M, pc, f, b \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, b \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `arrayload` $\tau$ | $b = \mathtt{null}$<br>$e \notin Dom(h)$ | $\langle M, pc, f, k \cdot b \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, k \cdot b \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `arrayload` $\tau$ | $h[b] = [[v_0, \ldots, v_{n-1}]]_{(\mathtt{Array}\ \tau')}$<br>$k < 0 \vee k > n-1 \qquad e \notin Dom(h)$ | $\langle M, pc, f, k \cdot b \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, k \cdot b \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `arraystore` $\tau$ | $b = \mathtt{null}$<br>$e \notin Dom(h)$ | $\langle M, pc, f, v' \cdot k \cdot b \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, v' \cdot k \cdot b \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `arraystore` $\tau$ | $h[b] = [[v_0, \ldots, v_{n-1}]]_{(\mathtt{Array}\ \tau')}$<br>$\Gamma \not\vdash Tag(h, v') <: \tau' \qquad e \notin Dom(h)$ | $\langle M, pc, f, v' \cdot k \cdot b \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, v' \cdot k \cdot b \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `arraystore` $\tau$ | $h[b] = [[v_0, \ldots, v_{n-1}]]_{(\mathtt{Array}\ \tau')}$<br>$k < 0 \vee k > n-1 \qquad e \notin Dom(h)$ | $\langle M, pc, f, v' \cdot k \cdot b \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, v' \cdot k \cdot b \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `invokevirtual`<br>$\{\!\|\varphi, m', \alpha \to \gamma\|\!\}_{\mathrm{M}}$ | $b = \mathtt{null} \qquad e \notin Dom(h)$<br>$\lvert\alpha\rvert = \lvert s_1\rvert$ | $\langle M, pc, f, s_1 \bullet (b \cdot s), z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, s_1 \bullet (b \cdot s), z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `invokeinterface`<br>$\{\!\|\omega, m', \alpha \to \gamma\|\!\}_{\mathrm{I}}$ | $b = \mathtt{null} \qquad e \notin Dom(h)$<br>$\lvert\alpha\rvert = \lvert s_1\rvert$ | $\langle M, pc, f, s_1 \bullet (b \cdot s), z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, s_1 \bullet (b \cdot s), z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `invokespecial`<br>$\{\!\|\varphi, m', \alpha \to \gamma\|\!\}_{\mathrm{M}}$ | $b = \mathtt{null} \qquad e \notin Dom(h)$<br>$\lvert\alpha\rvert = \lvert s_1\rvert$ | $\langle M, pc, f, s_1 \bullet (b \cdot s), z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, s_1 \bullet (b \cdot s), z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |
| `throw` | $b = \mathtt{null}$<br>$e \notin Dom(h)$ | $\langle M, pc, f, b \cdot s, z\rangle \cdot A; h$ | $\langle e\rangle_{\mathrm{exc}} \cdot \langle M, pc, f, b \cdot s, z\rangle \cdot$<br>$A; h[e \mapsto Blank(\mathtt{Throwable})]$ |

*Figure 10.* Operational semantics, part 3.

[CH 0]

$$\frac{\Gamma[M] = \langle P,\ H\rangle \quad \neg\exists i \in Dom(H).\ GoodHandler(\Gamma, pc, \sigma, H[i])}{CorrectHandler(\Gamma, M, pc, \sigma) = 0}$$

[CH 1]

$$\frac{\Gamma[M] = \langle P,\ H\rangle \quad GoodHandler(\Gamma, pc, \sigma, H[j]) \quad H[j] = \langle s,\ e,\ t,\ \sigma'\rangle \quad \neg\exists i \in Dom(H).\ i < j \wedge GoodHandler(\Gamma, pc, \sigma, H[i])}{CorrectHandler(\Gamma, M, pc, \sigma) = t}$$

[GH]

$$\frac{s \le pc < e \quad \Gamma \vdash \sigma <: \sigma'}{GoodHandler(\Gamma, pc, \sigma, \langle s,\ e,\ t,\ \sigma'\rangle)}$$

*Figure 11.* Judgments to identify exception handler for an exception.

caller's stack during the method invocation to simplify a few aspects of our formal development.

The `throw` instruction raises an exception by taking a `Throwable` argument off the top of the stack and pushing a new activation record containing that reference.

The second table in Figure 9 shows how exceptions are handled. If a valid handler is found in the topmost activation record, control is transferred to the target of that handler. Otherwise, the topmost activation record is popped off the stack, and we try again in the next activation record. Figure 11 defines the rules to determine the appropriate handler for an exception, if one exists.

Figure 10 contains rules that generate exceptions when run-time tests in the virtual machine fail, such as when `getfield` is passed a null pointer, an array is accessed out of bounds, and so on. These rules create a new `Throwable` object without calling its constructor, which enables us to treat the process of generating an exception for a run-time error as an atomic operation. Real Java Virtual Machines do call a constructor on the exception object, but this deviation allows us to model all instructions in a small-step operational semantics easily and does not impact the static semantics in any way.

### 3.3.1. *Object Initialization*

The JVML$_f$ `new` instruction (defined in Figure 8) allocates memory for a new class instance, but it does not initialize the object. The object is considered initialized only after successfully completing a constructor invocation on the new object. Moreover, the constructor must invoke either another constructor from the same class or from the superclass on the object being initialized before exiting and before using the object in any way. This requirement must be satisfied by all constructors, and it ensures that a constructor for each class in the inheritance chain is called,

in order from the new object's class up to `Object`, during object initialization. A program calls a constructor with the `invokespecial` instruction.

To capture these and other rules for initialization, the JVML$_f$ semantics do not allow a program to

1. use an object before it has been properly initialized,
2. invoke a constructor on an initialized object,
3. invoke a constructor from the wrong class on an object, or
4. return from a constructor without invoking another constructor from an appropriate class on the object being initialized.

For simplicity, our treatment of object initialization differs from the Java Virtual Machine Specification in two ways. First, we do not permit assignments to instance fields prior to the superclass constructor invocation. Second, our semantic rules do not resolve references to constructors named in *Method-Ref*s in the same way as method invocations, which is implied by the specification.

To identify and prevent the illegal operations listed above, the JVML$_f$ virtual machine stores the initialization status of an object in its tag. An initialized object of class $\sigma$ has the tag $\sigma$, and it may be the operand of any instruction expecting an initialized object of that type. Uninitialized object tags have the form $\sigma \Diamond$`(Uninit` $\sigma'$ $j$`)`, and objects with tags of this form are not valid operands for most instructions. As we shall see, the format of these tags makes it easy to map dynamic values to statically computed type information for a program. We explain the intuition behind uninitialized object tags first and then relate them back to the operational semantics rules.

When a `new` $\sigma$ instruction is executed, the tag on the newly allocated object is $\sigma \Diamond$`(Uninit` $\sigma$ *pc*`)`, where *pc* is the program counter for the currently executing method. The first part of such a tag, $\sigma$, indicates that the object was originally allocated by a `new` $\sigma$ instruction. The second half, `(Uninit` $\sigma$ *pc*`)`, indicates that the program allocated the object on line *pc* of the method and that it must call a constructor from class $\sigma$ on that object before it can do anything else to it. We store the specific value of *pc* for use in the soundness proof for the alias tracking uninitialized objects in the static semantics.

Inside a constructor for class $\sigma'$, the object being initialized will have a tag of the form $\sigma \Diamond$`(Uninit` $\sigma'$ 0`)`. In this situation, the next step toward properly initializing such an object is to invoke either another constructor for $\sigma'$ or a constructor from the super class of $\sigma'$. Any other operation on the object is not defined.

Invoking a constructor changes the tag of the receiving object to reflect which operations are allowed to be performed on the object next. For example, if a constructor of type $\sigma$ were called on an object with tag $\sigma \Diamond$`(Uninit` $\sigma$ *pc*`)`, the tag would become $\sigma \Diamond$`(Uninit` $\sigma$ 0`)`. If a constructor of $\sigma'$, the super class of $\sigma$, were then invoked on the object, the tag would become $\sigma \Diamond$`(Uninit` $\sigma'$ 0`)`, and so on. Invoking the `Object` constructor on the object changes its tag to $\sigma$. Constructors may exit only when the object being initialized has an initialized object tag.

To simplify our soundness proofs, we actually model object initialization in a different way, as demonstrated by the rules for `new`, `invokespecial`, and `return`. In order to avoid changing tags of existing objects in constructors, which complicates proving several key monotonicity properties of JVML$_f$ heaps, we create new objects at constructor call sites. More specifically, calling a constructor creates a new object with a new tag, and returning from a constructor performs a substitution of the initialized object for the old, uninitialized object in the caller's activation record. The $z$ component in activation records tracks these substitutions. If an activation record is for a normal method, $z$ is $\emptyset$. Otherwise, it is a tuple $\langle a, b \rangle$, where $a$ is the object passed into the constructor as the `this` argument, and $b$ is the properly initialized object to take its place, or `null` if it has not been initialized by the superclass constructor yet.

These substitutions are similar to those in the JVML$_i$ semantics from [18], and we are able to leverage the proof development in that paper with little change. The accuracy of this model is acceptable because no operations are performed on the intermediate objects, and the standard execution behavior can be considered an optimization of our semantics.

#### 3.3.2. *Subroutines*

Bytecode subroutines are a form of local call and return that provide space-efficient compilation of `try-finally` statements in Java programs. Without subroutines, the code for a `finally` block would have to be duplicated at every exit from the corresponding `try` block. Subroutines are easy to model in the dynamic semantics, and the rules for them appear in Figure 8. However, they are difficult to analyze in the static semantics.

## 4. Static Semantics

The static semantics determine whether a JVML$_f$ program can be assigned a valid type. If a program can be assigned a type, it will not cause a type error when executed. Thus, the static semantics form the core of a bytecode verifier for JVML$_f$ programs.

The static semantics place two key requirements on a program modeled by the environment $\Gamma$: (1) the declarations in $\Gamma$ must be consistent, and (2) each method body must not cause a type error when executed. We describe each requirement in this section.

### 4.1. WELL-FORMED ENVIRONMENTS

Well-formed environments are environments satisfying certain constraints, including the requirements that

– the class hierarchy contains no cycles,

[WT ENV]

$$\frac{\begin{array}{c}\texttt{Object}, \texttt{Throwable} \in \mathit{Dom}(\Gamma) \\ \forall d \in \mathit{Dom}(\Gamma).\ \Gamma \vdash d\ \texttt{ty} \\ \forall d \in \mathit{Dom}(\Gamma).\ \exists K \in \{\texttt{class}, \texttt{interface}, \texttt{method}\}.\ \Gamma \vdash d\ K\end{array}}{\Gamma \vdash \texttt{wt}}$$

[WT METH]

$$\frac{\Gamma[M] = \langle P,\ H\rangle \quad \Gamma, F, S \vdash P, H : M}{\Gamma \vdash M\ \texttt{method}}$$

[WT OBJ]

$$\frac{\Gamma[\texttt{Object}] = \langle \texttt{None}, \emptyset, \emptyset\rangle}{\Gamma \vdash \texttt{Object class}}$$

[WT THROWABLE]

$$\frac{\Gamma[\texttt{Throwable}] = \langle \texttt{Object}, \emptyset, \emptyset\rangle}{\Gamma \vdash \texttt{Throwable class}}$$

[WT CLASS]

$$\frac{\begin{array}{rll}\Gamma[\sigma] = \langle \sigma_s,\ \{\omega_i\}^{i\in I},\ \{\{\!|\sigma_j,\ l_j,\ \kappa_j|\!\}_{\mathrm{F}}\}^{j\in J}\rangle & & \\ \Gamma \not\vdash \sigma_s <: \sigma & & \textit{no cycles} \\ \Gamma[\sigma_s].\texttt{interfaces} \subseteq \{\omega_i\}^{i\in I} & & \textit{inherit all interfaces} \\ \Gamma[\sigma_s].\texttt{fields} \subseteq \{\{\!|\sigma_j,\ l_j,\ \kappa_j|\!\}_{\mathrm{F}}\}^{j\in J} & & \textit{inherit all fields} \\ \forall j \in J.\ \Gamma \vdash \sigma <: \sigma_j & & \textit{fields only from class or ancestor} \\ \forall i \in I.\ \forall m, \alpha, \gamma.\ \{\!|\omega_i,\ m,\ \alpha \to \gamma|\!\}_{\mathrm{I}} \in \Gamma[\omega_i].\texttt{methods} & & \\ \quad \Rightarrow \{\!|\sigma,\ m,\ \alpha \to \gamma|\!\}_{\mathrm{M}} \in \mathit{Dom}(\Gamma) & & \textit{implement all interfaces} \\ \forall m, \alpha, \gamma.\ \left(\left(\begin{array}{l}\{\!|\sigma_s,\ m,\ \alpha \to \gamma|\!\}_{\mathrm{M}} \in \mathit{Dom}(\Gamma) \\ \wedge\ m \neq \texttt{<init>}\end{array}\right) \Rightarrow \{\!|\sigma,\ m,\ \alpha \to \gamma|\!\}_{\mathrm{M}} \in \mathit{Dom}(\Gamma)\right) & & \textit{inherit / override all methods}\end{array}}{\Gamma \vdash \sigma\ \texttt{class}}$$

[WT INT]

$$\frac{\begin{array}{rll}\Gamma[\omega] = \langle \{\omega_i\}^{i\in I},\ \{\{\!|\omega,\ m_j,\ \alpha_j \to \gamma_j|\!\}_{\mathrm{I}}\}^{j\in J}\rangle & & \\ \Gamma \not\vdash \omega_i <: \omega^{\,i\in I} & & \textit{no cycles} \\ \forall j \in J.\ m_j \neq \texttt{<init>} & & \textit{no constructors} \\ \forall i \in I.\ \forall m, \alpha, \gamma.\ \{\!|\omega_i,\ m,\ \alpha \to \gamma|\!\}_{\mathrm{I}} \in \Gamma[\omega_i].\texttt{methods} & & \\ \quad \Rightarrow \{\!|\omega,\ m,\ \alpha \to \gamma|\!\}_{\mathrm{I}} \in \{\{\!|\omega,\ m_j,\ \alpha_j \to \gamma_j|\!\}_{\mathrm{I}}\}^{j\in J} & & \textit{include all methods}\end{array}}{\Gamma \vdash \omega\ \texttt{interface}}$$

*Figure 12.* Rules for well-formed environments.

- each class implements its declared interfaces by defining all methods listed in them, and
- each class inherits all field and interface declarations from its superclass and inherits or overrides all methods.

Basically, an environment is well formed if all declarations conform to the properties described in the Sun Java Virtual Machine Specification [29] that do not depend on the bodies of methods.

Figure 12 contains the inference rule [WT ENV], which shows that a JVML$_f$ environment $\Gamma$ is well formed. The rule requires that (1) `Object` and `Throwable` are present in $\Gamma$, (2) all declarations refer only to classes and interfaces defined in the environment (see Figure 13), and (3) each declaration in $\Gamma$ is well formed. The statement that declaration $d$ is well formed is written $\Gamma \vdash d\ \ K$, where $K$ is the kind of declaration.

Figure 12 also presents rules to conclude that a declaration is a well-formed interface, class, or method. In [WT METH], $F$ is a map from ADDR to functions

$$\frac{}{\Gamma \vdash \texttt{Object ty}}\ [\text{TY OBJ}] \qquad \frac{\Gamma[\sigma] = \langle \sigma_s,\ \{\omega_i\}^{i\in I},\ \{\{\!|\sigma_j,\ l_j,\ \tau_j|\!\}_{\texttt{F}}\}^{j\in J}\rangle \quad \Gamma \vdash \sigma_s\ \texttt{ok} \quad \forall i \in I.\ \Gamma \vdash \omega_i\ \texttt{ok} \quad \forall j \in J.\ \Gamma \vdash \{\!|\sigma_j,\ l_j,\ \tau_j|\!\}_{\texttt{F}}\ \texttt{ok}}{\Gamma \vdash \sigma\ \texttt{ty}}\ [\text{TY CLASS}]$$

$$\frac{\Gamma[\omega] = \langle \{\omega_i\}^{i\in I},\ \{\{\!|\omega,\ m_j,\ \alpha_j \to \gamma_j|\!\}_{\texttt{I}}\}^{j\in J}\rangle \quad \forall i \in I.\ \Gamma \vdash \omega_i\ \texttt{ok} \quad \forall j \in J.\ \Gamma \vdash \{\!|\omega,\ m_j,\ \alpha_j \to \gamma_j|\!\}_{\texttt{I}}\ \texttt{ok}}{\Gamma \vdash \omega\ \texttt{ty}}\ [\text{TY INT}]$$

$$\frac{\Gamma \vdash \{\!|\varphi,\ m,\ \alpha \to \gamma|\!\}_{\texttt{M}}\ \texttt{ok}}{\Gamma \vdash \{\!|\varphi,\ m,\ \alpha \to \gamma|\!\}_{\texttt{M}}\ \texttt{ty}}\ [\text{TY METH}]$$

$$\frac{\tau \in \{\texttt{int}, \texttt{float}\}}{\Gamma \vdash \tau\ \texttt{ok}}\ [\text{OK BASE}] \qquad \frac{\sigma \in \mathit{Dom}(\Gamma)}{\Gamma \vdash \sigma\ \texttt{ok}}\ [\text{OK CLASS}] \qquad \frac{\omega \in \mathit{Dom}(\Gamma)}{\Gamma \vdash \omega\ \texttt{ok}}\ [\text{OK INT}]$$

$$\frac{\Gamma \vdash \tau\ \texttt{ok}}{\Gamma \vdash \texttt{(Array}\ \tau\texttt{)}\ \texttt{ok}}\ [\text{OK ARRAY}] \qquad \frac{}{\Gamma \vdash \epsilon\ \texttt{ok}}\ [\text{OK } \epsilon] \qquad \frac{\Gamma \vdash \tau\ \texttt{ok} \quad \Gamma \vdash \alpha\ \texttt{ok}}{\Gamma \vdash \tau \cdot \alpha\ \texttt{ok}}\ [\text{OK SEQ}]$$

$$\frac{\Gamma \vdash \varphi\ \texttt{ok} \quad \Gamma \vdash \tau\ \texttt{ok}}{\Gamma \vdash \{\!|\varphi,\ l,\ \tau|\!\}_{\texttt{F}}\ \texttt{ok}}\ [\text{OK FREF}] \qquad \frac{\Gamma \vdash \varphi\ \texttt{ok} \quad \Gamma \vdash \alpha\ \texttt{ok} \quad \Gamma \vdash \gamma\ \texttt{ok} \vee \gamma = \texttt{void}}{\Gamma \vdash \{\!|\varphi,\ m,\ \alpha \to \gamma|\!\}_{\texttt{M}}\ \texttt{ok}}\ [\text{OK MREF}] \qquad \frac{\Gamma \vdash \omega\ \texttt{ok} \quad \Gamma \vdash \alpha\ \texttt{ok} \quad \Gamma \vdash \gamma\ \texttt{ok} \vee \gamma = \texttt{void}}{\Gamma \vdash \{\!|\omega,\ m,\ \alpha \to \gamma|\!\}_{\texttt{I}}\ \texttt{ok}}\ [\text{OK IREF}]$$

*Figure 13.* Rules to ensure that declarations refer only to defined types.

mapping local variables to types such that $F_i[y]$ is the type of local variable $y$ at line $i$. The function $S$ is a map from ADDR to stack types where $S_i$ is the type of the operand stack at location $i$ of the program. Finding $F$ and $S$ such that $\Gamma, F, S \vdash P, H : M$ is analogous to the verifier accepting the code of method $M$. Figure 14 shows the valid type information for a method, and Section 4.2 develops this judgment.

The rules for environments, partially based on the work of Syme [44] and Drossopoulou and Eisenbach [14], do not show how to build a well-formed environment incrementally. Incremental construction is necessary to handle dynamic class loading in the Java Virtual Machine, and this problem has been studied elsewhere. Building an environment for a complete program is straightforward, and we assume an environment has been built in this way because it is adequate to describe how to type check bytecode methods.

## 4.2. METHODS

Figure 15 contains the typing rule to check the body of a method. The judgment $\Gamma, F, S \vdash P, H : M$ means that, given the environment $\Gamma$ and type information

| $i$ | $P[i]$ | $S_i$ | $F_i[0]$ | $F_i[1]$ | $F_i[2]$ |
|---|---|---|---|---|---|
| 1 : | `new A` | $\epsilon$ | `A` | `int` | `Top` |
| 2 : | `store 2` | `(Uninit A 1)` $\cdot \epsilon$ | `A` | `int` | `Top` |
| 3 : | `load 2` | $\epsilon$ | `A` | `int` | `(Uninit A 1)` |
| 4 : | `load 1` | `(Uninit A 1)` $\cdot \epsilon$ | `A` | `int` | `(Uninit A 1)` |
| 5 : | `invokespecial` {\|`A`, `<init>`, `int` → `void`\|}$_{\mathtt{M}}$ | `int` $\cdot$ `(Uninit A 1)` $\cdot \epsilon$ | `A` | `int` | `(Uninit A 1)` |
| 6 : | `goto 11` | $\epsilon$ | `A` | `int` | `A` |
| 7 : | `pop` | `Throwable` $\cdot \epsilon$ | `A` | `int` | `Top` |
| 8 : | `load 0` | $\epsilon$ | `A` | `int` | `Top` |
| 9 : | `push 2` | `A` $\cdot \epsilon$ | `A` | `int` | `Top` |
| 10 : | `putfield` {\|`A`, `num`, `int`\|}$_{\mathtt{F}}$ | `int` $\cdot$ `A` $\cdot \epsilon$ | `A` | `int` | `Top` |
| 11 : | `push 6` | $\epsilon$ | `A` | `int` | `Top` |
| 12 : | `returnval` | `int` $\cdot \epsilon$ | `A` | `int` | `Top` |

```
Exception table:
   from   to  target  type
     1     6     7    Throwable
```

*Figure 14.* The type information for method {|`A`, `foo`, `int` → `int`|}$_{\mathtt{M}}$ in Figure 2.

[METH CODE]

$$\frac{\begin{array}{c} m \neq \texttt{<init>} \\ \Gamma \vdash F_{\mathrm{TOP}}[0 \mapsto \sigma, 1..|\alpha| \mapsto \alpha] <: F_1 \\ S_1 = \epsilon \\ G_1 = \{\epsilon\} \\ \forall i \in Dom(P).\ \Gamma, F, S, i \vdash P : \{|\sigma, m, \alpha \to \gamma|\}_{\mathtt{M}} \quad \textit{instructions well-typed} \\ \forall i \in Dom(H).\ \Gamma, F, S \vdash H[i] \text{ handles } P \quad \textit{handlers well-typed} \\ \forall i \in Dom(P).\ G, i \vdash P \text{ labeled} \quad \textit{labeling exists for instructions} \\ \forall i \in Dom(H).\ G, H[i] \vdash P \text{ labeled} \quad \textit{and handlers} \end{array}}{\Gamma, F, S \vdash P, H : \{|\sigma, m, \alpha \to \gamma|\}_{\mathtt{M}}}$$

[CSTR CODE]

$$\frac{\begin{array}{c} \Gamma \vdash F_{\mathrm{TOP}}[0 \mapsto \texttt{(Uninit } \sigma\ \texttt{0)}, 1..|\alpha| \mapsto \alpha] <: F_1 \\ S_1 = \epsilon \\ G_1 = \{\epsilon\} \\ Z_1 = \mathit{false} \\ \forall i \in Dom(P).\ \Gamma, F, S, i \vdash P : \{|\sigma, m, \alpha \to \gamma|\}_{\mathtt{M}} \\ \forall i \in Dom(H).\ \Gamma, F, S \vdash H[i] \text{ handles } P \\ \forall i \in Dom(P).\ P, i \vdash G \text{ labeled} \\ \forall i \in Dom(H).\ G, H[i] \vdash P \text{ labeled} \\ \forall i \in Dom(P).\ \Gamma, Z, S, i \vdash P \text{ constructs } \sigma \quad \textit{all paths call superclass constructor,} \\ \forall i \in Dom(H).\ \Gamma, Z, S, H[i] \vdash P \text{ constructs } \sigma \quad \textit{including exception paths} \end{array}}{\Gamma, F, S \vdash P, H : \{|\sigma, \texttt{<init>}, \alpha \to \gamma|\}_{\mathtt{M}}}$$

*Figure 15.* Rules for well-typed methods and constructors.

$F$ and $S$, executing the instruction array $P$ with handlers $H$ does not cause a type error and is consistent with the method type expressed in the *Method-Ref* $M$. In that rule, $F_{\mathrm{TOP}}$ maps all variables in VAR to `Top`. The map $F_1$ matches the types of the values stored in $f$ during the creation of a new activation record for a call to this method.

The fourth line requires that each instruction in the program be well typed according to the local judgments described in the next section, and the fifth line requires that each handler be well typed. Our rules do not identify dead code and require valid type information for every line, including lines after calls to subroutines that never return.

The final two lines ensure that the subroutines within a method are well structured. We elaborate on these judgments and those regarding object initialization in [CSTR CODE] below.

### 4.2.1. *Instructions*

Figures 16–18 present the instruction typing rules. These typing rules describe a set of constraints between the types of variables and stack slots at different locations in the program. The horizontal lines in those judgments break the constraints into logical groups. The format is as follows:

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] =$ instruction form |
|---|
| constraints on line $i$ |
| constraints on successor of $i$ |

The basic intuition behind these rules is that type information flows along execution paths. The types of variables and stack locations touched by an instruction change the type information for all successor instructions, and the types of untouched locations are the same or more general in the successor instructions. As an example, consider the rule for `getfield` $\{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}}$ in Figure 16, which concludes that $\Gamma, F, S, i \vdash P : M$ if all conditions listed in the box are satisfied. Given the requirements on well-formed environments, we know that as long as the object on top of the stack is a subclass of $\varphi$, a field named $\{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}}$ will be present in the object's record.

### 4.2.2. *Object Initialization*

The static semantics must guarantee that no well-typed program uses an object before it has been initialized. Since references to uninitialized objects may be stored in local variables or duplicated on the stack between allocation and initialization, a simple form of alias analysis is used to track all references to each uninitialized object.

The type of a variable or stack slot containing an uninitialized object reference is a type of the form `(Uninit` $\sigma$ $pc$`)`, which represents an uninitialized object of class $\sigma$ allocated on line $pc$ of the method. All references with the same uninitialized object type are assumed to be aliases, and when any of those references are initialized, the types of all references to the object are changed to an initialized object type. This analysis is sound because the same uninitialized object type is never assigned to two different objects during program execution.

[PUSH]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{push}\ v$ |
|---|
| $v \in$ values of type $\tau$<br>$\tau \in \mathit{Prim} \cup \{\mathtt{Null}\}$ |
| $\Gamma \vdash \tau \cdot S_i <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i+1 \in \mathit{Dom}(P)$ |

[POP]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{pop}$ |
|---|
| $S_i = \tau \cdot \beta$ |
| $\Gamma \vdash \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i+1 \in \mathit{Dom}(P)$ |

[IF]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{ifeq}\ L$ |
|---|
| $\tau \in \mathit{Simple\text{-}Ref} \cup \mathit{Prim} \cup \mathit{Array}$<br>$\Gamma \vdash S_i <: \tau \cdot \tau \cdot \beta$ |
| $\Gamma \vdash \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i+1 \in \mathit{Dom}(P)$ |
| $\Gamma \vdash \beta <: S_L$<br>$\Gamma \vdash F_i <: F_L$<br>$L \in \mathit{Dom}(P)$ |

[ADD]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{add}$ |
|---|
| $\tau \in \mathit{Prim}$<br>$\Gamma \vdash S_i <: \tau \cdot \tau \cdot \beta$ |
| $\Gamma \vdash \tau \cdot \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i+1 \in \mathit{Dom}(P)$ |

[LOAD]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{load}\ x$ |
|---|
| $x \in \mathit{Dom}(F_i)$ |
| $\Gamma \vdash F_i[x] \cdot S_i <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i+1 \in \mathit{Dom}(P)$ |

[STORE]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{store}\ x$ |
|---|
| $x \in \mathit{Dom}(F_i)$<br>$S_i = \tau \cdot \beta$ |
| $\Gamma \vdash \beta <: S_{i+1}$<br>$\Gamma \vdash F_i[x \mapsto \tau] <: F_{i+1}$<br>$i+1 \in \mathit{Dom}(P)$ |

[GET FIELD]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{getfield}\ \{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}}$ |
|---|
| $\Gamma \vdash S_i <: \varphi \cdot \beta$<br>$\{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}} \in \Gamma[\varphi].\mathtt{fields}$ |
| $\Gamma \vdash \kappa \cdot \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i+1 \in \mathit{Dom}(P)$ |

[PUT FIELD]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{putfield}\ \{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}}$ |
|---|
| $\Gamma \vdash S_i <: \kappa \cdot \varphi \cdot \beta$<br>$\{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}} \in \Gamma[\varphi].\mathtt{fields}$ |
| $\Gamma \vdash \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i+1 \in \mathit{Dom}(P)$ |

[GOTO]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{goto}\ L$ |
|---|
| $\Gamma \vdash S_i <: S_L$<br>$\Gamma \vdash F_i <: F_L$<br>$L \in \mathit{Dom}(P)$ |

[THROW]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{throw}$ |
|---|
| $\Gamma \vdash S_i <: \mathtt{Throwable} \cdot \beta$ |

*Figure 16.* Instruction typing rules, part 1.

The constraints in [NEW] eliminate this possibility by removing any old occurrences of the type $(\mathtt{Uninit}\ \sigma\ i)$ in the type information for successor instructions. This requirement (and similar requirements in [JSR] and [RET]) is phrased differ-

[NEW ARRAY]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{newarray}\ \tau$ |
|---|
| $\Gamma \vdash S_i <: \mathtt{int} \cdot \beta$ |
| $\Gamma \vdash (\mathtt{Array}\ \tau) \cdot \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i + 1 \in Dom(P)$ |

[ARRAY LEN]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{arraylength}$ |
|---|
| $\Gamma \vdash S_i <: (\mathtt{Array}\ \tau) \cdot \beta$ |
| $\Gamma \vdash \mathtt{int} \cdot \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i + 1 \in Dom(P)$ |

[ARRAY LOAD]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{arrayload}\ \tau$ |
|---|
| $\Gamma \vdash S_i <: \mathtt{int} \cdot (\mathtt{Array}\ \tau) \cdot \beta$ |
| $\Gamma \vdash \tau \cdot \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i + 1 \in Dom(P)$ |

[ARRAY STORE]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{arraystore}\ \tau$ |
|---|
| $\Gamma \vdash S_i <: \tau \cdot \mathtt{int} \cdot (\mathtt{Array}\ \tau) \cdot \beta$ |
| $\Gamma \vdash \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i + 1 \in Dom(P)$ |

[INV VIRT]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{invokevirtual}\ N$ |
|---|
| $N = \{\!|\varphi, m', \alpha \to \gamma|\!\}_{\mathtt{M}}$<br>$m' \neq \mathtt{<init>}$<br>$\Gamma \vdash S_i <: \alpha \bullet (\varphi \cdot \beta)$<br>$N \in Dom(\Gamma)$ |
| $\gamma \neq \mathtt{void} \Rightarrow \Gamma \vdash \gamma \cdot \beta <: S_{i+1}$<br>$\gamma = \mathtt{void} \Rightarrow \Gamma \vdash \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i + 1 \in Dom(P)$ |

[INV INT]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{invokeinterface}\ N$ |
|---|
| $N = \{\!|\omega, m', \alpha \to \gamma|\!\}_{\mathtt{I}}$<br>$\Gamma \vdash S_i <: \alpha \bullet (\omega \cdot \beta)$<br>$N \in \Gamma[\omega].\mathtt{methods}$ |
| $\gamma \neq \mathtt{void} \Rightarrow \Gamma \vdash \gamma \cdot \beta <: S_{i+1}$<br>$\gamma = \mathtt{void} \Rightarrow \Gamma \vdash \beta <: S_{i+1}$<br>$\Gamma \vdash F_i <: F_{i+1}$<br>$i + 1 \in Dom(P)$ |

[RETURN]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{return}$ |
|---|
| $\gamma_m = \mathtt{void}$ |

[RETURN VAL]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

| $P[i] = \mathtt{returnval}$ |
|---|
| $\gamma_m \neq \mathtt{void}$<br>$\Gamma \vdash S_i <: \gamma_m \cdot \beta$ |

*Figure 17.* Instruction typing rules, part 2.

ently from that in our previous work [16]. Our earlier formulation, which matches the Sun verifier specification, asserted that `(Uninit` $\sigma$ $i$`)` did not occur in the type information for the `new` instruction being checked. However, this restriction leads to a monotonicity problem in the dataflow algorithm derived from these rules that we describe in Section 6. The new versions avoid this problem and use checks similar to those in other frameworks [41, 25].

Constructor bodies require additional checks to ensure that they apply either a different constructor of the same class or a constructor from the parent class to the object that is being initialized (which is passed into the constructor in local variable 0) before they exit. The only deviation from this requirement is for constructors of class `Object`. Since `Object` has no superclass, constructors for `Object` need not call any other constructor.

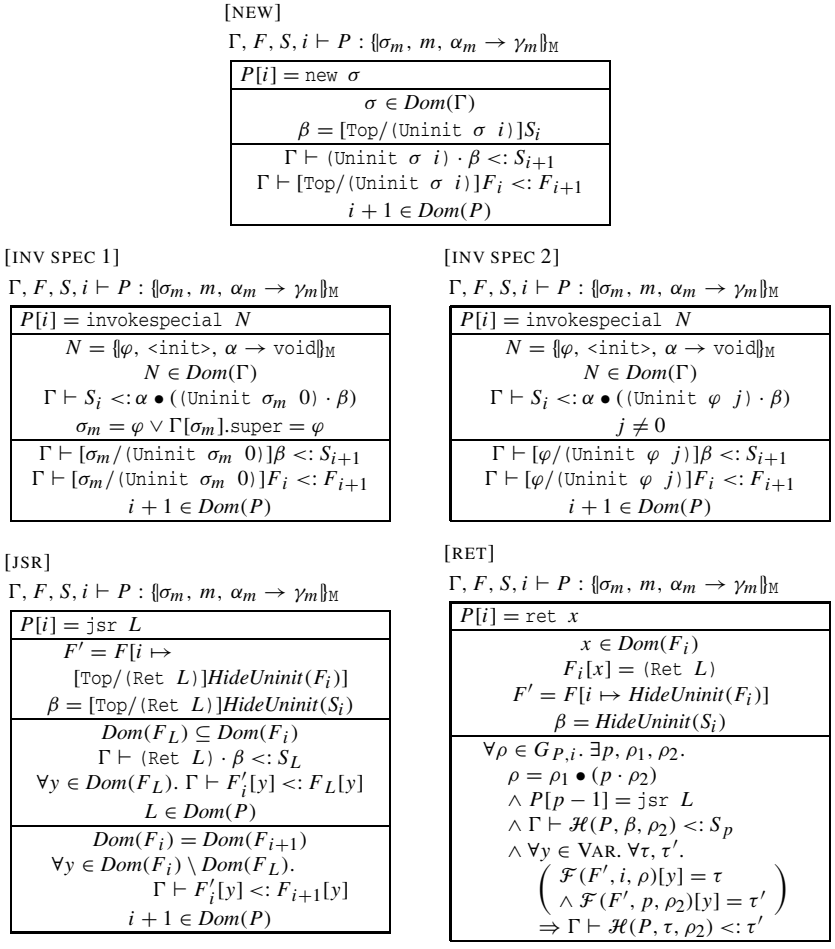


*Figure 18.* Instruction typing rules, part 3.

Our treatment of constructors uses additional uninitialized object types, and we add the following to the type system:

- special types of the form (Uninit $\sigma$ 0), which are assigned to the uninitialized objects passed into constructors,
- a variant of [INV SPEC] to check the usage of these types, and
- a set of rules to examine the structure of a constructor and determine whether all paths through the code call an appropriate superclass constructor.

The structural rules label each line of the constructor's body with *true* if a superclass constructor has been called on all paths to that line, and *false* otherwise. Figure 19 contains these rules, and Figure 20 shows the type and superclass constructor information ($Z_i$) for a constructor. The first rule in Figure 19 matches a

[CSTR LABEL INV SPEC1]

$$\frac{P[i] = \texttt{invokespecial}\ \{\!|\varphi, \texttt{<init>}, \alpha \to \texttt{void}|\!\}_{\texttt{M}} \quad \Gamma \vdash S_i <: \alpha \bullet ((\texttt{Uninit}\ \sigma_m\ 0) \cdot \beta) \quad Z_i = \mathit{false} \quad Z_{i+1} = \mathit{true}}{\Gamma, S, Z, i \vdash P \text{ constructs } \sigma_m}$$

[CSTR LABEL INV SPEC2]

$$\frac{P[i] = \texttt{invokespecial}\ \{\!|\varphi, \texttt{<init>}, \alpha \to \texttt{void}|\!\}_{\texttt{M}} \quad \forall \beta.\ \Gamma \not\vdash S_i <: \alpha \bullet ((\texttt{Uninit}\ \sigma_m\ 0) \cdot \beta) \quad Z_{i+1} = Z_i}{\Gamma, S, Z, i \vdash P \text{ constructs } \sigma_m}$$

[CSTR LABEL IFEQ]

$$\frac{P[i] = \texttt{ifeq}\ L \quad Z_L = Z_i = Z_{i+1}}{\Gamma, S, Z, i \vdash P \text{ constructs } \sigma_m}$$

[CSTR LABEL GOTO]

$$\frac{P[i] = \texttt{goto}\ L \quad Z_L = Z_i}{\Gamma, S, Z, i \vdash P \text{ constructs } \sigma_m}$$

[CSTR LABEL JSR]

$$\frac{P[i] = \texttt{jsr}\ L \quad Z_L = Z_i = Z_{i+1}}{\Gamma, S, Z, i \vdash P \text{ constructs } \sigma_m}$$

[CSTR LABEL RET-THROW]

$$\frac{P[i] \in \{\texttt{ret}\ x, \texttt{throw}\}}{\Gamma, S, Z, i \vdash P \text{ constructs } \sigma_m}$$

[CSTR LABEL RETURN]

$$\frac{P[i] = \texttt{return} \quad Z_i = \mathit{true}}{\Gamma, S, Z, i \vdash P \text{ constructs } \sigma_m}$$

[CSTR LABEL REST]

$$\frac{\forall x, L, M.\ P[i] \notin \{\texttt{invokespecial}\ M, \texttt{ifeq}\ L, \texttt{goto}\ L, \texttt{jsr}\ L, \texttt{ret}\ x, \texttt{throw}, \texttt{return}\} \quad Z_{i+1} = Z_i}{\Gamma, S, Z, i \vdash P \text{ constructs } \sigma_m}$$

[CSTR LABEL HANDLER]

$$\frac{\forall i \in [b, e).\ Z_i = Z_t}{\Gamma, S, Z, \langle b, e, t, \sigma \rangle \vdash P \text{ constructs } \sigma_m}$$

*Figure 19.* Structural rules to determine initialization status in a constructor.

| $i$ | $P[i]$ | $Z_i$ | $S_i$ | $F_i[0]$ | $F_i[1]$ |
|---|---|---|---|---|---|
| 1 : | load 0 | false | $\epsilon$ | (Uninit B 0) | int |
| 2 : | load 1 | false | (Uninit B 0) $\cdot\ \epsilon$ | (Uninit B 0) | int |
| 3 : | invokespecial {\|B, <init>, int → void\|}$_{\texttt{M}}$ | false | int $\cdot$ (Uninit B 0) $\cdot\ \epsilon$ | (Uninit B 0) | int |
| 4 : | load 0 | true | $\epsilon$ | B | int |
| 5 : | load 0 | true | B $\cdot\ \epsilon$ | B | int |
| 6 : | push 2 | true | B $\cdot$ B $\cdot\ \epsilon$ | B | int |
| 7 : | invokevirtual {\|B, foo, int → int\|}$_{\texttt{M}}$ | true | int $\cdot$ B $\cdot$ B $\cdot\ \epsilon$ | B | int |
| 8 : | putfield {\|B, num, int\|}$_{\texttt{F}}$ | true | int $\cdot$ B $\cdot\ \epsilon$ | B | int |
| 9 : | return | true | $\epsilon$ | B | int |

*Figure 20.* The type information for constructor {|B, <init>, int → void|}$_{\texttt{M}}$ in Figure 2.

valid call to a superclass constructor, and the rest simply preserve the initialization status of the object being constructed. Since all instructions guarded by a handler must agree with the target on initialization status, an exception handler may not guard the call to the superclass constructor.

#### 4.2.3. *Subroutines*

Checking programs with subroutines is difficult for the following reasons:

1. Subroutine calls and returns occur in a stack-like manner. In some cases, however, a subroutine return, exception, or branch instruction may cause a jump to a return address other than that of the most recently called subroutine's caller.
2. A specific form of local variable polymorphism introduced by subroutines must be used correctly. Local variables not touched by a subroutine may contain values of conflicting types at different calls to the subroutine. The types of these variables are preserved across the subroutine call so that they may be used again once the subroutine exits.

Our mechanism for checking subroutines uses information about the subroutine call graph, and, in particular, the set of all valid subroutine call stacks for each instruction $i$. A call stack is a sequence of return addresses representing the stack of subroutines that have been called in a method but that have not yet returned. We capture the set of all possible call stacks for line $i$, $G_i$, by examining the structure of the code array and exception handlers.

Figure 21 presents the rules to specify $G$, and Figure 22 shows a sample JVML$_f$ program with subroutines and its labeling and type information. The dominator set $D_{P,[b,e)}$ is the set of all subroutines that dominate every path to each instruction in the range $[b, e)$. The set is defined by the following equations:

$$D_{P,i} \stackrel{\text{def}}{=} \{L \mid \forall \rho \in G_i.\ \exists p \in \rho.\ P[p-1] = \texttt{jsr}\ \ L\},$$
$$D_{P,[b,e)} \stackrel{\text{def}}{=} \{L \mid \forall i \in [b,e).\ L \in D_{P,i}\}.$$

In essence, these labeling rules ensure that all instructions considered part of subroutine $L$ are associated with the same set of subroutine call stacks. The rule for `jsr` also rules out cycles in the subroutine call graph. One or more subroutines may be implicitly popped off the subroutine call stack if an exception causes a jump to a handler located outside of the currently executing subroutine. To be able to type check such situations, the labeling rules require that the target of an exception handler belong to a subroutine dominating every instruction protected by the handler.

More than one valid labeling may exist for a method body $P$. To simplify the static semantics, we shall refer to a canonical labeling for a method body, $G_P$. We define $G_P$ to be the labeling that conforms to all rules above and also contains no extra call stacks. In other words, no extraneous subroutine call stacks are incorporated into $G_{P,L}$ at the beginning of each subroutine $L$. Also, we assume that

[LAB JSR]

$$\frac{\begin{array}{c}P[i] = \texttt{jsr } L \\ \forall \rho \in G_i.\ \forall p \in \rho.\ P[p-1] \neq \texttt{jsr } L \\ \{(i+1)\cdot\rho \mid \rho \in G_i\} \subseteq G_L \\ G_{i+1} = G_i\end{array}}{G, i \vdash P \text{ labeled}}$$

[LAB IFEQ]

$$\frac{\begin{array}{c}P[i] = \texttt{ifeq } L \\ G_L = G_i = G_{i+1}\end{array}}{G, i \vdash P \text{ labeled}}$$

[LAB GOTO]

$$\frac{\begin{array}{c}P[i] = \texttt{goto } L \\ G_L = G_i\end{array}}{G, i \vdash P \text{ labeled}}$$

[LAB RET THROW]

$$\frac{\begin{array}{c}P[i] \in \{\texttt{ret } x, \texttt{return}, \\ \qquad \texttt{returnval}, \texttt{throw}\}\end{array}}{G, i \vdash P \text{ labeled}}$$

[LAB NORMAL]

$$\frac{\begin{array}{c}\forall x, L.\ P[i] \notin \{\texttt{jsr } L, \texttt{ifeq } L, \texttt{goto } L, \texttt{ret } x, \\ \qquad \texttt{return}, \texttt{returnval}, \texttt{throw}\} \\ G_i = G_{i+1}\end{array}}{G, i \vdash P \text{ labeled}}$$

[LAB HANDLER 0]

$$\frac{\begin{array}{c}G_t = \{\epsilon\}\end{array}}{G, \langle b,\ e,\ t,\ \sigma\rangle \vdash P \text{ labeled}}$$

[LAB HANDLER 1]

$$\frac{\begin{array}{c}G_t = G_L \\ L \in D_{P,[b,e)}\end{array}}{G, \langle b,\ e,\ t,\ \sigma\rangle \vdash P \text{ labeled}}$$

*Figure 21.* JVML$_f$ labeling rules.

| $i$ | $P[i]$ | $G_{P,i}$ | $S_i$ | $F_i[1]$ | $F_i[2]$ | $F_i[3]$ |
|---|---|---|---|---|---|---|
| 1 : | `jsr 3` | $\epsilon$ | $\epsilon$ | `Top` | `Top` | `Top` |
| 2 : | `return` | $\epsilon$ | $\epsilon$ | `Top` | `Top` | `Top` |
| 3 : | `store 1` | $2\cdot\epsilon$ | `(Ret 3)` $\cdot\,\epsilon$ | `Top` | `Top` | `Top` |
| 4 : | `jsr 7` | $2\cdot\epsilon$ | $\epsilon$ | `(Ret 3)` | `Top` | `Top` |
| 5 : | `jsr 10` | $2\cdot\epsilon$ | $\epsilon$ | `(Ret 3)` | `Top` | `Top` |
| 6 : | `ret 1` | $2\cdot\epsilon$ | $\epsilon$ | `(Ret 3)` | `Top` | `Top` |
| 7 : | `store 2` | $5\cdot2\cdot\epsilon$ | `(Ret 7)` $\cdot\,\epsilon$ | | `Top` | `Top` |
| 8 : | `jsr 10` | $5\cdot2\cdot\epsilon$ | $\epsilon$ | | `(Ret 7)` | `Top` |
| 9 : | `ret 2` | $5\cdot2\cdot\epsilon$ | $\epsilon$ | | `(Ret 7)` | `Top` |
| 10 : | `store 3` | $6\cdot2\cdot\epsilon,\ 9\cdot5\cdot2\cdot\epsilon$ | `(Ret 10)` $\cdot\,\epsilon$ | | | `Top` |
| 11 : | `ret 3` | $6\cdot2\cdot\epsilon,\ 9\cdot5\cdot2\cdot\epsilon$ | $\epsilon$ | | | `(Ret 10)` |

*Figure 22.* The type information computed for a method using subroutines.

the labeling rules for exception handlers always match the most recently called dominator in $G_P$. While this assumption prevents a small set of programs with valid labels from being assigned a canonical labeling, we have not encountered these programs in practice.

The typing rules for subroutine call and return appear in Figure 17. In those rules, the domains of the local variable maps are restricted inside subroutines, and the type `(Ret` $L$`)` is assigned to the return address generated by a `jsr` $L$ instruction. The types of local variables over which the current instruction is polymorphic depends on execution history. The auxiliary function $\mathcal{F}$, defined in Figure 23, recovers these types: $\mathcal{F}(F, pc, \rho)[y] = \tau$ only if local variable $y$ can be assigned type $\tau$ at line $pc$ of the program given subroutine call stack $\rho$ implicit in the execution history. In addition, the hiding function $\mathcal{H}(P, \tau, \rho)$ equals `Top` if $\tau$ is a return address type inconsistent with all return addresses is in $\rho$. If $\tau$ is the type

$$[\text{TT } 0]\ \frac{x \in Dom(F_{pc})}{\mathcal{F}(F, pc, \rho)[x] = F_{pc}[x]} \qquad [\text{TT } 1]\ \frac{x \notin Dom(F_{pc}) \quad \mathcal{F}(F, p, \rho)[x] = \tau}{\mathcal{F}(F, pc, p \cdot \rho)[x] = \tau}$$

$$[\text{HT } 0]\ \frac{\tau \notin Ret}{\mathcal{H}(P, \tau, \rho) = \tau} \qquad [\text{HT } 1]\ \frac{\forall p \in \rho.\ P[p-1] \neq \texttt{jsr}\ \ L}{\mathcal{H}(P, (\texttt{Ret}\ \ L), \rho) = \texttt{Top}}$$

$$[\text{HT } 2]\ \frac{P[p-1] = \texttt{jsr}\ \ L}{\mathcal{H}(P, (\texttt{Ret}\ \ L), \rho \bullet (p \cdot \rho')) = (\texttt{Ret}\ \ L)}$$

$$[\text{HT } \epsilon]\ \frac{}{\mathcal{H}(P, \epsilon, \rho) = \epsilon} \qquad [\text{HT SEQ}]\ \frac{\mathcal{H}(P, \tau, \rho) = \tau' \quad \mathcal{H}(P, \beta, \rho) = \beta'}{\mathcal{H}(P, \tau \cdot \beta, \rho) = \tau' \cdot \beta'} \qquad [\text{HT } <:]\ \frac{\mathcal{H}(P, \tau, \rho) = \tau'' \quad \Gamma \vdash \tau'' <: \tau'}{\Gamma \vdash \mathcal{H}(P, \tau, \rho) <: \tau'}$$

*Figure 23.* Auxiliary rules for `jsr` and `ret`.

of a return address appearing in $\rho$, or any type other than a return address type, then $\mathcal{H}(P, \tau, \rho) = \tau$.

We prohibit uninitialized object types from propagating to the type information for successors of `jsr` and `ret` instructions to prevent an error that we found in the original Sun verifier [18]. To do this, we compute the type information for successor instructions based on stack and local variable type information in which uninitialized object types present at line $i$ have been converted to `Top`. For any type $\tau$, the function $HideUninit(\tau)$ is defined as follows:

$$HideUninit(\tau) = \begin{cases} \tau & \text{if } \tau \notin Uninit, \\ \texttt{Top} & \text{if } \tau \in Uninit. \end{cases}$$

We extend the definition to cover sequence and map types in the obvious way. We discuss the typing rules for exception handlers in Section 4.2.5.

Soundness of these rules depends on invariants showing that the implicit subroutine call stack is always contained in the sets of statically computed call stacks and that a return address type `(Ret` $L$`)` is assigned only to a valid return address for $L$ present in the implicit call stack.

Our labeling strategy is based on the labeling technique of Stata and Abadi [43] and offers several improvements over their system. Stata and Abadi labeled each instruction with a linearization of all subroutine calling sequences leading to it, but this linearization contains too little information to check multilevel returns and exceptions handlers.

### 4.2.4. *Limitations of the Labeling Rules*

One limitation of our labeling technique is the treatment of branch-and-jump statements, such as `goto`. These instructions may implicitly pop subroutines off the subroutine call stack when they are executed. Figure 24 demonstrates such a program. The `goto` at line 16 implicitly leaves the subroutine starting at line 10. We cannot assign a valid labeling to the program because the labeling rules require that the `goto` instruction and its target belong to the same subroutine. Compilation of `break` statements within finally clauses can cause similar situations.

To partially address this limitation, we could relax the labeling rule for the `goto` instruction to require only that the target belong to a subroutine dominating the `goto` instruction, as we did in the labeling rule for exception handler targets. However, this approach would lead to complex typing rules for all branch instructions, and we have not fully explored the implications of such rules for our checking algorithm.

```
void f() {
  int i = 0;
  while (true) {
    try {
      ...
    } finally {
      if (i == 0) continue;
      ...
    }
  }
}
```

```
 1 : push 0
 2 : store 1     ; set i to 0
 3 : ...         ; code from try block
 4 : jsr 10      ; jump to subroutine
 5 : goto 3      ; goto top of loop
 6 : store 2     ; store exception value
 7 : jsr 10      ; jump to subroutine
 8 : load 2      ; load exception value
 9 : throw       ; re-throw exception
10 : store 3     ; store return address
11 : load 1      ; load i
12 : push 0      ; push 0
13 : ifeq 16     ; branch if i == 0
14 : ...         ;
15 : ret 3       ; return from subroutine
16 : goto 3      ; jump out of subroutine with goto
Exception table:
   from   to  target  type
     3     4    6     Throwable
```

*Figure 24.* A Java program whose translation into JVML is rejected by our type system.

Several recent approaches to bytecode verification handle programs like the one in Figure 24 more effectively. They forego labeling in favor of analysis techniques that enforce less stringent structure on code with subroutines [41, 27, 5]. We discuss them in detail in Section 9.

### 4.2.5. *Exception Handler Typing Rules*

Exception handlers place additional typing requirements on $F$ and $S$. Given $F$, $S$, and $P$, an exception handler $\langle b,\ e,\ t,\ \sigma \rangle$ is well typed if

- $[b, e)$ is a valid interval in $Dom(P)$, and $t \in Dom(P)$,
- $\sigma$ is a subclass of `Throwable`,
- $S_t$ is a valid type for a stack containing only a single reference to a $\sigma$ object, and
- $F_t$ assigns types to local variables that are at least as general as the types of those local variables at all program points protected by the handler.

We show the typing rules to capture these requirements in Figure 25. The *GoodTarget* judgment verifies that the domain of the local variable map at the instruction to which the handler jumps is the same as it is in the rest of the subroutine. The *GoodExceptionJump* judgment verifies local variable types at the target, given a particular subroutine call stack. It employs checks similar to rule [RET] because the process of catching an exception may change the implicit subroutine call stack in a way similar to a multilevel return.

[WT HANDLER]

$$\frac{\begin{array}{c}\Gamma \vdash \sigma <: \texttt{Throwable} \\ 1 \le b < e \\ b, e-1, t \in Dom(P) \\ GoodTarget(P, F, t) \\ \forall i \in [b, e).\ \forall \rho \in G_{P,i}.\ GoodExceptionJump(\Gamma, P, F, i, \rho, t) \\ \Gamma \vdash \sigma \cdot \epsilon <: S_t\end{array}}{\Gamma, F, S \vdash \langle b,\ e,\ t,\ \sigma \rangle \text{ handles } P}$$

[GT 0]

$$\frac{\begin{array}{c}G_{P,t} = \{\epsilon\} \\ Dom(F_t) = \text{VAR}\end{array}}{GoodTarget(P, F, t)}$$

GT 1

$$\frac{\begin{array}{c}p \cdot \rho \in G_{P,t} \\ P[p-1] = \texttt{jsr}\ \ L \\ Dom(F_t) = Dom(F_L)\end{array}}{GoodTarget(P, F, t)}$$

[EXC JUMP]

$$\frac{\begin{array}{c}\rho' \in G_{P,t} \\ \forall y \in \text{VAR}.\ \forall \tau, \tau'.\ \left(\begin{array}{c}\mathcal{F}(F, i, \rho \bullet \rho')[y] = \tau \\ \wedge\ \mathcal{F}(F, t, \rho')[y] = \tau'\end{array}\right) \Rightarrow \Gamma \vdash \mathcal{H}(P, \tau, \rho') <: \tau'\end{array}}{GoodExceptionJump(\Gamma, P, F, i, \rho \bullet \rho', t)}$$

*Figure 25.* Exception handler typing rules.

## 5. Soundness

In our machine model, a program that attempts to perform an operation causing a type error gets stuck because those operations are not defined. By proving that well-typed programs do not get stuck, we know that well-typed programs will not attempt to perform illegal operations when executed. The rest of this section gives a high-level overview of the formal statement and proof of this soundness property.

We first present in Figure 26 rules to map run-time values to types. We start with rules for primitive values and lead up to a rule that types an entire heap, which is well typed if every record in the heap is well typed. The function *TagToType* converts a tag to a type, erasing the extra information kept in the tag of uninitialized objects. Thus, $\mathit{TagToType}(\varphi\Diamond(\texttt{Uninit}\ \sigma\ j)) = (\texttt{Uninit}\ \sigma\ j)$, and $\mathit{TagToType}(\tau) = \tau$ if $\tau$ is any other kind of tag.

Execution steps preserve invariants relating the run-time state to the static type information, and each step maintains a well-typed heap.

THEOREM 1 (JVML$_f$ One-Step Soundness). *Given* $\Gamma \vdash \texttt{wt}$*:*

$$
\begin{array}{l}
\forall A, A', h, h'. \\
\quad \Gamma \vdash h\ \texttt{wt} \\
\quad\ \wedge\ \mathit{GoodStack}(\Gamma, A, h) \\
\quad\ \wedge\ \Gamma \vdash A; h \rightarrow A'; h' \\
\Rightarrow \Gamma \vdash h'\ \texttt{wt} \\
\quad\ \wedge\ \mathit{GoodStack}(\Gamma, A', h')
\end{array}
$$

$$
\frac{v \in \text{values of type } \tau \qquad \tau \in \mathit{Prim} \cup \{\texttt{Null}\}}{\Gamma, h \vdash v : \tau}\ [\textsc{const}]
\qquad
\frac{K, L \in \textsc{Addr}}{\Gamma, h \vdash K : (\texttt{Ret}\ L)}\ [\textsc{addr}]
$$

$$
\frac{h[a] = \langle\!\langle \{\!|\sigma_i, l_i, \kappa_i|\!\}_{\mathtt{F}} = v_i \rangle\!\rangle_{\sigma}^{i \in I} \qquad \Gamma[\sigma].\texttt{fields} = \{\!|\sigma_i, l_i, \kappa_i|\!\}_{\mathtt{F}}^{i \in I} \qquad \forall i \in I.\ \Gamma \vdash \mathit{Tag}(h, v_i) <: \kappa_i}{\Gamma, h \vdash a : \sigma}\ [\textsc{obj}]
$$

$$
\frac{h[a] = \langle\!\langle \{\!|\sigma_i, l_i, \kappa_i|\!\}_{\mathtt{F}} = v_i \rangle\!\rangle_{\varphi\Diamond(\texttt{Uninit}\ \sigma\ j)}^{i \in I} \qquad \Gamma[\varphi].\texttt{fields} = \{\!|\sigma_i, l_i, \kappa_i|\!\}_{\mathtt{F}}^{i \in I} \qquad \Gamma \vdash \varphi <:_C \sigma \qquad \forall i \in I.\ \Gamma \vdash \mathit{Tag}(h, v_i) <: \kappa_i}{\Gamma, h \vdash a : (\texttt{Uninit}\ \sigma\ j)}\ [\textsc{uninit obj}]
$$

$$
\frac{h[a] = [[v_i]]_{(\texttt{Array}\ \tau)}^{i \in [0..n-1]} \qquad \forall i \in [0..n-1].\Gamma \vdash \mathit{Tag}(h, v_i) <: \tau}{\Gamma, h \vdash a : (\texttt{Array}\ \tau)}\ [\textsc{array}]
$$

$$
\frac{}{\Gamma, h \vdash \epsilon : \epsilon}\ [\epsilon\ \textsc{val}]
\qquad
\frac{\Gamma, h \vdash v : \tau \qquad \Gamma, h \vdash s : \beta}{\Gamma, h \vdash v \cdot s : \tau \cdot \beta}\ [\textsc{seq val}]
$$

$$
\frac{\Gamma, h \vdash v : \tau_1 \qquad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma, h \vdash v : \tau_2}\ [\textsc{subsumption}]
\qquad
\frac{\forall a \in \mathit{Dom}(h).\ \Gamma, h \vdash a : \mathit{TagToType}(\mathit{Tag}(h, a))}{\Gamma \vdash h\ \texttt{wt}}\ [\textsc{wt heap}]
$$

*Figure 26.* Typing rules for values.

[GS 0]

$$\frac{}{\mathit{GoodStack}(\Gamma, \epsilon, h)}$$

[GS 1]

$$\frac{\begin{array}{c} M = \{\!|\sigma_m,\ m,\ \epsilon \to \texttt{void}|\!\}_{\texttt{M}} \\ m \neq \texttt{<init>} \\ \exists \rho.\ \mathit{GoodFrame}(\Gamma, F, S, P, pc, f, s, \rho, h) \end{array}}{\mathit{GoodStack}(\Gamma, \langle M,\ pc,\ f,\ s,\ \emptyset\rangle \cdot \epsilon, h)}$$

[GS METH]

$$\frac{\begin{array}{c} M = \{\!|\sigma_m,\ m,\ \alpha_m \to \gamma_m|\!\}_{\texttt{M}} \\ m \neq \texttt{<init>} \\ \Gamma[M] = \langle P,\ H,\ F,\ S\rangle \\ A' = \langle M',\ pc',\ f',\ s',\ z'\rangle \cdot A_1 \\ M'_\Gamma[pc'] = \texttt{invokevirtual}\ \{\!|\varphi_m,\ m,\ \alpha_m \to \gamma_m|\!\}_{\texttt{M}} \\ \Gamma \vdash \sigma_m <: \varphi_m \\ \mathit{GoodStack}(\Gamma, A', h) \\ \exists \rho. \mathit{GoodFrame}(\Gamma, F, S, P, pc, f, s, \rho, h) \end{array}}{\mathit{GoodStack}(\Gamma, \langle M,\ pc,\ f,\ s,\ \emptyset\rangle \cdot A', h)}$$

[GS INT]

$$\frac{\begin{array}{c} M = \{\!|\sigma_m,\ m,\ \alpha_m \to \gamma_m|\!\}_{\texttt{M}} \\ m \neq \texttt{<init>} \\ \Gamma[M] = \langle P,\ H,\ F,\ S\rangle \\ A' = \langle M',\ pc',\ f',\ s',\ z'\rangle \cdot A_1 \\ M'_\Gamma[pc'] = \texttt{invokeinterface}\ \{\!|\omega_m,\ m,\ \alpha_m \to \gamma_m|\!\}_{\texttt{I}} \\ \Gamma \vdash \sigma_m <: \omega_m \\ \mathit{GoodStack}(\Gamma, A', h) \\ \exists \rho. \mathit{GoodFrame}(\Gamma, F, S, P, pc, f, s, \rho, h) \end{array}}{\mathit{GoodStack}(\Gamma, \langle M,\ pc,\ f,\ s,\ \emptyset\rangle \cdot A', h)}$$

[GS CSTR]

$$\frac{\begin{array}{c} M = \{\!|\sigma_m,\ \texttt{<init>},\ \alpha_m \to \texttt{void}|\!\}_{\texttt{M}} \\ \Gamma[M] = \langle P,\ H,\ F,\ S\rangle \\ A' = \langle M',\ pc',\ f',\ s_1 \bullet (a \cdot s'),\ z'\rangle \cdot A_1 \\ M'_\Gamma[pc'] = \texttt{invokespecial}\ M \\ |\alpha_m| = |s_1| \\ \mathit{GoodStack}(\Gamma, A', h) \\ \exists \rho.\ \mathit{GoodFrame}(\Gamma, F, S, P, pc, f, s, \rho, h) \\ \mathit{GoodConstructor}(P, pc, \sigma_m, h, a, \langle b,\ c\rangle) \\ \mathit{Corresponds}(\Gamma, F_{pc}, S_{pc}, f, s, h, b, (\texttt{Uninit}\ \sigma_m\ 0)) \end{array}}{\mathit{GoodStack}(\Gamma, \langle M,\ pc,\ f,\ s,\ \langle b,\ c\rangle\rangle \cdot A', h)}$$

[GS EXC]

$$\frac{\begin{array}{c} \mathit{Tag}(h, a) = \sigma \\ \Gamma \vdash \sigma <: \texttt{Throwable} \\ \mathit{GoodStack}(\Gamma, A', h) \end{array}}{\mathit{GoodStack}(\Gamma, \langle a\rangle_{\text{exc}} \cdot A', h)}$$

$$\begin{array}{l} \mathit{GoodFrame}(\Gamma, F, S, P, pc, f, s, \rho, h) \stackrel{\text{def}}{=} \\ \quad pc \in \mathit{Dom}(P) \\ \quad \wedge\ \Gamma, h \vdash s : S_{pc} \\ \quad \wedge\ \forall y \in \textsc{Var}.\ \exists \tau.\ \mathcal{F}(F, pc, \rho)[y] = \tau \wedge \Gamma, h \vdash f[y] : \tau \\ \quad \wedge\ \mathit{ConsistentInit}(\Gamma, F_{pc}, S_{pc}, f, s, h) \\ \quad \wedge\ \mathit{ConsistentSub}(F, S, P, pc, f, s, \rho) \end{array}$$

*Figure 27.* *GoodStack* and *GoodFrame*.

The *GoodStack* judgment, defined in Figure 27, requires that the state of the machine be consistent with the static type information for the program. There is one rule for each of the six possible stack configurations. The stack may

- be empty,
- have exactly one activation record,
- have an activation record on the top that was created by either an `invokevirtual`, `invokeinterface`, or `invokespecial` instruction, or
- have an exception activation record on the top.

In these rules, it is convenient to assume that there is a canonical $F$ and $S$ for each method, and we let $\Gamma[M]$ now return a record $\langle P, H, F, S\rangle$ containing the code array, exception handlers, and canonical type information for method $M$.

The basic format for each rule is the same. If there is a preceding activation record below the top one, the rules ensure that the proper relationship between that activation record and the topmost one is met. For example, if the topmost activation record were created by a virtual method invocation, rule [GS METH] ensures that the method called to create it was properly dispatched. In addition, the *GoodFrame* invariant must be satisfied by each activation record on the stack.

The five conjuncts of *GoodFrame* have the following meaning:

$pc \in Dom(P)$: The program counter is within the code array for the method.

$\Gamma, h \vdash s : S_{pc}$: The stack has values of the expected types for the current line of execution.

$\forall y \in \text{VAR}.\ \exists \tau.\ \mathcal{F}(F, pc, \rho)[y] = \tau \wedge \Gamma, h \vdash f[y] : \tau$: The local variables contain values of the expected types for the current line of execution and subroutine call history captured in $\rho$.

*ConsistentInit*$(\Gamma, F_{pc}, S_{pc}, f, s, h)$: The invariants ensuring the correctness of the alias analysis for tracking uninitialized object references are satisfied by the current state of the machine. This judgment is defined in Figure 28, and it shows that each uninitialized object type maps to a unique run-time value.

[CONS INIT]

$$\frac{\forall \sigma, j.\ \exists b.\ \Gamma \vdash b : (\texttt{Uninit}\ \sigma\ j) \wedge \mathit{Corresponds}(\Gamma, F_i, S_i, f, s, h, b, (\texttt{Uninit}\ \sigma\ j))}{\mathit{ConsistentInit}(\Gamma, F_i, S_i, f, s, h)}$$

[CORR]

$$\frac{\forall x \in \mathit{Dom}(F_i).\ F_i[x] = (\texttt{Uninit}\ \sigma\ j) \implies \begin{pmatrix} f[x] = b \\ \wedge\ \Gamma, h \vdash b : (\texttt{Uninit}\ \sigma\ j) \end{pmatrix} \quad \mathit{StackCorresponds}(\Gamma, S_i, s, h, b, (\texttt{Uninit}\ \sigma\ j))}{\mathit{Corresponds}(\Gamma, F_i, S_i, f, s, h, b, (\texttt{Uninit}\ \sigma\ j))}$$

[SC 0]

$$\frac{}{\mathit{StackCorresponds}(\Gamma, \epsilon, \epsilon, h, b, (\texttt{Uninit}\ \sigma\ j))}$$

[SC 1]

$$\frac{\Gamma, h \vdash b : (\texttt{Uninit}\ \sigma\ j) \quad \mathit{StackCorresponds}(\Gamma, S_i, s, h, b, (\texttt{Uninit}\ \sigma\ j))}{\mathit{StackCorresponds}(\Gamma, (\texttt{Uninit}\ \sigma\ j) \cdot S_i, b \cdot s, h, b, (\texttt{Uninit}\ \sigma\ j))}$$

[SC 2]

$$\frac{\tau \neq (\texttt{Uninit}\ \sigma\ j) \quad \mathit{StackCorresponds}(\Gamma, S_i, s, h, b, (\texttt{Uninit}\ \sigma\ j))}{\mathit{StackCorresponds}(\Gamma, \tau \cdot S_i, v \cdot s, h, b, (\texttt{Uninit}\ \sigma\ j))}$$

*Figure 28.* Invariants for object initialization.

[RET CORR]

$$\frac{\forall y \in \text{VAR}.\ \forall L.\ \mathcal{F}(F, pc, \rho)[y] = (\texttt{Ret}\ \ L) \implies \begin{pmatrix} f[y] \in \rho \\ \wedge\ P[f[y]-1] = \texttt{jsr}\ \ L \end{pmatrix}}{\begin{array}{c} \mathit{StackRetCorresponds}(P, S_{pc}, s, \rho) \\ \hline \mathit{RetCorresponds}(P, F, S, pc, f, s, \rho) \end{array}}$$

[RSC 0]

$$\frac{}{\mathit{StackRetCorresponds}(P, \epsilon, \epsilon, \rho)}$$

[RSC 1]

$$\frac{\begin{array}{c} p \in \rho \\ P[p-1] = \texttt{jsr}\ \ L \\ \mathit{StackRetCorresponds}(P, S_i, s, \rho) \end{array}}{\mathit{StackRetCorresponds}(P, (\texttt{Ret}\ \ L) \cdot S_i, p \cdot s, \rho)}$$

[RSC 2]

$$\frac{\begin{array}{c} \tau \notin \mathit{Ret} \\ \mathit{StackRetCorresponds}(P, S_i, s, \rho) \end{array}}{\mathit{StackRetCorresponds}(P, \tau \cdot S_i, v \cdot s, \rho)}$$

[WF 1]

$$\frac{\begin{array}{c} \mathit{Dom}(F_{pc}) = \text{VAR} \\ G_{P,pc} = \{\epsilon\} \end{array}}{\mathit{WFCallStack}(P, F, pc, \epsilon)}$$

[WF 2]

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

[NHU]

$$\frac{\forall y \in \text{VAR} \setminus \mathit{Dom}(F_{pc}).\ \forall \tau.\ \mathcal{F}(F, pc, \rho)[y] = \tau\ \Rightarrow\ \tau \notin \mathit{Uninit}}{\mathit{NoHiddenUninit}(F, pc, \rho)}$$

*Figure 29.* Subroutine call stack invariants.

*ConsistentSub*$(F, S, P, pc, f, s, \rho)$: The subroutine return address stack $\rho$ is consistent with the program state and type information.

The *ConsistentSub* predicate is defined as follows:

$$\begin{aligned} &\mathit{ConsistentSub}(F, S, P, pc, f, s, \rho) \stackrel{\text{def}}{=} \\ &\quad \mathit{WFCallStack}(P, F, pc, \rho) \\ &\quad \wedge\ \mathit{RetCorresponds}(F, S, P, pc, f, s, \rho) \\ &\quad \wedge\ \mathit{NoHiddenUninit}(F, pc, \rho) \end{aligned}$$

The inference rules for these three statements appear in Figure 29. If the judgment *RetCorresponds*$(F, S, P, pc, f, s, \rho)$ is derivable, the return addresses stored in the local variables and stack correspond to return addresses appearing in $\rho$. The *WFCallStack* judgment ensures that the local variable map domain and set of possible subroutine call stacks is constant for the duration of a subroutine call. Finally, the last rule in the figure enforces the requirement that no uninitialized object types appear in local variables not visible to the current subroutine.

Constructors require one additional execution invariant to track which object is being constructed and where the initialized form is stored (recall that constructors

[GCSTR 0]

$$\frac{\begin{array}{c} Z_{P,pc} = \mathit{false} \\ \mathit{Tag}(h, a) = \varphi\Diamond(\texttt{Uninit}\ \sigma'\ j) \\ \mathit{Tag}(h, b) = \varphi\Diamond(\texttt{Uninit}\ \sigma\ 0) \end{array}}{\mathit{GoodConstructor}(P, pc, \sigma, h, a, \langle b, \texttt{null}\rangle)}$$

[GCSTR 1]

$$\frac{\begin{array}{c} Z_{P,pc} = \mathit{true} \\ \mathit{Tag}(h, a) = \varphi\Diamond(\texttt{Uninit}\ \sigma'\ j) \\ \mathit{Tag}(h, b) = \varphi\Diamond(\texttt{Uninit}\ \sigma\ 0) \\ \mathit{Tag}(h, c) = \varphi \end{array}}{\mathit{GoodConstructor}(P, pc, \sigma, h, a, \langle b, c\rangle)}$$

*Figure 30.* Constructor invariants.

create new objects to preserve a heap monotonicity property). The *GoodConstructor* judgment in Figure 30 captures this information, and it is used in the [METH CSTR] rule.

We prove Theorem 1 by case analysis on the operational semantics rule used by the execution step. The proof leverages the soundness proofs for $\text{JVML}_0$ [43] and $\text{JVML}_i$ [18]; but adding heaps, object references, arrays, and exceptions introduces many additional cases. We sketch the proof for `store` $x$ to motivate the structure of the invariants we have presented, but we refer the reader to Freund's thesis for a more thorough treatment [15].

We shall explore two key properties about the execution of `store`. First, we show that execution of a `store` instruction in a virtual machine configuration $A; h$, where $\Gamma \vdash h$ `wt` and $\mathit{GoodStack}(\Gamma, A, h)$, yields a new well-typed heap. In general, we prove this statement for any instruction by noting that all heap updates respect three properties: (1) the types of records themselves never change, (2) values written into heap records have the same types as the overwritten values, and (3) any new records introduced by allocation are well-typed records. If an instruction changes a heap $h$ into heap $h'$ according to these three rules, then $h'$ will be a well-typed heap. For `store`, this is trivial to show since $h$ is not modified. A related property guaranteed by all instructions is that if $\mathit{GoodStack}(\Gamma, A, h)$ is derivable, then $\mathit{GoodStack}(\Gamma, A, h')$ will also be derivable, as shown by induction over the derivation of $\mathit{GoodStack}(\Gamma, A, h)$. This observation is important for proving the second key property of instruction execution.

The second property is that executing a `store` instruction preserves all invariants listed in *GoodFrame*. Suppose a `store` instruction moves the virtual machine from configuration $\langle M, pc, f, v \cdot s, z\rangle \cdot A; h$ to $\langle M, pc + 1, f[x \mapsto v], s, z\rangle \cdot A; h$. Further suppose that $P$ is the method body for $M$ and that $F$ and $S$ comprise the type information used to show that $\Gamma, F, S, pc \vdash P : M$. We proceed by showing that if $\mathit{GoodFrame}(\Gamma, F, S, P, pc, f, v \cdot s, \rho, h)$ holds for some $\rho$, then $\mathit{GoodFrame}(\Gamma, F, S, P, pc + 1, f[x \mapsto v], s, \rho, h)$ will also hold. We consider each conjunct of $\mathit{GoodFrame}(\Gamma, F, S, P, pc + 1, f[x \mapsto v], s, \rho, h)$ separately:

$pc + 1 \in \mathit{Dom}(P)$: The conclusion $\Gamma, F, S, pc \vdash P : M$ could only be derived by rule [STORE]. As a requirement of that rule, $pc + 1 \in \mathit{Dom}(P)$.

$\Gamma, h \vdash s : S_{pc+1}$: As a requirement of rule [STORE], we know that $\Gamma \vdash S_{pc} <: \tau \cdot S_{pc+1}$ for some $\tau$. Since $\Gamma, h \vdash v \cdot s : S_{pc}$, we can conclude that $\Gamma, h \vdash v \cdot s : \tau \cdot S_{pc+1}$ by [SUBSUMPTION]. Thus, $\Gamma, h \vdash s : S_{pc+1}$.

$\forall y \in \text{VAR}.\ \exists \tau'.\ \mathcal{F}(F, pc+1, \rho)[y] = \tau' \wedge \Gamma, h \vdash f([x \mapsto v])[y] : \tau'$: The only variable of interest is $x$. The static semantics for [STORE] require that $x$ be in $Dom(F_{pc})$ and, therefore, in $Dom(F_{pc+1})$. (We know that $Dom(F_{pc}) = Dom(F_{pc+1})$ because $F_{pc}[x \mapsto v]$ is a subtype of $F_{pc+1}$, and a map can only be a subtype of another map with the same domain.) Thus, $\mathcal{F}(F, pc+1, \rho)[x] = F_{pc+1}[x]$. Pick $\tau$ such that $\Gamma \vdash S_{pc} <: \tau \cdot S_{pc+1}$, as required by rule [STORE]. Since $\Gamma, h \vdash v \cdot s : S_{pc}$, we know that $\Gamma, h \vdash v : \tau$. After the instruction executes, $(f[x \mapsto v])[x] = v$, and according to rule [STORE], $\Gamma \vdash \tau <: F_{pc+1}[x]$. Thus, $\Gamma, h \vdash (f[x \mapsto v])[x] : F_{pc+1}[x]$ by [SUBSUMPTION].

*ConsistentInit*$(\Gamma, F_{pc+1}, S_{pc+1}, f[x \mapsto v], s, h)$: If $v$ is not originally assigned an uninitialized object type when it is on top of the stack, then $F_{pc+1}[x]$ will not have an uninitialized object type, and we may show that *ConsistentInit*$(\Gamma, F_{pc+1}, S_{pc+1}, f[x \mapsto v], s, h)$, given the assumption *ConsistentInit*$(\Gamma, F_{pc}, S_{pc}, f, v \cdot s, h)$. On the other hand, suppose $v$ is an uninitialized object reference and is assigned type `(Uninit` $\sigma$ $j$`)` for some $\sigma$ and $j$. If $F_{pc+1}[x]$ is also assigned the type `(Uninit` $\sigma$ $j$`)`, we must show that all occurrences of `(Uninit` $\sigma$ $j$`)` in $F_{pc+1}$ and $S_{pc+1}$ map to $v$ in $f[x \mapsto v]$ and $s$. Since $v$ was previously on the stack, *ConsistentInit*$(\Gamma, F_{pc}, S_{pc}, f, v \cdot s, h)$ will guarantee that all occurrences of `(Uninit` $\sigma$ $j$`)` in $S_{pc}$ and $F_{pc}$ will correspond to the run-time value $v$ in $f$ and $v \cdot s$, and we may show this property for the post-state as well. The correspondences between all other uninitialized object types and unique run-time values that held in the prestate will still hold after the `store` instruction executes.

*ConsistentSub*$(F, S, P, pc+1, f[x \mapsto v], s, \rho)$: By the structural labeling rules for $P$, $G_{pc+1} = G_{pc}$, and we know that $Dom(F_{pc}) = Dom(F_{pc+1})$. These two facts are sufficient to prove that *WFCallStack*$(P, F, pc+1, \rho)$. The statement *NoHiddenUninit*$(F, pc+1, \rho)$ is also trivial to prove, given that *NoHiddenUninit*$(F, pc, \rho)$ and $\Gamma \vdash F_{pc}[x \mapsto \tau] <: F_{pc+1}$. Proving that *RetCorresponds*$(F, S, P, pc+1, f[x \mapsto v], s, \rho)$ is similar to proving that the alias analysis for object initialization works correctly. In essence, either the value $v$ does not originally have a return address type, or it has type `(Ret` $L$`)` and $v$ is in $\rho$. In the latter case, all occurrences of `(Ret` $L$`)` in the static type information for $pc$ match occurrences of $v$ in $v \cdot s$ and $f$ because *RetCorresponds*$(F, S, P, pc, f, v \cdot s, \rho)$. Given the inference rules for *RetCorresponds*, we know that the same correspondence holds between the type information for $pc+1$ and $f[x \mapsto v]$ and $s$.

Once we have shown that executing a `store` instruction preserves the *GoodFrame* invariants, it is routine to show that *GoodStack* is also preserved. The same general proof outline may be used to show that all instructions affecting

only the topmost activation record preserve the *GoodStack* invariant. The remain instructions, such as `invokevirtual`, `throw`, and `return`, change more than one activation record, but the design of the JVML$_f$ semantics ensure that the invariants are all preserved. For example, the initial state of a newly created activation record matches the initial conditions for the method type information specified in rule [METH CODE], and the rules for `returnval` ensure that values of the appropriate type are always returned to the caller of a method.

We now state multistep soundness and progress theorems for JVML$_f$.

THEOREM 2 (JVML$_f$ Multistep Soundness). *Given* $\Gamma \vdash \texttt{wt}$*:*

$$\begin{array}{l}\forall A_0, A', h_0, h'. \\ \quad \Gamma \vdash h \ \texttt{wt} \\ \quad\ \wedge\ \mathit{GoodStack}(\Gamma, A, h) \\ \quad\ \wedge\ \Gamma \vdash A_0; h_0 \rightarrow^* A'; h' \\ \Rightarrow \Gamma \vdash h' \ \texttt{wt} \\ \quad\ \wedge\ \mathit{GoodStack}(\Gamma, A', h')\end{array}$$

Proof is by induction on the number of steps taken by the virtual machine, and it is routine once Theorem 1 is established.

THEOREM 3 (JVML$_f$ Progress). *Given* $\Gamma \vdash \texttt{wt}$*:*

$$\begin{array}{l}\forall A, h. \\ \quad \Gamma \vdash h \ \texttt{wt} \\ \quad\ \wedge\ \mathit{GoodStack}(\Gamma, A, h) \\ \Rightarrow \exists A', h'.\ \Gamma \vdash A; h \rightarrow A'; h'\end{array}$$

We prove Theorem 3 by showing that if $\Gamma \vdash h \ \texttt{wt}$ and $\mathit{GoodStack}(\Gamma, A, h)$, then the hypotheses for at least one rule of the operational semantics will be satisfied.

We now state the main soundness theorem for JVML$_f$. In the Java Virtual Machine, execution begins by invoking a static method `main` for some class. Since we have not included static methods, JVML$_f$ programs start in a different way. A program begins by executing a method that takes no arguments and returns no value on an object with no fields. In addition, that object is the only object in the heap. The following theorem states that if a program begins in this way, it will not halt unless the activation record stack becomes empty.

THEOREM 4 (JVML$_f$ Soundness). *If* $\Gamma \vdash \texttt{wt}$*,* $M = \{\!|\sigma, m, \epsilon \rightarrow \texttt{void}|\!\}_{\mathrm{M}}$*,* $M \in \Gamma$*,* $\Gamma[\sigma].\texttt{fields} = \emptyset$*,* $a \in \textsc{Loc}$*, and* $\mathit{Dom}(h_0) = \emptyset$*, then*

$$\begin{array}{l}\forall A, h. \\ \quad \Gamma \vdash \langle M,\ 1,\ f_0,\ a \cdot \epsilon \rangle \cdot \epsilon; h_0[a \mapsto \langle\langle\rangle\rangle_\sigma] \rightarrow^* A; h \\ \quad\ \wedge\ \neg\exists A', h'.\ \Gamma \vdash A; h \rightarrow A'; h' \\ \Rightarrow A = \epsilon\end{array}$$

The proof is a direct application of Theorem 2 and Theorem 3.

## 6. Verifier Algorithm

A Java Virtual Machine must check the two major properties captured by the static semantics: (1) the program environment is well formed, and (2) the code of each method body is well typed. The first property has been extensively studied by others, especially in the presence of dynamic class loading and name resolution (see, for example, [10, 20, 22, 46, 38, 13]). Many of these mechanisms could be merged with our framework. We are primarily interested in how a bytecode verifier can check the second property for the JVML$_f$ type system presented in this paper, and we give an overview of a checking algorithm proven to accept only well-typed method bodies. Given a method body P and the environment for the program, our algorithm computes type information consistent with the rules of the type system in the following order:

1. $G_P$, the function to map an address to the set of possible subroutine calls stacks for that line of code.
2. The domain of the local variable type map $F_i$, for each address $i$.
3. $F$ and $S$, the functions to map an address to the local variable type map and stack type for that line of code.

We separate the algorithm into these three steps so that all edges in the flow graph and local variable uses are known before beginning the dataflow analysis to compute $F$ and $S$. If this information is not precomputed, it becomes more difficult to compute $F$ and $S$ efficiently and to prove that the dataflow analysis algorithm is correct, as discussed in Section 9 where we compare our algorithm to some of the other proposed algorithms.

We begin by computing $G_P$ with an iterative algorithm that uses the transfer function shown in Figure 31. We order sets of possible subroutine call stacks in a lattice by subset inclusion. The lattice is finite for any program because we do not permit call stacks to contain multiple return addresses for the same subroutine. The join operation $G_i \sqcup_G G'_i$ equals $G_i \cup G'_i$, provided that $G_i \cup G'_i$ is *consistent*. A set is consistent if all of its call stacks end with calls to the same subroutine. In other words, $G_i \cup G'_i$ is consistent if it is $\emptyset$, $\{\epsilon\}$, or $\{p_1 \cdot \rho_1, \ldots, p_n \cdot \rho_n\}$ such that, for some $L$, $P[p_i - 1] = \texttt{jsr}\ L$ for all $i$. If $G_i \cup G'_i$ is not consistent, then it contains call stacks created while entering two different subroutines, and $G_i \sqcup_G G'_i$ results in a special error element in the lattice. We define $G \sqcup_G G'$ to be the join of the subroutine call stack sets at each address.

Without exception handlers, we can find $G_P$ by setting each $G_{P,i}$ to $\emptyset$ and repeatedly computing $G_P = G_P \sqcup_G \texttt{CG}(P, G_P, i)$, picking $i$ from $Dom(P)$ on each iteration, until quiescence is reached or an error occurs. Errors occur when either an assertion in CG fails or when the join operation sets some $G_i$ to the error element. We use a slightly different approach to deal with exception handlers. Assigning

```
CG(P, G, i) {
 case P[i] of
  ifeq L:
    G_{i+1} := G_i
    G_L := G_i
  jsr L:
    G_{i+1} := G_i
    assert ∀ρ ∈ G_i. ∀p ∈ ρ. P[p − 1] ≠ jsr L
    G_L := G_L ∪ {(i + 1) · ρ | ρ ∈ G_i}
  ret x:
  ...
 return G
}
```

*Figure 31.* The transfer function for computing $G$.

a label to the target of a handler requires finding the dominating subroutines for it, but the set of dominators for an instruction is not a monotonic property of the lattice described above. Therefore, we fix the order in which our algorithm picks $i$ so that (1) an instruction is visited only after at least one predecessor, (2) all calls to subroutine $L$ are visited before instruction $L$, and (3) the target of an exception handler is visited only after all instructions protected by the handler. This ordering ensures that the full set $G_L$ for a subroutine $L$ is computed before propagating that information inside subroutine $L$, and that the set of dominators for a target is known exactly before visiting the target. When visiting a target $t$, `CG` can then identify the most recently called dominator and set $G_t$ accordingly. An ordering of this form will exist for any well-typed JVML$_f$ program generated by a reasonable Java compiler (the second and third requirements are satisfied because the type system requires that the subroutine call graph be acyclic and because the target of an exception handler will never be protected by the handler).

The next step is to compute the set of local variables in the domain of local variable maps for the instructions in each subroutine. The domain of the local variable map for an instruction in subroutine $L$ contains any variable accessed by the code of $L$, plus the variables accessed by code belonging to subroutines transitively called from inside $L$. A simple depth-first traversal of the subroutine call graph computes the information needed to construct these sets.

The final step, constructing $F$ and $S$, uses dataflow analysis to find a least fixed point in the lattice of potential types for $P$. We translate the JVML$_f$ instruction typing rules into a transfer function, part of which is presented in Figure 32. We assume a fixed environment $\Gamma$ and *Method-Ref* $M = \{\!|\varphi, m, \alpha \to \gamma|\!\}_{\mathtt{M}}$ in the rest of this section. Constraints in the typing rules become assertions and transformations on F and S. The function `First(`$S_i$`)` returns the top element of stack $S_i$, and `Rest(`$S_i$`)` returns stack $S_i$ without the top element.

`Verify` uses `Transfer` to find a valid type for the method if it exists. The initial values $F^0$ and $S^0$ are the least types consistent with the domains of the

```
Transfer(P, G, i, ⟨F, S⟩) {
 case P[i] of
  store x:
    assert x ∈ Dom(F_i)
    S_{i+1} := Rest(S_i)
    F_{i+1} := F_i[x ↦ First(S_i)]
  new σ:
    S_{i+1} := (Uninit σ i) · [Top/(Uninit σ i)]S_i
    F_{i+1} := [Top/(Uninit σ i)]F_i
  ...
 return ⟨F, S⟩
}

Verify(P, G) {
 ⟨F, S⟩ := ⟨F^0, S^0⟩
 repeat
   ⟨F, S⟩ := ⟨F, S⟩ ⊔ Transfer(P, G, i, ⟨F, S⟩) for i ∈ Dom(P)
 until ∀i ∈ Dom(P). Transfer(P, G, i, ⟨F, S⟩) = ⟨F, S⟩
}
```

*Figure 32.* Verify algorithm.

local variable maps and initial conditions on method entry. The type information for $F$ and $S$ is merged with the join operator. The join of two types, written $\tau_1 \sqcup \tau_2$, is the least upper bound of $\tau_1$ and $\tau_2$ in the type lattice induced by the JVML$_f$ subtyping relation, augmented with the least type `Bottom`. The join operation is also extended in the obvious way to $F$ and $S$. For simplicity, we do not present the transfer function for exception handlers, but it is constructed from the typing rules in a similar fashion, and when processing line $i$, `Transfer` would also apply the exception handler transfer function for each handler protecting line $i$.

Given $G_P$ and the domain of each local variable map, the `Verify` algorithm for $F$ and $S$ will accept only method bodies to which rule [METH CODE] may be successfully applied and will accept all method bodies typeable by that rule. The proof of these properties is straightforward, but tedious, because of the size of the type system. Therefore, we state the key theorems and provide a brief sketch of the proofs.

To prove that `Verify` is sound, we show any fixed point $\langle F, S\rangle$ computed by `Transfer` is consistent with all instructions in the code array, that is, $\forall i \in Dom(P).\ \Gamma, F, S, i \vdash P : M$. Demonstrating that the algorithm satisfies the other hypotheses of rule [METH CODE] is straightforward once this property has been established. We first present a theorem stating that `Transfer`$(P, G, i, \langle F, S\rangle)$ returns a type consistent with the typing rule for instruction $i$.

THEOREM 5 (`Transfer` Soundness). *Given code array $P$ with labeling $G$ and $i \in Dom(P)$, for all $F$, $S$, $F'$, and $S'$:*

$$\texttt{Transfer}(P, G, i, \langle F, S\rangle) = \langle F', S'\rangle \implies \Gamma, F', S', i \vdash P : M$$

Consider the `store` instruction. If $P[i] = \texttt{store}\ x$, then $S'_{i+1} = \texttt{Rest}(S_i)$ and $F'_{i+1} = F_i[x \mapsto \texttt{First}(S_i)]$, where $x \in Dom(F'_i)$. In addition, $i + 1 \in Dom(P)$, or else we could not have assigned types to $F'_{i+1}$ and $S'_{i+1}$. We shall use rule [STORE] to derive $\Gamma, F', S', i \vdash P : M$, and we dispatch the remaining hypotheses of that rule. First, if $S'_{i+1} = \texttt{Rest}(S_i)$ and $S'_i = S_i$, then $\Gamma \vdash \texttt{Rest}(S'_i) <: S'_{i+1}$. We may show that $\Gamma \vdash F'_i[x \mapsto \texttt{First}(S_i)] <: F'_{i+1}$ in a similar fashion. All other cases are similar.

During the computation of $F$ and $S$, `Transfer` is repeatedly applied to instructions in $P$ until either (1) an error occurs or (2) quiescence is reached. In the latter case, any additional calls to the function $\texttt{Transfer}(P, G, i, \langle F, S\rangle)$ for any $i$ yields $\langle F, S\rangle$. Therefore, we can use Theorem 5 to conclude that $\forall i \in Dom(P).\ \Gamma, F, S, i \vdash P : M$.

We rely on standard proof techniques for dataflow analysis problems to show that `Verify` is complete (see, for example, [24, 45]). In essence, any valid type $\langle F, S\rangle$ for $P$ will be a fixed point for `Transfer`, and we show that the algorithm will find a fixed point in the type lattice if any such $\langle F, S\rangle$ exists. Showing the following monotonicity property for the transfer function is sufficient.

THEOREM 6 (`Transfer` Monotone). *Given code array $P$ with labeling $G$ and $i \in Dom(P)$, for all $F$, $S$, $F'$, and $S'$:*

$$\begin{aligned} &\Gamma, F', S' \vdash P : M \\ &\quad \wedge\ \Gamma \vdash \langle F, S\rangle <: \langle F', S'\rangle \\ &\Rightarrow\ \Gamma \vdash \texttt{Transfer}(P, G, i, \langle F, S\rangle) <: \texttt{Transfer}(P, G, i, \langle F', S'\rangle) \end{aligned}$$

Suppose that $P[i] = \texttt{store}\ x$. Since $\Gamma, F', S', i \vdash P : M$ must be derivable by rule [STORE], $x$ is in the domain of $F'_i$. In addition, the hypothesis that $\Gamma \vdash F <: F'$ implies that $\Gamma \vdash F_i <: F'_i$, which means $Dom(F_i) = Dom(F'_i)$. Therefore, $x \in Dom(F_i)$. Thus, the `assert` succeeds in both $\texttt{Transfer}(P, G, i, \langle F, S\rangle)$ and $\texttt{Transfer}(P, G, i, \langle F', S'\rangle)$, and neither computation results in an error. Let $\texttt{Transfer}(P, G, i, \langle F, S\rangle)$ return $\langle \bar{F}, \bar{S}\rangle$ and $\texttt{Transfer}(P, G, i, \langle F', S'\rangle)$ return $\langle \bar{F}', \bar{S}'\rangle$. Given the transfer function, $\langle \bar{F}, \bar{S}\rangle$ and $\langle \bar{F}', \bar{S}'\rangle$ will differ from $\langle F, S\rangle$ and $\langle F', S'\rangle$ only at index $i + 1$. Therefore, to show the conclusion of the theorem, it is sufficient to prove that $\Gamma \vdash \bar{S}_{i+1} <: \bar{S}'_{i+1}$ and $\Gamma \vdash \bar{F}_{i+1} <: \bar{F}'_{i+1}$. In order to have concluded $\Gamma \vdash \langle F, S\rangle <: \langle F', S'\rangle$, it must be the case that $\Gamma \vdash S_i <: S'_i$ and $\Gamma \vdash F_{i+1} <: F'_{i+1}$. Given the definition of `Transfer`, the following two equations hold for some $\tau$ and $\tau'$: $S_i = \tau \cdot \bar{S}_{i+1}$ and $S'_i = \tau' \cdot \bar{S}'_{i+1}$, where $\Gamma \vdash \tau <: \tau'$ and $\Gamma \vdash \bar{S}_{i+1} <: \bar{S}'_{i+1}$. Moreover, this implies that $\Gamma \vdash F_{i+1}[x \mapsto \tau] <: F'_{i+1}[x \mapsto \tau']$. Thus, $\Gamma \vdash \bar{F}_{i+1} <: \bar{F}'_{i+1}$. The cases for all other instructions are similar.

## 7. Implementation

We have implemented a prototype JVML verifier that uses the three-phase type checker from the preceding section. Our verifier actually translates each JVML

method body into a slightly modified subset of the JVML$_f$ instruction set in order to handle all of JVML. The translation preserves the structure of the original method body but utilizes a small core verifier to perform the type synthesis. This approach removes as many details as possible from the dataflow analysis implementation. For example, the instruction

```
invokevirtual {|Vector, indexOf, Object → int|}M
```

is replaced by

```
pop<Object>  ; pop argument
pop<Vector>  ; pop receiver
push<int>    ; push integer as return value
```

where an instruction like `pop<Vector>` indicates that a value of type `Vector` should be popped from the top of the stack.

To capture those requirements not checked by this core verifier, such as the requirement that `Vector.indexOf` actually exists and has the correct type, our translator also generates a set of assertions that may be checked separately, either before or after the type checking has been performed. The one constraint generated by the above instruction is

$$\{|\texttt{Vector, indexOf, Object} \rightarrow \texttt{int}|\}_{\texttt{M}} \in \mathit{Dom}(\Gamma)$$

Goldberg has described the form of these assertions and demonstrates how they may be used to construct a typing environment for bytecode verification in the presence of dynamic loading [20].

Our checker has verified a large fraction of the JDK libraries, as well as many examples of Java programs using common idioms for exception handling and the other features of JVML$_f$. The behavior of our algorithm differs slightly from the Sun reference implementation. The most notable deviation is due to the issues described in Section 4.2.4.

The idea of translating a bytecode program into a "micro"-instruction set has been explored by others [49] and seems to be a promising way to keep the size and complexity of the core verifier small.

## 8. Applications

In this section, we explore applications of this work that are both within and outside of the current role of the bytecode verifier.

### 8.1. FORMALLY SPECIFYING THE VERIFIER

The first and most obvious direct application of this type system is to use it as a formal specification of the bytecode verifier. The original informal specification is insufficient to describe a correct implementation, and we have described several

situations in which verifiers incorrectly accepted bad programs in this paper. These errors were caused in part because the verifiers were implemented without a clear specification.

### 8.2. TESTING EXISTING IMPLEMENTATIONS

Another avenue in which to apply this work is to follow the direction of the Kimera project [40]. That project built on traditional software engineering techniques to test bytecode verifiers by automatically generating a large number of faulty class files and looking for inconsistencies in behavior between different bytecode verifier implementations when checking those programs.

Even without automated testing, we have identified several flaws and inconsistencies in existing implementations. In one case, a version of the Sun verifier accepted a program that used an uninitialized object. Another example is that the set of operations allowed on `null` references differed between some early commercial implementations. These issues were uncovered by translating difficult cases from our soundness proofs into sample test programs.

### 8.3. CHECKING ADDITIONAL SAFETY PROPERTIES

We can also check additional safety properties for bytecode programs. One example of this is to check that `monitorenter` and `monitorexit`, the instructions to acquire and release object locks, are used correctly. Correct use may simply mean that every lock acquired in a method is released prior to exit, or it may entail a stronger property, such as requiring that locks are acquired and released according to a specific locking policy. These checks may be added to our type system by first introducing a simple form of alias analysis for object references and then constructing the set of locks held at each line in the program. The alias information is needed to track multiple references to the same object within an activation record. In the time since we initially suggested this approach for checking monitors [16], Bigliardi and Laneve have developed an extension to our type system to model lock acquisition and release, as well as thread synchronization via `wait` and `notify` [3].

A second example is checking that class initializers are called at the appropriate time. According to the language specification, class initializers must be called only once and prior to any direct use of an object of that class. Determining where these calls should be made is left to the Java Virtual Machine implementor currently, and run-time tests are often employed to detect whether the necessary initializers have been called. By modifying the verifier slightly, however, we can specify and determine more precisely where initializers need to be called within a method. For example, the JVM could forego the class initialization test for any instruction reachable only by paths already including the appropriate test.

### 8.4. ELIMINATING UNNECESSARY RUN-TIME CHECKS

One final application of our type system is to extend the role of the verifier to identify locations where run-time checks may be eliminated. The type of analysis used to identify various unnecessary run-time checks may be phrased as a dataflow analysis problem, and given the style of our typing rules, it is fairly straightforward to embed dataflow problems into our system. We have extended our type system and prototype implementation to determine locations where the following run-time checks will always succeed:

- `null` pointer tests
- array bounds tests
- type checks for dynamic casts
- tests required because of the covariant subtyping of arrays

To perform this analysis, we incorporate additional type constructors to identify references known not to be `null` and dependent types to represent integer values falling within a specified range. Since array lengths are not known until run time, range types use alias information to refer to the lengths of arrays stored in local variables.

Figure 34 shows the type information computed for a program in this extended system. In that figure, the type `(Array+` $\tau$`)` is a subtype of `(Array` $\tau$`)` that is assigned to $\tau$-array references known not to be `null`. The type `(Range` $v_1$ $v_2$`)` is assigned to integer values that fall into the range $[v_1, v_2]$. The terms $v_1$ and $v_2$ may be numbers, expressions like $l(x)$ to indicate the length of the array stored in local variable $x$, or an arithmetic expression containing limited uses of addition or subtraction. The subscript $x$ on a stack element indicates that the stack slot will always contain the same value as local variable $x$. Given the type information, it is clear that the run-time tests for the `arraystore` operation will always succeed. To be useful in a general setting, it is necessary to expand the expression language for these dependent types to a more expressive fragment of arithmetic, possibly similar to what is used in the work of Xi and Pfenning [48].

This information may be applied in two ways. First, the verifier may pass it on to the interpreter or just-in-time compiler, which can then omit the unnecessary tests. Second, compilers can take advantage of the new verifier by performing optimizations with the knowledge that unneeded checks will not be performed. Eliminating

```
void f(int a[]) {
  int i;
  int n = a.length;
  for (i = 0; i < n; i++) {
    a[i] = 3;
  }
}
```

*Figure 33.* A simple method with array accesses that always succeed.

| $i$ | $P[i]$ | $S_i$ | $F_i[1]$ | $F_i[2]$ | $F_i[3]$ |
|---|---|---|---|---|---|
| 1 : | load 1 | $\epsilon$ | (Array int) | Top | Top |
| 2 : | arraylength | (Array int)$_1 \cdot \epsilon$ | (Array int) | Top | Top |
| 3 : | store 3 | (Range $l(1)$ $l(1)$) $\cdot \epsilon$ | (Array+ int) | Top | Top |
| 4 : | push 0 | $\epsilon$ | (Array+ int) | Top | (Range $l(1)$ $l(1)$) |
| 5 : | store 2 | (Range 0 0) $\cdot \epsilon$ | (Array+ int) | Top | (Range $l(1)$ $l(1)$) |
| 6 : | goto 15 | $\epsilon$ | (Array+ int) | (Range 0 0) | (Range $l(1)$ $l(1)$) |
| 7 : | load 1 | $\epsilon$ | (Array+ int) | (Range 0 $l(1)$-1) | (Range $l(1)$ $l(1)$) |
| 8 : | load 2 | (Array+ int)$_1 \cdot \epsilon$ | (Array+ int) | (Range 0 $l(1)$-1) | (Range $l(1)$ $l(1)$) |
| 9 : | push 3 | (Range 0 $l(1)$-1)$_2 \cdot$ (Array+ int)$_1 \cdot \epsilon$ | (Array+ int) | (Range 0 $l(1)$-1) | (Range $l(1)$ $l(1)$) |
| 10 : | arraystore int | (Range 3 3) $\cdot$ (Range 0 $l(1)$-1)$_2 \cdot$ (Array+ int)$_1 \cdot \epsilon$ | (Array+ int) | (Range 0 $l(1)$-1) | (Range $l(1)$ $l(1)$) |
| 11 : | load 2 | $\epsilon$ | (Array+ int) | (Range 0 $l(1)$-1) | (Range $l(1)$ $l(1)$) |
| 12 : | push 1 | (Range 0 $l(1)$-1)$_2 \cdot \epsilon$ | (Array+ int) | (Range 0 $l(1)$-1) | (Range $l(1)$ $l(1)$) |
| 13 : | add | (Range 1 1) $\cdot$ (Range 0 $l(1)$-1)$_2 \cdot \epsilon$ | (Array+ int) | (Range 0 $l(1)$-1) | (Range $l(1)$ $l(1)$) |
| 14 : | store 2 | (Range 1 $l(1)$) $\cdot \epsilon$ | (Array+ int) | (Range 0 $l(1)$-1) | (Range $l(1)$ $l(1)$) |
| 15 : | load 2 | $\epsilon$ | (Array+ int) | (Range 0 $l(1)$) | (Range $l(1)$ $l(1)$) |
| 16 : | load 3 | (Range 0 $l(1)$)$_2 \cdot \epsilon$ | (Array+ int) | (Range 0 $l(1)$) | (Range $l(1)$ $l(1)$) |
| 17 : | iflt 7 | (Range $l(1)$ $l(1)$)$_3$ $\cdot$ (Range 0 $l(1)$)$_2 \cdot \epsilon$ | (Array+ int) | (Range 0 $l(1)$) | (Range $l(1)$ $l(1)$) |
| 18 : | return | $\epsilon$ | (Array+ int) | (Range $l(1)$ $l(1)$) | (Range $l(1)$ $l(1)$) |

*Figure 34.* Extended type information for the method in Figure 33.

these checks can significantly improve performance in some situations. For example, execution speed of the method in Figure 33 running on the `srcjava` virtual machine [19] improves by approximately 20% when the array bounds check is removed. A more modest, but still nontrivial, performance increase can be obtained for larger programs.

One possible avenue for future work is to explore the relationship between static and dynamic checks further. For example, we may be able to improve resource management techniques [9] by incorporating some resource tracking into our static analysis. Bytecode analysis may also allow some security checks to be eliminated or moved to more optimal locations [47].

A current limitation of our system is that the bytecode verification process was designed to examine only one method at a time. Therefore, we have not included interprocedural analysis or global information in our framework. This limits the precision of our analysis and what additional properties we can check. Dynamic loading makes global analysis difficult because a newly loaded class may invalidate program invariants that previously held.

## 9. Related Work

Several projects have been established to develop a static type system for the Java programming language, the earliest of which include the work of Drossopoulou and Eisenbach [14], Syme [44], and Nipkow and von Oheimb [31]. Our definition of environments and the rules for describing well-formed environments are based on this body of work, but the overlap between Java and JVML does not extend much past the basic structure of declarations.

Our framework for type checking instructions is based on the type system originally developed by Stata and Abadi to study bytecode subroutines [43]. In our previous work, we first extended their system to study object initialization [18], and we made their semantics for subroutines more similar to the original Sun virtual machine specification in this paper [17]. This paper combines these previous projects to construct what we feel is a sufficiently large subset of JVML to cover all the interesting analysis problems. The rest of this section describes related studies of bytecode verification, focusing primarily on attempts to formalize Sun's original verification technique.

Hagiya and Tozawa present a type system for subroutines that has a similar technique for labeling and checking subroutines [21]. However, the types assigned to return addresses are dependent on the subroutine call stack height and must change upon entry or exit from a subroutine. In addition, special types are assigned to polymorphic variables instead of removing them from the local variable maps. Their paper also provides a brief overview of a dataflow algorithm to type check simple bytecode programs with subroutines. In contrast to the three steps in ours, their algorithm computes all information in one phase. Since they do not compute which variables are used in subroutines before starting the dataflow analysis, their

algorithm must guess whether each variable is used by a subroutine when it is first encountered. If an error occurs, the algorithm must back up and revise the guess before continuing. We are able to avoid backtracking because we precompute the variable usage information.

Qian [37] also presents an algorithm to verify subroutines based on a type system he developed for a subset of JVML similar to the subset studied here [36]. Our fixed-point computation in `Verify` follows the form of his algorithm, but Qian, like Hagiya and Tozawa, incorporates computation of control flow and local variable usage information into the iterative algorithm. As a result, it is more difficult to prove properties of the algorithm because special techniques are needed to show that the transfer function behaves monotonically. As demonstrated in Qian's paper, our system may overestimate the set of variables accessed in a subroutine because of the inability to identify dead code introduced by multilevel returns while constructing $G_P$ (when we have no type information). Although the cases in which this matters seem unlikely to become an issue, we could extend our system to handle them by either tracking return address types during the first phase of our checker or merging the second and third phase so that variable usage is discovered during dataflow analysis. We believe either approach would be straightforward to adopt if necessary.

A number of bytecode verification studies have departed more drastically from the original Sun specification. The Trusted Logic verifier for Java Card off-card verification, as described in [27], uses a polyvariant analysis in which subroutine bodies are checked multiple times, once for each calling context. A polyvariant analysis eliminates the need to construct the subroutine call graph before performing dataflow analysis but, since it analyzes each subroutine for each calling context, may be more computationally expensive. Leroy provides a thorough comparison of polyvariant analyses to other techniques in [28].

Coglio [5] uses a simple dataflow analysis algorithm for subroutines that does not merge the types of local variables on subroutine entry. Instead, multiple types are kept for each local variable while analyzing a subroutine. This allows a sufficient degree of context sensitivity to check several classes of programs not accepted by verifiers based on context-insensitive analysis of subroutines, such as the $\text{JVML}_f$ verifier presented in this paper.

The static semantics of Stärk, Schmid, and Börger [41, 42], which does not use subroutine call stacks, demonstrate the primary advantages of less restrictive approaches to subroutines. First, their more relaxed requirements on subroutines in the typing rules enable their system to accept programs with jumps that cause implicit returns from subroutines. In addition, the program invariants used to prove type soundness are simpler because they do not have to match the subroutine call stack implicit in the program execution history to a statically computed call stack.

Jones and Yelland independently developed ways of type checking bytecode programs using aspects of the Haskell type checker [49, 23], but these check-

ers are not easily realizable in a JVM implementation. A type system for Java bytecode subroutines based on the framework developed to study typed assembly language [30] has also been developed [32].

A number of studies have attempted to construct machine-verified proofs of correctness for bytecode verifier specifications and implementations. Pusch proves the soundness of a fragment of Qian's work automatically [35], and Bertot [2] validated the correctness of our soundness proofs for the fragment of $\text{JVML}_f$ concerned with object initialization [18]. Recently, Klein and Nipkow characterized dataflow-based type inference algorithms for low-level languages and proved in Isabelle/HOL that the standard iterative implementation of these algorithms yields a correct verifier [25]. They then constructed a Java bytecode verifier for a subset of $\text{JVML}_f$ as an instance of this general framework. They extended this approach to cover subroutines in [26]. Coglio et al. proposed building a complete JVML specification in the Specware system [7]. Specware could translate such a specification directly into an executable verifier. Pursuing an automatic translation of our typing rules into an executable verifier would be a useful extension to this work because it would reduce the possibility of introducing implementation errors.

Other work has focused more on developing verification techniques for bytecode programs under specific circumstances. For example, Rose and Rose discussed bytecode verification for Java Cards, which have limited resources available to the type checker [39]. By extending class files to contain stack and variable type information for branch targets, they were able to eliminate the need for the verifier to infer types. Instead, the verifier just checks the consistency of the provided type information. Posegga and Vogt also focused their attention on Java Cards, using model checking as the general framework for verification [34].

Precise specifications of the dynamic behavior of JVML programs have also been developed. Cohen's system is based on an ACL2 specification of the bytecode instruction set [8]. Bertelson presented a detailed dynamic semantics, although no formal properties of the system are shown [1]. Another dynamic semantics based on abstract state machines is presented in [4]. These systems, which cover various subsets of JVML, identify how programs should be compiled and executed, but they do not immediately produce a sound static semantics.

Stärk, Schmid, and Börger [41] have recently extended the work on abstract state machines for Java and the Java Virtual Machine to model not only the compilation process, but also bytecode verification for a subset of Java. This work demonstrates the potential of the abstract state machine methodology for proving the correctness of the whole compilation and verification process for nontrivial languages, such as Java and JVML.

## 10. Conclusions

Devious programs can circumvent the security checks built into run-time architectures like the Java Virtual Machine and Microsoft .NET platform [33] if they

can exploit run-time type errors. Thus, ensuring type-safe execution of low-level code is necessary for the security of such systems. In the case of the Java Virtual Machine, the component responsible for checking code, the bytecode verifier, was originally underspecified by the language designers and not fully understood by virtual machine implementors.

In this work, we have designed a static type system for a relatively complete subset of JVML that includes classes, interfaces, and methods, as well as a rich set of bytecode instructions that capture all difficult type checking problems. We have also shown that a bytecode verifier based on our type checker rejects all unsafe programs.

Our type system serves as the foundation for complete specification of the bytecode verifier. In addition, formalizing the JVML type system provides a setting in which other bytecode analysis problems can be explored, including both enforcement of stronger safety properties and identification of optimization opportunities. One clear avenue for future work is to develop a richer type system that enables interprocedural analysis techniques to analyze global properties of bytecode programs. The most significant challenge will be to model global analysis in the presence of dynamic loading effectively.

## References


1. Bertelsen, P.: Dynamic semantics of Java bytecode, in *Workshop on Principles of Abstract Machines*, 1998.
2. Bertot, Y.: Formalizing a JVML verifier for initialization in a theorem prover, in *CAV 01: Computer Aided Verification*, 2001, pp. 14–24.
3. Bigliardi, G. and Laneve, C.: A type system for JVM threads, in *Workshop on Types in Compilation*, 2000.
4. Börger, E. and Schulte, W.: Programmer friendly modular definition of the semantics of Java, in J. Alves-Foss (ed.), *Formal Syntax and Semantics of Java*, Lecture Notes in Comput. Sci. 1523, Springer-Verlag, 1999, pp. 353–404.
5. Coglio, A.: Simple verification technique for complex Java bytecode subroutines, in *Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs*, 2002.
6. Coglio, A. and Goldberg, A.: Type safety in the JVM: Some problems in the Java 2 SDK 1.2 and proposed solutions, *Concurrency and Computation: Practice and Experience* **13**(13) (2001), 1153–1171.
7. Coglio, A., Goldberg, A. and Qian, Z.: Toward a provably-correct implementation of the JVM bytecode verifier, in *Workshop on the Formal Underpinnings of the Java Paradigm*, 1998.
8. Cohen, R.: Defensive Java Virtual Machine version 0.5 alpha release, available from `http://www.cli.com/software/djvm/index.html`, 1997.
9. Czajkowski, G. and von Eicken, T.: JRes: A resource accounting interface for Java, in *Proceedings of the ACM Conference on Object Oriented Languages and Systems*, 1998, pp. 21–35.
10. Dean, D.: The security of static typing with dynamic linking, in *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, 1997, pp. 18–27.
11. Dean, D., Felten, E. W. and Wallach, D. S.: Java security: From HotJava to Netscape and beyond, in *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, 1996, pp. 190–200.

12. Dean, R. D.: Formal aspects of mobile code security, Ph.D. thesis, Princeton University, 1999.
13. Drossopoulou, S.: An abstract model of Java dynamic linking, loading and verification, in R. Harper (ed.), *Workshop on Types in Compilation*, Lecture Notes in Comput. Sci. 2071, 2001, pp. 53–84.
14. Drossopoulou, S. and Eisenbach, S.: Java is type safe – probably, in *European Conference On Object Oriented Programming*, 1997, pp. 389–418.
15. Freund, S. N.: Type systems for object-oriented intermediate languages, Ph.D. thesis, Stanford University, 2000.
16. Freund, S. N. and Mitchell, J. C.: A formal framework for the Java bytecode language and verifier, in *Proceedings of the ACM Conference on Object-Oriented Programming: Languages, Systems, and Applications*, 1999.
17. Freund, S. N. and Mitchell, J. C.: Specification and verification of Java bytecode subroutines and exceptions, Stanford Computer Science Technical Note STAN-CS-TN-99-91, 1999.
18. Freund, S. N. and Mitchell, J. C.: A type system for object initialization in the Java bytecode language, *ACM Transactions on Programming Languages and Systems* **21**(6) (1999), 1196–1250.
19. Ghemawat, S.: Srcjava implementation, 1999. Available from `http://www.research.digital.com/SRC/java`.
20. Goldberg, A.: A specification of Java loading and bytecode verification, in *ACM Conference on Computer and Communication Security*, 1998, pp. 49–58.
21. Hagiya, M. and Tozawa, A.: On a new method for dataflow analysis of Java virtual machine subroutines, in *Static Analysis Symposium*, 1998, pp. 17–32.
22. Jensen, T., Metayer, D. L. and Thorn, T.: Security and dynamic class loading in Java: A formalisation, in *Proceedings of the International Conference on Computer Languages*, 1998, pp. 4–15.
23. Jones, M.: The functions of Java bytecode, in *Workshop on the Formal Underpinnings of the Java Paradigm*, 1998.
24. Kildall, G. A.: A unified approach to global program optimization, in *Proceedings of ACM Symposium on Principles of Programming Languages*, 1973, pp. 194–206.
25. Klein, G. and Nipkow, T.: Verified bytecode verifiers, *Theoret. Comput. Sci.* (2002). To appear.
26. Klein, G. and Wildmoser, M.: Verified bytecode subroutines, *J. Automated Reasoning* (2003). To appear.
27. Leroy, X.: Java bytecode verification: An overview, in *CAV 01: Computer Aided Verification*, 2001, pp. 265–285.
28. Leroy, X.: Java bytecode verification: Algorithms and formalizations, *J. Automated Reasoning* (2003). To appear.
29. Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, 2nd edn, Addison-Wesley, 1999.
30. Morrisett, G., Crary, K., Glew, N. and Walker, D.: From system F to typed assembly language, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1998, pp. 85–97.
31. Nipkow, T. and von Oheimb, D.: Java$_{light}$ is type-safe – definitely, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1998, pp. 161–170.
32. O'Callahan, R.: A simple, comprehensive type system for Java bytecode subroutines, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999, pp. 70–78.
33. Platt, D.: *Introduducing Microsoft .NET*, Microsoft Press, 2001.
34. Posegga, J. and Vogt, H.: Byte code verification for Java smart cards based on model checking, in *5th European Symposium on Research in Computer Security (ESORICS)*, 1998, pp. 175–190.

35. Pusch, C.: Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL, in *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1999, pp. 89–103.
36. Qian, Z.: A formal specification of Java Virtual Machine instructions for objects, methods and subrountines, in J. Alves-Foss (ed.), *Formal Syntax and Semantics of Java*, Lecture Notes in Comput. Sci. 1523, Springer-Verlag, 1999, pp. 271–312.
37. Qian, Z.: Standard fixpoint iteration for Java bytecode verification, *ACM Transactions on Programming Languages and Systems* **22**(4) (2000), 638–672.
38. Qian, Z., Goldberg, A. and Coglio, A.: A formal specification of Java class loading, in *Proc. 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 325–336.
39. Rose, E. and Rose, K. H.: Toward a provably-correct implementation of the JVM bytecode verifier, in *Workshop on the Formal Underpinnings of the Java Paradigm*, 1998.
40. Sirer, E. G., McDirmid, S. and Bershad, B.: Kimera: A Java system architecture, 1997. Available from `http://kimera.cs.washington.edu`.
41. Stärk, R., Schmid, J. and Börger, E.: *Java and the Java Virtual Machine – Definition, Verification, Validation*, Springer-Verlag, 2001.
42. Stärk, R. F. and Schmid, J.: Completeness of a bytecode verifier and a certifying Java-to-JVM compiler, *J. Automated Reasoning* (2003). To appear.
43. Stata, R. and Abadi, M.: A type system for Java bytecode subroutines, *ACM Transactions on Programming Languages and Systems* **21**(1) (1999), 90–137.
44. Syme, D.: Proving Java type soundness, Technical Report 427, University of Cambridge, 1997.
45. Tarjan, R. E.: A unified approach to path problems, *J. ACM* **28** (1981), 577–593.
46. Tozawa, A. and Hagiya, M.: Careful analysis of type spoofing, in *Java-Informations-Tage*, 1999, pp. 290–296.
47. Wallach, D. S. and Felten, E. W.: Understanding Java stack inspection, in *Proceedings of IEEE Symposium on Security and Privacy*, 1998, pp. 52–63.
48. Xi, H. and Pfenning, F.: Dependent types in practical programming, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999, pp. 214–227.
49. Yelland, P.: A compositional account of the Java Virtual Machine, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999, pp. 57–69.