A Formal Framework for the Java Bytecode Language and Verifier

Stephen N. Freund* John C. Mitchell*
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
{freunds, mitchell}@cs.stanford.edu
Phone: (650) 723-8634, Fax: (650) 725-4671

Abstract

This paper presents a sound type system for a large subset of the Java bytecode language including classes, interfaces, constructors, methods, exceptions, and bytecode subroutines. This work serves as the foundation for developing a formal specification of the bytecode language and the Java Virtual Machine's bytecode verifier. We also describe a prototype implementation of a type checker for our system and discuss some of the other applications of this work. For example, we show how to extend our work to examine other program properties, such as the correct use of object locks.

1 Introduction

The bytecode language, which we refer to as JVML, is the platform independent representation of compiled Java programs. In order to prevent devious applets from causing security problems stemming from type errors, the Java Virtual Machine bytecode verifier performs a number of consistency checks on bytecode before it is executed [LY96]. This paper presents a type system that may serve as the foundation of a formal specification of the bytecode verifier for large fragment of JVML. Originally, the only specification was an informal English description that was incomplete or incorrect in some respects. Since then, a variety of more formal specifications for various subsets of JVML have been proposed. We discuss some of these in Section 9.

In our previous studies, we have examined several of the complex features of JVML in isolation. One study focused on object initialization and formalized the way in which a type system may prevent Java bytecode programs from using objects before they have been initialized [FM98]. In other work, we extended the work of Stata and Abadi on bytecode subroutines to develop a semantics for subroutines that is closer to the original Sun specification and includes exception handlers [FM99]. 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.

This paper builds on these studies to construct a formal model for a subset of JVML containing:

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

This subset captures the most difficult static analysis problems in bytecode verification. There are some features of JVML that we have not covered, including additional primitive types and control structures, final and static modifiers, access levels, concurrency, and packages. In some sense, these omitted features only contribute to the complexity of our system in the sheer number of cases that they introduce, but they do not appear to introduce any challenging or new problems. For example, the excluded primitive types and operations on them are all very 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

*Supported in part by NSF grants CCR-9303099 and CCR-9629754, and ONR MURI Award N00014-97-1-0505. Stephen Freund received additional support through a NSF Graduate Research Fellowship.

This paper appeared in OOPSLA '99.

use of final and access modifiers are well understood and straightforward to include.

We have applied our previous work on developing type checkers for the JVML type system to cover the subset presented in this paper as well [FM99] and have implemented a prototype verifier. This prototype demonstrates that our type system does reject faulty programs and also accepts virtually all programs generated by reasonable Java compilers.

Although the main contribution of this work is a framework in which bytecode verification can be formally specified, our type system is also useful for other purposes. We have extended our system to check additional safety properties, such as ensuring that object locks are acquired and released properly by each method. In addition, augmenting the verifier to track more information through the type system has allowed us to determine where run-time checks, such as `null` pointer and array bounds tests, may be eliminated.

This work may also lead to methods for verifying bytecode programs offline in situations where full bytecode verification cannot be done in the virtual machine due to resource constraints.

Section 2 introduces JVML$_f$, the fragment of JVML studied in this paper. Sections 3, 4 and 5 describes the formal dynamic and static semantics for JVML$_f$ and gives an overview of the soundness proof, and Section 6 highlights some of the technical details in handling object construction and subroutines. Section 7 gives a brief overview of our prototype verifier. Section 8 describes some applications of this work, Section 9 discusses some related work, and Section 10 concludes.

2 JVML$_f$

In this section, we informally introduce JVML$_f$, an idealized subset of JVML encompassing the features listed above. We shall use the Java program in Figure 1 as an example throughout the section. Compilation of this code produces a *class file* for each declared class or interface. In addition to containing the bytecode instructions for each method, a class file also contains the symbolic name and type information for any class, interface, method, or field mentioned in the source program. This information allows the Java Virtual Machine to verify and dynamically link code safely. To avoid some unnecessary details inherent in the class file format, we shall represent JVML$_f$ programs as a series of declarations somewhat similar to Java source code, as demonstrated in Figure 2.

The collection of declarations in Figure 2 contains the compiled Java language classes `Object` and `Throwable`. These are included in the JVML$_f$ program so that all referenced classes are present. As a convenience, 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 a list of exception handlers, and all methods in the superclass of a class are copied into the subclass unless they are overridden. Although real class files do not duplicate the code in this manner, it simplifies several aspects of our system by essentially flattening the class hierarchy when it comes to method lookup. The special name `<init>` is given to constructors.

The execution environment for JVML$_f$ programs consists of a stack of activation records and an object store. 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 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 σ and τ are a class name and array component type, respectively. We refer the reader to the Java Virtual Machine specification for a detailed discussion of these bytecode instructions [LY96].

The bytecode language uses *descriptors* to refer to method and field names from Java language programs. A descriptor contains three pieces of information about the method or field that it describes:

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

For example, the bytecode instruction used to set the `num` instance field in the constructor for `A` is `putfield` $\{\!|\texttt{A}, \texttt{num}, \texttt{int}|\!\}_{\texttt{F}}$. Descriptors are used in place of simple names to provide enough information to:

1. check uses of methods and fields without loading the class to which they belong.
2. dynamically link class files safely.

```
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.

```
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:  push 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 12
       7:  pop
       8:  load 0
       9:  push 2
      10:  putfield ⦃A, num, int⦄_F
      11:  goto 12
      12:  push 6
      13:  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.

$$
\begin{array}{rl}
\mathit{instruction} ::= & \texttt{push}\ v \mid \texttt{pop} \mid \texttt{store}\ x \mid \texttt{load}\ x \\
& \mid \texttt{add} \mid \texttt{ifeq}\ L \mid \texttt{goto}\ L \\
& \mid \texttt{new}\ \sigma \\
& \mid \texttt{invokevirtual}\ \mathit{MDescriptor} \\
& \mid \texttt{invokeinterface}\ \mathit{IDescriptor} \\
& \mid \texttt{invokespecial}\ \mathit{MDescriptor} \\
& \mid \texttt{getfield}\ \mathit{FDescriptor} \\
& \mid \texttt{putfield}\ \mathit{FDescriptor} \\
& \mid \texttt{newarray (Array}\ \tau\texttt{)} \mid \texttt{arraylength} \mid \texttt{arrayload} \mid \texttt{arraystore} \\
& \mid \texttt{throw}\ \sigma \mid \texttt{jsr}\ L \mid \texttt{ret}\ x \\
& \mid \texttt{return} \mid \texttt{returnvalue}
\end{array}
$$

Figure 3: The JVML$_f$ instruction set.

3. provide unique symbolic names to overloaded methods and fields. Overloading is resolved at compile time in Java.

Valid method, interface, and field descriptors are generated by the following grammar:

$$
\begin{array}{rcl}
\mathit{MDescriptor} & ::= & \{\!|\mathit{Class\text{-}Name},\ \mathit{Label},\ \mathit{Method\text{-}Type}|\!\}_{\mathtt{M}} \\
\mathit{IDescriptor} & ::= & \{\!|\mathit{Interface\text{-}Name},\ \mathit{Label},\ \mathit{Method\text{-}Type}|\!\}_{\mathtt{I}} \\
\mathit{FDescriptor} & ::= & \{\!|\mathit{Class\text{-}Name},\ \mathit{Label},\ \mathit{Field\text{-}Type}|\!\}_{\mathtt{F}}
\end{array}
$$

A *Field-Type* may be either `int`, `float`, any class or interface name, or an array type. A *Method-Type* is a type $\alpha \rightarrow \gamma$ were α is a possibly empty sequence of *Field-Type*'s and γ is the return type of the function (or `void`). Figure 4 shows the exact representation of all of these types, as well as 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 cannot be mentioned in a JVML$_f$ program. Note that we distinguish between methods declared in a class and methods declared in an interface using different types of descriptors.

Some JVML$_f$ instructions generate exceptions when the arguments are not valid. For example, if `null` is used as an argument to any instruction that performs an operation on an object, a run-time exception is generated. The value of an exception is an object whose class is `Throwable` or a subclass of it. To avoid introducing additional classes, we assume that all failed run-time checks generate `Throwable` objects. When an exception is generated, the list of handlers associated with the currently executing method is searched for an appropriate handler. 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 an appropriate handler is found, execution jumps to the first instruction of the exception handler's code. Otherwise, the top activation record is popped, and this process is repeated 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 an environment, Section 3.2 introduces a few notational conventions, and Section 3.3 describes the semantics of the bytecode instructions.

3.1 Environments

A JVML$_f$ program is represented formally by an environment Γ containing all the information about classes, interfaces, and methods found in the class files of the program. The environment, as defined in Figure 5, is broken into three components storing each of these items. Construction of Γ from the representation of class file information demonstrated in Figure 2 is straightforward.

We write $\Gamma \vdash \tau_1 <: \tau_2$ to indicate that τ_1 is a subtype of τ_2, given Γ. The Java Virtual Machine model uses this judgment as a way to perform run-time type tests. The subtyping rules are presented in the Appendix, and they follow the form of the rules used to model subtyping in the Java language [DE97, Sym97], with extensions to cover the JVML$_f$ specific types. This judgment, and all others presented in this paper, are summarized in Figure 6.

3.2 Notational Conventions

Before proceeding, we summarize a few notational conventions used throughout this paper. To access information about a method M declared in Γ, we write $\Gamma[M]$. Similar notation is used to access interface and class declarations. If $\Gamma[M] = \langle P, H\rangle$ for some method descriptor M, then $\mathit{Dom}(P)$ is the set of addresses from the set $\textsc{Addr}$ used in P, and $P[i]$ is the i^{th} instruction in P. $\mathit{Dom}(P)$ will always include address 1 and is usually a range $\{1, \ldots, n\}$ for some n. Likewise, H is a partial map from integer indexes to handlers, where

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

Figure 4: JVML$_f$ types.

$$\begin{array}{rrcl}
\Gamma^C : & \textit{Class-Name} & \to & \left\langle \begin{array}{l} \texttt{super} : \textit{Class-Name} \cup \{\textit{None}\}, \\ \texttt{interfaces} : \text{set of } \textit{Interface-Name}, \\ \texttt{fields} : \text{set of } \textit{FDescriptor} \end{array} \right\rangle \\
 & & & \\
\Gamma^M : & \textit{MDescriptor} & \to & \left\langle \begin{array}{l} \texttt{code} : \textit{instruction}^+, \\ \texttt{handlers} : \textit{handler}^* \end{array} \right\rangle \\
 & & & \\
\Gamma^I : & \textit{Interface-Name} & \to & \left\langle \begin{array}{l} \texttt{interfaces} : \text{set of } \textit{Interface-Name}, \\ \texttt{methods} : \text{set of } \textit{IDescriptor} \end{array} \right\rangle
\end{array}$$

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

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

$\Gamma \vdash \tau_1 <: \tau_2$	τ_1 is a subtype of τ_2.
$\Gamma \vdash M[pc] = I$	I is the instruction at address pc in the code array of M.
$\Gamma \vdash C_0 \to C_1$	A program in configuration C_0 evaluates to C_1 in a single step.
$\Gamma \vdash \texttt{wf}$	Γ is well-formed.
$\Gamma \vdash d\ \texttt{ty}$	The definition of d only refers to names of classes/interfaces defined in Γ.
$\Gamma \vdash d :: K$	The declaration of d is a valid K, $K \in \{\texttt{class}, \texttt{interface}, \texttt{method}\}$.
$\Gamma, F, S \vdash \langle P, H \rangle : M$	The method body $\langle P, H \rangle$ conforms to descriptor M given Γ, F, and S.
$\Gamma, F, S, i \vdash P : M$	Instruction i of P is well typed given Γ, F, and S.
$\Gamma, F, S \vdash h$ handles P	Handler h is well-typed given Γ, F, and S.
$\Gamma \vdash h\ \texttt{wt}$	h is a well-typed heap.
$\Gamma, h \vdash v : \tau$	The value v has type τ given environment Γ and heap h.
$\Gamma \vdash C\ \texttt{wt}$	The configuration C is well typed in environment Γ.

Figure 6: Summary of the judgements.

a handler has the form $\langle b,\, e,\, t,\, \sigma\rangle$ for addresses b, e, t, and class σ.

Update and substitution operations are defined as follows:

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

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

where $y \in \mathit{Dom}(f)$, and a, b, and v range over the codomain of f.

Sequences will also be used. The empty sequence is ϵ, and $v \cdot s$ represents placing v on the front of sequence s. Appending one sequence to another is written as $s_1 \bullet s_2$, and substitution is defined on sequences of in the same manner as substitution on partial maps.

3.3 Operational Semantics

We describe the JVML$_f$ virtual machine in the standard framework of operational semantics. A machine state is a configuration $C = A; h$, where A is a stack of activation records and h is a memory store. An activation record stack is a sequence of activation records of the form $\langle M,\, pc,\, f,\, s\rangle$, where each part has the following meaning:

M: the method descriptor 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.

Exceptions introduce a special form of activation record, $\langle e\rangle_{\mathrm{exc}}$, where e is a reference to the object that was raised.

Objects of class σ are represented as records of the form

$$\langle\!\langle \langle\sigma_1,\, l_1\rangle = v_1, \ldots, \langle\sigma_n,\, l_n\rangle = v_n \rangle\!\rangle_\sigma$$

An object's fields are indexed by pairs $\langle\sigma_i,\, l_i\rangle$ containing a class σ_i and a name l_i to correspond to how fields are named in field descriptors. To simplify this representation, we abbreviate this type of record as $\langle\!\langle \langle\sigma_i,\, l_i\rangle = v_i \rangle\!\rangle_\sigma^{i \in I}$ where the subscript $i \in I$ refers to the i^{th} field in the record. Arrays are stored in a similar type of record:

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

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

$$\begin{array}{lr} h : \textsc{Loc} \rightarrow & \\ \quad \langle\!\langle \langle\sigma_i,\, l_i\rangle = v_i \rangle\!\rangle_\sigma^{i \in I} & \text{(object)} \\ \quad \mid \langle\!\langle \langle\sigma_i,\, l_i\rangle = v_i \rangle\!\rangle_{\varphi \,\diamond\, \texttt{(Uninit } \sigma\ j\texttt{)}}^{i \in I} & \text{(uninit. object)} \\ \quad \mid [[v_i]]^{i \in [0..n-1]}_{\texttt{(Array } \tau\texttt{)}} & \text{(array)} \end{array}$$

The middle record form is used to represent uninitialized objects and is described in Section 6.

All records have tags to support run-time tests and a form of dynamic dispatch. The function *Tag* returns the tag of a reference or any other run-time value:

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

The rest of the section presents representative rules from the operational semantics. The full list of rules will be presented in an extended version of this paper.

The first rule in Figure 7 shows how to execute the `getfield` instruction, which pops an object reference off the stack and pushes the value stored in the specified field of that object.

That rule indicates that the program represented by Γ may move from configuration C_0 to C_1 in a single step if the configurations match the patterns listed in the table and if all other conditions in the first parts of a box are satisfied. The judgment $\Gamma \vdash M[pc] = \texttt{getfield}\ \{\!|\varphi,\, l,\, \kappa|\!\}_{\texttt{F}}$ means that the instruction at index pc in the code array belonging to M is $\texttt{getfield}\ \{\!|\varphi,\, l,\, \kappa|\!\}_{\texttt{F}}$. The inference rule to derive this judgment is

$$\frac{\Gamma[M] = \langle P,\, H\rangle \qquad P[pc] = I}{\Gamma \vdash M[pc] = I}$$

Two other simple cases for loading and storing values in the local variables are also shown in Figure 7.

The rule for `arraystore` requires runtime tests to verify that the array reference is a 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 due to a potential type loophole caused by the covariant subtyping of arrays. This is similar to the problem of covariant method specialization discussed in [Coo89].

The rule for `invokevirtual` demonstrates how the run-time tag of the receiver object is used to construct the method descriptor of the new activation record when a method is called. The notation $f[0 \mapsto b, 1..|\alpha| \mapsto s_1]$ is an abbreviation for storing the receiver object in local variable 0, and the arguments in a sequence of local variables starting at 1. The function f_0 maps the local variables to any arbitrary values.

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \mathtt{getfield}\ \{\!\vert\varphi,\ l,\ \kappa\vert\!\}_{\mathtt{F}}$		
Cond	$b \in \mathit{Dom}(h)$	*b is a valid reference*
C_0	$\langle M,\ pc,\ f,\ b \cdot s\rangle \cdot A; h$	*the initial configuration*
C_1	$\langle M,\ pc+1,\ f,\ h[b].\langle\varphi,\ l\rangle \cdot s\rangle \cdot A; h$	*the final configuration*

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \mathtt{load}\ x$	
C_0	$\langle M,\ pc,\ f,\ s\rangle \cdot A; h$
C_1	$\langle M,\ pc+1,\ f,\ f[x] \cdot s\rangle \cdot A; h$

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \mathtt{store}\ x$	
C_0	$\langle M,\ pc,\ f,\ v \cdot s\rangle \cdot A; h$
C_1	$\langle M,\ pc+1,\ f[x \mapsto v],\ v \cdot s\rangle \cdot A; h$

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \mathtt{arraystore}$		
Cond	$\mathit{Tag}(h, v') = \tau'$ $h[b] = [[v_0, \ldots, v_{n-1}]]_{(\mathtt{Array}\ \tau)}$ $\Gamma \vdash \tau' <: \tau$ $0 \leq k < n$	*find type of v'* *b is a valid reference* *type test for covariance* *index is in bounds*
C_0	$\langle M,\ pc,\ f,\ v' \cdot k \cdot b \cdot s\rangle \cdot A; h$	
C_1	$\langle M,\ pc+1,\ f,\ s\rangle \cdot A;$ $h[b \mapsto [[v_0, \ldots, v_{k-1}, v',$ $v_{k+1}, \ldots, v_{n-1}]]_{(\mathtt{Array}\ \tau)}]$	

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \mathtt{invokevirtual}\ \{\!\vert\varphi,\ m,\ \alpha \to \gamma\vert\!\}_{\mathtt{M}}$	
Cond	$\mathit{Tag}(h, b) = \sigma$ $\lvert\alpha\rvert = \lvert s_1\rvert$ $N = \{\!\vert\sigma,\ m,\ \alpha \to \gamma\vert\!\}_{\mathtt{M}}$
C_0	$\langle M,\ pc,\ f,\ s_1 \bullet (b \cdot s)\rangle \cdot A; h$
C_1	$\langle N,\ 1,\ f_0[0 \mapsto b, 1..\lvert\alpha\rvert \mapsto s_1],\ \epsilon\rangle \cdot$ $\langle M,\ pc,\ f,\ s_1 \bullet (b \cdot s)\rangle \cdot A; h$

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \mathtt{returnval}$	
Cond	$M = \{\!\vert\sigma,\ m,\ \alpha \to \gamma\vert\!\}_{\mathtt{M}}$ $\lvert\alpha\rvert = \lvert s_1\rvert$ $\gamma \neq \mathtt{void}$
C_0	$\langle M,\ pc,\ f,\ v \cdot s\rangle \cdot$ $\langle M',\ pc',\ f',\ s_1 \bullet (b \cdot s')\rangle \cdot A; h$
C_1	$\langle M',\ pc'+1,\ f',\ v \cdot s'\rangle \cdot A; h$

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \mathtt{throw}\ \sigma$	
Cond	$b \in \mathit{Dom}(h)$
C_0	$\langle M,\ pc,\ f,\ b \cdot s\rangle \cdot A; h$
C_1	$\langle b\rangle_{\text{exc}} \cdot \langle M,\ pc,\ f,\ b \cdot s\rangle \cdot A; h$

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \mathtt{arraystore}$	
Cond	$b \notin \mathit{Dom}(h)$ $e \notin \mathit{Dom}(h)$
C_0	$\langle M,\ pc,\ f,\ v' \cdot k \cdot b \cdot s\rangle \cdot A; h$
C_1	$\langle e\rangle_{\text{exc}} \cdot \langle M,\ pc,\ f,\ v' \cdot k \cdot b \cdot s\rangle \cdot A; h[e \mapsto$ $\mathit{Blank}_{\mathtt{Throwable}}]$

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \mathtt{arraystore}$	
Cond	$h[b] = [[v_0, \ldots, v_{n-1}]]_{(\mathtt{Array}\ \tau)}$ $\mathit{Tag}(h, v') = \tau'$ $\Gamma \not\vdash \tau' <: \tau$ $e \notin \mathit{Dom}(h)$
C_0	$\langle M,\ pc,\ f,\ v' \cdot k \cdot b \cdot s\rangle \cdot A; h$
C_1	$\langle e\rangle_{\text{exc}} \cdot \langle M,\ pc,\ f,\ v' \cdot k \cdot b \cdot s\rangle \cdot A; h[e \mapsto$ $\mathit{Blank}_{\mathtt{Throwable}}]$

$\Gamma \vdash C_0 \to C_1$

Cond	$\mathit{Tag}(h, b) = \sigma$ $\mathit{CorrectHandler}(\Gamma, M, pc, \sigma) = 0$
C_0	$\langle b\rangle_{\text{exc}} \cdot \langle M,\ pc,\ f,\ s\rangle \cdot A; h$
C_1	$\langle b\rangle_{\text{exc}} \cdot A; h$

$\Gamma \vdash C_0 \to C_1$

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

$\Gamma \vdash C_0 \to C_1$

C_0	$\langle b\rangle_{\text{exc}} \cdot \epsilon; h$
C_1	$\epsilon; h$

Figure 7: JVML$_f$ dynamic semantics (excerpts).

There are two ways in which an exception can be generated. Either a `throw` statement is executed or a runtime check fails. In either situation, the resulting configuration contains a special record on the top of the stack indicating the exception value, as demonstrated by the rule for `throw` and two of the three rules for the failure of an `arraystore` instruction. In those rules, the record $\mathit{Blank}_{\texttt{Throwable}}$ is an object record for class `Throwable` in which all fields are set to their initial values (either 0 or `null`). Introduction of this record essentially creates a new object without calling its constructor, but this deviation from the real Java Virtual Machine does not significantly affect the overall behavior of our execution model, nor does it affect the static semantics in anyway.

The last three rules in Figure 7 show 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. The *CorrectHandler* judgment is defined in [FM99]. We may conclude that $\mathit{CorrectHandler}(\Gamma, M, pc, \sigma) = t$ only if $\langle b, e, t, \sigma' \rangle$ is the first handler found in the list of handlers for M such that $b \leq pc < e$ and $\Gamma \vdash \sigma <: \sigma'$. If no such handler is found, $\mathit{CorrectHandler}(\Gamma, M, pc, \sigma) = 0$.

4 Static Semantics

This section gives a brief overview of the static semantics for JVML_f. We first define well-formed environments and then describe the typing rules for the JVML_f instruction set and exceptions. For simplicity, we omit subroutines and constructors from this section, but return to them in Section 6.

4.1 Well-formed Environments

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

- there are no circularities in the class hierarchy.
- all classes actually implement their declared interfaces by defining all methods listed in them.
- a class inherits all field and interface declarations from its superclass, and inherits or overrides all methods.

Basically, all class properties described in [LY96] that do not depend on the bodies of methods fall into this list.

Figure 8 contains the judgments used to show that a JVML_f environment Γ is well formed.

The (*wf env*) rule requires that `Object` and `Throwable` be defined in Γ. The auxiliary judgment $\Gamma \vdash d\ \texttt{ty}$ is derivable only if the definition of d refers only to classes or interfaces defined in Γ. This judgment does not look inside the bytecode arrays. The local typing judgments from the next section inspect individual instructions. Finally, each declaration d in Γ must be well formed, written $\Gamma \vdash d :: K$, where K is the kind of declaration. The rules for classes and methods, (*wt class*) and (*wt meth*), also appear in Figure 8.

In (*wt meth*), F is a map from $\textsc{Addr}$ to functions 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 $\textsc{Addr}$ 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 we can derive $\Gamma, F, S \vdash \langle P, H \rangle : M$ is analogous to the verifier accepting the code of method M, and it means that if the method M is called with arguments of the correct type, no type error will occur as the result of executing the code for the method. Figure 9 shows the typing information for a method, and the rest of the section develops this judgement.

These rules for environments, partially based on [Sym97, DE97], do not show how to incrementally build a well-formed environment. However, assuming that the environment is fully built prior to verification is adequate for describing how to type check bytecode methods. In reality, construction of Γ would be complicated by other Java-specific features like dynamic loading, which we have not addressed to date.

4.2 Methods

The judgment in Figure 10 concludes that a method is valid. In that rule, $F_{\textsc{Top}}$ is a function mapping all variables in $\textsc{Var}$ to `Top`. The initial conditions on F_1 match the types of the values values stored in f during the creation of new activation record for a call to this method. The fourth line requires that each instruction in the program being well typed according to the local judgments described in the next section, and the last line requires that each handler be well typed. There may be more than one F and S for which the above rule is applicable but, as a convenience, we shall assume that there is some unique canonical F and S for each method.

4.2.1 Instructions

The local typing rules are presented in Figure 11. These typing rules describe a set of constraints between the types of variables and stack slots at different locations in the program. The basic intuition behind these rules is that the type information flows along execu-

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

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

$$(\mathit{wt\ meth})\qquad \frac{\begin{array}{c}\Gamma[M]=\langle P,\,H\rangle\\ \Gamma,F,S\vdash\langle P,\,H\rangle : M\end{array}}{\Gamma\vdash M :: \texttt{method}}$$

Figure 8: Rules for well-formed environments (excerpts).

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

`Exception table:`

from	to	target	type
1	6	7	Throwable

Figure 9: The type information for method $\{\!|\texttt{A},\,\texttt{foo},\,\texttt{int}\to\texttt{int}|\!\}_{\texttt{M}}$ in Figure 2.

$$(\mathit{meth\ code})\qquad \frac{\begin{array}{cl} m\neq\texttt{<init>} & \textit{not a constructor}\\ F_1=F_{\mathrm{TOP}}[0\mapsto\sigma,1..|\alpha|\mapsto\alpha] & \textit{types of initial values in local vars}\\ S_1=\epsilon & \textit{initial stack}\\ \forall i\in\mathit{Dom}(P).\ \Gamma,F,S,i\vdash P : \{\!|\sigma,\,m,\,\alpha\to\gamma|\!\}_{\texttt{M}} & \textit{each instruction is well typed}\\ \forall i\in\mathit{Dom}(H).\ \Gamma,F,S\vdash H[i]\ \text{handles}\ P & \textit{each exception handler is well typed}\end{array}}{\Gamma,F,S\vdash\langle P,\,H\rangle : \{\!|\sigma,\,m,\,\alpha\to\gamma|\!\}_{\texttt{M}}}$$

Figure 10: Rule to type the body of a method.

$$(\textit{get field})\quad \begin{array}{|c|l} \multicolumn{1}{l}{\Gamma, F, S, i \vdash P : M} & \\ \hline P[i] = \mathtt{getfield}\ \{|\varphi,\ l,\ \kappa|\}_{\mathtt{F}} & \\ \hline \Gamma \vdash S_i <: \varphi \cdot \beta & \textit{top of stack contains class } \varphi \textit{ or subclass} \\ \{|\varphi,\ l,\ \kappa|\}_{\mathtt{F}} \in \Gamma[\varphi].\mathtt{fields} & \textit{field is present in declaration of } \varphi \\ \hline \Gamma \vdash \kappa \cdot \beta <: S_{i+1} & \textit{new stack has type of field on top} \\ \Gamma \vdash F_i <: F_{i+1} & \textit{variables are not updated} \\ i+1 \in \mathit{Dom}(P) & \textit{not stepping out of code array} \\ \hline \end{array}$$

$$(\textit{load})\quad \begin{array}{|c|} \multicolumn{1}{l}{\Gamma, F, S, i \vdash P : M} \\ \hline P[i] = \mathtt{load}\ x \\ \hline x \in \mathit{Dom}(F_i) \\ \hline \Gamma \vdash F_i[x] \cdot S_i <: S_{i+1} \\ \Gamma \vdash F_i <: F_{i+1} \\ i+1 \in \mathit{Dom}(P) \\ \hline \end{array}$$

$$(\textit{store})\quad \begin{array}{|c|} \multicolumn{1}{l}{\Gamma, F, S, i \vdash P : M} \\ \hline P[i] = \mathtt{store}\ x \\ \hline x \in \mathit{Dom}(F_i) \\ S_i = \tau \cdot \beta \\ \hline \Gamma \vdash \beta <: S_{i+1} \\ \Gamma \vdash F_i[x \mapsto \tau] <: F_{i+1} \\ i+1 \in \mathit{Dom}(P) \\ \hline \end{array}$$

$$(\textit{get field})\quad \begin{array}{|c|} \multicolumn{1}{l}{\Gamma, F, S, i \vdash P : M} \\ \hline P[i] = \mathtt{getfield}\ \{|\varphi,\ l,\ \kappa|\}_{\mathtt{F}} \\ \hline \Gamma \vdash S_i <: \varphi \cdot \beta \\ \{|\varphi,\ l,\ \kappa|\}_{\mathtt{F}} \in \Gamma[\varphi].\mathtt{fields} \\ \hline \Gamma \vdash \kappa \cdot \beta <: S_{i+1} \\ \Gamma \vdash F_i <: F_{i+1} \\ i+1 \in \mathit{Dom}(P) \\ \hline \end{array}$$

$$(\textit{array store})\quad \begin{array}{|c|} \multicolumn{1}{l}{\Gamma, F, S, i \vdash P : M} \\ \hline P[i] = \mathtt{arraystore} \\ \hline \Gamma \vdash S_i <: \tau \cdot \mathtt{int} \cdot (\mathtt{Array}\ \tau) \cdot \beta \\ \hline \Gamma \vdash \beta <: S_{i+1} \\ \Gamma \vdash F_i <: F_{i+1} \\ i+1 \in \mathit{Dom}(P) \\ \hline \end{array}$$

$$(\textit{inv virt})\quad \begin{array}{|c|} \multicolumn{1}{l}{\Gamma, F, S, i \vdash P : M} \\ \hline P[i] = \mathtt{invokevirtual}\ N \\ \hline N = \{|\varphi,\ m,\ \alpha \to \gamma|\}_{\mathtt{M}} \\ m \neq \mathtt{<init>} \\ \Gamma \vdash S_i <: \alpha \bullet (\varphi \cdot \beta) \\ N \in \mathit{Dom}(\Gamma) \\ \hline \gamma \neq \mathtt{void} \ \Rightarrow\ \Gamma \vdash \gamma \cdot \beta <: S_{i+1} \\ \gamma = \mathtt{void} \ \Rightarrow\ \Gamma \vdash \beta <: S_{i+1} \\ \Gamma \vdash F_i <: F_{i+1} \\ i+1 \in \mathit{Dom}(P) \\ \hline \end{array}$$

$$(\textit{return val})\quad \begin{array}{|c|} \multicolumn{1}{l}{\Gamma, F, S, i \vdash P : M} \\ \hline P[i] = \mathtt{returnval} \\ \hline M = \{|\sigma,\ m,\ \alpha \to \gamma|\}_{\mathtt{M}} \\ \gamma \neq \mathtt{void} \\ \Gamma \vdash S_i <: \gamma \cdot \beta \\ \hline \end{array}$$

$$(\textit{throw})\quad \begin{array}{|c|} \multicolumn{1}{l}{\Gamma, F, S, i \vdash P : M} \\ \hline P[i] = \mathtt{athrow}\ \sigma \\ \hline \Gamma \vdash \sigma <: \mathtt{Throwable} \\ \Gamma \vdash S_i <: \sigma \cdot \beta \\ \hline \end{array}$$

Figure 11: JVML$_f$ static semantics for instructions (excerpts).

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

4.2.2 Exception Handlers

An exception handler $\langle b,\ e,\ t,\ \sigma\rangle$ is well typed if the following conditions hold:

- $[b, e)$ is a valid interval in $\mathit{Dom}(P)$, and $t \in \mathit{Dom}(P)$.
- σ is a subclass of `Throwable`.
- S_t is a valid type for a stack containing only a single reference to a σ object.
- 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.

These checks ensure that when execution transfers to t from any i in $[b, e)$, S_t and F_t are valid types for the local variables and new stack. These requirements are summarized by the following exception rule:

$$(\textit{wt handler})\quad \frac{\begin{array}{c} \Gamma \vdash \sigma <: \mathtt{Throwable} \\ 1 \leq b < e \\ b, e-1, t \in \mathit{Dom}(P) \\ \forall i \in [b, e).\ \Gamma \vdash F_i <: F_t \\ \Gamma \vdash \sigma \cdot \epsilon <: S_t \end{array}}{\Gamma, F, S \vdash \langle b,\ e,\ t,\ \sigma\rangle \text{ handles } P}$$

We revisit this rule when we discuss subroutines in Section 6. Subroutines significantly complicate matters because the types of local variables become dependent upon execution history.

$$(\mathit{int})\quad \frac{n \in \text{integers}}{\Gamma, h \vdash n : \texttt{int}} \qquad (\mathit{obj})\quad \frac{\begin{array}{c} h[a] = \langle\!\langle \langle \sigma_i,\, l_i \rangle = v_i \rangle\!\rangle_{\sigma}^{i \in I} \\ \Gamma[\sigma].\texttt{fields} = \{\{\!| \sigma_i,\, l_i,\, \kappa_i |\!\}_{\texttt{F}}\}^{i \in I} \\ \forall i \in I.\ \Gamma \vdash \mathit{Tag}(h, v_i) <: \kappa_i \end{array}}{\Gamma, h \vdash a : \sigma} \qquad (\mathit{subsumption})\quad \frac{\begin{array}{c} \Gamma, h \vdash v : \tau_1 \\ \Gamma \vdash \tau_1 <: \tau_2 \end{array}}{\Gamma, h \vdash v : \tau_2}$$

Figure 12: Rules assigning types to values (excerpts).

5 Soundness

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 slightly differently. 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 run until the activation record stack is empty.

Theorem 1 (JVML$_f$ Soundness) *If* $\Gamma \vdash \texttt{wf}$*,* $M = \{\!|\sigma,\, m,\, \epsilon \rightarrow \texttt{void}|\!\}_{\texttt{M}}$*,* $M \in \Gamma$*,* $\Gamma[\sigma].\texttt{fields} = \emptyset$*,* $a \in \textsc{Loc}$*, and* $\mathit{Dom}(h_0) = \emptyset$*, then:*

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

In our machine model, programs that attempt to perform an operation leading to a type error get stuck because those operations are not defined in our operational semantics. By proving that well-typed programs always run until the last method exits, we know that well-typed programs will not attempt to perform any illegal operations.

The proof consists of two parts. The first shows that any single step from a well-formed configuration leads to another well-formed configuration. The second part shows that a transition can always be made from a well-formed configuration unless the activation record stack is empty. The rest of the section develops the notion of well-formed configurations for environment Γ.

5.1 Heaps

A heap h is well typed if all references in $\mathit{Dom}(h)$ may be assigned the type with which their records are tagged. Values and, in particular, references are assigned types with respect to both an environment and a heap. Figure 12 contains typing judgments for integers and objects. For objects, the value stored in each field must have a tag that is a subtype of the field's type. The following judgment concludes that a heap is well typed:

$$(\mathit{wt\ heap})\quad \frac{\forall a \in \mathit{Dom}(h).\ \Gamma, h \vdash a : \mathit{Tag}(h, a)}{\Gamma \vdash h\ \texttt{wt}}$$

These rules allow us to type heaps containing circular references by using only the tag information about field values in the hypotheses of the typing rules.

5.2 Configurations

We now describe the requirements for a configuration $A; h$ to be well formed with respect to Γ, written $\Gamma \vdash A; h\ \texttt{wt}$. Clearly, it must be the case that $\Gamma \vdash h\ \texttt{wt}$. Let us assume that A is not empty, meaning that $A = \langle M,\, pc,\, f,\, s \rangle \cdot A'$. If $\Gamma \vdash \texttt{wf}$, then it must be the case that $\Gamma[M] = \langle P,\, H \rangle$ and $\Gamma, F, S \vdash \langle P,\, H \rangle : M$. The following must be true for $A; h$ to be well formed:

- the program counter is within the bounds of the code array: $pc \in \mathit{Dom}(P)$.
- the values on the stack have the statically determined type: $\Gamma, h \vdash s : S_{pc}$.
- the local variables have the correct types: $\forall y \in \textsc{Var}.\ \Gamma, h \vdash f[y] : F_{pc}[y]$.
- either:
  - execution in the top activation record in A' is at a method invocation that resolves to a call to M, and A' is well formed, or
  - A' is empty and M is an acceptable method to start execution.

The inductive definition of these invariants ensures that they are true of each activation record on the stack. Other necessary invariants concerning subroutines and object initialization are described below.

6 Technical Details

This section briefly describes how we check subroutines and object initialization. This work builds on our previous studies [FM98, FM99], and we refer the reader to those sources for some of the details omitted from this discussion.

6.1 Object Initialization

The static semantics must guarantee that no well-typed program uses an object before it has been initialized. Objects are allocated with the `new` instruction

$$(\mathit{new}) \qquad \begin{array}{l} \Gamma, F, S, i \vdash P : M \\ \begin{array}{|c|l} \hline P[i] = \texttt{new}\ \sigma & \\ \hline \sigma \in \mathit{Dom}(\Gamma) & \\ (\texttt{Uninit}\ \sigma\ i) \notin S_i & \textit{no old values with type } (\texttt{Uninit}\ \sigma\ i) \\ \forall y \in \mathit{Dom}(F_i).\ F_i[y] \neq (\texttt{Uninit}\ \sigma\ i) & \\ \hline \Gamma \vdash (\texttt{Uninit}\ \sigma\ i) \cdot S_i <: S_{i+1} & \textit{push new reference onto stack} \\ \Gamma \vdash F_i <: F_{i+1} & \\ i+1 \in \mathit{Dom}(P) & \\ \hline \end{array} \end{array}$$

$$(\mathit{inv\ spec}) \qquad \begin{array}{l} \Gamma, F, S, i \vdash P : M \\ \begin{array}{|c|l} \hline P[i] = \texttt{invokespecial}\ N & \\ \hline N = \{\!|\varphi,\ \texttt{<init>},\ \alpha \to \texttt{void}|\!\}_{\texttt{M}} & \textit{N is defined} \\ N \in \mathit{Dom}(\Gamma) & \textit{receiver has type } (\texttt{Uninit}\ \varphi\ j) \\ \Gamma \vdash S_i <: \alpha \bullet ((\texttt{Uninit}\ \varphi\ j) \cdot \beta) & \textit{0 is special case (see below)} \\ j \neq 0 & \\ \hline \Gamma \vdash [\varphi/(\texttt{Uninit}\ \varphi\ j)]\beta <: S_{i+1} & \textit{replace with initialized object type} \\ \Gamma \vdash [\varphi/(\texttt{Uninit}\ \varphi\ j)]F_i <: F_{i+1} & \\ i+1 \in \mathit{Dom}(P) & \\ \hline \end{array} \end{array}$$

Figure 13: Static semantics for object creation and initialization instructions.

and are initialized by invoking a constructor with the `invokespecial` instruction. Since the reference to the uninitialized object 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 references to uninitialized objects.

In the static semantics, the type of a variable or stack slot containing an uninitialized object reference is a compound type expression that contains the line on which the object was allocated. For example, the type $(\texttt{Uninit}\ \sigma\ \mathit{pc})$ represents an uninitialized object of type σ 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 only if the same uninitialized object type is never assigned to two different objects during program execution. The rules for `new` and `invokespecial` are presented in Figure 13.

This analysis is correct only if constructors do in fact properly initialize objects. According to the Java language specification, an object is only initialized after a constructor for each ancestor in the class hierarchy is invoked, in order from the object's class up to `Object`. To enforce this at the bytecode level, we require that every constructor 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 the constructor exits. For simplicity, we may refer to either of these actions as invoking the superclass constructor. The only deviation from this requirement is for constructors of class `Object`. Since, by the Java language definition, `Object` is the only class without a superclass, constructors for `Object` need not call any other constructor.

Our treatment of constructors again uses uninitialized object types. For brevity, we do not present the technical details, but, in summary, we add the following pieces presented in [FM98] to the system:

- a special type of the form $(\texttt{Uninit}\ \sigma\ 0)$, which is assigned to the uninitialized object passed into a constructor, and a special form of (*inv spec*) to check the usage of this type.
- a set of rules to examine the structure of the constructor to determine whether all paths through the code call an appropriate superclass constructor.

One tricky part of developing the JVML$_f$ semantics is constructing the abstract machine in such a way that programs which do not properly initialize objects get stuck. In other words, the run-time representation of uninitialized or partially initialized objects must be distinguishable from initialized objects. If this is not the case, we cannot use our soundness theorem to conclude that programs always initialize objects correctly. The tag for an uninitialized object record, such as $\sigma \diamond (\texttt{Uninit}\ \varphi\ j)$, contains two pieces of information. The first part, σ, indicates that the object was originally allocated by a `new` σ instruction. The second half matches the type assigned to references to the object in the static type information for the code associated with the top activation record on the stack. When a constructor is called on an object, the tag is changed to match the new constructor's view. For example, if a constructor for φ' is called on the object from above, the tag would be changed to $\sigma \diamond (\texttt{Uninit}\ \varphi'\ 0)$. When the `Object` constructor is called, the tag is changed a σ, and the object is fully initialized. The operational semantics rules are then written to allow constructors to be called only on objects with appropriate tags.

To simplify our soundness proofs, we actually model this slightly differently. In order to avoid changing tags of existing objects, which complicates proving invariants about heaps, we create new objects. Specifically, calling the constructor actually creates a new object with the 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. This is similar to the substitution in the JVML$_i$ semantics from [FM98], and the invariants showing that the analysis works correctly are similar to those in that paper. 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. The extensions to the JVML$_i$ operational semantics to prove invariants about constructor execution are also incorporated into this work.

6.2 Subroutines

Bytecode subroutines are a form of local call and return that provide space efficient compilation of the `try-finally` statements in Java programs. Without subroutines, the code for the `finally` block would have to be duplicated at every exit from the `try` block. We refer the reader to our previous work for a full description of subroutines. Subroutines are easy to model in the dynamic semantics, and the rules for them appear in Figure 14. However, it is tricky to type check methods that use subroutines. It is necessary to check that:

1. subroutine calls and returns occur in a stack-like manner. In some cases, however, a return instruction may cause a jump the order of the subroutine calls stack than its immediate caller.

2. a specific form of local variable polymorphism introduced by subroutines is 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 involves the use of 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 which have been called in a method, which have not yet returned. We can capture the set of all possible call stacks for line i, G_i, by examining the structure of the program. The rules for doing this are presented in [FM99]. For this presentation, it is sufficient to assume that any valid method has an acyclic subroutine call graph and that execution cannot reach the beginning of a subroutine without calling `jsr`.

The typing rules for subroutine call and return appear in Figure 15. These rules combine checks from our previous study of object initialization with our work on subroutines.

In those rules, the domains of the local variable maps are restricted inside subroutines. The types of local variables over which the current instruction is polymorphic depends on execution history. To recover these types, and auxiliary function $\mathcal{F}$ is used: $\mathcal{F}(F, pc, \rho)[y] = \tau$ only if local variable y can be assigned type τ at line pc of the program given subroutine called stack ρ implicit in the execution history. In addition, $\mathcal{H}(P, \tau, \rho) = \texttt{Top}$ if τ is a return address type inconsistent with all return address is in ρ. In all other cases, it equals τ.

Figure 16 shows a sample JVML$_f$ program that uses subroutines and the type information for it.

Soundness of these rules depends on invariants showing that the implicit subroutine call stack is always contained in the sets of statically computed all stacks, and that a return address type `(Ret-from` L`)` is only assigned to addresses which are both valid return addresses for L and present in the implicit call stack. In addition, we prohibit uninitialized objects from being present in variables over which the currently executing subroutine is polymorphic to prevent error that was found in the Sun verifier [FM98]. The typing rule for exception handlers must be rewritten to be similar to rule (*ret*) since the process of catching an exception may affect the implicit subroutine call stack. One or more subroutines may be popped off the call stack because the handler for the exception may appear outside of the currently executing subroutine.

7 Implementation

We have implemented a prototype of the JVML$_f$ verifier. The algorithm uses the three phase type checker developed in [FM99]. The basic idea is to use a data flow analysis algorithm to compute a valid typing for each method in a class file, after performing several passes through the code to analyze subroutines.

Our verifier translates JVML methods into a subset of JVML$_f$. This translation preserves the structure and data flow of the original program, but utilizes a small core verifier to perform the type synthesis. In this way, as many details as possible are removed from the data flow analysis algorithm. For example, the instruction

$$\texttt{invokevirtual}\ \{\!|\texttt{Vector}, \texttt{indexOf}, \texttt{Object} \rightarrow \texttt{int}|\!\}_{\texttt{M}}$$

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \texttt{jsr}\ L$	
C_0	$\langle M,\ pc,\ f,\ s\rangle \cdot A; h$
C_1	$\langle M,\ L,\ f,\ (pc+1) \cdot s\rangle \cdot A; h$

$\Gamma \vdash C_0 \to C_1$

$\Gamma \vdash M[pc] = \texttt{ret}\ x$	
C_0	$\langle M,\ pc,\ f,\ s\rangle \cdot A; h$
C_1	$\langle M,\ f[x],\ f,\ s\rangle \cdot A; h$

Figure 14: Operational semantics for subroutines instructions.

$\Gamma, F, S, i \vdash P : M$

	$P[i] = \texttt{jsr}\ L$	
(jsr)	$\forall y \in Dom(F_i). F_i[y] \neq (\texttt{Ret-from}\ L)$ $(\texttt{Ret-from}\ L) \not\in S_i$ $\forall y \in Dom(F_i).\ F_i[y] \not\in Uninit$ $\forall y \in Dom(S_i).\ S_i[y] \not\in Uninit$	*no recursive subroutine calls* *no uninit. objects in variables* *or on stack*
	$Dom(F_L) \subseteq Dom(F_i)$ $\Gamma \vdash (\texttt{Ret-from}\ L) \cdot S_i <: S_L$ $\forall y \in Dom(F_L).\ \Gamma \vdash F_i[y] <: F_L[y]$ $L \in Dom(P)$	*accesses smaller set of vars* *push return address*
	$Dom(F_i) = Dom(F_{i+1})$ $\forall y \in Dom(F_i) \setminus Dom(F_L).$ $\Gamma \vdash F_i[y] <: F_{i+1}[y]$ $i+1 \in Dom(P)$	*subroutine returns to $i+1$* *variables over which subroutine is* *polymorphic*

$\Gamma, F, S, i \vdash P : M$

	$P[i] = \texttt{ret}\ x$	
(ret)	$x \in Dom(F_i)$ $F_i[x] = (\texttt{Ret-from}\ L)$ $\forall y \in Dom(F_i).\ F_i[y] \not\in Uninit$ $\forall i \in Dom(S_i).\ S_i[y] \not\in Uninit$	*returning from L* *not passing back uninitialized objects*
	$\forall \rho \in G_{P,i}.\ \exists p, \rho_1, \rho_2.$ $\rho = \rho_1 \bullet (p \cdot \rho_2)$ $\wedge\ P[p-1] = \texttt{jsr}\ L$ $\wedge\ \Gamma \vdash \mathcal{H}(P, S_i, \rho_2) <: S_p$ $\wedge\ \forall y \in Dom(F_p).\ \forall \tau.$ $\mathcal{F}(F, i, \rho)[y] = \tau$ $\Rightarrow \Gamma \vdash \mathcal{H}(P, \tau, \rho_2) <: F_p[y]$	*find return address for L* *in each possible call stack* *hide invalid return types* *in stack and variables* *at return location p*

Figure 15: Static semantics for subroutines instructions.

i	$P[i]$	G_i	$F_i[1]$	$F_i[2]$	$F_i[3]$	S_i
1 :	`jsr 3`	ϵ	`Top`	`Top`	`Top`	ϵ
2 :	`halt`	ϵ	`Top`	`Top`	`Top`	ϵ
3 :	`store 1`	$2 \cdot \epsilon$	`Top`	`Top`	`Top`	`(Ret-from 3)` $\cdot\ \epsilon$
4 :	`jsr 7`	$2 \cdot \epsilon$	`(Ret-from 3)`	`Top`	`Top`	ϵ
5 :	`jsr 10`	$2 \cdot \epsilon$	`(Ret-from 3)`	`Top`	`Top`	ϵ
6 :	`ret 1`	$2 \cdot \epsilon$	`(Ret-from 3)`	`Top`	`Top`	ϵ
7 :	`store 2`	$5 \cdot 2 \cdot \epsilon$		`Top`	`Top`	`(Ret-from 7)` $\cdot\ \epsilon$
8 :	`jsr 10`	$5 \cdot 2 \cdot \epsilon$		`(Ret-from 7)`	`Top`	ϵ
9 :	`ret 2`	$5 \cdot 2 \cdot \epsilon$		`(Ret-from 7)`	`Top`	ϵ
10 :	`store 3`	$6 \cdot 2 \cdot \epsilon,\ 9 \cdot 5 \cdot 2 \cdot \epsilon$			`Top`	`(Ret-from 10)` $\cdot\ \epsilon$
11 :	`ret 3`	$6 \cdot 2 \cdot \epsilon,\ 9 \cdot 5 \cdot 2 \cdot \epsilon$			`(Ret-from 10)`	ϵ

Figure 16: The type information computed for a program using subroutines.

is replaced by:

```
pop<Object>  ; pop argument
pop<Vector>  ; pop reciever
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 which may be checked separately, either before or after the type checking has been performed. The one constraint generated by the above instruction is:

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

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 [Gol98].

We have used this translator to verify 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, but the cases where our algorithm does differ have not been found in practice. Typically, the differences are only found in programs which use subroutines and exception handlers in nonstandard ways. Our previous work describes the situations in detail.

The idea of translating a bytecode program into a "micro"-instruction set has been explored by others [Yel99] and seems to be a promising way to keep the size and complexity of the core verifier small.

8 Applications

A formal framework for JVML has a number of different applications including aspects both within and outside of the current role of the Java Virtual Machine in program execution. This section provides a brief overview of some of these applications.

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. Given the previous problems found in the existing informal specification and reference implementations [DFWB97, FM98], there is clearly the need for a more precise description of it.

One of the major design goals for our system was to remain as consistent with the original Sun specification and reference implementation as possible so that our work could directly influence specification of the bytecode language. Although there are some minor deviations, mostly due to adding more structure to the subroutine mechanism [FM99], we have remained close enough to be confident that a useful formal specification of bytecode verification can be derived from this work.

Our work on constructing type checking algorithms from our specification may also prove useful in better describing how to implement a verifier based on our type system. In addition, by specifying verification separately from the linking process, we may be able to apply this work to construct verification techniques for situations in which resource constraints on either memory or computation power limit the ability to perform verification at runtime. This will become an issue as small embedded devices like Java cards and Java rings [Jav99] become more prevalent.

8.2 Testing Existing Implementations

Another avenue in which to apply this work is to follow the direction of the Kimera project [SMB97]. 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. We are currently developing an architecture to perform similar tests with our prototype implementation.

Even without automated testing, this work has 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 [FM98]. Another example is that the set of operations allowed on null references differs between some of the 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 are also investigating ways to 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 two-phase locking policy. These checks are added to our type system by first introducing a very simple form of alias analysis for object references and then constructing the set of locks held at each line the program. The alias information is used to track multiple references to the same object within an activation record. Recent work on type systems to detect race and deadlock conditions [FA99] may allow us to perform stronger checks in this area.

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 or not the necessary initializers have been called. However, by modifying the verifier slightly, we can specify and determine more precisely where initializers need to be called.

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 data flow analysis problem, and given the style of our typing rules, it is fairly straightforward to embed data flow problems into our system. We have extended our type system and prototype implementation to determine locations where the following runtime checks will always succeed:

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

To perform this analysis, we incorporate additional type constructors that identify references known not to be `null` and dependent types to represent integer values known to be within a certain 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. The details of these checks are omitted for space considerations, but Figure 18 shows the type information computed for a program in this extended system.

In Figure 18, the type `(Array+` τ`)` is a subtype of `(Array` τ`)` that is assigned to τ-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 [XP99].

There are two ways to apply this information. First, the verifier may pass it on to the interpreter or just-in-time compiler, which may 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 these checks can significantly improve performance in some situations. For example, execution speed of the method in Figure 17 running on the `srcjava` virtual machine [Ghe99] improves by approximately 20% when the array bounds check is removed. We are currently evaluating the impact of these techniques for larger programs.

One possibility 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 [CvE98] 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 [WF98].

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. Developing ways to verify and compute global properties incrementally as classes are loaded is left for future work.

9 Related Work

There have been several projects to develop a static type system for the Java programming language [DE97, Sym97, NvO98]. It is not surprising that the Java language and JVML are similar in some respects, and our definition of environments and the rules for describing well-formed environments are based on this body of work. However, since the Java language statements are completely different than the JVML instruction set, the overlap between the language does not extend much

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

Figure 17: A simple method.

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

Figure 18: Extended type information for the method in Figure 17.

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 [SA99]. In our previous work, we first extended their system to study object initialization [FM98], and we also studied ways to make the semantics of subroutines closer to the original virtual machine specification [FM99]. 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 some of the many other endeavors to specify the verifier.

Qian's system [Qia98, Qia99] is the closest in extent and coverage to our system. The main difference that distinguishes our work is our attempt to remain as close to the original specification as possible, particularly in the treatment of subroutines. Our system also has a more precise notion of a typing environment and what it means for an environment to be well formed. We have also attempted to structure our system as the composition of smaller, well understood systems, which we believe has led to a better understanding of how the different language features interact with each other. Pusch has attempted to prove the soundness of a fragment of his work automatically [Pus99]. Some of the technical differences between our work and Qian's are described in [FM99], where we also compare the type checking algorithms developed for both type systems.

Coglio et al. are currently building a complete JVML specification in the SpecWare system, although subroutines and exceptions are currently not modeled in their system [CGQ98]. Pursuing an automatic translation of our typing rules into an executable verifier would be a useful extension to this work since it reduces the possibility of introducing implementation errors. This work is partially based on Goldberg's earlier work which describes a verification framework that also handles dynamic loading [Gol98]. We believe that his approach may be adapted to our formal system, and our prototype implementation generates typing constraints for environments similar to those described in his paper.

Other work has focused more on developing verification techniques for bytecode programs under specific circumstances. For example, Rose and Rose discuss bytecode verification for Java cards, which have limited resources available to the type checker [RR98]. In earlier work, E. Rose described the static semantics for a small subset of JVML, and a compilation technique from a subset of Java to that bytecode language [Ros98].

Posegga and Vogt also focused their attention on Java cards, using model checking as the general framework for verification [PV98].

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 [Coh97]. Bertelson presents a detailed dynamic semantics, although no formal properties of the system are shown [Ber98]. Another dynamic semantics based on evolving algebras is presented in [BS98]. These systems, which have varying degrees of coverage of JVML features, are useful at identifying how programs should be compiled and executed, but they do not immediately produce a sound static semantics.

One final category of projects regarding JVML are those which have departed from the original Sun specification. O'Callahan, for example, presents a type system for Java bytecode subroutines based on the framework developed to study typed assembly language [O'C99]. Jones and Yelland independently developed ways of type checking bytecode programs using the Haskell type checker [Yel99, Jon98]. One potential area for future work is to combine some of the ideas from these studies and the type systems for typed assembly languages [TMC^{+}96, MCGW98] into our work as a way of better specifying and tying together the different phases of compilation and verification in the Java framework.

10 Conclusions

In our previous work, we studied type systems for object initialization [FM98] and subroutines [FM99] in isolation. The languages used in these studies consisted of a very small set of operations on a single activation record, and there was no notion of classes or methods. After studying the properties of these features, we have been able to compose them into a formal semantics 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. It appears to be relatively straightforward to extend the techniques presented here to cover the full bytecode language.

Developing a type system for the Java Virtual Machine bytecode language has several important uses. Clearly, the need to provide strong safety properties for mobile code execution necessitates the need for formal specification of the bytecode verifier. In addition, formalizing the JVML type system may provide an avenue through which we may examine how to apply recent results on proof carrying code and typed assembly languages to the Java compilation and execution process [MCGW98, Nec97].

We have also presented ways to extend the verifier to provide stronger checks and information to aid in optimizing program execution. The next 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.

A Subtyping Rules

This appendix introduces the subtyping judgments for our system, which are based on [Sym97]. We first present the subclass and subinterface relationships induced by the environment Γ:

$$(<:_C\ \textit{refl})\qquad \frac{\sigma \in \textit{Class-Name}}{\Gamma \vdash \sigma <:_C \sigma}$$

$$(<:_C\ \textit{super})\qquad \frac{\begin{array}{c}\Gamma \vdash \sigma_1 <:_C \sigma_2\\ \Gamma[\sigma_2].\texttt{super} = \sigma_3\end{array}}{\Gamma \vdash \sigma_1 <:_C \sigma_3}$$

$$(<:_I\ \textit{refl})\qquad \frac{\omega \in \textit{interface-Name}}{\Gamma \vdash \omega <:_I \omega}$$

$$(<:_I\ \textit{super})\qquad \frac{\begin{array}{c}\Gamma \vdash \omega_1 <:_I \omega_2\\ \omega_2 \in \Gamma[\omega_3].\texttt{interfaces}\end{array}}{\Gamma \vdash \omega_1 <:_I \omega_3}$$

The following rules determine the subtyping relationships for array components:

$$(<:_A\ \textit{prim})\qquad \frac{\tau \in \textit{Prim}}{\Gamma \vdash \tau <:_A \tau}$$

$$(<:_A\ \textit{class})\qquad \frac{\Gamma \vdash \sigma_1 <:_C \sigma_2}{\Gamma \vdash \sigma_1 <:_A \sigma_2}$$

$$(<:_A\ \textit{interface})\qquad \frac{\Gamma \vdash \omega_1 <:_I \omega_2}{\Gamma \vdash \omega_1 <:_A \omega_2}$$

These three relations are combined to create the subtyping rules for all reference types. Special rules are required to handle `Null` and `Object`:

$$(<:_R\ \textit{class})\qquad \frac{\Gamma \vdash \sigma_1 <:_C \sigma_2}{\Gamma \vdash \sigma_1 <:_R \sigma_2}$$

$$(<:_R\ \textit{interface})\qquad \frac{\Gamma \vdash \omega_1 <:_I \omega_2}{\Gamma \vdash \omega_1 <:_R \omega_2}$$

$$(<:_R\ \textit{class int})\qquad \frac{\begin{array}{c}\Gamma \vdash \sigma_1 <:_C \sigma_2\\ \omega_1 \in \Gamma[\sigma_2].\texttt{interfaces}\\ \Gamma \vdash \omega_1 <:_I \omega_2\end{array}}{\Gamma \vdash \sigma_1 <:_R \omega_2}$$

$$(<:_R\ \textit{array}\ \texttt{Obj})\qquad \frac{(\texttt{Array}\ \tau) \in \textit{Array}}{\Gamma \vdash (\texttt{Array}\ \tau) <:_R \texttt{Object}}$$

$$(<:_R\ \textit{array})\qquad \frac{\begin{array}{c}\Gamma \vdash \tau_1 <:_A \tau_2\\ n > 0\end{array}}{\Gamma \vdash (\texttt{Array}^n\ \tau_1) <:_R (\texttt{Array}^n\ \tau_2)}$$

$$(<:_R\ \textit{int}\ \texttt{Obj})\qquad \frac{\omega \in \textit{Interface-Name}}{\Gamma \vdash \omega <:_R \texttt{Object}}$$

$$(<:_R\ \texttt{Null})\qquad \frac{\tau \in \textit{Simple-Ref} \cup \textit{Array} \cup \{\texttt{Null}\}}{\Gamma \vdash \texttt{Null} <:_R \tau}$$

Using these initial judgments, subtyping is defined with the following rules, which also extend the definition of subtypes to include sequences and partial maps in the obvious way:

(<: *ref*) $$\frac{\Gamma \vdash \tau_1 <:_R \tau_2}{\Gamma \vdash \tau_1 <: \tau_2}$$

(<: *refl*) $$\frac{\tau \in \mathit{Uninit} \cup \mathit{Prim} \cup \mathit{Ret}}{\Gamma \vdash \tau <: \tau}$$

(<: `Top`) $$\frac{\tau \in \mathit{Ref} \cup \mathit{Prim} \cup \mathit{Ret} \cup \{\texttt{Top}\}}{\Gamma \vdash \tau <: \texttt{Top}}$$

(<: ϵ) $$\frac{}{\Gamma \vdash \epsilon <: \epsilon}$$

(<: *seq*) $$\frac{\begin{array}{c}\Gamma \vdash \alpha_1 <: \alpha_2 \\ \Gamma \vdash \tau_1 <: \tau_2\end{array}}{\Gamma \vdash \tau_1 \cdot \alpha_1 <: \tau_2 \cdot \alpha_2}$$

$$\begin{array}{rcl}\Gamma \vdash F_1 <: F_2 & \overset{\text{def}}{=} & \mathit{Dom}(F_1) = \mathit{Dom}(F_2) \\ & & \wedge\ \forall y \in \mathit{Dom}(F_1).\ \Gamma \vdash F_1[y] <: F_2[y]\end{array}$$

References

[Ber98] Peter Bertelsen. Dynamic semantics of Java bytecode. In *Workshop on Principles of Abstract Machines*, September 1998.

[BS98] E. Boerger and W. Schulte. Programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer Verlag LNCS 1523, 1998.

[CGQ98] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *Workshop on the Formal Underpinnings of the Java Paradigm*, October 1998.

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

[Coo89] W.R. Cook. A proposal for making Eiffel typesafe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. Jres: A resource accounting interface for Java. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, October 1998.

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

[DFWB97] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java security: Web browers and beyond. In Dorothy E. Denning and Peter J. Denning, editors, *Internet Beseiged: Countering Cyberspace Scofflaws*. ACM Press, New York, New York, October 1997.

[FA99] Cormac Flanagan and Martín Abadi. Types for safe locking. In *Proceedings of European Symposium on Programming*, March 1999.

[FM98] Stephen Freund and John Mitchell. A type system for object initialization in the Java bytecode language. In *Proc. ACM Conference on Object-Oriented Programming: Languages, Systems, and Applications*, October 1998. An extended version appears as Stanford University Technical Note STAN-CS-98-62, April 1998.

[FM99] Stephen Freund and John Mitchell. Specification and verification of Java bytecode subroutines and exceptions, August 1999. To appear as a Stanford University Technical Note. Currently available from `http://cs.stanford.edu/~freunds`.

[Ghe99] Sanjay Ghemawat. srcjava. Available from http://www.research.digital.com/SRC/java, August 1999.

[Gol98] Allen Goldberg. A specification of Java loading and bytecode verification. In *ACM Conference on Computer and Communication Security*, 1998.

[Jav99] Java-Powered Ring. Available from http://www.ibutton.com, March 1999.

[Jon98] Mark Jones. The functions of Java bytecode. In *Workshop on the Formal Underpinnings of the Java Paradigm*, October 1998.

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

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

[Nec97] George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.

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

[O'C99] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, January 1999.

[Pus99] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *TACAS*, 1999.

[PV98] Joachim Posegga and Harald Vogt. Byte code verification for Java smart cards based on model checking. In *5th European Symposium on Research in Computer Security (ESORICS)*, Louvain-la-Neuve, Belgium, 1998. Springer LNCS.

[Qia98] Zhenyu Qian. A Formal Specification of Java(tm) Virtual Machine Instructions. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer Verlag LNCS 1523, 1998.

[Qia99] Zhenyu Qian. Least types for memory locations in (Java) bytecode. In *Sixth Workshop on Foundations of Object-Oriented Languages*, January 1999.

[Ros98] Eva Rose. Towards secure bytecode verification on a Java card. Master's thesis, University of Copenhagen, 1998.

[RR98] Eva Rose and Kristoffer Høgsbro Rose. Toward a provably-correct implementation of the JVM bytecode verifier. In *Workshop on the Formal Underpinnings of the Java Paradigm*, October 1998.

[SA99] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. *Transaction on Programming Languages and Systems*, 1999. To appear. Currently available as Digital Equipment Corporation Systems Research Center Research Report 158, June 1998. An earlier version appeared in Proc. 25th ACM Symposium on Principles of Programming Languages, January 1998.

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

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

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

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of IEEE Symposium on Security and Privacy*, May 1998.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, January 1999.

[Yel99] Phillip Yelland. A compositional account of the Java Virtual Machine. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, January 1999.