A Type System for JVM Threads

Gaetano Bigliardi    Cosimo Laneve

Technical Report UBLCS-2000-06

June, 2000

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)
Recent Titles from the UBLCS Technical Report Series

A Type System for JVM Threads

Gaetano Bigliardi  Cosimo Laneve

Technical Report UBLCS-2000-06
June, 2000

Abstract

The current definition of the Java Bytecode Verifier, as well as the proposals to formalize it, do not include any check about consistency of critical sections (those between monitorenter and monitorexit instructions). So code is run, even if the nesting of critical sections is corrupted. In this paper we isolate a sublanguage of the Java Virtual Machine with thread creation and mutual exclusion. For this subset we define a semantics and a formal verifier that enforces basic properties of threads and critical sections. Our verifier integrates well with previous formalizations of the Java Bytecode Verifier.

1. Partial support for this work was provided by the Microsoft Grant “The design and the analysis of Concurrent Object Oriented Languages” and the Esprit WG-21836 – CONFER-2.
1 Introduction

The Java Programming Language is compiled into an intermediate language, called (Java) bytecode. The bytecode is then interpreted by the Java Virtual Machine (JVM, in the following) [1]. Most of the portability of the Java language relies on the fact that JVM’s have been defined for almost all the platforms. Indeed, the present scenario is that the bytecode on some machine may be shipped and executed on another one, which is the basis of Java code mobility.

However, bytecode mobility poses a sequel of problems, as bytecodes generated by hostile compilers, or created by attackers, or corrupted along the migration. To contrast possible harmful bytecodes, Java developers have defined the so-called bytecode verifier, which checks the format of downloaded classes, the presence of illegal conversions, jumps to invalid addresses, methods invoked with a wrong number or type of actual parameters.

The official definitions of the bytecode verifier consist of a prose description and an implementation [12]. As usual, such definitions are not satisfactory as regards formal reasoning and proof checking. Therefore, very recently, several authors have developed formal specifications of parts of the bytecode verifier, as well as possible extensions [17, 6, 8]. Remarkably, soundness proofs have been given, and a bug of the Sun implementation of the bytecode verifier has been identified.

There are at least two key features that escape from the present definitions of the bytecode verifier: concurrency and class loading (some progress in the formal definition of class loading has been recently done in [16]). In this paper we try to fill the first of them, leaving to a future work the formalization of a verifier for the class loading mechanism.

At the present time, the bytecode verifier does not filter out the following code:

```
  1    load 0
  2    monitorexit
  3     return
```

that releases a lock (monitorexit instruction) without having acquired it before. As a consequence, the JVM will execute the above code and generate the following runtime exception:

```
java.lang.IllegalMonitorStateException: current thread not owner
```

A similar error occurs with a jump to inside a monitorenter-monitorexit region, because the jump may skip the entry point (the monitorenter instruction) at runtime. Even more pernicious is the execution of the code:

```
  1    load 0
  2    monitorenter
  3     return
```

that may be the cause of a possible deadlock. However, no exception is shown up both by the verifier and by the JVM.

We supply to these and other deficiencies of the bytecode verifier by designing a type system, thus conforming with previous works of Stata-Abadi and Freund-Mitchell. To this purpose, we define the operational semantics of a fragment of the Java Virtual Machine Language (in the following shortened into JVML) encompassing multithreading and mutual exclusion, and we prove the correctness of our typing system with respect to the operational semantics.

The structure of the paper is the following. Section 2 overviews concurrency in the JVM and in the bytecode, and gives a detailed sketch of what has been achieved and how. Sections 3 and 4 define the syntax and the operational model of \( \text{JVML}_C \), the sublanguage of JVML with primitives for thread creation and mutual exclusion, and we prove the correctness of our typing system with respect to the operational semantics.

The structure of the paper is the following. Section 2 overviews concurrency in the JVM and in the bytecode, and gives a detailed sketch of what has been achieved and how. Sections 3 and 4 define the syntax and the operational model of \( \text{JVML}_C \), the sublanguage of JVML with primitives for thread creation and mutual exclusion, and we prove the correctness of our typing system with respect to the operational semantics. Our static semantics is defined in Section 5, where we also prove its correctness with respect to the operational semantics and provide a type inference algorithm. In Section 6 we discuss the extension of \( \text{JVML}_C \) with the synchronization primitives (\texttt{wait}, \texttt{notify} and \texttt{notifyAll}), with method invocations and with exception handlers. We comment related works and conclude in Section 7.
2 Threads and mutual exclusion in the bytecode Java

Java supports concurrent programming through threads and monitors [11]. A new thread of control may be created by (1) creating a (sub-)object of the class java.lang.Thread of the standard Java libraries, and then (2) invoking the start method of this class. The method start spawns a new thread and returns. The control of this new thread is given to the run method of the object by the Java Virtual Machine. A thread exits when the run method returns. Since the default run of Thread does nothing, to design a parallel behaviour one must define a sub-class of Thread with a new run method.

Synchronization across threads is implemented through monitors. That is, each object has an associated lock and synchronization is defined by acquiring and releasing locks. The Java language provides two forms of synchronization: through synchronized methods and through synchronized statements.

If a thread invokes a synchronized method of an object, the invocation locks the object and releases the lock when the method returns. In between, other invocations of synchronized methods will be blocked. Synchronized methods are usually implemented by the ACC_SYNCHRONIZED flag in the constant pool [12]. In particular, the method invocation checks whether the ACC_SYNCHRONIZED flag of the method is set. In this case, the lock of the object is implicitly acquired; and will be released when the method returns. Therefore no explicit bytecode instruction is used to this purpose.

The statement synchronized (x) { S } models partial synchronization, namely it is used in those cases when a part of the body of a method needs to be synchronized on an object x. Its semantics is to execute the parenthesized statement S in a mutual exclusive way on the object x, by acquiring and releasing the lock at the beginning and at the end of the execution.

The statement synchronized is implemented in the bytecode through the instructions monitorenter and monitorexit. To illustrate the implementation, consider the following Java program:

```java
public class Agent extends Thread {
    public Cell ref;
    public void run() {
        synchronized (ref) { ref.val ++; }
    }
}
```

The behaviour of an Agent is to compete for a Cell stored in the field ref (statement synchronized). When the competition succeeds, the Agent increments the field val of the Cell and terminates. (The code of Cell has been omitted.)

Figure 1 shows a bytecode implementation of Agent (actually it is almost the output of the Sun Java-compiler). The instructions 1–4 are used to load on the stack the Cell-object that the Agent wants to lock. Observe that this object is recorded into the variable 1 (instruction 3). The monitorenter at 5 acquires the lock of the Cell-object. The critical section goes from 6 to 14, and the instruction till 12 are used to increment the field val of the Cell. The following two instructions perform the unlock operation: the instruction 13 restores the locked object on the stack; the instruction 14 actually relinquishes the lock of the object on the stack. The method returns at line 15. The instructions from lines 16 to 19 are used to catch exceptions that occur along the lines from 6 to 13 (see the Exception table). They guarantee that, if an exception is raised in between the lines from 6 to 13, the lock is always released and then the exception is thrown (instruction athrow).

The code in Figure 1 fits properly with the following requirement of the JVM (see [12], section 3.11.11):

```
Proper implementation of synchronized blocks requires cooperation from a compiler targeting the Java virtual machine. The compiler must ensure that at any method invocation completion a monitorexit instruction will have been executed for each monitorenter instruction executed since the method invocation. This must be the
```

UBLCS-2000-06
case whether the method invocation completes normally or abruptly\(^2\).

Notwithstanding this strong statement, the bytecode verifier does not check any completion of critical sections (see [12], section 4.9.2). Also, no guarantee is given on exception handlers. To this respect, at every point inside the critical section, one should be able to release the lock, regardless of what is going on. Remark that this ability of releasing the right lock everywhere in the critical section bans codes like

\begin{verbatim}
  1 load x
  2 monitorenter
    ...
  i load y
  i+1 monitorenter
    ...
  k load x
  k+1 monitorexit
    ...
  h load y
  h+1 monitorexit
\end{verbatim}

(intertwined critical sections) because it is not clear what lock releasing in the section from 2 to

\(^2\) A method invocation completes “abruptly” when its body causes a JVM-exception that is not handled within the method.
To formalize the previous quotation, we have singled out the following three properties a bytecode should satisfy:

**Property 1.** Critical sections fit with the following pattern

\[
i \quad \text{load } x \\
i + 1 \quad \text{monitorenter} \\
\cdots \text{(critical section)} \\
i' \quad \text{load } x \\
i' + 1 \quad \text{monitorexit}
\]

where the variable \( x \) is protected, namely its updates inside the critical section are forbidden. Moreover, given two different critical sections, either their codes are disjoint or one of them is contained into the other.

**Property 2.** Jumps are always inside their own critical section, but outside inner critical sections.

**Property 3.** Every critical section is paired with a piece of code, its exception handler, that releases the acquired lock, as well as the locks of the outer critical sections.

Doubtless, these properties, all together, are sufficient conditions that guarantee the above quotation, as well as the prevention of "current thread not-owner" exception. For instance, the right bracketing of synchronization statements may be broken by dropping property 1. This is the case when the instructions load 0 store 1 are inserted in between instructions 12 and 13 of Figure 1.

It is also evident that Properties 1, 2 and 3 are not necessary conditions for correct Java bytecodes. For example, a safe bytecode is one that creates an alias of a protected variable and loads the alias before the monitorexit instruction, rather than the protected variable. Still, this bytecode is rejected because it does not comply with Property 1.

It is difficult to give striking arguments against such kind of criticisms. We hope the following arguments persuade the reader:

- our restrictions are supported by an efficient verifier, that can be easily integrated into the bytecode verifier;
- our choices are implicitly carried out by the Sun Java compiler;
- the programming practice of assembler languages must be suitably restricted to improve readability, verification and maintenance;
- codes that are rejected are small quibbles.

To implement Properties 1, 2 and 3, we design a type system, in the same style of [17] and [6]. Since a thread may modify objects which are in common with other threads, we must carefully check that thread updates of shared data (objects and object locks) do not invalidate the correctness of threads in parallel. To this aim, our model contains the heap, a runtime area shared among threads. And typability of a configuration also takes into account the type of objects stored into the heap. The other addition to systems in [17, 6] regards critical sections. To keep track of the nesting of critical sections, we supplement the type system with the block informations, namely a sequence of tuples \( (i, i', x) \) specifying the critical section boundaries \((i \text{ and } i')\) and the protected variable \((x)\). The type system verifies that every instruction can be properly typed with the block information defining the static nesting of critical sections.

In the first part of the paper, for simplicity, we don’t consider exception handlers. Therefore the type system in Section 5 does not assure Property 3. We discuss in Section 6.3 the integration dealing with exception handlers.

Besides synchronized methods and statements, Java offers other three primitives (library methods) to control thread execution: wait, notify and notifyAll. We discuss about waiting and notification in section 6. The analysis of these primitives is deferred because it may be easily combined with that of monitorenter and monitorexit instructions.
3 The syntax of JVML

The language JVML is a restriction of JVML that includes basic constructs and instructions for concurrency.

In JVML, a program is a collection of class declarations:

```
class C {
    super: Thread
    fields: F
    method run ()
    P
}
```

where each class is actually a subclass of Thread and only contains the method run. Fields F are finite sequences of pairs \( a \tau \), where \( a \in \text{IID} \) is an identifier, and \( \tau \) is an integer INT or an object type \( \sigma \in T \). Bodies P are partial maps from addresses ADDR to instructions. JVML has the following instructions:

```
Instruction ::= inc | pop | push0 | load x |
            | store x | if L |
            | new \( \tau \) | putfield \( \sigma.a \tau \) | getfield \( \sigma.a \tau \)
            | start \( \sigma \) | monitorenter | monitorexit |
            | return |
```

where \( x \) ranges over a finite set of variables VAR, and \( L \) ranges over ADDR. Variables will be represented by positive integers, but we keep separate the sets INT and VAR.

The informal meaning of these instructions is as follows:

- inc increments the content of the stack; pop and push0 perform the standard operations on the stack; if L pops the top value off the stack and either goes through when that value is the integer 0 or jumps to the address L otherwise;
- new \( \tau \) allocates a new object of type \( \tau \), initializes it and puts it on top of the stack; putfield \( \sigma.a \tau \) pops the value on the stack and the underlying object value, and assigns the former to the field \( a \) of the latter; getfield \( \sigma.a \tau \) pops the object on the stack and pushes the value in the field \( a \);
- start \( \sigma \), monitorenter and monitorexit are the concurrent instructions. The first one creates and starts a new thread for the object on top of the stack. This operation corresponds to

```
invokevirtual java/lang/Thread/start()
```

namely, the standard operation to trigger new threads in the Java bytecode. The instructions monitorenter and monitorexit are the synchronization primitives that lock and unlock the object on top of the stack;
- return terminates program execution.

The restriction to consider classes as extensions of Thread simplifies our analysis, since the method start may be safely invoked inside our programs. For classes with constructors and initializers we refer to the analysis in [6] and for classes with other methods see [7]. We assume that initialization is performed at the same time of object creation by Java default initializers that put 0 in every integer field and null in every object field.

4 The operational semantics

The bytecode interpreter for JVML is defined using the same framework as in [17, 6]. We briefly review the framework before defining our concurrent model.

3. Unlike the bytecode, our instruction start carries an argument, which is the type (or the class name) of the object whose method must be triggered. This is an artifact to define the semantics of start: in the JVM the first address of the right method run is found in the heap. Here, we prefer to keep the heap as simple as possible and, therefore, we derive the address of run from the argument of start (see the operational semantics in Figure 3).
4.1 Notation, types and values

We start with addresses ADDR. We assume that ADDR and positive integers are different, even if we use the constant 1 and the operation + for the formers. We write \( \text{dom}[P] \) for the set of addresses of \( P \), \( P[i] \) is \( i \)-th instruction in \( P \), if \( i \in \text{dom}[P] \). If \( \sigma \) is the type of the class of a program \( P \), then we let \( 1_\sigma \in \text{dom}[P] \), for every \( P \), be the first instruction of \( P \). By extension, when \( Q \) is a set of class bodies, we let \( 1_\sigma \in \text{dom}[Q] \) be the first instruction of the class body of the class \( \sigma \).

We use partial maps to represent most of our entities (heaps, memory functions, etc.). If \( f \) is partial map, we let \( f[x \mapsto v] \) be the updating operation, that gives the function \( f \) where the value of \( x \) is \( v \). The symbol \( \varepsilon \) denotes the empty map (the map undefined everywhere).

Types \( \tau \) are \( \text{TOP} \), integers \( \text{INT} \) and object types \( T \). The latter, ranged over by \( \sigma \), include all the class names of the program. We assume, for every \( \sigma \), a countable set of object names, ranged over \( o, d, \ldots \). Let \( O \) be the set of all object names. Values \( v \) are integer constants or object names. The type \( \text{TOP} \) includes all the values.

Each type \( \sigma \) is the record of fields \( [a_i : \tau_i] \) that are defined in the corresponding class definition. We address fields with the usual dot notation; therefore we write \( \sigma.a : \tau \) if \( \sigma = [a : \tau, a_i : \tau_i] \). For every type \( \sigma = [a_i : \tau_i] \), we define the family of record values of \( \sigma \) as the set of \( [a_i = v_i] \), where \( v_i \) are values of type \( \tau_i \). In the following, record values are denoted by \( \rho \).

4.2 The operational model

Each instruction performs a transformation of machine states, that are configurations

\[
H \langle \rho c_1, f_1, s_1, z_1 \rangle, \ldots, \langle \rho c_n, f_n, s_n, z_n \rangle
\]

with the following meaning:

- \( H \) is the heap, namely a partial function whose domain is the set \( O \) and whose range is the set of record values. The special field \( \ell, \ell \notin \text{FD} \), represents the lock associated with the object. We assume that record values always have a field \( \ell \). Observe that \( \text{getfield} \) and \( \text{putfield} \) cannot read/write on the locks.
- each tuple \( \langle \rho c_i, f_i, s_i, z_i \rangle \) is a thread; \( \rho c_i \) is the address of the instruction to be executed; \( f_i \) is a total map from the set \( \text{VAR} \) of local variables to the set of values; \( s_i \) is a stack of values; \( z_i \) is a finite subset of \( O \) and represents the set of objects locked by the thread.
- Let \( \sigma \) be the type of the class whose \( \text{run} \) is invoked. The machine begins its execution in the state \( H \langle 1_\sigma, f_0[0 \mapsto o], \varepsilon, \theta \rangle \), where
  - \( H \) is the heap, \( \langle 1_\sigma, f_0[0 \mapsto o], \varepsilon, \theta \rangle \) is the record value of \( \sigma \)-objects that are initialized to 0 and null, according to they are integers or objects. The special field \( \ell \) is initialized to 0. We use the notation \( H[0,a \mapsto v] \) to update the field \( a \) of \( o \) to the value \( v \).
  - \( f_0 \) maps the local variables to any values;
  - \( \varepsilon \) is the empty stack.

The rules that define the operational semantics of JVMLC are shown in Figure 2. In Figure 2 we let \( Q \) be a set of class bodies, each class body being identified by a different set of addresses. We also leave the set \( Q \) implicit in the rewritings. Every rule in Figure 2 actually mentions the components that participate to the rewriting. Of course the rewriting applies to every configuration that contains the components. More explicitly, for every rewriting rule of Figure 2, we use the rule:

\[
\frac{(\text{CONTEXT})}{H \langle T_1, T \rangle \rightarrow H \langle T_2, T \rangle}
\]

where \( T, T_1 \) and \( T_2 \) are sets of threads.

We overlook the first ten rules of Figure 2, because standard (see [17, 6]) and almost intelligible. On the contrary, the last five rules are new and peculiar of our contribution.

The rule modelling the instruction \( \text{start} \ \sigma \) creates a new thread of control. The new thread begins into a state where the first instruction of the class \( \sigma \) must be executed, and with the variable 0 containing the object on top of the stack. Notice that none of the locks held by the caller thread is retained by the new thread.
Figure 2. The operational semantics of JVMLC:

\[
\begin{align*}
Q[pc] &= \text{inc} \\
\vdash_H \langle pc, f, n \cdot s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f, (n + 1) \cdot s, z \rangle \\
Q[pc] &= \text{push0} \\
\vdash_H \langle pc, f, s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f, 0 \cdot s, z \rangle \\
Q[pc] &= \text{pop} \\
\vdash_H \langle pc, f, v \cdot s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f, s, z \rangle \\
Q[pc] &= \text{if } L \\
\vdash_H \langle pc, 0 \cdot s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f, s, z \rangle \\
Q[pc] &= \text{load } x \\
\vdash_H \langle pc, f, s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f, f(x) \cdot s, z \rangle \\
Q[pc] &= \text{store } x \\
\vdash_H \langle pc, f, v \cdot s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f[x \mapsto v], s, z \rangle \\
Q[pc] &= \text{new } \sigma \\
&\quad \text{with } o' \notin \text{dom}[H] \\
\vdash_H \langle pc, f, s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f, o' \cdot s, z \rangle \\
Q[pc] &= \text{putfield } \sigma.a \tau \\
&\quad \text{with } H'[o.a \mapsto v] \\
\vdash_H \langle pc, f, v \cdot o \cdot s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f, s, z \rangle \\
Q[pc] &= \text{getfield } \sigma.a \tau \\
&\quad \text{with } H(o.a) = v \\
\vdash_H \langle pc, f, o \cdot s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f, v \cdot s, z \rangle \\
Q[pc] &= \text{start } \sigma \\
&\quad \text{with } o \in \text{dom}[H] \\
\vdash_H \langle pc, f, o \cdot s, z \rangle &\rightarrow \vdash_H \langle pc + 1, f, s, z \rangle, \langle 1, f, 0[1] \mapsto o \rangle, \varepsilon, \theta \\
Q[pc] &= \text{monitorenter} \\
&\quad \text{with } H(o.\ell) = 0 \\
&\quad \text{and } H'[o.\ell \mapsto 1] \\
\vdash_H \langle pc, f, o \cdot s, z \setminus \{o\} \rangle &\rightarrow \vdash_H' \langle pc + 1, f, s, z \cup \{o\} \rangle \\
Q[pc] &= \text{monitorexit} \\
&\quad \text{with } H(o.\ell) = 1 \\
&\quad \text{and } H'[o.\ell \mapsto 0] \\
\vdash_H \langle pc, f, o \cdot s, z \cup \{o\} \rangle &\rightarrow \vdash_H' \langle pc + 1, f, s, z \rangle \\
Q[pc] &= \text{monitorenter} \\
&\quad \text{with } H(o.\ell) = n \cdot (n > 0) \\
&\quad \text{and } H'[o.\ell \mapsto n + 1] \\
\vdash_H \langle pc, f, o \cdot s, z \cup \{o\} \rangle &\rightarrow \vdash_H' \langle pc + 1, f, s, z \cup \{o\} \rangle \\
Q[pc] &= \text{monitorexit} \\
&\quad \text{with } H(o.\ell) = n \cdot (n > 1) \\
&\quad \text{and } H'[o.\ell \mapsto n - 1] \\
\vdash_H \langle pc, f, o \cdot s, z \cup \{o\} \rangle &\rightarrow \vdash_H' \langle pc + 1, f, s, z \cup \{o\} \rangle
\end{align*}
\]
The other rules define the semantics of \texttt{monitorenter} and \texttt{monitorexit}. The two rules for \texttt{monitorenter} state that a thread can acquire a lock of an object if (1) the object is unlocked; or (2) it already owns the lock of that object. Observe that we have sufficient information to establish whether a thread can acquire the lock or wait for it. In particular, the special field \( \ell \) stores the number of times the object has been locked (by the same thread); whilst the field \( z \) in the thread collects the set of locks it owns. When the object \( o \) is unlocked, \texttt{monitorenter} sets the lock field to 1 and stores \( o \) in the component \( z \) of the thread. Note that, in this case, the rule is defined for configurations with \( z \) such that \( o \not\in z \). Alternatively, when the thread holds the object lock, \texttt{monitorenter} only increases the lock value.

The rules for \texttt{monitorexit} performs the reverse operations with respect to those of \texttt{monitorenter}.

The following proposition formalizes the property of mutual exclusion for critical sections.

**Proposition 1** Let \( Q \) be a JVMLC program and let \( F, T \) be a configuration reached along its execution. Then \( H(o, \ell) > 0 \) if and only if there is exactly one thread \( (i, f, s, z) \in T \) such that \( o \in z \).

## 5 The static semantics

In this section we develop the static semantics of JVMLC, according to the proposals in [17, 6].

Let \( \Sigma \) be the set of classes of our program \( Q \) in JVMLC and let \( \{ P_\sigma \mid \sigma \in \Sigma \} \) be the collection of bodies therein. We conclude that the program \( Q \) is well typed, in notation \( F, S, B \vdash Q \), if, for every \( \sigma \in \Sigma \), there exist \( F^\sigma \), \( S^\sigma \) and \( B^\sigma \) such that:

\[
F^\sigma, S^\sigma, B^\sigma \vdash P_\sigma.
\]

The partial maps \( F^\sigma \) and \( S^\sigma \) give the types of local variables, of the stack and of the set of locks when the program points at a given address. The partial map \( B^\sigma \) defines critical sections, that enclose all the instructions between the one following a \texttt{monitorenter} and the matching \texttt{monitorexit}.

Precisely, these maps have domain \texttt{ADDR} and codomain defined as follows \((i \in \texttt{ADDR})\):

1. \( F^\sigma[i] \) is a map from local variables to types at location \( i \);
2. \( S^\sigma[i] \) is a sequence of types of the operand stack at location \( i \);
3. \( B^\sigma[i] \) is a sequence \((a_1, a'_1, x_1) \ldots (a_k, a'_k, x_k)\), where \( a_i, a'_i \in \texttt{ADDR} \) and \( x_i \in \texttt{VAR} \). These sequences are well-formed, namely \([a_1 \ldots a'_k] \subseteq [a_i, a'_i, \ldots, a_i, a'_i]\) (extremes \( a_{i+1} \) and \( a'_{i+1} \) are not in \([a_1 \ldots a'_k]\)) and variables are pairwise different. We address to variables in \( B^\sigma[i] \) with \( \texttt{Var}[B^\sigma[i]] \), and we write \( j \in \texttt{Block}[B^\sigma[i]] \) if (a) \( B^\sigma[i] = \varepsilon \) or (b) \( B^\sigma[i] = (a, a', x) \cdot B' \) and \( j \notin [a, a'] \).

We let \( F_{\text{TOP}}^\sigma \) be the function that maps 0 to \( \sigma \) and all the other variables to \texttt{TOP}. The application of a partial map \( G \) to address \( i \) is often abbreviated into \( G_i \).

The rule that proves \( F^\sigma, S^\sigma, B^\sigma \vdash P_\sigma \) is:

\[
F_{\text{TOP}}^\sigma = F_{\text{TOP}}^\sigma \\
S^\sigma[i] = \varepsilon \\
B^\sigma[i] = \varepsilon \\
\forall i \in \text{dom}[P_\sigma], F^\sigma, S^\sigma, B^\sigma, i \vdash P_\sigma \\
\frac{}{F^\sigma, S^\sigma, B^\sigma \vdash P_\sigma}
\]

The top four premises regard the first instruction of the body \( P_\sigma \). They just set that a new thread starts with a value in the local variable 0 of type \( \sigma \) (see the semantics of \texttt{start} \( \sigma \)); whilst the stack and the set of acquired locks are empty. The lowest premise checks that every instruction is well typed. Figure 3 defines the rules for the judgment \( F^\sigma, S^\sigma, B^\sigma, i \vdash P_\sigma \) (in the figure we have always drop the index \( \sigma \)).

To be as much as possible conservative with respect to previous proposals, we have arranged premises of rules in such a way that those in the top conform with the premises of the corresponding rules in [17] and in [6]. The new premises concern the function \( B \). Among these rules, we discuss \texttt{IF} and \texttt{STORE}.
Figure 3. The static semantics of JVML

(Inc)
\[
P[i] = \text{inc} \\
F_{i+1} = F_i \\
S_{i+1} = \text{INT} \cdot \alpha \\
i + 1 \in \text{dom}[P] \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} \\
F,S,B,i \vdash P
\]

(Push)
\[
P[i] = \text{push} \theta \\
F_{i+1} = F_i \\
S_{i+1} = \text{INT} \cdot \theta \\
i + 1 \in \text{dom}[P] \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} \\
F,S,B,i \vdash P
\]

(Pop)
\[
P[i] = \text{pop} \\
F_{i+1} = F_i \\
S_i = \tau \cdot S_{i+1} \\
i + 1 \in \text{dom}[P] \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} \\
F,S,B,i \vdash P
\]

(if)
\[
P[i] = \text{if } L \\
F_{i+1} = F_L = F_i \\
S_i = \text{INT} \cdot S_{i+1} = \text{INT} \cdot S_L \\
i + 1, L \in \text{dom}[P] \\
\text{P}[L] \neq \text{monitorenter} \\
\text{P}[L] \neq \text{monitorexit} \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} = B_L \\
F,S,B,i \vdash P
\]

(load)
\[
P[i] = \text{load } x \\
x \in \text{dom}[F_i] \\
F_{i+1} = F_i \\
S_{i+1} = F_i[x] \cdot S_i \\
i + 1 \in \text{dom}[P] \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} \\
F,S,B,i \vdash P
\]

(store)
\[
P[i] = \text{store } x \\
x \in \text{dom}[F_i] \\
F_{i+1} = F_i[x \mapsto \tau] \\
S_i = \tau \cdot S_{i+1} \\
i + 1 \in \text{dom}[P] \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} \\
F,S,B,i \vdash P
\]

(new)
\[
P[i] = \text{new } \sigma \\
F_{i+1} = F_i \\
S_{i+1} = \sigma \cdot S_i \\
i + 1 \in \text{dom}[P] \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} \\
F,S,B,i \vdash P
\]

(putfield)
\[
P[i] = \text{putfield } \sigma.a: \tau \\
\sigma.a : \tau \\
F_{i+1} = F_i \\
S_i = \sigma \cdot S_{i+1} \\
i + 1 \in \text{dom}[P] \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} \\
F,S,B,i \vdash P
\]

(getfield)
\[
P[i] = \text{getfield } \sigma.a: \tau \\
\sigma.a : \tau \\
F_{i+1} = F_i \\
S_i = \sigma \cdot S_{i+1} \\
i + 1 \in \text{dom}[P] \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} \\
F,S,B,i \vdash P
\]

(return)
\[
P[i] = \text{return} \\
F,S,x,i \vdash P
\]

(start)
\[
P[i] = \text{start } \sigma \\
F_{i+1} = F_i \\
S_i = \sigma \cdot S_{i+1} \\
i + 1 \in \text{dom}[P] \\
i \in \text{Block}[B_i] \\
B_i = B_{i+1} \\
F,S,B,i \vdash P
\]

(monitorenter)
\[
P[i] = \text{monitorenter} \\
F_i = F_{i+1} \\
S_i = F_i[x] \cdot S_{i+1} \\
i + 1 \in \text{dom}[P] \\
\text{P}[i-1] = \text{load } x \\
x \notin \text{Var}[B_i] \\
B_{i+1} = (i+1,i',x) \cdot B_i \\
F,S,B,i \vdash P
\]

(monitorexit)
\[
P[i] = \text{monitorexit} \\
F_i = F_{i+1} \\
S_i = F_i[x] \cdot S_{i+1} \\
i + 1 \in \text{dom}[P] \\
\text{P}[i-1] = \text{load } x \\
B_{i+1} = (i',i,x) \cdot B_{i+1} \\
F,S,B,i \vdash P
\]
To type if \( L \) at \( i \), one must verify that both the instruction at \( i + 1 \) and that at \( L \) can be typed with the same values of \( F \), \( S \) and \( B \). This allows to abstract out of the branch that will be taken at run-time. Remark that \( B_i = B_{i+1} = B_L \) means that the instructions at \( i, i + 1 \) and \( L \) belong to the same critical section. Said otherwise, it is not possible to jump outside a critical section (because the top triple of \( B_L \) should have different addresses) or inside inner critical sections (because the lengths of \( B_{i+1} \) and \( B_L \) should be different). The premises \( P[L] \neq \text{monitorenter} \) and \( P[L] \neq \text{monitorexit} \) also disable jumps to \text{monitorenter} and \text{monitorexit} instructions. These instructions always come with a foregoing \text{load} \ instruction and these pairs have to be considered as a whole.

The rule \text{STORE} \ also verifies that the updated variable is free (premise \( x \not\in \text{Var}[B_i] \)), namely it has not been used to record a locked object. This is crucial to keep consistent \( B \) and for the correctness of our analysis.

The rules \text{START}, \text{MONITORENTER} and \text{MONITOREXIT} are new, and we examine them one by one. Rule \text{START} is straightforward because no static check is undertaken for the new thread, that is a run-time entity. Rules \text{MONITORENTER} as well as \text{MONITOREXIT} require that \text{monitorenter} and \text{monitorexit} are preceded by a \text{load} \ instruction, and the type \( F_i[x] \) matches the top of the stack \( S_i \). Moreover, \text{monitorenter} also checks that the variable \( x \) has not been used to record the object locked in an outer control section and sets the critical section for the next instruction (premise \( B_{i+1} = (i + 1, i', x) \cdot B_i \)). Remark that \( i' \) is not restrained: the constraint about \( i' \) is imposed by the matching \text{monitorexit} instruction.

### 5.1 A program and its typing

To discuss our type system we compute type informations for the instructions of the program in Figure 1. The resulting labeled program is illustrated in Figure 4. (We have specified the value of \( F \) for variables 0 and 1, since the other variables are always mapped to \text{TOP}.) The rightmost column defines the value of the function \( B \). Observe that \( B \) is not empty in the critical section only (instructions from 6 to 14) and, therein, \( B \) keeps the name of variable 1, to forbid possible updates. Remark that \( B[15] = \varepsilon \), this enforces the property that objects locked by the method have been properly unlocked on exiting.
5.2 The correctness of the verifier

To establish the correctness of the verifier in Figure 3 we define well typed configurations. To this aim, let \( H \) be a heap and let

\[
\mathcal{H}(H) = \{ o : \sigma \mid o \in \text{dom}[H] \text{ and } H(o) \text{ is a record value of type } \sigma \}
\]

Then we let \( \vdash_{\text{heap}} \) be the judgment defined by the following rules (\( \mathcal{H} \) is a set of assumptions on object names):

<table>
<thead>
<tr>
<th>( n ) integer</th>
<th>( \rho ) is a record value of type ( \sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{H} \vdash_{\text{heap}} n : \text{INT} )</td>
<td>( \mathcal{H} \vdash_{\text{heap}} \rho : \sigma )</td>
</tr>
<tr>
<td>( \mathcal{H} + o : \sigma \vdash_{\text{heap}} o : \sigma )</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{H} \vdash_{\text{heap}} o : [a_i : \tau_i \in 1..n] )</td>
<td>( \mathcal{H} \vdash_{\text{heap}} v : \tau )</td>
</tr>
</tbody>
</table>

**Definition 1** Let \( \models_H T \) be a configuration of a program \( Q \). We define \( F, S, B \models_H (T) \) if, for every \( (i, f, s, o) \in T \),

1. \( i \in \text{dom}[Q_o], \) for some \( \sigma \);
2. \( x \in \text{dom}[F^{f}_{i}] \) implies \( \mathcal{H}(H) \vdash_{\text{heap}} f(x) : F^{f}_{i}(x) \);
3. \( S^{u}_{i} = \tau_1 \cdots \tau_n \) implies \( s = v_1 \cdots v_n \cdot s' \) and \( \mathcal{H}(H) \vdash_{\text{heap}} v_i : \tau_i \), for every \( i \);
4. \( \{ f(x) \mid x \in \text{var}[B^{u}_{i}] \} \subseteq z \) and, for every \( x \in \text{var}[B^{u}_{i}] \), \( \text{card}([x' \mid f(x') = f(x) \text{and } x' \in \text{var}[B^{u}_{i}]]) \leq \mathcal{H}(f(x), o, l) \), where \( \text{card}() \) gives the cardinality of the argument.

The first three conditions of Definition 1 regard the existence of an instruction at \( i \) and the correctness of \( f \) and \( s \). The last condition deserves some discussion. First of all, note that condition 4 admits \( f(x_i) = f(x_j) \) with \( i \neq j \), which models the case of a thread that has acquired the lock of an object several times. It also admits that \( z \) is a super-set of the objects stored in \( \text{var}[B^{u}_{i}] \) and the actual lock value of an object \( o \) is an upper bound to the number of variables in \( \text{var}[B^{u}_{i}] \) that store \( o \). As concerns this paper, these two disequalities can be turned into equalities, because, in JVM\(_C\), the nesting of critical sections can be syntactically traced. However, as soon as method invocation is promoted in the language, the equality does not hold any more and the proof should be rewritten to generalize our results. See Section 6.2 for a discussion of this issue.

As an immediate consequence of the definition, we have a form of weakening for the judgment \( F, S, B \models_H (T) \).

**Lemma 1** Let \( F, S, B \models_H (T) \) and let \( H' \) be such that \( \mathcal{H}(H) \subseteq \mathcal{H}(H') \) and, for every \( (i, f, s, o) \in T \), \( o \in z \) implies \( \mathcal{H}(o, l) = \mathcal{H}'(o, l) \). Then \( F, S, B \models_H (T) \).

The following lemma establishes the invariance of the types of the objects in the heap.

**Lemma 2** Let \( Q \) be a JVM\(_C\) program, \( F, S, B \models Q \) and \( F, S, B \models_H (T) \) be a configuration reached along an execution of \( Q \). If \( \models_H T \rightarrow (T') \) then, for every \( o \in \text{dom}[H] \cap \text{dom}[H'], \mathcal{H}(H) \vdash_{\text{heap}} o : \sigma \) if and only if \( \mathcal{H}(H') \vdash_{\text{heap}} o : \sigma \).

We are ready to demonstrate the one-step soundness theorem, which guarantees that computation steps preserve types.

**Theorem 1** Let \( Q \) be a JVM\(_C\) program and \( F, S, B \models Q \). Let also \( \models_H T \) be a configuration reached along an execution of \( Q \). If \( F, S, B \models_H (T) \) and \( \models_H T \rightarrow (T') \) then \( F, S, B \models_H (T') \).

**Proof:** By induction on the number of applications of rule CONTEXT to infer \( \models_H T \rightarrow (T') \).
Basic case. The rule CONTEXT is not used; therefore $T$ consists of exactly one thread. By cases on the instruction to be executed. We always omit to check condition 1 of Definition 1 because trivial consequence of $F, S, B \vdash Q$.

Cases inc, push0, pop, load and if follow by the hypotheses $F, S, B \vdash (F_H \langle \delta, f, s, z \rangle)$ and the well typing of the whole program $Q$. We detail the instruction if, where the top of the stack is a non-null integer. In this case the rewriting is $F_H \langle \delta, f, n \cdot s, z \rangle \rightarrow F_H \langle L, f, s, z \rangle$ and we must verify that $F, S, B \vdash (F_H \langle L, f, s, z \rangle)$. Conditions 2 and 4 of Definition 1 are immediate consequence of $F, S, B \vdash (F_H \langle \delta, f, n \cdot s, z \rangle)$ and $F, S, B \vdash Q$. As regards condition 3, by $I_F$, $S_t'' = \text{INT} \cdot S_{t+1}$ and then use the same condition for $S_t''$.

Case store $x$. Let $T = \langle \delta, f, v_0 \cdot s, z \rangle$ and $T' = \langle i + 1, f[x \mapsto v_0], s, z \rangle$. By $F, S, B \vdash (F_H \langle \delta, f, s, z \rangle)$, we derive $S_t'' = \langle \tau_0, \tau_1 \cdots \tau_k, s, v_0 \ldots v_k \cdot s' \rangle$ and $(H(H) \vdash_{\text{heap}} v_i : \tau_i) \land \text{card}(x) \leq k$ (a). Therefore $(H(H) \vdash_{\text{heap}} v_i : \tau_i) \land \text{card}(x) \leq k$ which is condition 3 of Definition 1 for the configuration $F_H \langle \delta, f, s, z \rangle$.

By (a), we immediately derive condition 3. By (b), the previous instruction that has been performed is $\text{load } x$ (jumps to monitorexit are forbidden). Therefore $f(x) = o$. By $F, S, B \vdash (F_H \langle \delta, f, s, z \rangle)$, we derive:

- $S_t'' = F_t^\sigma [x] \cdot S_{t+1}''$ (a)
- $P[i - 1] \equiv \text{load } x$ (b)
- $x \notin \text{Var}(B_t''')$ (c)
- $\text{Var}(B_t''') = \text{Var}(B_t'''' \cup \{x\})$ (d)

By (b), the former instruction that has been executed is $\text{load } x$ (because jumps to monitorexit are forbidden). Therefore, by (a) and $F, S, B \vdash (F_H \langle \delta, f, o \cdot s, z \rangle)$, we derive condition 3. For the same reasons we also infer $f(x) = o$. Now, by $o \notin z$ and $\{f(x) | x \in \text{Var}(B_t'') \} \subseteq z$, that come from $F, S, B \vdash (F_H \langle \delta, f, o \cdot s, z \rangle)$, and (d) we conclude $\{f(x) | x \in \text{Var}(B_t'') \} \subseteq z \cup \{o\}$. The well typing of the initial state also gives that there is no $x$ in $\text{Var}(B_t'')$ such that $f(x) = o$. Henceforth card($\{x | f(x) = o$ and $x \in \text{Var}(B_t''')\}$) $\leq 1 = H(o, \ell)$. The case when the thread already holds the lock is similar and therefore omitted.

Case monitorexit. As before, there are two subcases that are similar. We only show the case when the thread releases the lock. Let $F_H \langle \delta, f, o \cdot s, z \sqcup \{o\} \rangle \rightarrow F_H' \langle i + 1, f, s, z \rangle$, where $H' = H(o, \ell) \rightarrow 0$. We check conditions 3 and 4 of Definition 1 for $F_H' \langle i + 1, f, s, z \rangle$. By rule MONITOREXIT we derive:

- $S_t'' = F_t^\sigma [x] \cdot S_{t+1}''$ (a)
- $P[i - 1] \equiv \text{load } x$ (b)
- $\text{Var}(B_t''') = \text{Var}(B_t'''' \setminus \{x\})$ (c)

By (a) we immediately derive condition 3. By (b), the previous instruction that has been performed is $\text{load } x$ (jumps to monitorexit are forbidden). Therefore $f(x) = o$. By $F, S, B \vdash (F_H \langle \delta, f, s, z \rangle)$, we derive:
Proof: \(\forall i, f, o \cdot s, z \in \{o\}\) we also derive that \(\text{card}\{x \mid f(x) = o \text{ and } x \in \text{Var}[B'_{\ell}]\} \leq 1 = H(o, \ell)\). Condition 4 follows by this property and (c).

Inductive case. Let \(\models_H T_1, T \rightarrow \models_H T_2, T, F, S, B \vdash (\models_H T_1, T)\) and \(F, S, B \vdash (\models_H T_2)\). To prove that \(F, S, B \vdash (\models_H T, T)\) we must verify that \(F, S, B \vdash (\models_H t)\). By hypothesis we derive \(F, S, B \vdash (\models_H t), \forall t \in T\). Now, observe that \(H^t = H^t_1 + H^t_2\), where \(\text{dom}[H^t_1] = \text{dom}[H^t_2]\) (the value of corresponding fields in \(H^t_1\) and \(H^t_2\) may be different). By Lemma 2, we obtain \(\mathcal{H}(H) = \mathcal{H}(H^t_1)\). Moreover, by Proposition 1, locks held by threads in \(T\) cannot be modified along \(\models_H T_1 \rightarrow \models_H T_2\). Therefore, we can conclude by Lemma 1.

\[\Box\]

5.3 Main properties

As a consequence of Theorem 1, we can derive some correctness properties of bytecodes that we collect in the following proposition.

Proposition 2 Let \(Q\) be a well-typed JVMLC program. Then

1. (Prevention of "current thread not-owner" exception) \(Q\) always releases locks that have been previously acquired.
2. (Objects are locked by active threads) Every method in \(Q\), upon termination, releases all the locks it acquires.
3. (Correct nesting of critical sections) Every method in \(Q\) always releases locks in the reverse ordering it has acquired them.

Proof: (Sketch) To verify the properties of Proposition 2, we refine the component \(z\) of threads into a stack of objects (where objects may also occur replicated). Let \(F, S, B \vdash Q\). The above properties clearly hold at the beginning of the execution. Then, assuming the properties hold at \((i, f, s, z)\) and \((i, f, s, z) \rightarrow (i', f', s', z')\), we conclude by means of Theorem 1.

Our static semantics does not cover a number of relevant properties. One for all, deadlock-freeness. Actually, the aim of the analysis in the present paper is to integrate the bytecode verifier and, consequently, the efficiency of the verification is a major issue. Efficiency can be hardly obtained with checks that require heavy model checking to be pursued on. In any case, the operational semantics in Section 4 is a ground basis for undertaking any analysis of concurrent bytecodes.

5.4 Type inference

In this section, we briefly describe the behaviour of our verifier. Its main purpose is to integrate the present Sun bytecode verifier (see [12] for an informal specification, and [17, 6] for formal ones).

The verifier TI:

1. The code is scanned method by method. While the procedure scans the code of a method, it builds the partial maps \(F\), \(S\) and \(B\). Entries of \(B\) are triples \((i, X, x)\), where \(i\) is the initial address of the critical section; \(X\) is a fresh variable that will be instantiated with the final address of the critical section; and \(x \in \text{Var}\) is the variable of the critical section. To each \((i, X, x)\) the procedure associates a set \(A_i\) of pairs \((a, a')\) of addresses that must belong to the critical section.
2. At each instruction \(i'\), TI makes the following steps. Let \(B_{\ell} = (i, X, x) \cdot B'\).
   a. if the instruction at \(i'\) is one of inc, pusho, pop, load, store, new, putfield, getfield, return, start, then it performs the checks stated in Figure 3 and builds the sets \(F\), \(S\) and \(B\) correspondingly. Moreover, every pair \((i, a)\) is removed from \(A\), and, for everyone of them, the verifier checks that \(F_{i'} = F_{i}, S_{i'} = S_{i + 1}, B_{i'} = B_i\), as well as that the instruction at \(i'\) is not a monitor enter or a monitor exit.
   b. if the instruction is if \(L\) then TI verifies that \(F_{\ell + 1} = F_{i'}, S_{\ell'} = \text{INT} \cdot S_{i' + 1}, i' + 1\) is a valid address, \(i' > i\) and \(B_{i'} = B_{i + 1}\). Moreover, if \(L < i'\) (backward branch) then it also verifies that \(F_{L} = F_{\ell}, S_{L} = S_{i + 1}, B_{L} = B_{i}\) and that the instruction at \(L\) is
We collect in this section our studies of extensions of JVML.

**6 Extensions**

We collect in this section our studies of extensions of JVML and its type system. The next subsection deals with three Java instructions for synchronization: wait, notify and notifyAll.

**6.1 The synchronization primitives**

Let JVML$^F$ be the language whose set of instruction is:

$$\text{Instruction} ::= \cdots \text{(as in section 3)} \mid \text{wait} \mid \text{notify} \mid \text{notifyAll}$$

The methods wait, notify and notifyAll are final in Java (their code cannot be modified) and they shorten the following invocations in the bytecode:

- `invokevirtual java/lang/Object/wait()`
- `invokevirtual java/lang/Object/notify()`
- `invokevirtual java/lang/Object/notifyAll()`

The execution of wait, notify and notifyAll succeed for threads that already hold the lock of the object $o$ on the stack. In this case, the `wait` instruction moves its own thread to the wait set of the object $o$ and the object is relinquished by performing as many unlock operations as the integer stored in the lock field of $o$. The instructions notify and notifyAll respectively wake up one thread and all the threads in the wait set of $o$. These threads are re-enabled for thread scheduling, which means competing for acquiring the lock of $o$. The winner will lock the object $o$ as many unmatched `monitorenter` it did on $o$ before the `wait-operation`.

To model the new operators we extend configurations as follows:

$$\Gamma \vdash H \vdash T \vdash \mathcal{W}$$

where $\mathcal{W}$ is a collection of indexed sets $W_o$, called wait sets. The set $W_o$, if not empty, contains indexed threads $\langle \nu', f, s, z \rangle_m$, where $n$ is a positive integer, representing the number of unmatched `monitorenter` instructions the thread $\langle \nu', f, s, z \rangle$ did before waiting. We denote the set $W_o$ with $\{\langle \nu'_k, f_k, s_k, z_k \rangle_m \mid k \in K \}$, with $\emptyset$, when it is empty (in this case we usually omit the wait set).

The semantics of JVML$^F$ is defined in Figure 2 and Figure 5. Note that there are two rules for `notify`: one when the corresponding wait set is not-empty and the other when it is empty.

---

4. Because `wait`, `notify` and `notifyAll` are final, they can be safely considered as bytecode instructions. This is not the case for `start`, which is not final and, therefore, can be modified by the code. For this reason we let start be the invocation of the start method of the class Thread.
The static semantics.

In the latter case no notification is done. The last rule of Figure 5 deals with notified threads

\[ Q[p(e)] = \text{notify} \]
\[ H(e) = n \quad (n > 0) \]

\[ \text{if } H(e) = n \quad (n > 0) \]

\[ \text{then } W_o \cup \{p(e) + 1, f(s, z) \cup \{o\}\}_{n+1} \]

\[ \text{and } W_o \cup \{(p(e)' + 1, f(s', z') \cup \{o\}\}_{n+1} \]

\[ H(e) = 0 \]
\[ H' = H(e) \]

\[ \text{if } H(e) = n \quad (n > 0) \]

In the latter case no notification is done. The last rule of Figure 5 deals with notified threads \( p(e), f(s, z) \cup \{o\} \). Remark also that the woken-up thread can continue when the object on top of the stack is unlocked. In this case, the index \( n \) is used to set the lock field.

To complete the operational semantics, it remains to generalize rule CONTEXT to deal with wait sets. Let \( W \) and \( W' \) be two collections of wait sets, and let \( \text{dom}[W] \) be the set of indexes of wait sets in \( W \). When \( \text{dom}[W] \cap \text{dom}[W'] = \emptyset \), we define \( (T; W) + (T'; W') \) to be \( T, T' ; W, W' \). Therefore the new context rule becomes:

\[ (\text{CONTEXT}) \]
\[ \text{if } H(e) = n \quad (n > 0) \]

\[ W_o \cup \{p(e), f(s, z) \cup \{o\}\}_n \]

\[ \text{and } W_o \cup \{p(e)', f(s', z') \cup \{o\}\}_n \]

The static semantics. Most of the checks about the instructions \text{wait}, \text{notify} and \text{notifyAll} are performed by the JVM at run-time. The Java-compiler only verifies that this instructions are invoked with stacks containing an object whose type is not primitive (in our case, primitive types are integers). Therefore, the extension of the analyzer of Figure 3 is almost straightforward. The three inference rules for \text{wait}, \text{notify} and \text{notifyAll} have been collected in Figure 6 below.

In order to establish the correctness of the extension in Figure 6, we patch Definition 1 as follows:

\[ \text{Definition 2 Let } F, S, B \vdash (\text{Context } T; W) \text{ if, for every } (i, f, s, z) \in T, (i, f, s, z) \in W, \text{ with } i \in \text{dom}[P]\sigma, \text{ the conditions 1, 2, 3 and 4 of Definition 1 hold.} \]

We conclude by generalizing Theorem 1 to JVM\(_L_\). The proof has been omitted because it rely on the one for Theorem 1, for the instructions that are in JVM\(_L_\), plus three cases for \text{wait},
notify and `notifyAll` that are straightforward.

**Theorem 3** Let \( Q \) be a set of bodies in \( \text{JVML}_{C}^{m} \), and let \( F, S \) and \( B \) such that \( F^\sigma, S^\sigma, B^\sigma \vdash P_\sigma \), for every body \( P_\sigma \) in \( Q \). Let also \( \vdash_{H'} T' ; W \) be a configuration reached along an execution of \( Q \). If \( F, S, B \vdash (\langle F_{H} T ; W \rangle ; W) \) and \( \vdash_{H} T ; W \rightarrow \vdash_{H'} T' ; W' \) then \( F, S, B \vdash (\langle F'_{H'} T' \rangle ; W') \).

Remark that the type system of Figure 6 does not enforce any check over methods that perform a `wait`, `notify` or `notifyAll` without owning a lock. In other words, the property 1 of Proposition 2 is false for this extension. The reason is that, in Java, the instructions `wait`, `notify` or `notifyAll` may well occur inside not synchronized statements (or methods). For example, imagine of a method \( m \) with a `wait` instruction that will wait on the lock owned by the caller. If no lock is owned by the caller, an `IllegalMonitorStateException` will be raised at runtime. To ban these kind of errors, one should check that every invocation of \( m \) falls inside a critical section, that should definitely tangle our type system.

### 6.2 Method invocation

The JVM provides four instructions to model method invocation, according to the called method is public (instruction `invokevirtual`) or is implemented by an interface (instruction `invokespecial`) or is private (instruction `invokeinterface`) or is static (instruction `invokestatic`). We focus here on the instruction `invokevirtual` (the other ones may be dealt with in similar ways) and, for simplicity, but without loss of generality, we consider void methods with no argument.

Consider the language \( \text{JVML}_{C}^{m} \), that extends \( \text{JVML}_{C} \) with the instruction

\[
\text{Instruction} ::= \cdots \text{(as in sections 3 and 6.1)} \\
| \text{invokevirtual } \sigma.m
\]

where \( \sigma.m \) is a method descriptor, namely a pair (class-name, method-name) that uniquely identifies a method body. (In general, descriptors also contain method types.) In this section, we assume that the initial address of a method \( \sigma.m \) is \( l_{\sigma.m} \).

The formal definition of `invokevirtual` amounts to introduce the concept of frame, that is used to store data and partial results. More precisely, a frame is a tuple \( \langle p.e, f, s \rangle \), where \( p.e \) is the address of the instruction to be executed, \( f \) is the map of local variables and \( s \) is the stack. A thread is now a pair \( \langle r, z \rangle \), where \( r \) is a nonempty sequence of frames and \( z \) is the set of locked objects. The intended meaning is that the control is owned by the initial frame of \( r \). It ceases to own the control either when it invokes another method or when it returns. In the former case, a new tuple is added at the beginning of \( r \); in the latter case the first frame is removed and the control is given back to the second frame. Notice that all the frames share the same set of locks. Indeed, in the JVM, locks are per-thread, rather than per-method.

The rules defining the operational semantics of \( \text{JVML}_{C}^{m} \) are those of Figures 2 and 5, patched with frames, plus those in Figure 7. Observe that the instruction `invokevirtual` uses the object

---

**Figure 6. The static semantics of `wait`, `notify` and `notifyAll`.**

\[
\begin{array}{lll}
\text{(WAIT)} & \text{(NOTIFY)} & \text{(NOTIFYALL)} \\
F[i] = \text{wait} & F[i] = \text{notify} & F[i] = \text{notifyAll} \\
B_i = B_{i+1} & B_i = B_{i+1} & B_i = B_{i+1} \\
\end{array}
\]

\[
\begin{array}{l}
S_i = \sigma \cdot S_{i+1} \\
i + 1 \in \text{dom}[P] \\
\end{array}
\]
Figure 7. The operational semantics of JVML

\[ P[p] = \text{invokevirtual } \sigma.m \]

\[ \Gamma_H \langle (p', f, o \cdot s) \cdot \rho, z \rangle \rightarrow \Gamma_H \langle (l_{\sigma.m}, f_0 \downarrow o], \varepsilon) \cdot (p', f, o \cdot s) \cdot \rho, z \rangle \]

\[ P[p] = \text{return} \]

\[ \Gamma_H \langle (p', f, s) \cdot (p', f', o' \cdot s') \cdot \rho, z \rangle \rightarrow \Gamma_H \langle (p' + 1, f', s') \cdot \rho, z \rangle \]

Figure 8. The static semantics of invokevirtual.

\[(\text{INVOKEVIRTUAL})\]

\[ P[i] = \text{invokevirtual } \sigma.m \]

\[ F_i = F_{i+1} \]

\[ S_i = \sigma \cdot S_{i+1} \]

\[ i + 1 \in \text{dom}[P], \quad 1_{\sigma.m} \in \text{dom}[Q] \]

\[ B_i = B_{i+1} \]

\[ F, S, B, i \vdash P \]

on top of the stack of the caller to initialize the variable 0 of the new frame. This is the self-object of the invoked method. Observe also that return is unspecified when the length of the sequence of frames is 1.

The static semantics. Since the bytecode verifier checks a program method by method, the analysis of invokevirtual does not require any involved test. The inference rule in Figure 8 only verifies that the type of the object on top of the stack matches with the class of the invoked method. (We let \( Q \) be the whole program.)

More significative, the definition of well-typed configuration must now encompass frame sequences that are finite, but not limited. (These sequences model recursive methods.) As a consequence, the nesting of critical sections cannot be syntactically traced anymore.

To detail this issue, let \( \Gamma_H \langle (i_1, f_1, s_1) \cdots (i_n, f_n, s_n), z \rangle \) be a (one-thread) configuration. Observe that the locked objects in \( z \) are exactly all those objects that have been acquired (but not released) at the instructions \( i_1, \cdots, i_n \). Similarly, the lock value of an object is the sum of the times it has been acquired (but not released) at the above instructions. Therefore, to infer whether a configuration is well-typed or not, it suffices to verify Definition 1 on every frame. This leads to the following definition.

Definition 3 Let \( \Gamma_H \vdash T; W \) be a configuration of a program \( Q \). We define \( F, S, B \vdash (\Gamma_H \vdash T; W) \) if, for every \( (\rho, z) \in T, \rho, z \) in a wait set of \( W \), with \( i \in \text{dom}[P_{\sigma}], \) the conditions 1, 2, 3 and 4 of Definition 1 hold for every \( (i, f, s) \) in the sequence \( \rho \).

Remark that the soundness of Definition 3 strongly relies on the disequalities of item 4 in Definition 1, which establish that each frame has locked a subset of objects acquired by the thread.

Theorem 3 holds in JVML\(_m\). We omit the statement, because similar, as well as the proof.

6.3 Exception handlers

When an exception occurs during the execution of a program, the JVM looks for an handler in the corresponding exception table. The exception table is a sequence of tuples \((\text{from}, \text{to}, \text{target}, \text{type})\), where the first three components are addresses and the last is the type of the exception. So, if the
exception occurred at \( j \), the JVM looks for the first tuple \((i, i', i'', \text{type})\) in the exception table such that \( i \leq j \leq i' \) and the type of the exception fits with \( \text{type} \). In this case the control is given to the instruction at \( i'' \), otherwise the exception is rethrown to the caller. If no handler is found for that exception, the program terminates (in an abrupt way).

To model the type of exceptions, we augment the set of object types \( T \) with the type \( \text{THROWABLE} \). For simplicity, we drop the last component of entries of exception tables (the \( \text{type} \) component), assuming that the type is always \( \text{THROWABLE} \).

We also assume that a method body is now a pair \((P; E)\), where \( E \) is the exception table, namely a partial map from \( \text{dom}[P] \) to \( \text{dom}[P] \times \text{dom}[P] \times \text{dom}[P] \). If \( E[p] = (i, i', k) \), then the tuple \((i, i', k)\) is the first one in \( E \) such that \( i \leq p \leq i' \). By extension, \( E \) will also denote the function that is the union of the exception tables in every method of the program \( Q \). Finally, exceptions are supposed to be thrown only by instructions \text{athrow}.

Let \( \text{JVML}_Q \) be the following extension of \( \text{JVML}_W \):

\[
\text{Instruction} ::= \cdots \text{as in section 6.2) | \text{athrow}}
\]

The operational semantics of \text{athrow} is defined in Figure 9. Observe that, when an exception is thrown, the JVM looks for the first frame which has an entry in the exception table. If such a frame is found, the control is given to the corresponding handler that begins with a stack containing the exception. All the previous frames are deleted.

**Static semantics.** To type exception tables, we begin by refining the notion of well-typed program (see also [7] for a similar development). Let \( \Sigma \) be the set of classes of the program \( Q \) and let \( \{(P_\sigma; E_\sigma) \mid \sigma \in \Sigma\} \) be the collection of bodies therein. The program \( Q \) is well typed, in notation \( F, S, B \vdash Q \), if, for every \( \sigma \in \Sigma \), there exist \( F^\sigma, S^\sigma \) and \( B^\sigma \) such that:

\[
F^\sigma, S^\sigma, B^\sigma \vdash (P_\sigma; E_\sigma).
\]

and for every \((i, i', k) \in E[\text{dom}[P]], F^\sigma, S^\sigma, B^\sigma \vdash (i, i', k)\) handles \( P_\sigma \). This last judgment is defined by the rule:

\[
\begin{align*}
  j, j', k &\in \text{dom}[P_\sigma] \\
  S_k &\equiv \text{THROWABLE} \\
  B_j &\equiv B_k \\
  \text{dom}[F_k] &\equiv \text{Var}[B_j] \\
  \forall x \in \text{Var}[B_j], F_j[x] &\equiv F_k[x] \\
  \forall x \notin \text{Var}[B_j], F_j[x] &\equiv \text{TOP}
\end{align*}
\]

The above rule is more restrictive with respect to rule (\text{wht handler}) in [7]. The reason is twofold. Firstly, we have a rudimentary form of subtyping polymorphism, namely \text{TOP} is the type of every value. Variables that are not used by the handler are given type \text{TOP}, rather than a suitable supertype. Secondly, it is not possible to relinquish variables that are protected when the exception is thrown because they will be used by the handler to unlock the right objects. For this reason we also impose the constraint \( B_j = B_k \).
Figure 10. The static semantics of `athrow` and (the refinement of) `MONITORENTER`.

\[
\begin{align*}
\text{Rule `ATHROW`} & \quad P[i] = \text{athrow} \\
\text{S}_i &= \text{THROWABLE} \cdot S' \\
B_i &= E \Rightarrow \exists i \in \text{dom}[E] \\
F, S, B, i & \vdash (P; E)
\end{align*}
\]

(MONITORENTER-exc)

(as the premises of MONITORENTER)

\[
\begin{align*}
B_{i+1} &= (i + 1, i', x) \cdot B_i \\
E[i + 1] &= (i + 1, i' - 1, k) \\
F, S, B, i & \vdash (P; E)
\end{align*}
\]

Figure 10 defines the rule for the static correctness of `athrow` and a refinement of `MONITORENTER`. Rule `ATHROW` checks that the object on top of the stack is of type `THROWABLE` and the existence of an handler if the instruction `athrow` occurs inside a critical section. Rule `MONITORENTER-exc` adds one constraint to `MONITORENTER` affirming that a suitable entry of the exception table must exist to handle exceptions occurring in the critical section. Notice that, the well-typing of exception tables and rule `MONITORENTER-exc` guarantee that every critical section must be paired with an exception handler. This handler is typed with the same block information of the critical section. Therefore, by rules `RETURN` and `ATHROW`, the handler may safely return (either normally or abruptly) provided its code releases every lock in the block information.

Theorem 3 can also be proved for JVML. The details are omitted. This extension allows us to conclude with the following property (that yields Property 3 in Section 2).

**Proposition 3** Let Q be a well-typed JVML program. If exceptions raised along the execution of Q are due to `athrow` instructions then Q always releases every lock that has acquired.

7 Related works and concluding remarks

As remarked in several places, our work is strongly based on the framework developed in a series of papers by Stata-Abadi [17] and Freund-Mitchell [6, 7, 8], with the admitted aim of covering most of the static analysis problems of JVML. Other approaches to bytecode verification, that don’t cover concurrency issues, are based on data flow analysis [9], typed assembly languages [14] and the Haskell type checker [19].

As regards the bytecode, a very detailed semantics can be found in Bertelsen’s works [2]. However Bertelsen does not address the semantics of multi-threading, as well as that of `MONITOREXIT` and `MONITORENTER` (in his work these instructions have been regarded with the same semantics of `POP`). Another formal semantics of a sublanguage of JVML has been independently defined by Qian [15]. Also Qian misses the concurrent fragment.

Moving away from the bytecode, other works that share the same approach about static analysis concern the Java language [3, 13, 18]. To be fair, we admit that most of the problems addressed in this paper disappear in the high-level language. Because the synchronization statement explicitly defines the critical section and the locked object. Therefore, the integration of the previous works with the concurrent primitives should not be difficult. Remark that this comment does not weaken at all the results of the present paper. What makes the Java language a distinguished programming language is that its bytecode may be transmitted across different machines. A security layer—the bytecode verifier—is needed to safeguard machines from executing hostile bytecodes.
There are two kinds of extensions that have not been considered yet.

The first one concerns the integration of our verifier with other features of the JVM, in the same style of Section 6. Among the others, subroutines seem problematic because they require a form of polymorphism on local variables that are not used therein. We are confident that methods already developed in [17, 8] should be easily integrated inside our verifier.

The second kind of extensions concerns behavioural properties (i.e. safety and liveness properties, see Chapters 2 and 3 of [11]). To this respect, the studies for detecting race conditions among threads in [4, 5] and those about deadlock freeness [10] provide a source of inspiration because they strongly rely on type systems. In any case, the formal model defined here should be a ground basis for every verifier aiming at checking concurrent properties.

Finally, we have started the implementation of the algorithm in Section 5.4. Then we will test a bunch of bytecodes resulting from the Sun Java compiler and the IBM Jikes compiler. This will provide a further argument on the plausibility of the restrictions imposed by our type system (Properties 1, 2 and 3 of Section 2).

References


