A Type System for JVM Threads

Cosimo Laneve [a,1]

[a] *Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy.* laneve@cs.unibo.it

Abstract

The current definition of the Java bytecode verifier, as well as the proposals to formalize it, does not include any check about the structured use of locks by `monitorenter` and `monitorexit` instructions. So code is run, even if critical sections are 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 lock and unlock operations. The verifier integrates well with previous formalizations of the Java bytecode verifier. Our analysis of structured use of locks reveals the presence of bugs in the current compilers from Sun, IBM and Microsoft.

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 during the migration. To detect possible harmful bytecodes, Java developers have defined the *bytecode verifier*, which checks the format of downloaded classes, the

[1] This work is partially supported by the Microsoft Research Grant "Network Aware Programming and Interoperability" and by the Italian MIUR Research Program NAPOLI.

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 [17,7,9]. 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 formal specifications 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 address the first of them.

At the present time, the bytecode verifiers do not filter out the following codes:

```
1 aload_0                    1 aload_0
2 monitorexit                2 monitorenter
3 return                     3 return
```

where, the one on the left releases a lock (`monitorexit` instruction) without having acquired it before, whilst the one on the right acquires the lock (`monitorenter` instruction) without releasing it. It turns out that the execution of the above codes raises the run-time exception `IllegalMonitorStateException` (Sun's JDK 1.2 raises no exception for the bytecode on the right). That is, the JVM performs a number of run-time checks to guarantee proper locks usages by method invocations. In particular, these checks verify that exactly the objects which are locked are released upon termination—the *structured locking property*.

Extending the bytecode verifier with static checks of structured locking is not straightforward because:

- locks may be acquired and released by using different objects, which are aliases;
- the order of lock releases may be different from the order of lock acquirings;
- instructions performed in between a `monitorenter` and a `monitorexit` may be not contiguous in the bytecode;
- the normal execution may be deviated while a lock is owned because an exception occurs.

We address the problem of enforcing the structured locking property by designing a suitable type system which integrates the bytecode verifier, thus following 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. In particular, we demonstrate that a well-typed program never raises an `IllegalMonitorStateException` at run-time. As a consequence, the robustness of the JVM may be improved by integrating our checks in the bytecode verifier itself. In addition, JVM speed-up is increased because tests which nowadays are performed at run-time may be omitted. Finally, our analysis reveals the presence of bugs in the current compilers from Sun, IBM and Microsoft.

The structure of the paper is the following. Section 2 overviews concurrency in the JVM and in the bytecode, and gives a detailed account of the compilation of Java `synchronized` statement. Sections 3 and 4 define the syntax and the operational model of JVML$_C$, the sublanguage of JVML with primitives for thread creation and mutual exclusion. The static semantics is defined in Section 5. In Section 6 we discuss the extension of JVML$_C$ with exception handlers. The extension with synchronization primitives (`wait`, `notify`, and `notifyAll`) is discussed in Section 7. In Section 8 we analyze the bugs we have found in Sun, IBM, and Microsoft compilers. In Section 9 we discuss recipes for conforming bytecodes to the structured locking property. In Section 10 we present the current version of our prototype extension of the bytecode verifier with structured locking checks. We comment related works and conclude in Section 11.

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 an instance of a class deriving from `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 implementation of `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. Two forms of synchronization are provided: through *synchronized methods* and through *synchronized statements*.

If a thread invokes a synchronized method m of an object o, the invocation locks o and releases the lock when the method returns. In between, other

```
public void onlyMe(Foo f) {
  synchronized (f)
    { doSomething(); }
}
```

```
Method void onlyMe(Foo)
  0  aload_1
  1  astore_2
  2  aload_2
  3  monitorenter
  4  aload_0
  5  invokevirtual doSomething()
  8  aload_2
  9  monitorexit
  10 return
  11 aload_2
  12 monitorexit
  13 athrow
 Exception table:
  from  to  target  type
   4    8    11     any
```

Fig. 1. The method `onlyMe` and its sample bytecode of [12], section 7.14

invocations of m by other threads will be blocked, as well as invocations of other synchronized methods on o. Synchronized methods are implemented by the `ACC_SYNCHRONIZED` flag in the constant pool [12] [2]. 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 those cases where parts of the method bodies need to be synchronized. Its semantics is to execute the body `S`—the *critical section*—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 sample compilation of `synchronized` in [12], section 7.14, is recalled in Figure 1. There, the method `onlyMe` acquires the lock of the argument `f`, does something on the self object, then releases the lock and terminates. The JVM begins its execution in the state

$$\langle 0, \varepsilon, [0 \mapsto o,\ 1 \mapsto \mathit{foo}], \emptyset \rangle$$

where the first item is the program counter—the initial value is 0—; the second item is the stack—initially the stack is empty—; the third item are the local variables—at the beginning the variable 0 contains the self object o and 1 contains the actual parameter *foo* of type *Foo*—; the last item represents the set of locks owned by the thread—this set is empty in the initial state. Instructions

[2] The constant pool is a run-time representation of classes and interfaces which contains constants and field references that must be solved at run-time.

0-1 of the bytecode copy the argument of `onlyMe` in the variable `2`. This allows the object that is going to be locked to be recorded in a "safe" place. Therefore we obtain the state

$$\langle 2, \varepsilon, [0 \mapsto o,\, 1 \mapsto foo, 2 \mapsto foo], \emptyset\rangle$$

The next two instructions attempt to acquire the lock of the argument. Instruction 2 copies the value of the variable 2 on the stack; the `monitorenter` instruction at 3 is executed provided the object on the stack is unlocked. Once the `monitorenter` terminates the JVM state becomes

$$\langle 2, \varepsilon, [0 \mapsto o,\, 1 \mapsto foo, 2 \mapsto foo], \{foo\}\rangle$$

where the set of locks is not empty (the thread possesses the lock of the object *foo*). Then, the self object is loaded on the stack, and the control is given to the method `doSomething` (instructions 4-5). When `doSomething` returns, the lock is released: at 8 the reference in the protected variable is copied on the stack, and the `monitorexit` at 9 takes this copy to relinquish the lock. In the section from 4 to 9 no other thread may acquire the lock of *foo*. These lines correspond to the body of `synchronized` in the source code. `onlyMe` returns at 10.

Lines from 11 to 13 seem out of place, since they don't correspond to any operation in the source code. These instructions catch exceptions that occur at instructions 4 and 5, as prescribed by the exception table. (Line 8 is not covered by the exception table, [12], section 7.12.) They release the lock and re-bounce the exception to the caller (instruction `athrow`). Instructions 11–13 are added by compilers because the JVM specification is very demanding as regards the implementation of `synchronized` statements. In [12], section 3.11.11, we find the following statement:

> 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 case whether the method invocation completes normally or abruptly [3].

However, while instruction 5 must be covered by the exception table because the invocation of `doSomething` may complete abruptly, it is unclear why the exception table also covers instruction 4, which by no means can rise an exception. In facts, there are two types of exceptions in the JVM ([12], section 2.16.1):

[3] A method invocation completes "abruptly" when its body causes a JVM-exception that is not handled within the method.

(1) *synchronous exceptions*, generated by the evaluation of the code;
(2) *asynchronous exceptions*, generated "asynchronously" with respect to the evaluation of the code. There are two kinds of asynchronous exceptions:
    (a) `Thread.stop` invocation which gives a `java.lang.ThreadDeathException`. (`Thread.stop` has been deprecated and will be removed in a future JDK.)
    (b) internal error in the JVM implementation, raising a `java.lang.InternalErrorException`. It is raised, for instance, in `interpreter.c`, if the signature of a constructor is wrong, or in `string_util.c`, if there is no converter from a C-native string to Java, or *vice versa*.

Therefore, the exception table covers line 4 in Figure 1 in case an asynchronous exception occurs there.

Actually, the scheme described in Figure 1 is faulty, as it is possible for an asynchronous exception to occur between the `aload_2` instruction at 8 and the following `monitorexit`, in which case the monitor may be left in a locked state. (A similar remark concerns instructions 11 and 12.) As a matter of facts, none of the present compilers from Sun, IBM, and Microsoft fulfil the above statement (see Section 8).

The JVM specification is not so demanding as concerns JVM implementations. Indeed, JVM implementations may, but this is not required, enforce the following two rules, guaranteeing *structured locking* ([12], section 8.13). Let t be a thread and ℓ be a lock. Then:

(1) The numbers of locks and unlocks performed by t on ℓ during a method invocation must be equal, whether the method invocation completes normally or abruptly.
(2) At no point during a method invocation, the number of unlocks performed by t on ℓ may exceed the number of locks performed by t on ℓ.

Needless to say, a JVM without the structured locking may be exposed to runtime errors such as `IllegalMonitorState` exceptions and deadlocks, which are notoriously hard to diagnose.

In this paper we integrate the Java Bytecode Verifier with static checks of structured locking. This integration is implemented by a type system, in the same style as [17] and [7]. Our main result states that well-typed programs never violate the structured locking property.

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 run-time 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 [17,7] regards locks. To keep track of them, we supplement the type system with the multiset of object types locked by the thread. The type system verifies that every instruction can be properly typed with the right lock information.

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 7. 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$_C$

The language JVML$_C$ is a restriction of JVML that includes basic constructs and instructions for concurrency. In JVML$_C$, a program is a collection Σ of class declarations:

$$\begin{array}{l}\texttt{class C } \{ \\ \quad \texttt{super: Thread} \\ \quad \texttt{fields: } F \\ \quad \texttt{method run ()} \\ \quad\ \ P^C \\ \} \end{array}$$

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 $a \in \text{FID}$ is an identifier, and τ is an integer INT or an object type $\sigma \in T$. Bodies P^C are finite sequences of instructions, namely partial maps from *addresses* $\{n_1, \cdots, n_2\}$ in ADDR, for some $n_1 \leq n_2$, to instructions. JVML$_C$ has the following instructions:

$$\begin{array}{lll} \mathit{Instruction} & ::= & \texttt{inc} \mid \texttt{pop} \mid \texttt{push0} \mid \texttt{load}\ x \mid \texttt{store}\ x \mid \texttt{if}\ L \\ & & \mid \texttt{new}\ \tau \mid \texttt{putfield}\ \sigma.a\ \tau \mid \texttt{getfield}\ \sigma.a\ \tau \\ & & \mid \texttt{start}\ \sigma \mid \texttt{monitorenter} \mid \texttt{monitorexit} \\ & & \mid \texttt{return} \end{array}$$

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` respectively pops and pushes the integer 0 on the stack; `if` L pops the top value off the stack and either falls through when that value is the integer 0 or jumps to the address L otherwise;

- `new` τ allocates a new object of type τ, 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` σ, `monitorenter`, and `monitorexit` are the concurrency 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. [4] In this paper, we let σ be one of the classes of the program (namely $\sigma \in \Sigma$); in JVML, σ must be a subclass of `Thread`. 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 [7] and for classes with other methods see [8]. We assume that initialization is performed *at the same time* as object creation by Java default initializers that put 0 in every integer field and `null` in every object field.

In the first part of the paper, for simplicity, we don't consider exception handlers. We discuss in Section 6 the extension dealing with exception handlers.

4 The operational semantics

The bytecode interpreter for JVML_C is defined using the same framework as in [17,7]. We briefly review the framework before defining our concurrent model.

[4] 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 artifice to define the semantics of `start`: in 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 2).

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 former. (Addresses will be considered integers in the type system of Section 5.) Our programs P are collections of classes σ. We write $\mathsf{dom}[P]$ for the set of addresses of P, $P[i]$ being the i-th instruction in P, if $i \in \mathsf{dom}[P]$. We write 1_σ to identify the first instruction of the method `run` of the class σ. We assume that programs P are well-formed: two successive addresses in $\mathsf{dom}[P]$ belong to the same method body and jumps between different method bodies are not allowed.

Partial maps are used to represent most of our entities (heaps, memory functions, etc.). If f is a 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 ε denotes the empty map (the map undefined everywhere).

Types τ are TOP, integers INT, object types T and indexed object types $\widehat{T}$. Object types, ranged over by σ, include all the class names of the program. Indexed object types, ranged over by $\widehat{\sigma}$, are defined as follows:

$$\widehat{T} = \{\sigma_i \mid \sigma \in T \text{ and } i \in \text{ADDR}\}$$

The type σ_i is used for typing variables whose value of type σ has been copied at line i. Namely, all the variables that have the same type σ_i are aliases. Indexed object types do not occur in the syntax of section 3.

To conform with previous formalizations of Stata-Abadi and Freund-Mitchell [17,7], in this paper types are not ordered with a subtyping relation. As a consequence, the constraints in the type system of Figure 3, as well as those in [17,7], reject too many programs. On the contrary, our verifier prototype [4], relaying on subtypes, uses weaker constraints. We refer to section 10 for a thorough discussion of this issue.

We define an indexing operation over types τ, in order to mark types when variables are copied. Let τ_i, where $i \in$ ADDR, be the following:

(1) $\tau_i = \tau$, when $\tau =$ INT or $\tau =$ TOP (*only copies of references are relevant*);
(2) $\tau_i = \sigma_i$, if $\tau = \sigma$ (*the first copy of the variable changes the type*);
(3) $\tau_i = \widehat{\sigma}$, if $\tau = \widehat{\sigma}$ (*successive copies of a variable keep the type of the first copy*).

We also let *Ind* and *Type* be two partial functions from types to ADDR and types, respectively, defined as follows: $\mathit{Ind}[\sigma_i] = i$, *Ind* is undefined otherwise; $\mathit{Type}[\sigma_i] = \sigma$, and $\mathit{Type}[\tau] = \tau$ otherwise.

For every σ, we assume a countable set of *object names*, ranged over $o, o', \cdots$. Let $\mathcal{O}$ be the set of all object names. Values v are integer constants or object names. The type TOP includes all the values. The values of types σ and σ_i, for every i, are the same. As usual, we write $v : \tau$ if v is a value of τ.

Finally, for each type σ, we define $\overline{\sigma} = [a_j : \tau_j{}^{j \in J}]$, namely the record of fields that are specified in the corresponding class definition. We address fields with the usual dot notation; therefore, $[a : \tau, a_j : \tau_j{}^{j \in J}].a : \tau$. *Record values* are records $[a_j = v_j{}^{j \in J}]$, where v_j are values.

4.2 The operational model

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

$$\Vdash_H \langle pc_1, f_1, s_1, z_1 \rangle, \cdots, \langle pc_n, f_n, s_n, z_n \rangle$$

with the following meaning:

- H is the *heap*, namely a partial function whose domain is the set $\mathcal{O}$ and whose range is the set of record values. The special field ℓ, $\ell \notin$ FID, represents the *lock* associated with the object. (The instructions `getfield` and `putfield` cannot read/write on the locks.) We assume that record values always have a field ℓ. The notation $H(o).a$ is used to retrieve the value of the field a in $H(o)$; and the notation $H(o).a \mapsto v$ is used to update the field a of o to the value v.
- each tuple $\langle pc_i, f_i, s_i, z_i \rangle$ is a *thread*; pc_i is the address of the instruction to be executed; f_i is a total map from the set VAR of local variables to the set of values; s_i is a stack of values; z_i is a finite subset of $\mathcal{O}$ and represents the set of objects locked by the thread.

Let σ be the type of the class whose `run` is invoked. The machine begins its execution in the state $\Vdash_{H_0} \langle 1_\sigma, f_0[0 \mapsto o], \varepsilon, \emptyset \rangle$, where

- $H_0 = [o \mapsto \rho_\sigma]$, where o is the self object of the method `run` and ρ_σ is the record value of type $\overline{\sigma}$ with fields initialized to 0 and null, according to they are integers or objects. The special field ℓ is initialized to 0.
- f_0 maps the local variables to any values.

The rules that define the operational semantics of JVML$_C$ are shown in Figure 2. In this figure we let P be a set of class bodies, each class body being identified by a different set of addresses. Every rule in Figure 2 actually mentions the components that participate in the rewriting. Of course the rewriting applies to every configuration that contains the components. More explic-

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

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

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

$$\frac{P[pc] = \texttt{load}\ x}{P \Vdash_H \langle pc, f, s, z\rangle \rightarrow \Vdash_H \langle pc+1, f, f(x) \cdot s, z\rangle}$$

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

$$\frac{\begin{array}{c}P[pc] = \texttt{new}\ \sigma\\ o' \notin \mathsf{dom}[H]\\ H' = H[o' \mapsto \rho_\sigma]\end{array}}{P \Vdash_H \langle pc, f, s, z\rangle \rightarrow \Vdash_{H'} \langle pc+1, f, o' \cdot s, z\rangle}$$

$$\frac{\begin{array}{c}P[pc] = \texttt{putfield}\ \sigma.a\ \tau\\ H' = H(o).a \mapsto v\end{array}}{P \Vdash_H \langle pc, f, v \cdot o \cdot s, z\rangle \rightarrow \Vdash_{H'} \langle pc+1, f, s, z\rangle} \qquad \frac{\begin{array}{c}P[pc] = \texttt{getfield}\ \sigma.a\ \tau\\ H(o).a = v\end{array}}{P \Vdash_H \langle pc, f, o \cdot s, z\rangle \rightarrow \Vdash_H \langle pc+1, f, v \cdot s, z\rangle}$$

$$\frac{\begin{array}{c}P[pc] = \texttt{start}\ \sigma\\ o \in \mathsf{dom}[H]\end{array}}{P \Vdash_H \langle pc, f, o \cdot s, z\rangle \rightarrow \Vdash_H \langle pc+1, f, s, z\rangle, \langle 1_\sigma, f_0[0 \mapsto o], \varepsilon, \emptyset\rangle}$$

$$\frac{\begin{array}{c}P[pc] = \texttt{monitorenter}\\ H(o).\ell = 0\\ H' = H(o).\ell \mapsto 1\end{array}}{\begin{array}{l}P \Vdash_H \langle pc, f, o \cdot s, z \setminus \{o\}\rangle \rightarrow\\ \qquad \Vdash_{H'} \langle pc+1, f, s, z \cup \{o\}\rangle\end{array}} \qquad \frac{\begin{array}{c}P[pc] = \texttt{monitorenter}\\ H(o).\ell = n \ \ (n > 0)\\ H' = H(o).\ell \mapsto n+1\end{array}}{\begin{array}{l}P \Vdash_H \langle pc, f, o \cdot s, z \cup \{o\}\rangle \rightarrow\\ \qquad \Vdash_{H'} \langle pc+1, f, s, z \cup \{o\}\rangle\end{array}}$$

$$\frac{\begin{array}{c}P[pc] = \texttt{monitorexit}\\ H(o).\ell = 1\\ H' = H(o).\ell \mapsto 0\end{array}}{\begin{array}{l}P \Vdash_H \langle pc, f, o \cdot s, z \uplus \{o\}\rangle \rightarrow\\ \qquad \Vdash_{H'} \langle pc+1, f, s, z\rangle\end{array}} \qquad \frac{\begin{array}{c}P[pc] = \texttt{monitorexit}\\ H(o).\ell = n \ \ (n > 1)\\ H' = H(o).\ell \mapsto n-1\end{array}}{\begin{array}{l}P \Vdash_H \langle pc, f, o \cdot s, z \cup \{o\}\rangle \rightarrow\\ \qquad \Vdash_{H'} \langle pc+1, f, s, z \cup \{o\}\rangle\end{array}}$$

Fig. 2. The operational semantics of JVML_C

itly, let $\mathit{Lock}[\mathcal{T}]$ be the collection of objects locked by threads in $\mathcal{T}$, namely $\mathit{Lock}[\mathcal{T}] = \bigcup_{\langle pc,f,s,z\rangle \in \mathcal{T}} z$. Then, for every rewriting rule of Figure 2, we use the rule:

$$\text{(Context)}\quad \frac{P \Vdash_H \mathcal{T}_1 \rightarrow \Vdash_{H'} \mathcal{T}_2 \qquad (\mathit{Lock}[\mathcal{T}_1] \cup \mathit{Lock}[\mathcal{T}_2]) \cap \mathit{Lock}[\mathcal{T}] = \emptyset}{P \Vdash_H \mathcal{T}_1, \mathcal{T} \rightarrow \Vdash_{H'} \mathcal{T}_2, \mathcal{T}}$$

where $\mathcal{T}$, $\mathcal{T}_1$ and $\mathcal{T}_2$ are sets of threads. The side-condition $(\mathit{Lock}[\mathcal{T}_1] \cup \mathit{Lock}[\mathcal{T}_2]) \cap \mathit{Lock}[\mathcal{T}] = \emptyset$ bans moves of configurations where two threads lock the same object.

The first eight rules of Figure 2 are already discussed in [17,7]. Among them, we briefly comment the two rules for the `if` L instruction and the rule for the object creation. When $P[pc] = \texttt{if}\ L$, the virtual machine pops the value on top of the stack and checks whether it is 0 or not. In the first case no branch is performed and the program counter is updated to $pc + 1$. In the last case, the execution continues from line L. When $P[pc] = \texttt{new}\ \sigma$, a fresh object name o' is taken ($o' \notin \mathsf{dom}[H]$). In practice this means that a new memory slot is taken from the heap to allocate records of type σ. In addition to the standard semantics of new, our rule also initializes the object o', by assigning a default record ρ_σ to it.

Rules for putfield and getfield respectively update and retrieve object fields stored in the heap. In case of putfield, the stack is shaped as follows: the value to be stored is on top of it, and the object name is right below.

The last five rules are new and peculiar of our contribution. The rule modelling the instruction `start` σ creates a new thread of control. The new thread begins into a state where the first instruction of the class σ must be executed, and with the variable 0 containing the object on top of the stack. In particular, none of the locks held by the caller thread are retained by the new thread.

The other rules define the semantics of `monitorenter` and `monitorexit`. The two rules for `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 ℓ 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, `monitorenter` sets the lock field to 1 and stores o in the component z of the thread. In this case, the rule is defined for configurations with z such that $o \notin z$. Alternatively, when the thread holds the object lock, `monitorenter` only increases the lock value. In the two rewriting rules, the argument of `monitorenter` is never used.

The rules for `monitorexit` performs the reverse operations with respect to those of `monitorenter`.

The operational semantics of Figure 2 closely mimicks the behaviour of the JVM. Other descriptions are also possible, such as turning z into a multiset and eliminating two rules (one monitorenter and one monitorexit) and the special field ℓ. However, the resulting semantics should depart from the real one, and the formal correspondence should then be established.

We conclude the section with the formal definition of erroneus configurations we want to avoid.

Definition 1 *A* structured locking failure *for a program P is one of the following configurations:*

- $\langle pc, f, s, z \cup \{o\}\rangle$ *and* $P[pc] = \texttt{return}$ *(normal completion);*
- $\langle pc, f, s, z \cup \{o\}\rangle$ *and* $pc \notin \mathsf{dom}[P]$ *(abrupt completion);*
- $\langle pc, f, o \cdot s, z\rangle$*,* $P[pc] = \texttt{monitorexit}$ *and* $o \notin z$*.*

We observe that a structured locking failure raises a `java.lang.IllegalMonitor-StateException`, as discussed in the Introduction.

5 The static semantics

Let Σ be the set of classes of our program P in JVML_C and let $\{P^\sigma \mid \sigma \in \Sigma\}$ be the collection of bodies therein. We conclude that the program P is *well-typed* in $F = \{F^\sigma \mid \sigma \in \Sigma\}$, $S = \{S^\sigma \mid \sigma \in \Sigma\}$, and $Z = \{Z^\sigma \mid \sigma \in \Sigma\}$, in notation $F, S, Z \vdash P$, if for every $\sigma \in \Sigma$:

$$F^\sigma, S^\sigma, Z^\sigma \vdash P^\sigma \ .$$

The partial maps F^σ, S^σ and Z^σ give the types of local variables, of the stack and of the set of locks when the program points at a given address. Precisely, these maps have domain $\textsc{Addr}$ and codomain defined as follows ($i \in \textsc{Addr}$):

(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) $Z^\sigma[i]$ is a multiset of indexed object types.

We let F^σ_{TOP} be the function that maps 0 to σ and all the other variables to $\textsc{top}$. The application of a partial map G to address i is often abbreviated into G_i.

$$\text{(Inc)}\quad \frac{\begin{array}{c} P[i] = \texttt{inc} \\ F_i = F_{i+1} \\ S_i = \text{INT} \cdot \alpha = S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Push)}\quad \frac{\begin{array}{c} P[i] = \texttt{push0} \\ F_i = F_{i+1} \\ \text{INT} \cdot S_i = S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Pop)}\quad \frac{\begin{array}{c} P[i] = \texttt{pop} \\ F_i = F_{i+1} \\ S_i = \tau \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(If)}\quad \frac{\begin{array}{c} P[i] = \texttt{if } L \\ F_i = F_{i+1} \\ F_L = F_{i+1} \\ S_i = \text{INT} \cdot S_{i+1} = \text{INT} \cdot S_L \\ i+1, L \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} = Z_L \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Load)}\quad \frac{\begin{array}{c} P[i] = \texttt{load } x \\ x \in \mathsf{dom}[F_i] \\ F_i[x] = \tau \\ F_i[x \mapsto \tau_i] = F_{i+1} \\ \tau_i \cdot S_i = S_{i+1} \\ \forall \widehat{\sigma} \in S_i.\ i \neq \mathit{Ind}[\widehat{\sigma}] \\ \forall y \in \mathsf{dom}[F_i].\ i \neq \mathit{Ind}[F_i[y]] \\ \forall \widehat{\sigma} \in Z_i.\ i \neq \mathit{Ind}[\widehat{\sigma}] \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Store)}\quad \frac{\begin{array}{c} P[i] = \texttt{store } x \\ x \in \mathsf{dom}[F_i] \\ F_i[x \mapsto \tau] = F_{i+1} \\ S_i = \tau \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(New)}\quad \frac{\begin{array}{c} P[i] = \texttt{new } \sigma \\ F_i = F_{i+1} \\ \sigma \cdot S_i = S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Putfield)}\quad \frac{\begin{array}{c} P[i] = \texttt{putfield } \sigma.a\ \tau \\ \overline{\sigma}.a : \tau \\ F_i = F_{i+1} \\ S_i = \nu \cdot \nu' \cdot S_{i+1} \\ \tau = \mathit{Type}[\nu] \\ \sigma = \mathit{Type}[\nu'] \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Getfield)}\quad \frac{\begin{array}{c} P[i] = \texttt{getfield } \sigma.a\ \tau \\ \overline{\sigma}.a : \tau \\ F_i = F_{i+1} \\ S_i = \nu \cdot S'_i \\ \sigma = \mathit{Type}[\nu] \\ \tau \cdot S'_i = S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Return)}\quad \frac{\begin{array}{c} P[i] = \texttt{return} \\ Z_i = \emptyset \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Start)}\quad \frac{\begin{array}{c} P[i] = \texttt{start } \sigma \\ F_i = F_{i+1} \\ S_i = \tau \cdot S_{i+1} \\ \sigma = \mathit{Type}[\tau] \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Monitorenter)}\quad \frac{\begin{array}{c} P[i] = \texttt{monitorenter} \\ F_i = F_{i+1} \\ S_i = \widehat{\sigma} \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_{i+1} = Z_i \uplus \{\widehat{\sigma}\} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(Monitorexit)}\quad \frac{\begin{array}{c} P[i] = \texttt{monitorexit} \\ F_i = F_{i+1} \\ S_i = \widehat{\sigma} \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ \widehat{\sigma} \in Z_i \\ Z_{i+1} = Z_i \setminus \{\widehat{\sigma}\} \end{array}}{F, S, Z, i \vdash P}$$

Fig. 3. The static semantics of JVML_C

The rule that proves $F^\sigma, S^\sigma, Z^\sigma \vdash P^\sigma$ is:

$$\frac{\begin{array}{c} F^\sigma[1_\sigma] = F^\sigma_{\mathrm{TOP}} \\ S^\sigma[1_\sigma] = \varepsilon \\ Z^\sigma[1_\sigma] = \varepsilon \\ \forall i \in \mathsf{dom}[P^\sigma].\ F^\sigma, S^\sigma, Z^\sigma, i \vdash P^\sigma \end{array}}{F^\sigma, S^\sigma, Z^\sigma \vdash P^\sigma}$$

The top three premises concern the first instruction of the body P^σ. They just say that a new thread starts with a value in the local variable 0 of type σ (see the semantics of `start` σ); 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 $F^\sigma, S^\sigma, Z^\sigma, i \vdash P^\sigma$ (in the figure we always drop the index σ).

To be as compatible as possible with 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 [7]. The new premises mostly concern the function Z. Among these rules, we discuss IF and STORE.

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 Z. This allows to abstract out of the branch that will be taken at run-time. Remark that $Z_i = Z_{i+1} = Z_L$ means that the instructions at i, $i+1$ and L lock a same multiset of objects. Said otherwise, it is not possible to jump outside a critical section (because the multiset Z_L should be smaller) or inside an inner critical section (because Z_L should be greater).

The rule STORE verifies that the type stack at $i+1$ is the one at i without the topmost type. This type is recorded into $F_{i+1}[x]$. STORE also verifies that Z_{i+1} is equal to Z_i.

The rules LOAD, START, MONITORENTER, and MONITOREXIT are new, and they deserve a detailed discussion. We begin with rule START. This rule should verify that σ is actually a subclass of `Thread`. This check is omitted because σ is always in Σ, which by definition are subclasses of `Thread`. As regards the new thread, no static check is undertaken because it is a run-time entity.

Rules LOAD, MONITORENTER, and MONITOREXIT are discussed all together. We begin by recalling the corresponding rule LOAD in [17,7], and providing

```
new T
store x
new T
store y
load x
monitorenter
load y
monitorexit
```

Fig. 4. An erroneous code

two tentative rules for MONITORENTER and MONITOREXIT:

$$
\text{(Load-[17,7])}\quad
\frac{\begin{array}{c} P[i] = \texttt{load}\ x \\ x \in \mathsf{dom}[F_i] \\ F_i = F_{i+1} \\ \tau \cdot S_i = S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ (Z_i = Z_{i+1}) \end{array}}{F, S, Z, i \vdash P}
\qquad
\text{(Monitorenter-attp)}\quad
\frac{\begin{array}{c} P[i] = \texttt{monitorenter} \\ F_i = F_{i+1} \\ S_i = \sigma \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_{i+1} = Z_i \uplus \{\sigma\} \end{array}}{F, S, Z, i \vdash P}
\qquad
\text{(Monitorexit-attp)}\quad
\frac{\begin{array}{c} P[i] = \texttt{monitorexit} \\ F_i = F_{i+1} \\ S_i = \sigma \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ \sigma \in Z_i \\ Z_{i+1} = Z_i \setminus \{\sigma\} \end{array}}{F, S, Z, i \vdash P}
$$

where the lowest premise of LOAD-[17,7] has been written in parentheses because it misses in [17,7] since they don't deal with locks. Notice that, according to LOAD-[17,7], to verify `load` x at i, the top element of the type stack S_{i+1} must be the type of the variable x. Rules MONITORENTER-ATTP as well as MONITOREXIT-ATTP require that the type stack have an object type σ at the top. According to MONITORENTER-ATTP, σ is added to Z_i, which keeps track of the locked objects, as well as the number of times a same object has been locked. Rule MONITOREXIT-ATTP requires that the object type on top of S_i belongs to the multiset Z_i. In this case, this element is removed (which is the static counterpart of lock release).

However, consider the code in Figure 4. This code performs a `monitorenter` and a corresponding `monitorexit` on two objects of the same class, but *different*. The reader is invited to check that rules LOAD-[17,7], MONITORENTER-ATTP, and MONITOREXIT-ATTP type correctly the above erroneous code.

At this stage, one could think to assign a different type to objects which are created by the method, therefore altering the typing rule of `new` as follows—

see [7] for a similar rule:

$$
\textsc{(New-Attp)}\quad
\frac{\begin{array}{c}
P[i] = \mathtt{new}\ \sigma \\
F_i = F_{i+1} \\
\widehat{\sigma_i} \cdot S_i = S_{i+1} \\
\forall \widehat{\sigma'} \in S_i.\ i \neq \mathit{Ind}[\widehat{\sigma'}] \\
\forall y \in \mathsf{dom}[F_i].\ i \neq \mathit{Ind}[F_i[y]] \\
i+1 \in \mathsf{dom}[P] \\
(\forall \widehat{\sigma'} \in Z_i.\ i \neq \mathit{Ind}[\widehat{\sigma'}]) \\
(Z_i = Z_{i+1})
\end{array}}{F, S, Z, i \vdash P}
$$

Notice that, new objects of type σ are tagged with the address of the instruction creating them. To ensure unicity of the type, we check that the type σ_i does not already occur in the stack S_i—this is the premise $\forall \widehat{\sigma'} \in S_i.\ i \neq \mathit{Ind}[\widehat{\sigma'}]$—, in the local variables F_i—this is the premise $\forall y \in \mathsf{dom}[F_i].\ i \neq \mathit{Ind}[F_i[y]]$—, and in the objects in Z_i—this is the premise $\forall \widehat{\sigma'} \in Z_i.\ i \neq \mathit{Ind}[\widehat{\sigma'}]$. The lowest two premises have been written in parentheses because they do not appear in [7].

This solution is also flawed, because the two objects could be arguments of the method invocation, rather than created in the body. Indeed, consider a method taking two objects of the same class as arguments. The code

```
1   load 0
2   monitorenter
3   load 1
4   monitorexit
```

is clearly wrong because the `monitorexit` releases an object which has not been locked. Nevertheless the typing system with New-Attp, Load-[17,7], Monitorenter-attp, and Monitorexit-attp do not reject such code.

Notice that instruction `1` *copies* the content of the variable `0` on the stack. Therefore, at line `2`, a `monitorenter` is performed on *this copy*. Since we want to enforce a `monitorexit` on a copy of the same variable `0`, we should trace the *copying mechanism* in order to reject the above program pattern. To this end, the typing rule of the `load` instruction should tag object types, as well. This yields the rule Load in Figure 3.

With respect to rules Monitorenter-attp and Monitorexit-attp, the rules Monitorenter and Monitorexit in Figure 3 also enforce locked objects to have indexed types.

i	$P[i]$	$F_i[0]$	$F_i[1]$	S_i	Z_i
1	`load` 0	Sample	TOP	ϵ	$\emptyset$
2	`store` 1	$\mathtt{Sample}_1$	TOP	$\mathtt{Sample}_1$	$\emptyset$
3	`load` 1	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	ϵ	$\emptyset$
4	`monitorenter`	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	$\emptyset$
5	`load` 0	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	ϵ	$\{\mathtt{Sample}_1\}$
6	`push0`	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	$\{\mathtt{Sample}_1\}$
7	`putfield Sample.val` INT	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	$\mathrm{INT} \cdot \mathtt{Sample}_1$	$\{\mathtt{Sample}_1\}$
8	`load` 1	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	ϵ	$\{\mathtt{Sample}_1\}$
9	`monitorexit`	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	$\{\mathtt{Sample}_1\}$
10	`return`	$\mathtt{Sample}_1$	$\mathtt{Sample}_1$	ϵ	$\emptyset$

Fig. 5. A program and its static typing

To conclude, we discuss the inutility of the tags introduced by NEW-ATTP. To this aim, consider rule NEW in Figure 3, and remark that the code of Figure 4 is refused by the typing system of Figure 3. Indeed, a new object reference *needs to be copied before being locked*, because the copy will be used later on to release the object. For instance, a code as

```
1   new T
2   monitorenter
```

is going to fail *in any case*, regardless of the tag it has, because the reference created at line `1` has not been saved—*i.e.* copied somewhere. In JVML_C, the copying operation may be performed by storing the reference in a local variable, and then loading (thus *copying*) the variable on the stack [5]. Therefore, since `load` is going to introduce tagged types, it suffices to rely on these tags to check the structured locking property.

5.1 Sample programs and their typings

To illustrate merits and demerits of our type system we show the typing for sample programs.

In Figure 5 we analyze a bytecode similar to the body of the method `onlyMe` in Figure 1. They differ for two reasons: (1) there is no reference stored in the variable 1; (2) the invocation to `doSomething` has been replaced by an updating of the field `val` of the object stored in the variable 0.

[5] In JVML there is further operation making copies: `dup`. We discuss this instruction in Section 10.

i	$P[i]$	$F_i[1]$	$F_i[2]$	$F_i[3]$	$F_i[4]$	S_i	Z_i
1	`load 2`	`Object`	`Object`	`boolean`	TOP	ε	$\emptyset$
2	`if 6`	`Object`	`Object`	`boolean`	TOP	`boolean`	$\emptyset$
3	`load 0`	`Object`	`Object`	`boolean`	TOP	ε	$\emptyset$
4	`store 3`	$\texttt{Object}_3$	`Object`	`boolean`	TOP	$\texttt{Object}_3$	$\emptyset$
5	`goto 8`	$\texttt{Object}_3$	`Object`	`boolean`	$\texttt{Object}_3$	ε	$\emptyset$
6	`load 1`	`Object`	`Object`	`boolean`	TOP	ε	$\emptyset$
7	`store 3`	`Object`	$\texttt{Object}_6$	`boolean`	TOP	$\texttt{Object}_6$	$\emptyset$
8	`load 3`						
9	`monitorenter`						
10	`load 3`						
11	`monitorexit`						
12	`return`						

Fig. 6. A type system failure

In Figure 5, we have specified the value of F for the variables 0 and 1, since the other variables are always mapped to TOP. The rightmost column defines the value of the function Z. Observe that Z is not empty in the critical section only (instructions from 5 to 9) and, therein, Z keeps the type of variable 1, to forbid possible updates. Remark that $Z_{10} = \emptyset$, this enforces the property that objects locked by the method have been properly unlocked on exiting.

Next, consider the following method `m`, which takes two objects `x` and `y`, and a boolean `z`. It locks `x` or `y`, according to `z` is true or false, respectively.

```
public void m(Object x, Object y, boolean z) {
  Object w ;
  if (z) w = x; else w = y;
  synchronized(w) { } ;
}
```

In Figure 6, we illustrate the bytecode, together with a partial type assignment to the instructions (we use boolean types and the `goto L` instruction, which corresponds to `push0 ; inc ; if L`). We omit $F_i[0]$ because it is always equal to `Object`. The type assignment stops at line `8` because, at this line, the two branches of the conditional have mismatching types in $F_8[1]$, $F_8[2]$, and $F_8[4]$. Nevertheless, there is a standard solution to such kind of failures, which relies on subtypes. We discuss in detail this issue in Section 10.

The type system of Figure 3 does not type those bytecodes which enter two nested critical sections by using the same `monitorenter` instruction. This bytecode, together with a partial type assignment, is illustrated in Figure 7. In this case, the failure is due to the mismatch between types at line `3`, which marks the `monitorenter` with the type of the variable `0`, and line `8`, which attempts to mark the `monitorenter` with the type of the variable `1`. We notice

i	$P[i]$	$F_i[0]$	$F_i[1]$	S_i	Z_i
1	push0	Object	Object	ε	$\emptyset$
2	load 0	Object	Object	INT	$\emptyset$
3	monitorenter	$\texttt{Object}_2$	Object	$\texttt{Object}_2 \cdot \text{INT}$	$\emptyset$
4	if 9	$\texttt{Object}_2$	Object	INT	$\{\texttt{Object}_2\}$
5	push 0	$\texttt{Object}_2$	Object	ε	$\{\texttt{Object}_2\}$
6	inc	$\texttt{Object}_2$	Object	INT	$\{\texttt{Object}_2\}$
7	load 1	$\texttt{Object}_2$	Object	INT	$\{\texttt{Object}_2\}$
8	goto 3	$\texttt{Object}_2$	$\texttt{Object}_7$	$\texttt{Object}_7 \cdot \text{INT}$	$\{\texttt{Object}_2\}$
9	load 1				
10	monitorexit				
11	load 0				
12	monitorexit				
13	return				

Fig. 7. An erroneous bytecode

that considering erroneous such a bytecode is not a big deal, since it has no counterpart in the Java programming language. Indeed, the bytecode of Figure 7 should correspond to entering a synchronized statement and, within its body, jumping to the same synchronized statement. Nevertheless, a bytecode as in Figure 7 may outcome from optimizations of just-in-time compilers. We discuss this issue in Section 9.

Another bytecode which is considered erroneous is one that

(1) creates an alias with a type having the index of a locked object—therefore using *the same* `load` instruction;
(2) locks the alias created in (1).

In this case the (indexed) types of two different objects become equal, and the type system could no more guarantee the releasing of the right lock on exiting the critical section. Such bytecodes have no counterpart in Java, too, because they require that nested synchronizations are performed with the same synchronized statement of outer ones.

5.2 The soundness property

Our result is the following soundness property:

Theorem 2 *For all P, F, S and Z such that $F, S, Z \vdash P$, and for all classes σ in P, if $P \Vdash_{H_0} \langle 1_\sigma, f_0[o \mapsto \rho_\sigma], \varepsilon, \emptyset\rangle \rightarrow^* \Vdash_H \mathcal{T}$ and no exception has occurred during the computation, then $\mathcal{T}$ does not contain a structured locking failure.*

We do not prove this theorem because it is a consequence of Theorem 3, in the next section. The technique is standard. Firstly the notion of well-typing is extended to configurations. Then, one proves that the typing is an invariant of the operational semantics (subject reduction). Finally, it is shown that well-typed configurations have no structured locking failure.

6 Exceptions and Exception Handlers

When an exception occurs during the execution of a program, the JVM removes from the stack all the elements except the topmost one (the exception object), and looks for an handler in the corresponding exception table. The exception table is a sequence of tuples ($\mathtt{from}, \mathtt{to}, \mathtt{target}, \mathtt{type}$), where the first three components are addresses and the last one is the type of the exception. So, if the exception occurred at j, the JVM looks for the first tuple $(i, i', i'', \mathtt{type})$ in the exception table such that $i \leq j < i'$ and the type of the exception matches `type`. (To be remarked: the clause is inclusive on the "from" end and is exclusive on the "to" end—see [12], section 7.12.) If a tuple $(i, i', i'', \mathtt{type})$ is found, 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).

The exception table may not meet the structured locking property in Section 2. Indeed, this property requires that critical sections have a corresponding exception handler, which releases every lock before returning, or when the exception is rethrown to the caller (these are the cases of normal and abrupt completion in Definition 1). Therefore, the verifier of Figure 3 must be extended with suitable checks, as detailed below.

We begin with augmenting the set of object types T with the type THROWABLE of exceptions and, for simplicity, we assume that exceptions have only type THROWABLE. In the operational semantics, exception tables E are represented by total functions from $\mathsf{dom}[P]$ to ADDR, rather than sequences of tuples. When $E[pc] = k$ and an exception occurs at pc, the control transfers to k. Abrupt completions are modelled by choosing k such that $k \notin \mathsf{dom}[P]$.

Programs P are now collections of method bodies of the form $(P^\sigma \,;\, E^\sigma)$, where E^σ is the *exception table*, with the constraint that if $E^\sigma[pc] = k$ and $k \in \mathsf{dom}[P]$ then $k \in \mathsf{dom}[P^\sigma]$ (exception handlers are part of the method body).

As discussed in Section 2, there are two types of exceptions: synchronous and asynchronous. To explicitly model synchronous exceptions, we extend the language JVML_C with the instruction `athrow`. Let JVML_C^e be the following

$$\frac{P[pc] = \texttt{athrow} \quad E[pc] = k}{P \Vdash_H \langle pc, f, o \cdot s, z\rangle \;\to\; \Vdash_H \langle k, f, o, z\rangle} \qquad \frac{o \notin \mathsf{dom}[H] \quad H' = H[o \mapsto \rho_{\text{THROWABLE}}] \quad E[pc] = k}{P \Vdash_H \langle pc, f, s, z\rangle \;\to\; \Vdash_{H'} \langle k, f, o, z\rangle}$$

Fig. 8. The operational semantics of synchronous and asynchronous exceptions

extension of JVML_C:

$$\begin{aligned} \textit{Instruction} ::= &\cdots \text{(as in section 3)} \\ &\mid \texttt{athrow} \end{aligned}$$

The instruction `athrow` gives the control to the exception handler, once the stack is emptied, except for the topmost object. If no exception handler exists, the method completes abruptly. Asynchronous exceptions are not modelled by explicit instructions. Rather, they will be simulated in the operational semantics by a suitable (nondeterministic) rule.

6.1 The operational semantics

The operational semantics of synchronous and asynchronous exceptions are defined in Figure 8. The rule on the right simulates asynchronous exceptions, that can occur at any moment. It says that any configuration $\langle pc, f, s, z\rangle$ may depart from the standard behaviour described in Figures 2 and 8(left)—because an asynchronous exception has occurred—, and branches to its own exception handler, with a stack containing a fresh object of type THROWABLE. Of course, this rule has no counterpart in the JVM implementations.

6.2 The static semantics

We begin by refining the notion of well-typed program (see also [8] for a similar development). Let Σ be the set of classes of the program P and let $\{(P^\sigma; E^\sigma) \mid \sigma \in \Sigma\}$ be the collection of bodies therein. The program P is well-typed in $F = \{F^\sigma \mid \sigma \in \Sigma\}$, $S = \{S^\sigma \mid \sigma \in \Sigma\}$, and $Z = \{Z^\sigma \mid \sigma \in \Sigma\}$, in notation $F, S, Z \vdash P$, if for every $\sigma \in \Sigma$:

$$F^\sigma, S^\sigma, Z^\sigma \vdash P^\sigma$$

and for every $(i, k) \in E^\sigma$,

$$F^\sigma, S^\sigma, Z^\sigma \vdash (i, k) \text{ handles } P^\sigma.$$

$$\frac{\begin{array}{c}\text{(Exc-Handler)}\\ k \in \mathsf{dom}[P] \;\Rightarrow\; \begin{cases} S_k = \text{THROWABLE} \\ \forall x.\, F_k[x] \in \{F_i[x], \text{TOP}\} \\ Z_i = Z_k \end{cases} \\ Z_i \neq \emptyset \;\Rightarrow\; k \in \mathsf{dom}[P]\end{array}}{F, S, Z \vdash (i,k) \text{ handles } P} \qquad \frac{\begin{array}{c}\text{(Athrow)}\\ P[i] = \mathtt{athrow} \\ S_i^\sigma = \text{THROWABLE} \cdot S'\end{array}}{F, S, Z, i \vdash P}$$

Fig. 9. The static semantics of exceptions

This last judgment is defined by the rule Exc-Handler in Figure 9 (superscripts σ have been omitted). Rule Exc-Handler constraints the type of the first instruction of exception handlers, when they belong to $\mathsf{dom}[P]$. In particular, when $(i, k) \in E$, types of variables at k may be TOP or the same as the corresponding variable in F_i. Also, we impose the constraint $Z_i = Z_k$.

Rule Exc-Handler is more restrictive with respect to rule (*wt handler*) in [8]. The reason is twofold. Firstly, we have a rudimentary form of subtyping polymorphism, namely TOP is the type of every value. In the exception handler, local variables may be given type TOP, rather than a suitable super-type. Secondly, we enforce the presence of exception handlers for instructions in critical sections.

Figure 9 also defines the rule for the static correctness of `athrow`. Rule Athrow checks that the object on top of the stack is of type THROWABLE. The check for the existence of an handler when `athrow` occurs inside a critical section is demanded to the rule Exc-Handler.

6.3 A program and its typing

To illustrate our extensions, we supplement the sample bytecode in Figure 5 with an exception handler. The resulting program and its typing is shown in Figure 10. With respect to the program in Figure 5, there are three more lines (11, 12, and 13) and an exception table. The lines 11, 12, and 13 define the exception handler in case an (asynchronous) exception occurs at lines [5..10[(see the first line of the exception table). In this case, the variable 1, storing the lock, is loaded on the stack (line 11), the lock is released (line 12), and the exception is thrown to the caller (line 13). We remark that rule Exc-Handler holds for instruction in [5..10[. However, according to this rule, every instruction i with a not empty Z_i should have a suitable exception handler. In particular, also instructions at lines 11 and 12. The second entry in the exception table catches exactly exceptions occurring at these lines. So, for instance, in case a (second, asynchronous) exception occurs at line 11, the lock

i	$P[i]$	$F_i[0]$	$F_i[1]$	S_i	Z_i
1	`load` 0	`Sample`	TOP	ϵ	$\emptyset$
2	`store` 1	$\texttt{Sample}_1$	TOP	$\texttt{Sample}_1$	$\emptyset$
3	`load` 1	$\texttt{Sample}_1$	$\texttt{Sample}_1$	ϵ	$\emptyset$
4	`monitorenter`	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\emptyset$
5	`load` 0	$\texttt{Sample}_1$	$\texttt{Sample}_1$	ϵ	$\{\texttt{Sample}_1\}$
6	`push0`	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\{\texttt{Sample}_1\}$
7	`putfield` `Sample.val` INT	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\text{INT} \cdot \texttt{Sample}_1$	$\{\texttt{Sample}_1\}$
8	`load` 1	$\texttt{Sample}_1$	$\texttt{Sample}_1$	ϵ	$\{\texttt{Sample}_1\}$
9	`monitorexit`	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\texttt{Sample}_1$	$\{\texttt{Sample}_1\}$
10	`return`	$\texttt{Sample}_1$	$\texttt{Sample}_1$	ϵ	$\emptyset$
11	`load` 1	TOP	$\texttt{Sample}_1$	THROWABLE	$\{\texttt{Sample}_1\}$
12	`monitorexit`	TOP	$\texttt{Sample}_1$	$\texttt{Sample}_1 \cdot \text{THROWABLE}$	$\{\texttt{Sample}_1\}$
13	`athrow`	TOP	$\texttt{Sample}_1$	THROWABLE	$\emptyset$

```
Exception table
from    to    target      type
   5    10        11  throwable
  11    13        11  throwable
```

Fig. 10. A program and its static typing

must still be released before completing. This amounts to performing the same sequence of operations as we were at lines [5..10[. For this reason the handlers of lines [11..13[and [5..10[are the same (the handler is self-protecting: see Section 8).

6.4 The soundness property

We may conclude with the main soundness property, which generalizes Theorem 2:

Theorem 3 *For all P, F, S and Z such that $F, S, Z \vdash P$, and for all classes σ in P, if $P \Vdash_{H_0} \langle 1_\sigma, f_0[o \mapsto \rho_\sigma], \varepsilon, \emptyset\rangle \to^* \Vdash_H \mathcal{T}$ then $\mathcal{T}$ does not contain a structured locking failure.*

Theorem 3 guarantees the structured locking property for well-typed programs, even in presence of exceptions. The proof is found in Appendix A.

7 The synchronization primitives

For completeness, in this section we discuss the extension of JVML_C^e and its type system with the three Java primitives to control thread execution: `wait`, `notify` and `notifyAll`. This extension is almost straightforward as concerns the type system because the JVM considers synchronization primitives as standard method invocations. Therefore this section may be safely skipped in a first reading.

Let JVML_C^s be the language whose set of instruction is:

$$\begin{aligned} \mathit{Instruction} ::= &\ \cdots \text{(as in Sections 3 and 6)} \\ &\mid \texttt{wait} \mid \texttt{notify} \mid \texttt{notifyAll} \end{aligned}$$

The methods `wait`, `notify` and `notifyAll` are *final* in Java (their code cannot be modified) and they shorten the following invocations in the bytecode [6]:

```
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.

7.1 The operational semantics

To model the new operators we extend the configurations as follows:

$$\Vdash_H \mathcal{T}; \mathcal{W}$$

[6] 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`.

$$\frac{\begin{array}{c} P[pc] = \mathtt{wait} \\ H(o).\ell = n \ \ (n > 0) \\ H' = H(o).\ell \mapsto 0 \end{array}}{P \Vdash_H \langle pc, f, o \cdot s, z \cup \{o\}\rangle ; W_o \to \Vdash_{H'} ; W_o \uplus \langle pc, f, s, z \cup \{o\}\rangle_n}$$

$$\frac{\begin{array}{c} P[pc] = \mathtt{notify} \\ H(o).\ell = n \ \ (n > 0) \end{array}}{\begin{array}{l} P \Vdash_H \langle pc, f, o \cdot s, z \cup \{o\}\rangle ; W_o \uplus \{\langle pc', f', s', z'\rangle_n\} \to \\ \qquad \Vdash_H \langle pc+1, f, s, z \cup \{o\}\rangle, \langle pc', f', s', z'\rangle_n \ ; \ W_o \end{array}}$$

$$\frac{\begin{array}{c} P[pc] = \mathtt{notify} \\ H(o).\ell = n \ \ (n > 0) \end{array}}{P \Vdash_H \langle pc, f, o \cdot s, z \cup \{o\}\rangle \ ; \ \emptyset_o \to \Vdash_H \langle pc+1, f, s, z \cup \{o\}\rangle \ ; \ \emptyset_o}$$

$$\frac{\begin{array}{c} P[pc] = \mathtt{notifyAll} \\ H(o).\ell = n \ \ (n > 0) \\ W_o = \{\langle pc_k, f_k, s_k, z_k\rangle_{n_k} \mid k \in K\}_o \end{array}}{P \Vdash_H \langle pc, f, o \cdot s, z \cup \{o\}\rangle \ ; \ W_o \to \Vdash_H \langle pc+1, f, s, z \cup \{o\}\rangle, (\langle pc_k, f_k, s_k, z_k\rangle_{n_k})^{k \in K} \ ; \ \emptyset_o}$$

$$\frac{\begin{array}{c} H(o).\ell = 0 \\ H' = H(o).\ell \mapsto n \end{array}}{P \Vdash_H \langle pc, f, o \cdot s, z\rangle_n \to \Vdash_{H'} \langle pc+1, f, s, z\rangle}$$

Fig. 11. The operational semantics of JVML_C^s

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

The semantics of the synchronizing primitives is defined in Figure 11. Note that there are two rules for `notify`: one when the wait set is not empty and the other when it is empty. In the latter case no notification is done. The last rule of Figure 11 deals with notified threads $\langle pc, f, o \cdot s, z\rangle_n$. 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 $\mathcal{W}$ and $\mathcal{W}'$ be two collections of wait sets, and let $\mathsf{dom}[\mathcal{W}]$ be the set of indexes of wait sets in $\mathcal{W}$. When $\mathsf{dom}[\mathcal{W}] \cap \mathsf{dom}[\mathcal{W}'] = \emptyset$, we define $(\mathcal{T}; \mathcal{W}) + (\mathcal{T}'; \mathcal{W}')$ to be $\mathcal{T}, \mathcal{T}' \ ; \ \mathcal{W}, \mathcal{W}'$. Therefore the new context

$$\frac{\begin{array}{c} P[i] = \texttt{wait} \\ F_i = F_{i+1} \\ S_i = \sigma \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}\ (\textsc{Wait}) \qquad \frac{\begin{array}{c} P[i] = \texttt{notify} \\ F_i = F_{i+1} \\ S_i = \sigma \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}\ (\textsc{Notify}) \qquad \frac{\begin{array}{c} P[i] = \texttt{notifyAll} \\ F_i = F_{i+1} \\ S_i = \sigma \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}\ (\textsc{NotifyAll})$$

Fig. 12. The static semantics of `wait`, `notify` and `notifyAll`.

rule becomes:

$$(\textsc{Context})\ \frac{P \Vdash_H \mathcal{T}_1; \mathcal{W}_1 \to \Vdash_{H'} \mathcal{T}_2; \mathcal{W}_2 \qquad (\mathsf{dom}[\mathcal{W}_1] \cup \mathsf{dom}[W_2]) \cap \mathsf{dom}[\mathcal{W}] = \emptyset}{P \Vdash_H (\mathcal{T}_1; \mathcal{W}_1) + (\mathcal{T}; \mathcal{W}) \to \Vdash_{H'} (\mathcal{T}_2; \mathcal{W}_2) + (\mathcal{T}; \mathcal{W})}$$

7.2 The static semantics

As for method invocations, the Java-compiler only verifies that `wait`, `notify` and `notifyAll` are invoked with stacks containing the right argument types (in this case, reference objects). The extension of the analyzer in Figure 3 with inference rules for `wait`, `notify` and `notifyAll` is illustrated in Figure 12.

Remark that the type system of Figure 12 does not enforce any check that methods invoking a synchronizing primitive actually own the lock. In other words, the Theorem 3 is false for this extension because, in Java, the instructions `wait`, `notify` or `notifyAll` may well occur inside not synchronized statements (or methods). For example, consider a method 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 run-time. To ban these kind of errors, one should check that every invocation of the synchronization primitives falls inside a critical section, that should definitely tangle our type system.

8 Compilation of `synchronized` in current Java compilers

We now analyze the outputs of four Java compilers: the Sun JDK 1.2 and 1.3, the IBM Jikes and Microsoft Visual J++. As specified in [12] these compilers should conform to the commitment in Section 2. However, all of them definitely fail this goal.

```
Method void onlyMe(Foo)
   0  aload_1
   1  astore_2
   2  aload_2
   3  monitorenter
   4  aload_0
   5  invokevirtual doSomething()
   8  aload_2
   9  monitorexit
   10 goto 16
   13 aload_2
   14 monitorexit
   15 athrow
   16 return

  Exception table:
   from  to  target  type
     4    8    13     any

     JDK 1.2
```

```
Method void onlyMe(Foo)
   0  aload_1
   1  astore_2
   2  aload_2
   3  monitorenter
   4  aload_0
   5  invokevirtual doSomething()
   8  aload_2
   9  monitorexit
   10 goto 18
   13 astore_3
   14 aload_2
   15 monitorexit
   16 aload_3
   17 athrow
   18 return

  Exception table:
   from  to  target  type
     4   13    13     any

     JDK 1.3
```

Fig. 13. Compilations of the `synchronized` statement by Sun's JDK 1.2 and JDK 1.3

We consider the method `onlyMe` in Figure 1. Figure 13 illustrates the outputs of JDK 1.2 and 1.3. The output of JDK 1.2 is almost the same as the bytecode compiling the `onlyMe` method in Figure 1 (that is taken from [12], section 7.14): the differences are not meaningful for the following discussion.

JDK 1.2 and JDK 1.3 are correct in case of synchronous exceptions; they are faulty in the presence of asynchronous exceptions. In particular, in JDK 1.2, if an asynchronous exception occurs at 8–9 (or at 13–14), the monitor may be left in a locked state because the exception table does not protect these lines. To remedy to this problem, JDK 1.3 has a different output: the block protected by the exception handler is widened from 4 to 13 (and the exception handler begins at 13 itself). Therefore, if an exception is raised at 10, the lock of the object released at line 9 will be released a second time at line 14, thus causing an exception `IllegalMonitorStateException`.

The IBM Jikes 1.11 compiler follows the same pattern of JDK 1.3, therefore it shares the same criticisms with JDK 1.3.

Figure 14 shows the output of the compilation of `onlyMe` by the Microsoft Visual J++ compiler. Apart the use of subroutines for lock releases, we observe that the exception handler leaves lines 9, 17, 18, and 19 unprotected. Therefore, also in this case, the method invocation may complete abruptly,

```
Method void onlyMe(Foo)
   0  aload_1
   1  dup
   2  astore_2
   3  monitorenter
   4  aload_0
   5  invokevirtual doSomething()
   9  jsr 17
   11 goto 22
   14 aload_2
   15 monitorexit
   16 athrow
   17 astore_3
   18 aload_2
   19 monitorexit
   20 ret 3
   22 return

  Exception table:
   from  to  target  type
     4    9     14    any
```

Fig. 14. Compilation of the `synchronized` statement by Microsoft Java++

without having released the lock before.

The common problem in the bytecodes of Figures 13 and 14 is the management of asynchronous exceptions raised when the interpreter is running the code of the exception handler. In Figure 13, for example, if the exception occurs at line 14 of the two bytecodes, the flow of the computation is deviated to the caller, thus preventing any release of the lock.

Next, we illustrate what should be correct solution. Consider the JDK 1.3 bytecode, and replace the exception table with:

```
Exception table:
   from  to  target  type
     4   10    13     any
    13   16    13     any
```

According to this table, the protected block of the critical section goes, correctly, from 4 to 9. In fact, these are the instructions where the lock acquired at line 3 is held. In addition, there is a further entry which guarantees a "self-protection" for the exception handler itself: if an exception is raised while the exception handler is running and the handler has still not released the lock then the exception must be managed by the same handler. In our case, if an exception is raised from 13 to 15, the old exception is discarded and the new exception is handled in the usual way at line 13.

This solution seems at odd with Sun's compilers, which generate code without "self-protecting" exception handlers ([12], section 4.9.5). Nevertheless, very recently Sun has released the new `JDK 1.4`, where the compiler uses "self-protecting" exception handlers to cope with the above mentioned bugs. We have tested `JDK 1.4` with a set of sample programs and the bytecodes correctly conform with the structured locking property discussed in this paper. However a detailed analysis of `JDK 1.4` has still not undertaken.

9 Compelling the structured locking property

Since outputs of present compilers do not meet the structured locking property (with the exception of Sun `JDK 1.4`), here we provide few recipes which ensure correct bytecodes. We remark that correct bytecodes may still rise an `IllegalMonitorStateException`, because optimizations performed by just-in-time compilers may compromise the structural locking property. We discuss this issue in Subsection 9.3.

9.1 Avoiding synchronizing statements

The problems addressed in this paper are due to `monitorenter`/`monitorexit` blocks, which compile synchronized statements in Java programs. In particular, a Java program without synchronized statements does conform with our type system. Below we discuss a technique for removing synchronized statements from a Java program. This technique cannot be applied when the body of the synchronized statement accesses private methods or fields of the class. In these cases, in order to conform with the structured locking property, the programmer should patch the Java bytecode itself, as discussed in the next subsection.

The details of the technique are the following ones:

(1) Let `synchronized(x){ S }` belong to a method `m` of the class `A`, and `x` be an object of the class `B`. For simplicity, we assume the global variables in `S` are exactly `this` and `x`, and the dot-notation is always used to access to fields and methods. The general case is discussed in item (4).
(2) Replace the statement `synchronized(x){ S }` in `m` with the statement `x.m_aux(this, x)`, where `m_aux` is a fresh name in `A`.
(3) In the class `B`, add a new synchronized method as follows

```
public synchronized void m_aux(A y, B x) {
    S'
}
```

where `m_aux` and `y` are fresh names, and `S'` is the statement `S` with every occurrence of `this` replaced by `y`.

(4) If `S` accesses global variables different from `this` and `x`, there are two subcases:
   (a) *the global variables are references.* In this case, let $\mathtt{x}_1, \cdots, \mathtt{x}_n$ be the global variables, and $\mathtt{T}_1, \cdots, \mathtt{T}_n$ be their corresponding types. It sufficies to change the header of `m_aux` into

   ```
   public synchronized void m_aux(A y, B x, T1 x1,···, Tn xn)
   ```

   and the invocation into $\mathtt{x.m\_aux(this,\ x,\ x_1,\ \cdots\ ,\ x_n)}$.
   (b) *a subset of global variables are primitive data types, which are local in* `m`. In this case, (i) all the primitive data types must be collected in a new class, let it be `C`; (ii) an object of the new class, say `u`, must be created before the invocation to `m_aux` in `m`; (iii) `u` must be initialized with the corresponding values of the primitive data and it is passed as actual parameter to `m_aux`; (iv) `m_aux` is turned into a function which returns values of type `C`; (v) in `m`, after the invocation of `m_aux`, the returned value is assigned to `u`, and hence to the variables whose types are predefined. All these steps are almost standard, thus they do not require further details.

As an example, we consider the method `onlyMe` in Figure 1. The output of the conversion discussed above is:

```
// let T be the class of onlyMe

public void onlyMe(Foo f) {
  f.onlyMe_aux(this, f) ;
}

// in the class Foo add onlyMe_aux

public void synchronized onlyMe_aux(T x, Foo f) {
    { x.doSomething(); }
}
```

The drawback of this solution is that the code could be fragmented into several small methods, without no apparent rationale, and thus hindedring the understanding of the whole program.

9.2 Patching bytecodes

In alternative to modify the source Java code, one may change the bytecode. As discussed in Section 8, these changes only concern the exception tables. Precisely, for every method containing a synchronized statement:

(1) Identify the pair `monitorenter/monitorexit` which correspond to the synchronized statement, and let `i` and `j` be the addresses of `monitorenter` and `monitorexit`, respectively.
(2) Change the entry `i+1 u v T` of the exception table into `i+1 j+1 v T`. We observe that in bytecodes discussed in Section 8 there is exactly one entry with `i+1` in the field "`From`".
(3) Add a new entry `v k v T` in the exception table. The value `k` depends on the compiler which has been used. If the compiler is Sun `JDK 1.2` or Microsoft Java++, then `k` is `v+2`; if the compiler is Sun `JDK 1.3` then `k` is `v+3`.

Identifying the pair `monitorenter/monitorexit` which correspond to the synchronized statement may be difficult. In particular when synchronized statements are nested. This is the danger of this solution: if the identified pair is not the right one, the patched bytecode is still erroneous.

9.3 What optimizers should not do

Java bytecodes are converted by just-in-time compilers into object codes, which are then executed. The just-in-time compilers usually perform code optimizations which shorten the code or minimize the resources that are used (such as local variables). It turns out that these optimizations may compromise the structured locking property.

As a first example consider optimizations that factorize out the `monitorenter` code or the `monitorexit` one, as shown in Figure 7, possibly by means of subroutines (see Figure 14).

As a second example take a method which has two different critical sections, which lock different objects, but with a same body. Then, an optimization could factorize out these bodies into a subroutine, and replace them with a `jsr` instruction. This optimization is wrong because, if an asynchronous exception occurs while the subroutine is executed, the exception handler cannot determine which lock must be relased.

There is a simple general solution to certify optimizer correctness with respect to the structured locking property. It sufficies to run the verifier of Figures 3, 9, and 12 on the transformed code. If this code is well-typed, Theorem 3 guarantees that no `IllegalMonitorStateException` will be raised at runtime. Alternatively, one may demonstrate that optimization patterns preserve the structured locking property. This demonstration avoids to run our verifier each time the code is optimized.

10 The prototype verifier with structured locking checks

Our verifier prototype [4] extends the bytecode verifier in Java 2 SDK 1.3 with structured locking checks, as detailed in this paper. It is written in Java, using the BCEL library (`bcel.sourceforge.net`), a high level tool for analyzing, creating, and manipulating (binary) Java class files.

The prototype also extends the verifier of Figures 3, 9, and 12 in several ways.

10.1 First extension: a larger set of instructions

The set of instructions which are verified is extended with arithmetic and relational operations, subclasses, method invocations, and other primitive instructions of JVML. This richer set of instructions has allowed us to typecheck a large class of Java programs. The typing rules of these instructions are standard, except those of the operations `dup` and `new`.

As regards `dup`, it copies the top of the stack and puts the result on the stack. These copies must traced statically, in case the two items are used for acquiring and releasing a same object. This means that remarks similar to `load` apply to `dup`, too. The typing rule for this instruction is therefore the following:

$$\text{(Dup)}\quad \frac{\begin{array}{c} P[i] = \mathtt{dup} \\ F_i = F_{i+1} \\ S_i = \tau \cdot S' \\ S_{i+1} = \tau_i \cdot \tau_i \cdot S' \\ \forall \widehat{\sigma} \in S_i.\ i \neq \mathit{Ind}[\widehat{\sigma}] \\ \forall y \in \mathsf{dom}[F_i].\ i \neq \mathit{Ind}[F_i[y]] \\ \forall \widehat{\sigma} \in Z_i.\ i \neq \mathit{Ind}[\widehat{\sigma}] \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

The type rule for the instruction `new` of the prototype is different from that in Figure 3. Indeed, in this paper we over-simplified the semantics of `new`, by assuming that object creation and initialization occur at the same time (see Section 3). In the JVML, these two operations are different, and it may happen that an object is created, a number of copies are created, and one of them is initialized. After the initialization of one copy, every copy may be used. To guarantee that objects are only used after the initialization, Freund and Mitchell have defined a verifier in [7], which:

(1) gives a tagged type to the created object,
(2) enforces the copying operation to preserve the tagged type,
(3) makes the initialization operation erase the tag from every type in the stack and in the local variables
(4) constraints the “object usage instructions” to apply to untagged types only.

There are several analogies between this mechanism and the one developed in the present paper. Indeed the machineries which are used are quite similar. Nevertheless, there is a deep difference, that we discuss briefly.

In this paper, the copying operation must be traced in order to accept those programs that perform a `monitorenter` on a lock and a `monitorexit` on a copy of that lock. However, the present type system has no operation which corresponds to that labelled “3” above. Namely there is no way to destroy the tag of object types, except when erasing the content of a variable or of a stack item. Indeed the operation “3” is conceptually erroneous in the system of Figure 3, since copies may be used several times to lock and unlock objects.

In order to merge Freund-Mitchell’s type analysis with our one, we could uses object types with *two* tags: one for object initialization and the other for structured use of locks. Actually, the prototype uses an optimization proposed by Gaetano Bigliardi with types $UNINIT\text{-}T_i$. The initialization operation just removes the prefix “*UNINIT*” from types, and the istructions which use objects apply to types without the prefix “*UNINIT*”. We are confident of the correctness of this solution, even if it has not been proved, yet.

10.2 Second extension: a more reasonable verifier

The verifier in this paper, as well as those in [17,7], rejects codes which, reasonably, the Java bytecode verifier accepts. An example appears in Figure 6, where it was not possible to type a bytecode which locks one variable or another, according to the value of a boolean. In that case, it was not possible to type the join of two branches of a conditional because of a mismatch in the types of the local variables. As we anticipated, there is a standard solution to this problem, which is also used in the bytecode verifier and in our prototype. This solution relies on subtypes and relaxes the equality constraint of types of successor instructions in Figures 3, 9, and 12. We briefly outline this solution.

Types are equipped with a binary relation <: which is the reflexive and transitive closure of the following pairs:

$$\sigma_i <: \sigma \;\; (\forall i) \qquad\qquad \sigma <: \mathrm{TOP} \qquad\qquad \mathrm{INT} <: \mathrm{TOP}$$

$$\text{(IF)}\quad \frac{\begin{array}{c} P[i] = \texttt{if } L \\ F_i <: F_{i+1} \\ F_L = F_{i+1} \\ S_i <: \text{INT} \cdot S_{i+1} = \text{INT} \cdot S_L \\ i+1, L \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} = Z_L \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(LOAD)}\quad \frac{\begin{array}{c} P[i] = \texttt{load } x \\ x \in \mathsf{dom}[F_i] \\ F_i[x] = \tau \\ F_i[x \mapsto \tau_i] <: F_{i+1} \\ \tau_i \cdot S_i <: S_{i+1} \\ \forall \widehat{\sigma} \in S_i.\ i \neq \mathit{Ind}[\widehat{\sigma}] \\ \forall y \in \mathsf{dom}[F_i].\ i \neq \mathit{Ind}[F_i[y]] \\ \forall \widehat{\sigma} \in Z_i.\ i \neq \mathit{Ind}[\widehat{\sigma}] \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(STORE)}\quad \frac{\begin{array}{c} P[i] = \texttt{store } x \\ x \in \mathsf{dom}[F_i] \\ F_i[x \mapsto \tau] <: F_{i+1} \\ S_i <: \tau \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_i = Z_{i+1} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(MONITORENTER)}\quad \frac{\begin{array}{c} P[i] = \texttt{monitorenter} \\ F_i <: F_{i+1} \\ S_i <: \widehat{\sigma} \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ Z_{i+1} = Z_i \uplus \{\widehat{\sigma}\} \end{array}}{F, S, Z, i \vdash P}$$

$$\text{(MONITOREXIT)}\quad \frac{\begin{array}{c} P[i] = \texttt{monitorexit} \\ F_i <: F_{i+1} \\ S_i <: \widehat{\sigma} \cdot S_{i+1} \\ i+1 \in \mathsf{dom}[P] \\ \widehat{\sigma} \in Z_i \\ Z_{i+1} = Z_i \setminus \{\widehat{\sigma}\} \end{array}}{F, S, Z, i \vdash P}$$

Fig. 15. A sample of rules in Figure 3 with the subtyping relation

The relation $<:$ is extended to functions $F^\sigma[i]$ and stacks $S^\sigma[i]$ as follows. Let $f <: g$, if f and g have the same domain and, for every x, $f(x) <: g(x)$. Let also $s <: s'$, if $s = \tau_1 \cdots \tau_n$, $s' = \tau'_1 \cdots \tau'_n$ (s and s' have the same length), and, for every i, $\tau_i <: \tau'_i$.

Equalities in the typing rules are therefore relaxed, in favour of the relation $<:$. In Figure 15 we rewrite a subset of rules in Figure 3 which is useful to typecheck the bytecode in Figure 6.

In Figure 16, we illustrate the types of the bytecode in Figure 6 (as usual, the instruction `goto L` corresponds to the sequence `push0 ; inc ; if L`).

We observe that the type of variables $F_8[0]$, $F_8[1]$, and $F_8[3]$ is `Object`, which is an upper bound of the type $\texttt{Object}_3$ (obtained when the execution comes from line 5) and $\texttt{Object}_6$ (obtained when the execution comes from line 7).

We conclude with two remarks. Firstly the technicalities of this paper are not altered in a sensible way by relaxing the equalities in the typing rules. In particular, the content of Appendix A holds *as it is* for these weakened typing rules. To this aim, remark that if s, s' are sequences of types with $s <: s'$, and $i \notin \mathit{Ind}[s]$ then $i \notin \mathit{Ind}[s']$, too. Similarly for functions from local variables to types.

i	$P[i]$	$F_i[0]$	$F_i[1]$	$F_i[2]$	$F_i[3]$	S_i	Z_i
1	`load 2`	`Object`	`Object`	`boolean`	TOP	ε	$\emptyset$
2	`if 6`	`Object`	`Object`	`boolean`	TOP	`boolean`	$\emptyset$
3	`load 0`	`Object`	`Object`	`boolean`	TOP	ε	$\emptyset$
4	`store 3`	$\texttt{Object}_3$	`Object`	`boolean`	TOP	$\texttt{Object}_3$	$\emptyset$
5	`goto 8`	$\texttt{Object}_3$	`Object`	`boolean`	$\texttt{Object}_3$	ε	$\emptyset$
6	`load 1`	`Object`	`Object`	`boolean`	TOP	ε	$\emptyset$
7	`store 3`	`Object`	$\texttt{Object}_6$	`boolean`	TOP	$\texttt{Object}_6$	$\emptyset$
8	`load 3`	`Object`	`Object`	`boolean`	`Object`	ε	$\emptyset$
9	`monitorenter`	`Object`	`Object`	`boolean`	$\texttt{Object}_8$	$\texttt{Object}_8$	$\emptyset$
10	`load 3`	`Object`	`Object`	`boolean`	$\texttt{Object}_8$	ε	$\{\texttt{Object}_8\}$
11	`monitorexit`	`Object`	`Object`	`boolean`	$\texttt{Object}_8$	$\texttt{Object}_8$	$\{\texttt{Object}_8\}$
12	`return`	`Object`	`Object`	`boolean`	$\texttt{Object}_8$	ε	$\emptyset$

Fig. 16. The static typing of the program in Figure 6

Second, the verifier [4], as well as the Java bytecode verifier, uses a data flow analysis: when the type inference algorithm reaches a same instruction twice, it attempts to find a super type for the mappings F, S and Z. This may require several scannings of the code. On the contrary, as usual, the type inference algorithm underlining the system of this paper scans the code exactly once.

10.3 Third extension: no analysis of dead code

Our verifier avoids to analyze dead codes. Therefore it is less demanding than the static semantics of this paper, but it complies with the Java bytecode verifier.

11 Related works and concluding remarks

As already remarked, our work is strongly based on the framework developed in a series of papers by Stata-Abadi [17] and Freund-Mitchell [7–9], 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 [10], typed assembly languages [14], the Haskell type checker [19], and abstract state machines [5].

As regards the bytecode, a 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 `monitorenter` and `monitorexit` (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 [6,13,18]. Since the `synchronized` statement in Java explicitly defines the critical section and the locked object, most of the problems addressed in this paper disappear in the high-level language. Nevertheless, the structured locking checks are relevant. 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. The extension with method invocations is not problematic (indeed, the current prototype already include this feature [4]). On the contrary, the extension with 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 [17,9] should be easily integrated inside our verifier.

The second kind of extensions concern behavioural properties (i.e. safety and liveness properties, see Chapters 2 and 3 of [11]). Our static semantics does not cover a number of relevant properties. Among others, race conditions and 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.

Acknowledgments. Preliminary results in collaboration with Gaetano Bigliardi appeared in [3]. I thank Gaetano for the pleasant discussions about the prototype implementation. I am also very grateful to the referees, whose suggestions improved a lot the submitted version of this paper.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[2] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.

[3] G. Bigliardi and C. Laneve. A type system for JVM threads. In *Third ACM SIGPLAN Workshop on Types in Compilation (TIC 2000), Montreal*, September 2000.

[4] G. Bigliardi and C. Laneve. The java bytecode verifier with structured locking checks. Software and documentation available from `www.cs.unibo.it/~laneve/html/software.html`, April 2001.

[5] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim and J. Gruska and J. Zlatuska, editor, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic*, number 1450 in LNCS. Springer, Aug. 1998.

[6] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. In *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418. Springer-Verlag, 1997.

[7] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 310–328, 1998.

[8] S. N. Freund and J. C. Mitchell. A formal framework for the Java bytecode language and verifier. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 147–166, 1999.

[9] S. N. Freund and J. C. Mitchell. A type system for java bytecode subroutines and exceptions. Technical Report STAN-CS-TN-99-91, Stanford Computer Science Technical Note, August 1999.

[10] M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. *Lecture Notes in Computer Science*, 1503:17–32, 1998.

[11] D. Lea. *Concurrent programming in Java: design principles and patterns.* The Java series. Addison-Wesley, 1996.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[13] T. Nipkow and D. von Oheimb. Java$_{light}$ is type-safe — definitely. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 161–170, 1998.

[14] R. O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–78. ACM Press, 1999.

[15] Z. Qian. Formal specification of a large subset of Java virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, 1998.

[16] Z. Qian, A. Goldberg, and A. Goglio. A Formal Specification of Java Class Loading. Technical report, Kestrel Institute, Palo Alto, March 2000.

[17] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[18] D. Syme. Proving java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, June 1997.

[19] P. M. Yelland. A compositional account of the Java virtual machine. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 57–69, 1999.

A The correctness of the verifier for $\mathbf{JVML}_C^e$

We begin with the notion of well-typed configurations. Let H be a heap and let

$$\mathcal{H}(H) = \{o : \overline{\sigma} \mid o \in \mathsf{dom}[H] \text{ and } o : \sigma\}$$

where that $\overline{\sigma}$ is the record of fields that are specified in the class definition of σ (see Section 4). We also let $\vdash_{\text{heap}}$ be the judgment defined by the following rules ($\mathcal{H}$ is a set of assumptions $o : \overline{\sigma}$):

$$\frac{n \text{ integer}}{\mathcal{H} \vdash_{\text{heap}} n : \text{INT}} \qquad \frac{v \text{ is a value (integer or object name)}}{\mathcal{H} \vdash_{\text{heap}} v : \text{TOP}} \qquad \frac{o : \overline{\sigma} \in \mathcal{H}}{\mathcal{H} \vdash_{\text{heap}} o : \sigma}$$

$$\frac{\mathcal{H} \vdash_{\text{heap}} o : \tau \qquad i \in \textsc{Addr}}{\mathcal{H} \vdash_{\text{heap}} o : \tau_i}$$

The first three judgments allow to infer types for integers, generic values (integers and object names) and object names declared in $\mathcal{H}$. The last judgment is about indexed types: if an object name has type σ in $\mathcal{H}$ then it has also type σ_i, for every address i.

Definition 4 *(Well-typed configuration) Let* $P = \{(P^\sigma; E^\sigma) \mid \sigma \in \Sigma\}$, $F = \{F^\sigma \mid \sigma \in \Sigma\}$, $S = \{S^\sigma \mid \sigma \in \Sigma\}$, $Z = \{Z^\sigma \mid \sigma \in \Sigma\}$. *A configuration* $\Vdash_H \mathcal{T}$ *is* $[P, F, S, Z]$-well-typed, *written* $P, F, S, Z \vdash (\Vdash_H \mathcal{T})$, *if, for every* $\langle i, f, s, z\rangle \in \mathcal{T}$*:*

(1) if $i \in \mathsf{dom}[P^\sigma]$, for some σ, then

(a) for every $x \in \text{VAR}$, $\mathcal{H}(H) \vdash_{\text{heap}} f(x) : F_i^\sigma[x]$;

(b) $s = v_1 \cdots v_k$ implies $S_i^\sigma = \tau_1 \cdots \tau_k$ and $(\mathcal{H}(H) \vdash_{\text{heap}} v_j : \tau_j)^{j \in 1..k}$;

(c) $z = \{o_j \mid j \in J\}$ and $H(o_j).\ell = k_j$ $(k_j > 0)$, for $j \in J$, imply $(\mathcal{H}(H) \vdash_{\text{heap}} o_j : \widehat{\sigma_j})^{j \in J}$ and $Z_i^\sigma = \{k_j \cdot \widehat{\sigma_j} \mid j \in J\}$;

*(d) (*aliasing*) let*

- *$o = f(x)$, when $F_i^\sigma[x] = \widehat{\sigma}$, or $o = v_j$, $j \in 1..k$, when $s = v_1 \cdots v_k$, $S_i^\sigma = \tau_1 \cdots \tau_k$ and $\tau_j = \widehat{\sigma}$;*
- *$o' = f(y)$, when $F_i^\sigma[y] = \widehat{\sigma'}$, or $o' = v_{j'}$, $j' \in 1..k$, when $s = v_1 \cdots v_k$, $S_i^\sigma = \tau_1 \cdots \tau_k$ and $\tau_{j'} = \widehat{\sigma'}$.*

If $\widehat{\sigma} = \widehat{\sigma'}$ then $o = o'$. Furthermore, if $\widehat{\sigma} \in Z_i$ then $o \in z$, too.

(2) if $i \notin \mathsf{dom}[P]$ then $z = \emptyset$.

Conditions 1.a, 1.b, and 1.c of Definition 4 regard the correctness of f, s, and z. Condition 1.d concerns the correctness of our aliasing technique: if two variables have the same indexed type then they must be aliases. Similarly for two items in the stack, for a variable and an item in the stack, and for a variable (or an item in the stack) and a locked object.

Next, we begin by stating a couple of preliminary lemmas about types and a simple proposition about heaps and locks (proofs are omitted because trivial). Then we demonstrate the one step soundness theorem for the operational semantics in Figures 2 and 8.

In the following, P is always a JVML_C^e program, namely $P = \{(P^\sigma; E^\sigma) \mid \sigma \in \Sigma\}$, for some σ. The first lemma gives a form of weakening for the judgment $P, F, S, Z \vdash (\Vdash_H \mathcal{T})$.

Lemma 5 *Let $P, F, S, Z \vdash (\Vdash_H \mathcal{T})$ and let H' be such that $\mathcal{H}(H) \subseteq \mathcal{H}(H')$ and, for every $o \in Lock[\mathcal{T}]$, $H(o).\ell = H'(o).\ell$. Then $P, F, S, Z \vdash (\Vdash_{H'} \mathcal{T})$.*

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

Lemma 6 *For all P, F, S, and Z such that $F, S, Z \vdash P$, if $P, F, S, Z \vdash (\Vdash_H \mathcal{T})$ and $P \Vdash_H \mathcal{T} \to \Vdash_{H'} \mathcal{T}'$ then, for every $o \in \mathsf{dom}[H] \cap \mathsf{dom}[H']$, $\mathcal{H}(H) \vdash_{\text{heap}} o : \sigma$ if and only if $\mathcal{H}(H') \vdash_{\text{heap}} o : \sigma$.*

The next proposition states that heaps do not decrease during the computation and locks which are modified are those in the sets z.

Proposition 7 *Let $P \Vdash_H \mathcal{T} \to \Vdash_{H'} \mathcal{T}'$ then*

(1) $\mathsf{dom}[H] \subseteq \mathsf{dom}[H']$;

(2) if $o \notin Lock[\mathcal{T}] \cup Lock[\mathcal{T}']$ then $H(o).\ell = H'(o).\ell$.

To prove the soundness theorem 3 we use the following one-step soundness property.

Theorem 8 *For all P, F, S, and Z such that $F, S, Z \vdash P$, if $P, F, S, Z \vdash (\Vdash_H \mathcal{T})$ and $P \Vdash_H \mathcal{T} \to \Vdash_{H'} \mathcal{T}'$ then $P, F, S, Z \vdash (\Vdash_{H'} \mathcal{T}')$.*

PROOF. By induction on the number of applications of rule CONTEXT to infer $P \Vdash_H \mathcal{T} \to \Vdash_{H'} \mathcal{T}'$.

Basic case. The rule CONTEXT is not used; therefore $\mathcal{T} = \langle i, f, s, z \rangle$. Since $\Vdash_H \mathcal{T}$ reduces, $i \in \mathsf{dom}[P]$ and, by $P, F, S, Z \vdash (\Vdash_H \mathcal{T})$, we derive:

(i) for every y, $\mathcal{H}(H) \vdash_{\text{heap}} f(y) : F_i[y]$;
(ii) let $s = v_1 \cdots v_k$, then $S_i = \tau_1 \cdots \tau_k$ and $(\mathcal{H}(H) \vdash_{\text{heap}} v_i : \tau_i)^{i \in 1..k}$;
(iii) let $z = \{o_j \mid j \in J\}$ and $H(o_j).\ell = k_j$ $(k_j > 0)$, for $j \in J$, then $(\mathcal{H}(H) \vdash_{\text{heap}} o_j : \widehat{\sigma_j})^{j \in J}$ and $Z_i^\sigma = \{k_j \cdot \widehat{\sigma_j} \mid j \in J\}$;
(iv) the aliasing condition

By cases on the rule used. In every case we use the corresponding typing judgment in Figures 3 and 9.

Cases $P[i]$ is `inc`, or `push0`, or `pop`, or `if` follow by the hypotheses $P, F, S, Z \vdash (\Vdash_H \mathcal{T})$ and the well typing of the whole program P. We detail the instruction `if` , when the top of the stack is a non-null integer. In this case $H' = H$ and $\mathcal{T}'$ is $\langle L, f, v_2 \cdots v_k, z \rangle$. We must verify that $P, F, S, Z \vdash (\Vdash_H \mathcal{T}')$, in particular condition 1 of Definition 4, because, by rule IF, $L \in \mathsf{dom}[P]$. Conditions 1.a, 1.c and 1.d of Definition 4 are immediate consequences of $F, S, Z \vdash P$ and (i), (iii), (iv), respectively. As regards condition 1.b, by rule IF, $S_i = \text{INT}\cdot S_{i+1} = \text{INT}\cdot S_L$, and then use (ii) for S_L.

Case $P[i] = \texttt{load}\ x$. In this case $H' = H$ and $\mathcal{T}'$ is $\langle i+1, f, f(x) \cdot s, z \rangle$. By rule LOAD, $i + 1 \in \mathsf{dom}[P]$, therefore we verify condition 1 of Definition 4 for $P, F, S, Z \vdash (\Vdash_H \mathcal{T}')$. Condition 1.c follows by $Z_i = Z_{i+1}$ (because of LOAD) and (iii). For the other conditions, we detail the case when $f(x)$ is an object name; the other case is simpler. Let $F_i[x] = \tau$. There are two subcases, according to $\tau = \widehat{\sigma}$ and $\tau = \sigma$. In the former case, $S_{i+1} = \widehat{\sigma} \cdot \tau_1 \cdots \tau_k$ and $F_{i+1} = F_i$. Therefore conditions 1.a, 1.b, and 1.d of Definition 4 are immediate from (i), (ii), and (iv), respectively. When $\tau = \sigma$, $S_{i+1} = \sigma_i \cdot \tau_1 \cdots \tau_k$ and $F_{i+1} = F_i[x \mapsto \sigma_i]$. Since $\mathcal{H}(H) \vdash_{\text{heap}} f(x) : \sigma$, we derive $\mathcal{H}(H) \vdash_{\text{heap}} f(x) : \sigma_i$. This fact, (i), and (ii) imply condition 1.a and 1.b of Definition 4. To verify condition 1.d, note that rule LOAD requires that no i-indexed type occurs in S_i, F_i, and Z_i. Therefore F_{i+1}, S_{i+1}, and Z_{i+1} contain exactly two items σ_i: $F_{i+1}[x]$ and the topmost element of S_{i+1}, which have a same corresponding value in f and $f(x) \cdot s$. Condition 1.d for the other indexed types in F_{i+1} and S_{i+1} follows from (iv).

Case $P[i] = \texttt{store}\ x$. In this case, $H' = H$ and $\mathcal{T}'$ is $\langle i{+}1, f[x \mapsto v_1], v_2 \cdots v_k, z\rangle$. By rule STORE, $i+1 \in \mathsf{dom}[P]$, which requires to check condition 1 of Definition 4 for $\Vdash_H \mathcal{T}'$. As regards 1.b, it follows by (ii). As regards 1.a, observe that $y \in \mathsf{dom}[F_i]$ and $y \neq x$ imply $\mathcal{H}(H) \vdash_{\text{heap}} (f[x \mapsto v_1])(y) : F_{i+1}(y)$ because of (i) and rule STORE. The judgment $\mathcal{H}(H) \vdash_{\text{heap}} (f[x \mapsto v_1])(x) : F_{i+1}(x)$ follows by $\mathcal{H}(H) \vdash_{\text{heap}} (f[x \mapsto v_1])(x) : \tau_1$, which is a consequence of (ii), and $F_{i+1}(x) = \tau_1$, which is consequence of rule STORE. Condition 1.c and 1.d follow with a similar reasoning.

Case $P[i] = \texttt{new}\ \sigma$. In this case $H' = H[o \mapsto \rho_\sigma]$, where o is a fresh object name, and $\mathcal{T}'$ is $\langle i+1, f, o \cdot s, z\rangle$. By rule NEW we derive $i+1 \in \mathsf{dom}[P]$, therefore we verify condition 1 of Definition 4 for $\Vdash_H \mathcal{T}'$. By rule NEW, $S_{i+1} = \sigma \cdot S_i$, and, by definition of $\vdash_{\text{heap}}$, we derive $\mathcal{H}(H') \vdash_{\text{heap}} o : \sigma$. This fact and (ii) give condition 1.b. The other conditions are immediate.

Cases $P[i]$ is putfield or getfield. We discuss the case of $\texttt{putfield}\ \sigma.a\ \tau$ (getfield is similar). Let $H' = H(v_2).a \mapsto v_1$ and $\mathcal{T}'$ be $\Vdash_{H'} \langle i+1, f, v_3 \cdots v_k, z\rangle$. To derive $P, F, S, Z \vdash (\Vdash_{H'} \mathcal{T}')$, we verify condition 1 of Definition 4 because, by rule PUTFIELD, $i+1 \in \mathsf{dom}[P]$. By (ii) and rule PUTFIELD, $\tau = \mathit{Type}[\tau_1]$, $\sigma = \mathit{Type}[\tau_2]$, and $\overline{\sigma}.a : \tau$. Therefore, by definition of $\mathcal{H}(H)$ and $\vdash_{\text{heap}}$, $v_2 : \overline{\sigma} \in \mathcal{H}(H)$ and $\mathcal{H}(H) \vdash_{\text{heap}} v_2 : \sigma$. Similarly for v_1, if it is an object. As a consequence, $\mathcal{H}(H) = \mathcal{H}(H')$ and the conditions 1.a, 1.b, 1.c, and 1.d for $(\Vdash_{H'} \mathcal{T}')$ follow from (i), (ii), (iii), and (iv), respectively.

Case $P[i] = \mathsf{return}$. Vacuous (there is no transition).

Case $P[i] = \texttt{start}\ \sigma$. In this case $H' = H$ and $\mathcal{T}'$ is $\langle i{+}1, f, v_2 \cdots v_k, z\rangle, \langle 1_\sigma, f_0[0 \mapsto v_1], \varepsilon, \emptyset\rangle$. We focus on the new thread, omitting the check for the old one. To infer $P, F, S, Z \vdash (\Vdash_H \langle 1_\sigma, f_0[0 \mapsto o], \varepsilon, \emptyset\rangle)$, we recall that $1_\sigma \in \mathsf{dom}[P]$, because $\sigma \in \Sigma$ by definition of JVML$_C$ (see Section 3). Therefore we verify condition 1 of Definition 4. By rule START and (ii) we derive $\sigma = \mathit{Type}[\tau_1]$. Therefore, by definition of $\mathcal{H}(H)$ and $\vdash_{\text{heap}}$, $v_1 : \overline{\sigma} \in \mathcal{H}(H)$ and $\mathcal{H}(H) \vdash_{\text{heap}} v_1 : \sigma$. This conclusion implies condition 1.a. The other conditions are trivial.

Case $P[i] = \texttt{monitorenter}$. There are two subcases, according to the thread already holds the lock or not. We demonstrate the latter case, the other is similar. In this case, $H(v_1).\ell = 0$, $v_1 \notin z$, $H' = H(v_1).\ell \mapsto 1$, and $\mathcal{T}'$ is $\langle i+1, f, v_2 \cdots v_k, z \cup \{v_1\}\rangle$. By rule MONITORENTER, $i+1 \in \mathsf{dom}[P]$, therefore we check condition 1 of Definition 4. Since $\mathcal{H}(H) = \mathcal{H}(H')$, conditions 1.a and 1.b are immediate consequences of (i) and (ii), respectively. Condition 1.d follows from (iv) because $F_{i+1} = F_i$ and S_{i+1} is shorter than S_i. As regards condition 1.c, by rule MONITORENTER and (ii), $\tau_1 = \widehat{\sigma}$, for some σ. In addition, (iii) and $\mathcal{H}(H) = \mathcal{H}(H')$ imply $\mathcal{H}(H') \vdash_{\text{heap}} o_j : \widehat{\sigma_j}$, for every $j \in J$. By rule MONITORENTER, $Z_{i+1} = Z_i \uplus \{\widehat{\sigma}\}$. Therefore, we conclude $Z_i \uplus \{\widehat{\sigma}\} = \{k_j \cdot \widehat{\sigma_j} \mid j \in J\} \uplus \{1 \cdot \widehat{\sigma}\}$, by (iii), $H'(o).\ell = H(o).\ell$, for every $o \neq v_1$, and

$H'(v_1).\ell = 1$.

Case $P[i] = \texttt{monitorexit}$. As before, there are two subcases that are similar. We show the case when the thread releases the object, namely $H(v_1).\ell = 1$. Let $z = z' \uplus \{v_1\}$, $H' = H(v_1).\ell \mapsto 0$, and $\mathcal{T}' = \langle i+1, f, v_2 \cdots v_k, z'\rangle$. By rule MONITOREXIT, $i+1 \in \mathsf{dom}[P]$, hence we verify condition 1 of Definition 4. As before, $\mathcal{H}(H) = \mathcal{H}(H')$, thus conditions 1.a and 1.b are immediate consequences of (i) and (ii), respectively. Similarly for 1.d. As regards condition 1.c, let $z' = \{o_{j'} \mid j' \in J'\}$. Since $\mathcal{H}(H) = \mathcal{H}(H')$, we derive $\mathcal{H}(H') \vdash_{\text{heap}} o_{j'} : \widehat{\sigma_{j'}}$, for all $j' \in J'$. It remains to verify that $Z_{i+1} = \{k_{j'} \cdot \widehat{\sigma_{j'}} \mid j' \in J'\}$, where $k_{j'} = H'(o_{j'}).\ell$, for every $j' \in J'$. By (iii), $H'(o).\ell = H(o).\ell$ for every $o \neq v_1$, and $H(v_1).\ell = 1$, we have $Z_i = \{k_{j'} \cdot \widehat{\sigma_{j'}} \mid j' \in J'\} \uplus \{1 \cdot \widehat{\sigma_j}\}$. Therefore we conclude by rule MONITOREXIT which enforces $Z_{i+1} = Z_i \setminus \{\widehat{\sigma}\}$.

Case $P[i] = \texttt{athrow}$. In this case $H' = H$ and $\mathcal{T}'$ is $\langle E[i], f, v_1, z\rangle$. Let $E[i] = k$; there are two subcases, either $k \in \mathsf{dom}[P]$ or $k \notin \mathsf{dom}[P]$. When $k \in \mathsf{dom}[P]$, we verify condition 1 of Definition 4. Condition 1.a follows by rule EXC-HANDLER and (i). Condition 1.b follows by (ii) and rule ATHROW, which yield $\mathcal{H}(H) \vdash_{\text{heap}} v_1 : \text{THROWABLE}$. Condition 1.c and 1.d are consequences of rule EXC-HANDLER and, respectively, (iii) and (iv). When $k \notin \mathsf{dom}[P]$, we verify condition 2 of Definition 4, namely $z = \emptyset$. Actually this equality follows by (iii) and rule EXC-HANDLER, which admits $k \notin \mathsf{dom}[P]$ only if $Z_i = \emptyset$.

Case of asynchronous exception. Similar to the case when $P[i] = \texttt{athrow}$.

Inductive case. Let $\Vdash_H \mathcal{T}_1, \mathcal{T} \to \Vdash_{H'} \mathcal{T}_2, \mathcal{T}$ and $P, F, S, Z \vdash (\Vdash_H \mathcal{T}_1, \mathcal{T})$ and $P \Vdash_H \mathcal{T}_1 \to \Vdash_{H'} \mathcal{T}_2$ and $(\mathit{Lock}[\mathcal{T}_1] \cup \mathit{Lock}[\mathcal{T}_2]) \cap \mathit{Lock}[\mathcal{T}] = \emptyset$. By inductive hypotheses, $P, F, S, Z \vdash (\Vdash_{H'} \mathcal{T}_2)$. By Proposition 7(1) and Lemma 6, $\mathcal{H}(H) \subseteq \mathcal{H}(H')$ (the types of corresponding fields in H and H' are the same, values may be different). In addition, the condition $(\mathit{Lock}[\mathcal{T}_1] \cup \mathit{Lock}[\mathcal{T}_2]) \cap \mathit{Lock}[\mathcal{T}] = \emptyset$ and Proposition 7(2) yield $H'(o).\ell = H(o).\ell$, for every $o \in \mathit{Lock}[\mathcal{T}]$. We conclude $P, F, S, Z \vdash (\Vdash_{H'} \mathcal{T})$ by Lemma 5. $\square$

The following proposition says that well-typed configurations do not contain structured locking failures:

Proposition 9 *Let $F, S, Z \vdash P$. If $P, F, S, Z \Vdash_H \mathcal{T}$ then configurations in $\mathcal{T}$ are not structured locking failures for P.*

PROOF. We verify that no case of structured locking failure may occur. Let $\langle i, f, s, z\rangle \in \mathcal{T}$.

(1) If $P[i] = \texttt{return}$ then $z = \emptyset$. Since $F, S, Z \vdash P$, by rule RETURN, $z_i = \emptyset$. Therefore, by Definition 4, condition 1.c, $z = \emptyset$.
(2) If $i \notin \mathsf{dom}[P]$ then $z = \emptyset$. By condition 2 of Definition 4.
(3) If $P[i] = \texttt{monitorexit}$ and $s = o \cdot s'$ then $o \in z$. Since $F, S, Z \vdash P$, by rule MONITOREXIT, $S_i = \widehat{\sigma} \cdot S_{i+1}$ and $\widehat{\sigma} \in Z_i$. Therefore, by Definition 4, condition 1.d, $o \in z$. □

Our main soundness theorem can now be proved. For completeness, we also restate it.

Theorem 3. *For all P, F, S and Z such that $F, S, Z \vdash P$, and for all classes σ in P, if $P \Vdash_{H_0} \langle 1_\sigma, f_0[o \mapsto \rho_\sigma], \varepsilon, \emptyset\rangle \rightarrow^* \Vdash_H \mathcal{T}$ then $\mathcal{T}$ does not contain a structured locking failure.*

PROOF. The configuration $\Vdash_{H_0} \langle 1_\sigma, f_0[o \mapsto \rho_\sigma], \varepsilon, \emptyset\rangle$ is $[P, F, S, Z]$-well-typed (see the demonstration of the case $P[i] = \texttt{start}$ in Theorem 8). Then, by Theorem 8, $\Vdash_H \mathcal{T}$ is $[P, F, S, Z]$-well-typed, too. We conclude by Proposition 9. □