Specification and Verification of Java Bytecode Subroutines and Exceptions (summary)

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

July 26, 2001

Abstract

This paper develops a formal description of verifying Java bytecode subroutines and exceptions. We first present a type system for subroutines which extends the work of Stata and Abadi to encompass multilevel returns [SA98]. Second, we describe a data flow analysis algorithm proven to accept exactly the set of well-typed programs. Finally, we extend our type system and algorithm to include exception handlers and exceptions. We also give a brief overview of an implementation of our type checker.

1 Introduction

Execution of mobile Java code can potentially lead to significant security problems if the code is not carefully monitored to prevent it from accessing sensitive data, deleting files, or corrupting the runtime system in other ways. The Java Virtual Machine has security mechanisms in place to prevent these kinds of attacks [Gon98], but they can be circumvented by a devious applet if it can cause certain run-time type errors, such as casting an integer to a pointer, as demonstrated in [DFW96, Sar97].

The Java Virtual Machine *bytecode verifier* prevents these types of errors by performing a number of consistency checks on compiled programs before they are executed. Compiled programs are written in what we shall call JVML, a platform-independent bytecode language with some low-level instructions to reflect high-level features and constructs of the Java language, such as exception handlers, classes, etc. The verifier checks for these programs include determining that:

- every instruction opcode is valid.
- all jumps are to valid instruction addresses.

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

```
void f() {
  try {
    something();
  } catch(Exception e) {
    oops();
  } finally {
    done();
  }
}
```

Figure 1: A method using a `try-finally` statement.

- operations are not performed on values of incorrect type.

This work studies the most complex parts of JVML and develops a foundation for formal specification of the verifier in terms of a type system and type checking algorithm.

We first present a new type system for JVML_0, a small fragment of JVML encompassing some simple operations and subroutines [SA98]. Bytecode subroutines are a form of local call and return that provide space efficient compilation of the `try-finally` statements in Java programs, as demonstrated in Figures 1 and 2. Without subroutines, the code for the `finally` block would have to be duplicated at every exit from the `try` block. However, verification of programs using subroutines is complicated by the need to check that

1. subroutine calls and returns occur in a stack-like manner, with the exception of multilevel returns, which are discussed below.

2. a specific form of local variable polymorphism introduced by subroutines is used correctly. In particular, local variables not touched by a subroutine may contain values of conflicting types at different calls to the subroutine.

We use some of the ideas from the type system of Hagiya and Tozawa [HT98] to eliminate one of the major simplifications to the subroutine mechanism made by Stata and Abadi and to develop a type system based on their work that is closer to the original Sun specification. In comparison to Stata and Abadi, our system is able to type check multilevel returns, which were not allowed in [SA98].

A multilevel return occurs when a subroutine returns not to its caller, but to its caller's caller, or even further up the stack of subroutines which have been called but which have not exited. As described in Section 9, our JVML_0 type system differs technically from Hagiya and Tozawa and the wide variety of other type systems for JVML_0 (see, for example, [HT98, O'C99, Qia98]), and our system offers some advantages over the others. Most importantly, we feel that ours is the closest to the Sun specification and existing implementations.

We also present a verification algorithm proven to accept exactly the set of well-typed programs. The algorithm uses standard data flow analysis techniques to synthesize type information for a program. However, subroutines again complicate matters. Our verifier computes flow control and variable usage information about subroutines before beginning data flow analysis. Precomputation of this information simplifies both the implementation and correctness proofs for the data flow analysis aspect of the type checker in contrast to some of the other suggested type checking algorithms [HT98, Qia99] that compute all information at once.

```
 Method void f()
 // code for try block
 0 aload_0                                 // load this
 1 invokevirtual <Method void something()>  // call something
 4 jsr 23                                  // call subroutine
 7 return

 // code for exception handler
 8 pop                                     // ignore exception value
 9 aload_0                                 // load this
10 invokevirtual <Method void oops()>       // call oops
13 jsr 23                                  // call subroutine
16 return

// compiler-inserted handler
17 astore_1                                // store exception value to rethrow
18 jsr 23                                  // call subroutine
21 aload_1                                 // load exception
22 athrow                                  // rethrow

// subroutine
23 astore_2                                // store return address
24 aload_0                                 // load this
25 invokevirtual <Method void done()>       // call done()
28 ret 2                                   // return from subroutine

Exception table:
   from   to  target type
     0     4     8   <Class java.lang.Exception>
     0    13    17   any
```

Figure 2: The translation of the program in Figure 1 into JVML. Control is transfered to the subroutine beginning on line 23 at the end of the `try` block, at the end of the exception handler, and when an uncaught exception (handled at line 17) causes an abrupt exit from the method.

The second half of this paper presents a type system and verifier for JVML_e, an extension of JVML_0 containing exceptions and exception handlers. The machinery developed to check multilevel returns may also be applied to type check exception handlers in the presence of subroutines. A jump in execution caused by throwing and catching an exception may, in some cases, behave very similar to a multilevel return. This is the first treatment of exception handlers and subroutines in a framework derived from the work of Stata and Abadi and the first presentation of a provably correct type checking algorithm similar to the original Sun specification.

We also summarize our experience with prototype implementations of the type checking algorithms.

Our type system and verification algorithm for JVML_e serves as the core of a specification and implementation of a bytecode verifier for all of JVML. We have extended this work to include our previous work on object initialization [FM98], and we have in fact been able to construct a sound type system for a much larger fragment of JVML that includes classes, interfaces, methods, constructors, subroutines, and exceptions based on these smaller systems.

Sections 2, 3, and 4 present JVML_0, its formal semantics, and a type checker for it. We extend our type system and verifier to include exception handlers in Sections 5 and 6. Section 7 describes an implementation of our algorithms, and Section 8 compares our verifier to the Sun implementation. Section 9 relates our contributions to other work on the bytecode verifier, and Section 10 draws some conclusions about this project.

2 JVML_0

A JVML_0 program is an array P of instructions drawn from the following list, where the basic execution model consists of a program counter, a set of local variables, and an operand stack:

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

where x is a local variable name and $L \in \textsc{Addr}$, the set of instruction addresses. Informally, `push 0`, `inc`, `pop`, and `if` L perform the obvious operations on the stack; `store` x and `load` x move values between the static and local variables; `jsr` L jumps to instruction L, pushing the return address onto the stack; and `ret` x jumps to the instruction address stored in local variable x.

3 Semantics of JVML_0

This section describes the dynamic and static semantics for JVML_0.

3.1 Dynamic Semantics

The bytecode interpreter for JVML_0 is modeled using the standard framework of operational semantics. The rules characterize transitions between machine states of the form $\langle pc, f, s\rangle$ where

- pc is a program counter.
- f is a total map from $\textsc{Var}$, the set of local variables, to values.

- s is a stack of values.

One example rule is

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

The machine begins execution in state $\langle 1,\, f_0,\, \epsilon\rangle$ where f_0 maps the local variables to any values. The Appendix contains the full set of rules.

3.2 Static Semantics

The types for JVML_0 are integers, return addresses, and $\textsc{Top}$. Integer and return address values will be assigned types according to the following rules:

$$\frac{n \text{ is an integer}}{n : \textsc{Int}} \qquad \frac{K, L \in \textsc{Addr}}{K : (\textsc{Ret-from}\ L)}$$

The type $(\textsc{Ret-from}\ L)$ will be used as the static type for value known to be a valid return address for subroutine L. The typing rule for return addresses allows any address to be interpreted as a return address for any subroutine. While the flexibility of this rule makes some parts of the JVML_0 type system simpler, it will require the proof of an invariant guaranteeing that a return address type for a subroutine will only be assigned to addresses which are valid for that subroutine.

The type $\textsc{Top}$ is the supertype of all types and represents unusable values in our static analysis. We define a simple subtyping relation as follows:

$$\frac{}{\textsc{Int} <: \textsc{Int}} \qquad \frac{L \in \textsc{Addr}}{(\textsc{Ret-from}\ L) <: (\textsc{Ret-from}\ L)}$$

$$\frac{\tau \text{ is a type}}{\tau <: \textsc{Top}}$$

We extend subtyping to cover sequences and partial maps:

$$\frac{}{\epsilon <: \epsilon} \qquad \frac{\tau <: \tau' \quad \alpha <: \alpha'}{\tau \cdot \alpha <: \tau' \cdot \alpha'}$$

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

Also, we have a subsumption rule for basic and sequence types:

$$\frac{v : \tau \quad \tau <: \tau'}{v : \tau'}$$

3.2.1 Well-typed Programs

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

$$F, S \vdash P$$

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

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

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

$$(halt)\quad \frac{P[i] = \mathtt{halt}}{F, S, i \vdash P}$$

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

$$(ret)\quad \frac{\begin{array}{c} P[i] = \mathtt{ret}\ x \\ x \in \mathit{Dom}(F_i) \\ F_i[x] = (\textsc{Ret-from}\ L) \\ \forall \rho \in G_{P,i}.\ \exists p, \rho_1, \rho_2. \left[\begin{array}{l} \rho = \rho_1 \bullet (p \cdot \rho_2) \\ \wedge\ P[p-1] = \mathtt{jsr}\ L \\ \wedge\ \mathcal{H}(P, S_i, \rho_2) <: S_p \\ \wedge\ \forall y \in \mathit{Dom}(F_L).\ \forall \tau.\ \mathcal{F}(F, i, \rho)[y] = \tau \ \Rightarrow \mathcal{H}(P, \tau, \rho_2) <: F_p[y] \end{array} \right] \end{array}}{F, S, i \vdash P}$$

Figure 3: JVML_0 static semantics.

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

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

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

$$\frac{\begin{array}{c} P[i] \in \{\texttt{inc},\ \texttt{pop},\ \texttt{push 0},\ \texttt{load}\ x,\ \texttt{store}\ x\} \\ G_i = G_{i+1} \end{array}}{G, i \vdash P \text{ labeled}}$$

Figure 4: Labeling rules for JVML$_0$.

i	$P[i]$	G	$F[1]$	$F[2]$	$F[3]$	S
1 :	`jsr 3`	ϵ	Top	Top	Top	ϵ
2 :	`halt`	ϵ	Top	Top	Top	ϵ
3 :	`store 1`	$2 \cdot \epsilon$	Top	Top	Top	(R 3) $\cdot\, \epsilon$
4 :	`jsr 7`	$2 \cdot \epsilon$	(R 3)	Top	Top	ϵ
5 :	`jsr 10`	$2 \cdot \epsilon$	(R 3)	Top	Top	ϵ
6 :	`ret 1`	$2 \cdot \epsilon$	(R 3)	Top	Top	ϵ
7 :	`store 2`	$5 \cdot 2 \cdot \epsilon$		Top	Top	(R 7) $\cdot\, \epsilon$
8 :	`jsr 10`	$5 \cdot 2 \cdot \epsilon$		(R 7)	Top	ϵ
9 :	`ret 2`	$5 \cdot 2 \cdot \epsilon$		(R 7)	Top	ϵ
10 :	`store 3`	$6 \cdot 2 \cdot \epsilon,\ 9 \cdot 5 \cdot 2 \cdot \epsilon$			Top	(R 10) $\cdot\, \epsilon$
11 :	`ret 3`	$6 \cdot 2 \cdot \epsilon,\ 9 \cdot 5 \cdot 2 \cdot \epsilon$			(R 10)	ϵ

Figure 5: The subroutine call stack and types for a JVML$_0$ program. (R L) is an abbreviation for (Ret-from L).

In this judgment, 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 such that S_i is the type of the operand stack at location i of the program. The judgment which allows us to conclude this is

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

where $F_{\textsc{Top}}$ is a function mapping all variables in $\textsc{Var}$ to $\textsc{Top}$. The third line requires that each instruction in the program is well typed according to the local judgments presented in Figure 3. These local judgments are close to Stata and Abadi, with the exception of the treatment of subroutines, which are discussed in Section 3.2.3. The fourth line places restrictions on the subroutine call graph of the program. A program with a valid labeling will not have recursive subroutines or paths reaching the beginning of a subroutine without calling `jsr`. To check these properties, we use the auxiliary judgment below:

$$(\mathit{lab\ prog}) \qquad \frac{\begin{array}{c} G_1 = \{\epsilon\} \\ \forall i \in \mathit{Dom}(P).\ G, i \vdash P \text{ labeled} \end{array}}{G \vdash P \text{ labeled}}$$

where G_i is the set of all valid subroutine call stacks valid for instruction i. Intuitively, if we think of ρ as the stack of subroutines which have been called but which have not yet returned, then $\rho \in G_i$ if P may start in state $\langle 1,\ f_0,\ \epsilon\rangle$ can reach state $\langle i,\ f,\ s\rangle$ with ρ implicit in the execution history.

The next section describes how the contents of G may be captured in the static semantics.

3.2.2 Labeling Rules

The rules defining the auxiliary judgment

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

are presented in Figure 4, and the first part of the program in Figure 5 shows an example of this labeling strategy.

In addition to enumerating all possible call stacks, the labels introduce the notion of which instructions belong to each subroutine in a program. We say that instruction i *belongs* to subroutine L only if G_i is the same as G_L. Stata and Abadi labeled each instruction with the linearization of all subroutine calling sequences leading to it. We shall show why this labeling information is insufficient to check multilevel returns in the next section.

The most important labeling rule is the rule for `jsr` L. If $P[i] =$ `jsr` L, then the labeling rule requires that, for each ρ in G_i, $(i+1)\cdot\rho$ is in G_L. This captures the notion of pushing a new return address onto the implicit subroutine call stack when jumping to a subroutine.

For any given program P, there may be more than one G satisfying the constraints of the labeling relation due to the presence of unreachable code and because the labeling rule for `jsr` L does not prevent the introduction of extraneous call stacks in G_L.

Since the second situation does affect the soundness of our type system, we require that there be some canonical G for any program P. This unique G will be written as G_P, where $G_{P,i}$ is the unique set of subroutine call stacks for instruction i in P. In addition, G_P has the following property:

```
Transfer(P,G,i,⟨F, S⟩)
  case P[i] of
    inc:     if i + 1 ∈ Dom(P)
                ∧ ∃α. S_i <: INT · α
             then
               F_{i+1} := F_i ⊔ F_{i+1}
               S_{i+1} := S_i ⊔ S_{i+1}
             else
               error
    ret x:   if F_i[x] = (RET-FROM L) for some L
             then
               for each ρ ∈ G_i do
                 choose ρ_1, ρ_2, p such that ρ_1 • (p · ρ_2) = ρ ∧ P[p − 1] = jsr L
                 S_p := H(P, S_i, ρ_2) ⊔ S_p
                 F_p := F̄_p ⊔ F_p
               where
                 Dom(F̄_p) = Dom(F_p) ∧ ∀x. F̄_p[x] = H(P, F(F, i, ρ)[x], ρ_2)
             else
               error
  return ⟨F, S⟩
```

Figure 6: `Transfer`

Property 3.1 (Smallest Sets) *Given program P, if there exists some i where $P[i] =$ `jsr` L, then:*

(1) $\epsilon \notin G_{P,L}$

(2) $\forall p \cdot \rho \in G_{P,L}.\ P[p-1] = \texttt{jsr}\ L \wedge \rho \in G_{P,p-1}$

This property ensures that no extraneous subroutine call stacks are incorporated into G_P at the beginning of a subroutine.

3.2.3 Typing Rules

The (*jsr*) rule is taken from Stata and Abadi with little change. For our purposes, the most important aspects of the rule are that the domain of local variables used by the subroutine is smaller than the domain of local variables in the calling subroutine and variables not touched by the subroutine have the same or more general type at the instruction following `jsr` L.

Before examining the (*ret*) rule, we introduce two additional judgments. We first define a judgment that assigns types to local variables for a program at line *pc* taking into account a subroutine call stack ρ. This judgment is defined by the following rules:

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

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

As long as ρ satisfies some well-formedness conditions, this judgment will be defined on all local variables. We do not discuss these conditions here, but as long as ρ belongs to $G_{P,i}$ for some i, the conditions will be satisfied. This definition matches the definition given in [SA98].

The second auxiliary judgment needed for rule (*ret*) is a judgment which hides return address types not corresponding to any return address in the subroutine call stack. This judgment is defined as

$$(ht\ 0) \qquad \frac{\tau \in \{\textsc{Int},\ \textsc{Top}\}}{\mathcal{H}(P, \tau, \rho) = \tau}$$

$$(ht\ 1) \qquad \frac{\forall p \in \rho.\ P[p-1] \neq \texttt{jsr}\ \ L}{\mathcal{H}(P, (\textsc{Ret-from}\ L), \rho) = \textsc{Top}}$$

$$(ht\ 2) \qquad \frac{\exists p \in \rho.\ P[p-1] = \texttt{jsr}\ \ L}{\mathcal{H}(P, (\textsc{Ret-from}\ L), \rho) = (\textsc{Ret-from}\ L)}$$

We also extend its definition to cover sequence types in the obvious way. As a convenience, we define subtyping involving $\mathcal{H}$ as

$$\mathcal{H}(P, \tau, \rho) <: \tau' \stackrel{\text{def}}{=} \exists \tau''.\ \mathcal{H}(P, \tau, \rho) = \tau'' \wedge \tau'' <: \tau'$$

We now turn our attention to the (*ret*) rule. The first three hypotheses of this rule are satisfied only if the variable x contains a return address for some subroutine L. The fourth hypothesis places constraints on the stack and variable types in the possible successors of i. To illustrate this, consider some ρ in $G_{P,i}$. We first decompose ρ into ρ_1, p, and ρ_2 such that

$$\rho = \rho_1 \bullet (p \cdot \rho_2)$$

and

$$P[p-1] = \texttt{jsr}\ \ L$$

An invariant preserved by program execution in our system is that if execution reaches instruction i with implicit call stack ρ where $F_i[x] = (\textsc{Ret-from}\ L)$, the value stored in $f[x]$ will be p, the return address for L in ρ. Thus, we may assume that the successor of i is p when the call stack is ρ, and Property 3.1 allows us to show that $\rho_2 \in G_{P,p}$. For S_p, we require that

$$\mathcal{H}(P, S_i, \rho_2) <: S_p$$

Intuitively, when we jump to p, we are popping subroutine return addresses of the subroutine call stack until we find p, and any return address types not corresponding to a return address in ρ_2 must be hidden.

We make a similar check for local variables with the last line:

$$\begin{array}{l} \forall y \in \mathit{Dom}(F_L).\ \forall \tau. \\ \qquad \mathcal{F}(F, i, \rho)[y] = \tau \ \Rightarrow \mathcal{H}(P, \tau, \rho_2) <: F_p[y] \end{array}$$

Note that these checks cannot be performed without knowing all the possible subroutine call stacks for instruction i. Thus, the original labeling techniques of Stata and Abadi do not capture enough information to perform these checks.

```
void f() {
  try {
    try {
      ...
    } finally {
      throw new RuntimeError;
    }
  } catch (RuntimeError e) {
    ...
  }
}
```

Figure 7: A program where an exception is thrown from inside a subroutine but caught outside of it.

4 A JVML_0 Verifier

In this section, we describe an algorithm to type check JVML_0 programs. The algorithm is divided into three parts. For program P, we first compute G_P, followed by the domain of F_i for each program point i. Finally, a data flow analysis algorithm is used to compute F and S. We separate the algorithm into these three steps so that all edges in the flow graph and uses of local variables are known before beginning the data flow analysis. If this information is not precomputed, it becomes more difficult to compute F and S efficiently and to prove that the data flow analysis algorithm is correct, as discussed in Section 9 where we compare our algorithm to some of the other proposed algorithms.

For brevity, we omit the details for the first two phases of the algorithm. G_P may be computed by generating constraints based on the labeling rules and solving them. These constraints may be solved efficiently by taking advantage of the fact that the subroutine call graph must be acyclic. Computing local variable uses is trivial once we have G_P.

4.1 `Verify`

F and S are computed by a data flow analysis algorithm that finds the fixed point in a lattice of program types. We construct a lattice of types based on the subtype relation extended to include a least type BOTTOM where the join operation $\tau_1 \sqcup \tau_2$ yields the smallest common supertype of τ_1 and τ_2. This operation may be easily extended to stack types and local variable maps based on the subtyping rules.

It is important to note that the lattice of program types $\langle F, S \rangle$ generated by the join operation has finite height.

Figure 6 shows excerpts from the `Transfer` function, which computes a new program type $\langle F, S \rangle$ from the current type by enforcing the conditions of the typing rule for instruction i. Each case in the transfer function closely resembles the corresponding typing rule. In that figure, $\mathcal{F}$ and $\mathcal{H}$ are functions based on the logical judgments from Section 3.2.3.

To verify a program P, we simply compute G and the local variable map domains, initialize F and S to appropriate initial types, and then use the transfer function to compute a fixed point in the program type lattice. We omit formal statements about the correctness of the algorithm from this summary, but soundness of the algorithm with respect to the typing rules is easy to show, and completeness relies on the finite height of the lattice and the fact that `Transfer` is monotonic.

5 JVML$_e$

The second half of this paper studies JVML$_e$, a form of JVML$_0$ with exception handlers and exceptions. For simplicity, exception values will be integers, unlike the real JVML exceptions which are objects. The `throw` instruction is added to raise an exception having the value on the top of the operand stack.

We also associate a list of exception handlers with a program. Each handler in this list is a tuple $\langle b,\, e,\, t,\, n\rangle$ where:

- $[b, e)$ is the range of instructions protected by this handler
- t is the index of the start of the handler code
- n matches the value of the exceptions to be caught by the handler.

If a program in state $\langle pc,\, f,\, n\cdot s'\rangle$ throws an exception, the entries in the list of handlers are examined in order to find the first handler $\langle b,\, e,\, t,\, n\rangle$ such that $b \leq pc < e$. If such a handler exists, control jumps to instruction t and the operand stack is set to $n\cdot\epsilon$. If no handler is found, execution halts. The JVML$_e$ dynamic semantics appear in the Appendix. The asymmetry in the inclusion and exclusion of the end points for the instruction range is due to the Java Virtual Machine specification [LY96].

The next two sections describe the new labeling and typing rules, and Figure 8 contains an example program using exceptions.

5.1 Labeling Rules

The labeling rules for `throw` is

$$\frac{P[i] = \texttt{throw}}{G, i \vdash P \text{ labeled}}$$

The more problematic issue is properly labeling exception handler targets.

An interesting situation arises when an exception handler protects a range of instructions belonging to a subroutine. It is possible that an exception thrown from inside the subroutine is caught in code outside of the subroutine. This occurs in Figure 7. We will consider the subroutine to exit implicitly as part of catching the exception. In more complex situations, more than one subroutine may need to exit to keep the implicit subroutine call stack consistent with the current line of execution.

With this in mind, we describe how to label some exception handler $\langle b,\, e,\, t,\, n\rangle$. When a program at instruction pc with subroutine call stack ρ implicit in the execution history throws an exception that causes a jump to instruction t, we require that ρ, or some initial sequence of ρ, be a valid subroutine call stack for t. This restriction matches the intuition from the example in the previous paragraph. The correct initial sequence depends on whether or not t *belongs* to any subroutine.

The first possibility is that t does not *belong* to any subroutine, i.e. $G_t = \{\epsilon\}$. In this case, all subroutines are implicitly popped off the subroutine call stack when control transfers to t. This leads to our first labeling rule:

$$\frac{G_t = \{\epsilon\}}{G \vdash \langle b,\, e,\, t,\, n\rangle \text{ labeled}}$$

The second possibility is that instruction t *belongs* to some subroutine L. In this case, the labeling rule will require that L *dominates* all instructions in the range $[b, e)$, where L *dominates* instruction i only if every ρ in G_i contains a return address for L.

i	$P[i]$	G	D	$F[1]$	$F[2]$	$F[3]$	S
1 :	`jsr 3`	ϵ	ϵ	Top	Top	Top	ϵ
2 :	`halt`	ϵ	ϵ	Top	Top	Top	ϵ
3 :	`store 1`	$2 \cdot \epsilon$	$3 \cdot \epsilon$	Top	Top	Top	$(\mathrm{R}\ 3) \cdot \epsilon$
4 :	`jsr 7`	$2 \cdot \epsilon$	$3 \cdot \epsilon$	(R 3)	Top	Top	ϵ
5 :	`jsr 10`	$2 \cdot \epsilon$	$3 \cdot \epsilon$	(R 3)	Top	Top	ϵ
6 :	`ret 1`	$2 \cdot \epsilon$	$3 \cdot \epsilon$	(R 3)	Top	Top	ϵ
7 :	`store 2`	$5 \cdot 2 \cdot \epsilon$	$7 \cdot 3 \cdot \epsilon$	(R 3)	Top	Top	$(\mathrm{R}\ 7) \cdot \epsilon$
8 :	`jsr 10`	$5 \cdot 2 \cdot \epsilon$	$7 \cdot 3 \cdot \epsilon$	(R 3)	(R 7)	Top	ϵ
9 :	`ret 2`	$5 \cdot 2 \cdot \epsilon$	$7 \cdot 3 \cdot \epsilon$	(R 3)	(R 7)	Top	ϵ
10 :	`store 3`	$6 \cdot 2 \cdot \epsilon,\ 9 \cdot 5 \cdot 2 \cdot \epsilon$	$10 \cdot 3 \cdot \epsilon$	(R 3)		Top	$(\mathrm{R}\ 10) \cdot \epsilon$
11 :	`ret 3`	$6 \cdot 2 \cdot \epsilon,\ 9 \cdot 5 \cdot 2 \cdot \epsilon$	$10 \cdot 3 \cdot \epsilon$	(R 3)		(R 10)	ϵ
12 :	`pop`	$2 \cdot \epsilon$	$3 \cdot \epsilon$	(R 3)	Top	Top	$\mathrm{INT} \cdot \epsilon$
13 :	`ret 1`	$2 \cdot \epsilon$	$3 \cdot \epsilon$	(R 3)	Top	Top	ϵ
14 :	`pop`	$6 \cdot 2 \cdot \epsilon,\ 9 \cdot 5 \cdot 2 \cdot \epsilon$	$10 \cdot 3 \cdot \epsilon$	(R 3)		(R 10)	$\mathrm{INT} \cdot \epsilon$
15 :	`ret 3`	$6 \cdot 2 \cdot \epsilon,\ 9 \cdot 5 \cdot 2 \cdot \epsilon$	$10 \cdot 3 \cdot \epsilon$	(R 3)		(R 10)	ϵ
16 :	`throw`	ϵ	ϵ	Top	Top	Top	$\mathrm{INT} \cdot \epsilon$
17 :	`throw`	ϵ	ϵ	(R 3)		(R 10)	$\mathrm{INT} \cdot \epsilon$

Exception Table:

Range	Target	Exception Value
[8,11)	12	3
[11,12)	14	1
[1,6)	16	2
[11,12)	17	2

Figure 8: The subroutine call stack, dominator, and type information computed for a JVML_e program.

If this condition is met, every possible call stack for each instruction protected by the handler will contain a return address for L. Therefore, when an exception causes a jump to t, we will always be able to pop return addresses off the subroutine call stack until a return address for L is reached. Moreover, examination of the instruction labeling rules shows that if L is the top element on the remaining call stack, the subroutine will belong to G_L, and since t *belongs* to L, the new subroutine call stack will also be present in G_t.

The labeling rule for this case is

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

where $D_{b,e}$ if the set of all subroutines dominating the instructions in the range $[b, e)$.

Two approaches to construct a unique G_P by selecting the appropriate L from $D_{b,e}$ are described in Section 6.

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

$$(gt\ 0)\quad \frac{\begin{array}{c}G_{P,t} = \{\epsilon\}\\ Dom(F_t) = \text{VAR}\end{array}}{GoodTarget(P, F, t)}$$

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

$$(exc\ jump)\quad \frac{\begin{array}{c}\rho_2 \in G_{P,t}\\ \forall y \in Dom(F_t).\ \forall \tau.\ \mathcal{F}(F, i, \rho_1 \bullet \rho_2)[y] = \tau \Rightarrow \mathcal{H}(P, \tau, \rho_2) <: F_t[i]\end{array}}{GoodExceptionJump(P, F, \rho_1 \bullet \rho_2, i, t)}$$

Figure 9: Typing rules for exception handlers.

5.1.1 Typing Rules

The typing rule for `throw` is:

$$(throw)\quad \frac{\begin{array}{c}P[i] = \texttt{throw}\\ S_i = \text{INT} \cdot \alpha\end{array}}{F, S, i \vdash P}$$

Each exception handler is type checked using the (*wt handler*) judgment in Figure 9. The third line uses the *GoodTarget* judgment to verify that $Dom(F_t)$ is the same as at the beginning of the subroutine to which t belongs, or VAR if t is not in any subroutine as required for our analysis to be sound.

The last two lines of the judgment check whether S_t and F_t contain valid types given the information about the program points in the range of the exception handler. For the stack, it suffices to require that S_t be a supertype of $\text{INT} \cdot \epsilon$. The case for local variables turns out to have conditions fairly similar to rule (*ret*).

6 JVML$_e$ Verifier

The type checker from Section 4 may be extended to cover exception handlers with some modification. This section discusses how to compute G. Changes to the other parts of the algorithm are relatively straight forward.

The major difficulty in computing G is to determine G_t for each handler $\langle b,\ e,\ t,\ n\rangle$. This boils down to setting G_t to either $\{\epsilon\}$ or G_L for some subroutine L in $D_{b,e}$, but determining the correct set is tricky. For example, in Figure 8, the second and fourth exception handlers protect the same range of code. However, the second exception handler's target must belong to the subroutine for which local variable 3 stores a valid return address, while the target for the fourth exception handler does not need to belong to any subroutine. To make this distinction, we need to know which previously stored return addresses are still live at the beginning of the exception handler. No algorithm can effectively determine G_t without this liveness information.

One way to avoid this problem altogether is to guess the value of G_t for each exception handler without computing any additional information. The set of possible values for G_t is typically very small [Fre98], and in most cases there is only one choice. All guesses can be tried until one of them allows the program to be type checked.

Another option is to compute the approximate types of the values stored in the local variables and on the stack by traversing through the code array once. A single pass over the code provides enough information to determine which return addresses are used in the exception handlers. The algorithm can then pick the most recently called dominator from these return addresses and consider the exception handler target to belong to that subroutine. This method is slightly more restrictive than the non-deterministic algorithm because, in some cases, choosing the most recently called dominator will prevent a program from being typed. However, all common Java compilers seem to output only bytecode programs for which this method works.

7 Implementation

We have implemented prototypes of the JVML_0 and JVML_e verifiers. These prototypes not only show that our specification does in fact reject faulty programs, but, also, that our rules are permissive enough to accept virtually all bytecode programs generated by Java compilers. To demonstrate this, we have developed a translator that converts any JVML method into a slightly extended form of JVML_e. This translation preserves the structure and data flow of the original program, but utilizes a small core verifier to perform the type synthesis.

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. We have found only one case of a Java program whose compiled form is accepted by the Sun verifier but not by ours. This case is described in the next section.

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 Sun's Verifier

The most significant difference between our work and the Sun verifier stems from how the verifier determines which instructions belong to subroutines. This is one area where the original specification was ambiguous, and the Sun implementation uses a slightly less restrictive mechanism than our labeling technique. However, their mechanism is more difficult to model in a framework like ours.

The basic difference is that the Sun labeling technique allows `goto` instructions to be treated as exits from subroutines. This case occurs in compiled code when a `continue` or `break` statement in a Java program causes execution to abruptly jump out of a `finally` block. We could change our system to accept this behavior by rewriting our branch rules to be similar to the exception rules. This would not significantly affect the type system or soundness proof, but it may impact the computation of G in a way similar to exception handlers. Alternatively, the specification could be strengthened to forbid this scenario, and different compilation strategies could be used to avoid them.

9 Related Work

This work is an extension and refinement of the Stata and Abadi type system which labeled instructions with a linearization of all subroutine calling sequences leading to that instruction. Their labeling strategy made extending their system to multilevel returns and exception handling difficult. By changing the labeling of instructions to be the set of possible subroutine call stacks, we were able to develop the static semantics for these more advanced features.

Hagiya and Tozawa have presented a type system for subroutines that has a similar technique for labeling and checking subroutines [HT98]. The main technical difference is that the types assigned to return addresses are dependent on the call stack 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.

These features make their soundness proofs simpler than ours, but our approach is closer to the original specification of the verifier [LY96], which makes it easier to relate our system and soundness proofs to existing verifiers and reason about their implementations. This has proven to be an effective way at discovering trouble spots in implementations [FM98]. Our system also goes beyond their work by considering exceptions. In addition, as we have studied larger subsets of JVML, we have found that not having the types associated with return addresses depend on a specific subroutine call stack allows us to better isolate the impact of subroutines on reasoning about the type system.

Their paper also provides a brief overview of a data flow algorithm to type check JVML_0 programs. In contrast to the three steps in ours, their algorithm computes all information in one phase. While this may seem simpler, there is a disadvantage. Since they do not compute which variables are used in subroutines before starting the data flow analysis, their algorithm must guess whether or not each variable is used by a subroutine when the subroutine is first encountered. If an error occurs, the algorithm must back up and revise the guess before continuing. We are able to avoid back tracking because we precompute the variable usage information.

A type system for JVML has also been developed by Qian [Qia98]. The scope of his original type system covers a larger fragment of JVML. Qian enforces the restrictions on subroutine calling sequences and local variable usage by constructing modification histories, structures containing information about which variables are modified between the beginning of each subroutine call and each program point. This difference has a significant impact on the structure of the type system.

Qian's recent work presents an algorithm to verify subroutines based on this type system [Qia99]. 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 the completeness of the algorithm with respect to the typing rules because the transfer function used to compute the type information plus local variable usage information is not obviously monotonic. His algorithm does not consider exceptions.

Colgio, Goldberg, and Qian have used the SpecWare system to specify the verifier and then derive an implementation, although subroutines and exceptions are not modeled in their system currently [CGQ98]. One possible direction for future work is to use a tool like SpecWare to to replace our by-hand translation of our static semantics into an executable verifier. In related work, Goldberg studied the connection between dynamic loading and bytecode verification [Gol98].

A variety of other projects have addressed Java bytecode semantics in different settings, but many of these do not consider subroutines [Pus99, Ros98, PV98, Coh97, Ber98]. Several recent projects have departed from Sun's original specification and have developed static semantics for JVML based on ideas from the TIL and TAL type systems of Morrisett et al. [O'C99, TMC+96, MCGW98], and the type system of Haskell [Jon98, Yel99].

10 Conclusions

In summary, this paper makes several contributions to the development of a formal specification for the bytecode verifier including:

- a type system for subroutines close to the original Sun specification in its treatment of multilevel returns.
- a verifier proven to accept exactly the set of well typed programs for this subset of JVML.
- extensions to the type system and verifier to include exception handlers.

Incorporating other instructions, such as those for object initialization [FM98], is very straightforward, and we have in fact constructed a type system and verifier for a much larger subset of the bytecode language based on the work presented here.

Beyond covering larger subsets of JVML, the next challenge is to extend our system to model other parts of the JVM that affect or depend upon bytecode verification, such as dynamic loading and the stack inspection-based security model. We are also investigating ways to enforce additional safety properties, such as the adherence to certain locking conventions, and ways to identify places where run-time checks, such as null pointer tests, may be eliminated. Our prototype implementation has been extended to cover some of these items.

Acknowledgments: Thanks to Zhenyu Qian for comments on a draft of this paper.

References

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

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

[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 Report STAN-CS-98-62, April 1998.

[Fre98] Stephen Freund. The costs and benefits of Java bytecode subroutines. In *Workshop on the Formal Underpinnings of the Java Paradigm*, October 1998.

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

[Gon98] Li Gong. Java Security Architecture (JDK1.2) . available from http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html, October 1998.

[HT98] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of java virtual machine subroutines. available from http://nicosia.is.s.u-tokyo.ac.jp/members/hagiya.html, 1998.

[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.

[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.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998. An extended version appears as Digital Equipment Corporation Systems Research Center Research Report 158, June 1998.

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

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

A JVML$_0$ Semantics

This appendix contains the judgments omitted from the earlier discussion of the semantics of JVML$_0$ and JVML$_e$ and statements of the main soundness theorems for each system. A full presentation of the proofs for prototype systems and algorithms appears in the extended version of this paper. We begin with the JVML$_0$ operational semantics.

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

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

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

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

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

Any well-typed JVML$_0$ program will not cause a run-time type error. We may state this formally as the following theorem:

Theorem A.1 (JVML$_0$ Soundness) *Given P, F, and S such that $F, S \vdash P$:*

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

This theorem implies that the only way for a program to get stuck is to reach a `halt` instruction. As mentioned in Section 3.1, a program would get stuck if it were about to perform an illegal operation, such as any operation leading to a type error. If a well-typed program never gets stuck until it halts, then it never causes a type error. The proof follows the same form as those presented in [SA98] and [FM98].

B JVML$_e$ Semantics

Representative cases from the JVML$_e$ dynamic semantics are given below. The rest of the rules are similar to those for JVML$_0$.

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

$$\frac{\begin{array}{c} P[pc] = \mathtt{throw} \\ CorrectHandler(H, pc, n) = t \\ t \neq 0 \end{array}}{P, H \vdash \langle pc,\ f,\ n \cdot s\rangle \rightarrow \langle t,\ f,\ n \cdot \epsilon\rangle} \qquad \frac{\begin{array}{c} P[pc] = \mathtt{throw} \\ CorrectHandler(H, pc, n) = 0 \end{array}}{P, H \vdash \langle pc,\ f,\ n \cdot s\rangle \rightarrow \mathtt{UncaughtException}}$$

$$\frac{\neg\exists i \in Dom(H).\ GoodHandler(H[i], pc, n)}{CorrectHandler(H, pc, n) = 0}$$

$$\frac{\begin{array}{c} GoodHandler(pc, \sigma, H[j]) \\ H[j] = \langle b,\ e,\ t,\ n\rangle \\ \neg\exists i \in Dom(H).\ i < j \wedge GoodHandler(H[i], pc, n) \end{array}}{CorrectHandler(H, pc, n) = t}$$

$$\frac{b \leq pc < e}{GoodHandler(\langle b,\ e,\ t,\ n\rangle, pc, n)}$$

The next two rules are analogous to (*wt prog*) and (*lab prog*) in the JVML$_e$ system:

$$(wt\ exc) \qquad \frac{\begin{array}{c} F_1 = F_{\mathrm{TOP}} \\ S_1 = \epsilon \\ \forall i \in Dom(P).\ F, S, i \vdash P \\ \forall i \in Dom(H).\ F, S \vdash H[i] \text{ handles } P \\ G \vdash P, H \text{ labeled} \end{array}}{F, S \vdash P, H}$$

$$(lab\ exc) \qquad \frac{\begin{array}{c} G_1 = \{\epsilon\} \\ \forall i \in Dom(P).\ G, i \vdash P \text{ labeled} \\ \forall i \in Dom(H).\ G \vdash H[i] \text{ labeled} \end{array}}{G \vdash P, H \text{ labeled}}$$

Our soundness theorem for JVML$_e$ is the same as it was for JVML$_0$, except that there are now two possible reasons why a program gets stuck. The first is, as before, when a halt instruction is reached. The second corresponds to throwing an uncaught exception, which causes a program to move into a special state for which no transitions have been defined.

Theorem B.1 (JVML$_e$ Soundness) *Given P, H, F, and S such that $F, S \vdash P, H$:*

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

In the full JVML, an uncaught exception would cause the top activation record to be popped off the method call stack, and the exception would be rethrown in the caller of the current method.