A New Type System for JVM Lock Primitives

Futoshi IWAMA and Naoki KOBAYASHI

Tohoku University
Aramaki aza Aoba 09, Aoba-ku Sendai Miyagi-pref. 980-8579, Japan

iwama@kb.ecei.tohoku.ac.jp and koba@ecei.tohoku.ac.jp

Abstract A bytecode verifier for the Java virtual machine language (JVML) statically checks that bytecode does not cause any fatal error. However, the present verifier does not check correctness of the usage of lock primitives. To solve this problem, we extend Stata and Abadi's type system for JVML by augmenting types with information about how each object is locked and unlocked. The resulting type system guarantees that when a thread terminates, it has released all the locks it has acquired and that a thread releases a lock only if it has acquired the lock previously. We have implemented a prototype Java bytecode verifier based on the type system. We have tested the verifier for several classes in the Java run time library and confirmed that the verifier runs efficiently and gives correct answers.

Keywords Type System, Java Bytecode Verifier, Lock

§1 Introduction

There has recently been a growing interest in verification of low-level code: proof-carrying code [23], typed assembly languages [22], Java bytecode verification [25, 19, 11, 12], to name a few. The verified properties range from memory-safety to more advanced properties like information flow [1, 17]. The verification of low-level code has several significant advantages over verification of source programs. First, the correctness of a compiler need not be assumed for the safe execution of compiled code. Second, source programs may be written in any programming languages – even the low-level language itself, as long as there exists a compiler into the low-level language. Third, the safety is guaranteed even in the situation where source programs are not available (as is often the case for libraries and programs downloaded from the Internet).

As an instance of the research on verification of low-level code, in this paper, we propose a type-based method for verifying safe usage of JVM lock primitives. The Java virtual machine language (JVML) has instructions for acquiring and releasing object locks. The `monitorenter` instruction acquires the lock on the object stored in the stack top, while the `monitorexit` instruction releases the lock on the object stored in the stack top. Using these instructions, a synchronized statement `synchronized (`x`)`$\{S\}$ in Java is compiled into the following bytecode:

```
load x
monitorenter
...
load x
monitorexit
```

The aim of our verification is to check, for each method, (1) all the locks that have been acquired will be released within the same method execution, and (2) non-acquired locks will not be released. Violation of the first property causes a deadlock; other threads that try to acquire the same lock will be blocked forever. Violation of the second property causes a run-time exception to be raised and the program to be aborted. The present Java bytecode verifier does not check those properties [20].

The safe usage of lock primitives is syntactically guaranteed (by `synchro nized` statements) for Java source programs, but it is less trivial for Java bytecode. For example, consider the six pieces of code in Figure 1. Here, for the sake of simplicity, we write `monitorenter x` for the combination of two instructions `load x; monitorenter`, and `monitorexit x` for `load x; monitorexit`.[*1] Code 1 first locks the object stored in variable x, tests whether the value of y is 0 (by the 2nd and 3rd instructions), and then, depending on the result of the test, executes the 4th or 6th instruction. Since the lock is released in each branch, the code is valid. On the other hand, Code 2 is invalid since the lock is not released if the value of y is not 0. Code 3 is valid; it first locks the object stored in x, and then move it from x to y by the 2nd and 3rd instructions, and then releases the object stored in y. On the other hand, Code 4 is invalid, since the value of x is changed by the 3rd instruction, so that the object that is unlocked by the 4th instruction may not be identical to the object locked by the 1st instruction.

[*1] We use these compound instructions also in the formalization in later sections.

Code 5 and 6 are both valid. Code 5 corresponds to a Java source program `synchronized(x){ synchorinzed(y){...}}`. Code 6 appears in hand-over-hand locking strategies as provided in `java.util.concurrent.locks`.

We use a type system to check the safe usage of lock primitives. The main idea of our type system is to augment the type of an object with a *usage*, which expresses how the object is locked and unlocked. For example, we express by $L.\widehat{L}.\mathbf{0}$ the usage of an object that will be locked, unlocked, and then neither locked nor unlocked afterwards. The usage $L\&\widehat{L}$ describes an object that will be either locked or unlocked. Let us reconsider *Code 1* in Figure 1. The following type is assigned to the object stored in x at each address.

Address	Type of x
1	$\sigma/L.\widehat{L}.\mathbf{0}$
2	$\sigma/\widehat{L}.\mathbf{0}$
3	$\sigma/\widehat{L}.\mathbf{0}$
4	$\sigma/\widehat{L}.\mathbf{0}$
5	$\sigma/\mathbf{0}$
6	$\sigma/\widehat{L}.\mathbf{0}$
7	$\sigma/\mathbf{0}$

Here, types are of the form σ/U, where σ is the ordinary object type (i.e., the class name) of x and U is a usage. The type $\sigma/L.\widehat{L}.\mathbf{0}$ at address 1 indicates that the object stored in x at address 1 will be locked once and then unlocked once in the method. So, we know that lock primitives are properly used. Based on this extension of types with usages, we extend Stata and Abadi's type system [25], so that any well-typed program uses lock primitives in a safe manner. Thus, the problem of verifying safe usage of lock primitives is reduced to the type-checking problem in the extended type system. Our type system accepts Code 1, 3, 5 and 6 in Figure 1, and rejects Code 2 and 4.

Our type system guarantees that well-typed bytecode satisfies the following properties: (1) when a thread terminates it has released all the locks it has acquired, and (2) a thread releases a lock only if it has acquired the lock previously. Our type system does *not* guarantee the absence of deadlocks caused by conflicting locking orders. The type system neither guarantees that appropriate locks are acquired before shared objects are accessed. However, our type system may be used as a building block for deadlock and race analyses; Once it is checked by our type system that each lock instruction is followed by an unlock

```
1  monitorenter x
2  load y
3  if 6
4  monitorexit x
5  return
6  monitorexit x
7  return
```

(*code 1*)

```
1  monitorenter x
2  load y
3  if 5
4  return
5  monitorexit x
6  return
```

(*code 2*)

```
1  monitorenter x
2  load x
3  store y
4  monitorexit y
5  return
```

(*code 3*)

```
1  monitorenter x
2  load y
3  store x
4  monitorexit x
5  return
```

(*code 4*)

```
1  monitorenter x
2  monitorenter y
3  monitorexit y
4  monitorexit x
5  return
```

(*code 5*)

```
1  monitorenter x
2  monitorenter y
3  monitorexit x
4  monitorexit y
5  return
```

(*code 6*)

Fig. 1 Programs that use lock primitives

instruction, then deadlock and race analyses for JVM become similar to those for source languages where only structured lock commands (like `synchronized` in Java) are used [7, 8, 9]. In fact, Permandla and Boyapati [24] have recently proposed a type system for JVM that guarantees deadlock- and race-freedom, and their type system is based on a type system (not ours but Laneve's one, mentioned below) for checking the correspondence between lock/unlock instructions.

Another type system for checking the safe usage of JVML lock primitives have already been proposed by Bigliardi and Laneve [2, 19]. The idea of their type system is quite different from ours. We believe that our type system is more intuitive, and can be more easily applied to languages other than JVML. More detailed comparison of our type system and theirs is found in Section 8.

The rest of this paper Section 2 introduces our target language. Section 3 defines our type system and Section 4 shows the correctness of the type system. Section 5 describes a type inference algorithm, and discusses the time complexity of the algorithm. Based on the type inference algorithm, we have implemented a prototype verifier for Java bytecode. Section 6 reports on experiments with that verifier. Our type system checks that each acquired lock is released within the same method, according to the JVM specification [20]. Section 7 discusses an extension of the type system to lift that restriction. Section 8 discusses related work and Section 9 concludes.

§2 The Target Language $JVML_L$

In this section, we introduce our target language $JVML_L$ and define its operational semantics. The language $JVML_L$ is a subset of the Java byte-code language JVML and is similar to $JVML_C$ introduced by Bigliardi and Laneve [2, 19]. For the sake of simplicity, the language $JVML_L$ has only basic instructions of JVML including lock operations and other main instructions.

2.1 The language $JVML_L$

In $JVML_L$, a program is represented by a set of class definitions of the form:

```
Class σ {
   super: Thread
   field: FD
   method run(D)
      B ; E
}
```

Here, meta-variable σ denotes a *class name* and each class is defined as a subclass of *Thread* class that has only one method *run*. Formally, a program is regarded as a mapping from class names to quadruples of the form (FD, D, B, E), and denoted by meta-variable P. FD, D, B, and E are called a *class field*, a *method descriptor*, a *method body*, and an *exception table* respectively; they are defined below.

We write Σ for the set of class names σ. The set $\mathbf{Ar}$ of array class names, ranged over by A, is defined by $A ::= \mathbf{Int}[\,] \mid \sigma[\,] \mid A[\,]$. We write $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{V}$ for the set of natural numbers, the set of program addresses, and the set of local variables, respectively. $\mathcal{A}$ and $\mathcal{V}$ are finite subsets of $\mathcal{N}$. We use a meta-variable l to denote an element of $\mathcal{A}$ and meta-variables $x, y, \ldots$ to denote elements of $\mathcal{V}$. We write d for an element of $\{\mathbf{Int}\} \cup \Sigma \cup \mathbf{Ar}$, and call it a *descriptor*.

A *class field*, denoted by FD, is a sequence $a_1 : d_1, \ldots, a_k : d_k$, where a is a field name and d is a descriptor. When $P(\sigma) = (FD, D, B, E)$, we often write $\overline{\sigma_P}$ for FD. We also write $\overline{\sigma_P}.a_i : d_i$ if $\overline{\sigma_P}$ is of the the form $[\ldots, a_i : d_i, \ldots]$.

A *method descriptor* D is a mapping from the set $\{0, \ldots, n-1\} (\subseteq \mathcal{V})$ to the set $\{\mathbf{Int}\} \cup \Sigma \cup A$, where n is a natural number that denotes the number of arguments of a method. $D(x)$ denotes the type of the x-th argument of a method. For example, $D(x) = \mathbf{Int}$ means that the type of x-th argument is integer.

A *method body* B is a mapping from a finite subset of $\{1, 2, \ldots, n\}$ ($n \in \mathcal{N}$) of $\mathcal{A}$ to the set of instructions **Inst**, where **Inst** is defined as follows:

Definition 2.1 (Instruction)
The set **Inst** of *instructions* is defined by:

$$
\begin{array}{rcl}
I & ::= & \texttt{inc} \mid \texttt{pop} \mid \texttt{push0} \mid \texttt{load}\ x \mid \texttt{store}\ x \mid \texttt{if}\ l \\
 & & \mid \texttt{putfield}\ \sigma.a\ \ d \mid \texttt{getfield}\ \sigma.a\ \ d \mid \texttt{aaload} \mid \texttt{aastore} \\
 & & \mid \texttt{monitorenter}\ x \mid \texttt{monitorexit}\ x \\
 & & \mid \texttt{new}\ \sigma \mid \texttt{start}\ \sigma \mid \texttt{athrow} \mid \texttt{return}
\end{array}
$$

These instructions have the same meanings as the corresponding instructions of JVML. Examples of method bodies are found in Figure 1.

A *JVML$_L$* program is executed by threads. Each thread has its own operand stack and local variables. A thread manipulates its own stack and local variables, creates new threads, etc, by executing the instructions. We explain each instruction briefly below. Instruction `inc` increments the integer stored at the top of the operand stack. Instruction `pop` pops a value from the operand stack and `push0` pushes the integer 0 onto the operand stack. Instruction `load` x pushes the value stored in the local variable x onto the operand stack, and `store` x removes the top value from the operand stack and stores the value into the local variable x. Instruction `if` l pops the top value from the operand stack and jumps to the address l if the value is not 0, and proceeds to the next address if the value is 0.

Instruction `putfield` $\sigma.a\ \ d$ pops two values from the operand stack and stores the first value into the field a of the second value. The first value must have type d and the second value must be a σ-class object. Instruction `getfield` $\sigma.a\ \ d$ pops an object from the operand stack and then pushes the value stored in field a of the object onto the operand stack, where the object must be a σ-class object with field a of descriptor d. Instruction `aaload` pops two values v_1 and v_2 from the operand stack, where v_1 must be an integer and v_2 must be an array object, and pushes the v_1-th element of the array v_2 onto the stack. Instruction `aastore` pops three values v_1, v_2, and v_3 from the operand stack, where the first value v_1 must be a value that is used as a component value of array v_3 and the second value v_2 must be a integer, and replaces the v_2-th element of array v_3 with v_1.

Instruction `new` σ allocates a new σ-class object, initializes it, and then puts a reference to the object on top of the operand stack. If the allocation or initialization fails, then an exception is raised. Instruction `start` σ creates a new σ-class thread and invokes the *run* method of the thread. Arguments of the method are taken from the top of the operand stack and stored in the local variables of the new thread (where the number of arguments is determined by the class name σ). Instruction `athrow` raises an exception and jumps to the address specified by the exception table (see below). Instruction `return` returns from the current method.

Instructions `monitorenter` x and `monitorexit` x respectively locks and unlocks the object stored in the local variable x. As in JVML (and unlike the usual semantics of locks, i.e., unlike non-reentrant locks), a thread can lock the same object more than once without unlocking it. An object has a lock counter to record how many times it has been locked. The lock counter is incremented and decremented respectively when `monitorenter` and `monitorexit` are executed, and the object becomes unlocked when the counter becomes 0.

We assume that every method body is well-formed, i.e., that if $B(l) = \texttt{if}\ l'$ then $l' \in dom(B)$ holds and that if $B(l) \neq \texttt{athrow}, \texttt{return}$ then $l + 1 \in dom(B)$.

An *exception table* E is a total mapping from $dom(B) (\subset \mathcal{A})$ to $\mathcal{A}$. If an exception is raised at address l, the control jumps to address $E(l)$. We do *not* require that $E(l) \in dom(B)$. The case $E(l) \notin dom(B)$ expresses that an exception handler is not defined for the address l. In that case, while the exception is propagated to the callee of the method in JVML, the thread is just aborted in our model.

Restriction on our language: As defined above, we assume that each class has only one method *run* and it is invoked only by the instruction `start` . (Thus, a thread terminates when it executes the instruction `return`.) In other words, there is no normal method invocation. Our type system introduced in the next section checks that all the locks that have been acquired in a method execution will be released within the same method. It is easy to extend our model to incorporate method invocations without any significant change of the type system; as far as locks are considered, a method invocation can be regarded as a nop. Because of this reason, we have also ignored the subclass relation (every class is treated as a direct sub-class of `Thread`). If we were to allow non-structured locking (where a lock acquired in one method may be released in another method), the restriction above (of not having method invocations) would be significant. We shall discuss how to deal with non-structured locking in Section 7.

For the sake of simplicity, we do not consider null pointer and array bound exceptions, and assume that only `new` σ and `athrow` may throw exceptions. The `athrow` instruction always raises an exception and instruction `new` σ may throw an exception when allocation or initialization fails. We also assume that there is only a single kind of exception. Section 6 discusses how those restrictions are removed in the actual implementation of our verifier.

2.2 The operational semantics of $JVML_L$

We define an operational semantics of the language in a manner similar to previous formalizations of JVML [2, 25, 19].

To define the semantics formally, we define several notations. First, we define notations about functions and stacks. we write $dom(f)$ and $codom(f)$ for the domain and the co-domain of function f, respectively. Let $f\{x \mapsto v\}$ denotes the function such that $dom(f\{x \mapsto v\}) = dom(f) \cup \{x\}$, $(f\{x \mapsto v\})(y) = f(y)$ if $y \neq x$, and $(f\{x \mapsto v\})(x) = v$. $f \setminus x$ denotes the function such that $dom(f \setminus x) = dom(f) \setminus \{x\}$ and $(f \setminus x)(y) = f(y)$ for each $y \in dom(f \setminus x)$. A *stack* is a partial mapping from $\mathcal{N}$ to **VAL** whose domain is of the form $\{i \in \mathcal{N} \mid 0 \leq i < n\}$ for some $n \in \mathcal{N}$. If s is a stack, $s(i)$ denotes the value stored at the i-th position of the stack. If s is a stack and v is a value, we write $v \cdot s$ for the stack defined by $(v \cdot s)(n+1) = s(n)$ and $(v \cdot s)(0) = v$. We write ϵ for the stack whose domain is empty.

Next, we define *values*,*objects*, and *array objects*. We write **I** for the set of integers. We assume that there is a countably infinite set **O** of *references* (to objects or arrays). A *value* is either an integer or a reference. We write **VAL** for the set $\mathbf{I} \cup \mathbf{O}$ of values. An *object* is a record of the form $[\mathtt{class} = \sigma, \mathtt{flag} = b, a_1 = v_1 : d_1, \cdots, a_m = v_m : d_m]$, where σ denotes the class name of the object, and b is either 0, indicating that the object is not locked, or 1, indicating that the object is locked. If $\rho = [\mathtt{class} = \sigma, \mathtt{flag} = b, a_1 = v_1 : d_1, \cdots, a_m = v_m : d_m]$, we write $\rho.\mathtt{class}$, $\rho.\mathtt{flag}$ and $\rho.a_i$ for σ, b and v_i respectively. We also write $\rho\{a \mapsto v\}$ ($\rho\{\mathtt{flag} \mapsto b\}$, resp.) for the record obtained by replacing a value stored in field a (`flag`, resp.) of record ρ with v (b, resp.). An *array* is a record of the form $[\mathtt{class} : A, \mathtt{flag} : b, \mathtt{1} = v_1 : d, \cdots, \mathtt{m} = v_m : d]$, where A is of the form

$d[\ ]$ and denotes the array class name of the array, and m is the length of the array. We write **RCD** for the set of objects and arrays and use a meta-variable ρ to denote an element of the set.

As stated in Section 2.1, a $JVML_L$ program is executed by threads. We express a *thread state* with a tuple

$$\langle l, f, s, z, \sigma \rangle.$$

Here, $l (\in \mathcal{A})$ denotes the current program counter, f maps each local variable to the value stored in the variable, s is a stack, and z maps each heap address o to a natural number expressing how many locks the thread holds for the object pointed to by o (in other words, how many locks of the object the thread needs to release in future). σ is the class name of the thread.

We write **T** for the set of thread states. We extend a partial mapping z to a total mapping $z^{\#}$ by:

$$z^{\#}(o) = \begin{cases} z(o) & o \in dom(z) \\ 0 & o \notin dom(z) \end{cases}$$

Unless it is confusing, we write z for $z^{\#}$.

A *machine state* is a pair

$$\langle \Psi, H \rangle$$

where Ψ is a partial mapping from the set of natural numbers to **T**, and H is a partial mapping from **O** to the set **RCD** of objects. $\Psi(i)$ represents the state of the thread whose identifier is i. $H(o)$ denotes the object pointed to by reference o. We assume that the execution of a program starts when the method of class *main* is invoked, and that the method has no argument. So, the initial machine state is represented by

$$\langle \{0 \mapsto \langle 1, \emptyset, \epsilon, \emptyset, main_P \rangle\}, \emptyset \rangle.$$

where, address 1 denotes the first instruction of the method `run` of the main class defined in program P.

We define the operational semantics of $JVML_L$ using a one-step reduction relation

$$P \vdash \langle \Psi, H \rangle \rightarrow \langle \Psi', H' \rangle$$

The relation $P \vdash \langle \Psi, H \rangle \rightarrow \langle \Psi', H' \rangle$ means that a machine state $\langle \Psi, H \rangle$ changes to $\langle \Psi', H' \rangle$ in one-step execution of program P. It is defined as the least relation closed under the rules in Figures 3 and 4. In the figures, $P[\sigma](l)$

denotes the instruction at address l of the method of σ-class thread in P, i.e., $B(l)$ if $P(\sigma) = (FD, D, B, E)$, and $1_{\sigma'}$ denotes the address of the first instruction of the method `run` of the σ class. We denote by $\bar{t}$ an element of the set $\mathbf{T}$. If $i \notin dom(\Psi)$ then $\Psi \uplus \{i \mapsto \bar{t}\}$ denotes a mapping defined by:

$$\Psi \uplus \{i \mapsto \bar{t}\}(i') = \begin{cases} \bar{t} & i' = i \\ \Psi(i') & i' \neq i \end{cases}$$

We explain some key rules.

Rules $(ment_1)$, $(ment_2)$: These are the rules for acquiring a lock. The rule $(ment_1)$ states that a thread can acquire the lock of an object if the object is not locked. The rule $(ment_2)$ states that a thread can acquire the lock of an object if the lock is held by the same thread.

Rules $(mext_1)$, $(mext_2)$: These are the rules for releasing a lock. The rule $(mext_1)$ covers the case where the thread has acquired the lock only once before; in this case, the object becomes unlocked. The rule $(mext_2)$ covers the other case, where a thread has acquired the lock more than once; in this case, the lock counter is just decremented.

Rules (new), (new_{exc}): These are the rules for creating (and initializing) a new object. The rule (new) covers the case where the object creation succeeds. Unlike JVM, the created object is also initialized by the instruction. The relation $P \vdash H$ `ok` is defined in Figure 2. The rule (new_{exc}) covers the case where the object creation fails and an exception is raised.

In the operational semantics, a thread may get stuck in the following situations

- *Type mismatch* : The type of an operand does not math the type specified by the current instruction (e.g. the rule (*putfield*)). [*2]
- *Uncaught exceptions* : An exception is raised by the current instruction, but, there is no handler. This occurs when $P[\sigma](l)$ is undefined after an application of the rule (new_{exc}) or $(throw)$.
- *Lock error* : The current instruction is `return`, but the thread has not released the lock and the current instruction is `monitorexit`, but the

[*2] In the actual JVM, a type mismatch raises an exception instead of the execution getting stuck; This discrepancy is not important since such programs are rejected by the original bytecode verifier as well as by the type system we describe later.

$$\frac{H(c).\mathtt{class} = d \vee (c \in \mathbf{I} \wedge d = \mathbf{Int})}{\vdash_H o : d}$$

$$\frac{\overline{\sigma_P} = a_1 : d_1, \ldots, a_m : d_m \qquad \vdash_H v_1 : d_1, \ldots, \vdash_H v_m : d_m}{P, H \vdash [\mathtt{class} = \sigma, \mathtt{flag} = b, a_1 = v_1 : d_1, \cdots, a_m = v_m : d_m]\ \mathtt{ok}}$$

$$\frac{\vdash_H v_1 : d, \ldots, \vdash_H v_m : d}{P, H \vdash [\mathtt{class} = d[\,], \mathtt{flag} = b, 1 = v_1 : d, \cdots, m = v_m : d]\ \mathtt{ok}}$$

$$\frac{\forall x \in dom(H). P, H \vdash H(x)\ \mathtt{ok}}{P \vdash H\ \mathtt{ok}}$$

Fig. 2 Definition of $P \vdash H$ ok

thread has not acquired the lock.

Our type system in this paper guarantees that, during the execution of a well-typed program, no thread gets stuck because of "type mismatch" or "lock error" and a thread may get stuck because of "uncaught exceptions", but at that time, the thread has released all the locks it acquired.

§3 Type System

In this section, we give a type system for checking that programs use lock primitives safely. As mentioned in Section 1, we extend an object type with a usage expression, which represents in which order the object is locked and unlocked. We first introduce usages and types in Section 3.1. In Section 3.2, we define relations on usages and types. In Section 3.3 and 3.4, we give typing rules for the extended types. Our type system is an extension of Stata and Abadi's type system [25] for JVML.

3.1 Usages and types

As mentioned above, we augment the type of an object with a usage expression, which expresses how the object will be locked and unlocked.

Definition 3.1 (usages)

The set $\mathcal{U}$ of usage expressions (*usages*, in short) is defined by:

$$U ::= \mathbf{0} \mid \alpha \mid L.U \mid \widehat{L}.U \mid U_1 \otimes U_2 \mid U_1 \& U_2 \mid \mu\alpha.U \mid \bot_{\mathbf{rel}}$$

$$\frac{P[\sigma](l) = \mathtt{inc} \quad c \in \mathbf{I}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, c \cdot s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, c+1 \cdot s, z, \sigma\rangle\}, H\rangle} \quad (\mathit{inc})$$

$$\frac{P[\sigma](l) = \mathtt{push0}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, 0 \cdot s, z, \sigma\rangle\}, H\rangle} \quad (\mathit{push}0)$$

$$\frac{P[\sigma](l) = \mathtt{pop}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, v \cdot s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z, \sigma\rangle\}, H\rangle} \quad (\mathit{pop})$$

$$\frac{P[\sigma](l) = \mathtt{if}\ l' \quad c \in \mathbf{I}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, 0 \cdot s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z, \sigma\rangle\}, H\rangle} \quad (\mathit{if}_{\mathit{proceed}})$$

$$\frac{P[\sigma](l) = \mathtt{if}\ l' \quad c \in \mathbf{I} \quad c \neq 0}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, v \cdot s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l', f, s, z, \sigma\rangle\}, H\rangle} \quad (\mathit{if}_{\mathit{branch}})$$

$$\frac{P[\sigma](l) = \mathtt{load}\ x}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, f(x) \cdot s, z, \sigma\rangle\}, H\rangle} \quad (\mathit{load})$$

$$\frac{P[\sigma](l) = \mathtt{store}\ x}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, v \cdot s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f\{x \mapsto v\}, s, z, \sigma\rangle\}, H\rangle} \quad (\mathit{store})$$

$$\frac{P[\sigma](l) = \mathtt{new}\ \sigma' \quad H'(o).\mathtt{class} = \sigma' \quad \forall x \in \mathit{dom}(H').H'(x).\mathtt{flag} = 0 \qquad P \vdash H'\ \mathtt{ok}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, o \cdot s, z, \sigma\rangle\}, H \uplus H'\rangle} \quad (\mathit{new})$$

$$\frac{P[\sigma](l) = \mathtt{new}\ \sigma' \quad P(\sigma) = (\mathit{FD}, D, B, E) \quad E(l) = l'}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l', f, \epsilon, z, \sigma\rangle\}, H\rangle} \quad (\mathit{new}_{\mathit{exc}})$$

$$\frac{\begin{array}{c} P[\sigma](l) = \mathtt{start}\ \sigma' \quad P(\sigma) = (\mathit{FD}, D, B, E) \quad j \notin \mathit{dom}(\Psi) \cup \{i\} \\ o \in \mathit{dom}(H) \quad H(o).\mathtt{class} = \sigma' \quad \mathit{dom}(D) = \{0, \ldots, n-1\} \quad f' = \emptyset\{0 \mapsto v_0, \ldots, n-1 \mapsto v_{n-1}\} \end{array}}{P \vdash \begin{array}{c} \langle \Psi \uplus \{i \mapsto \langle l, f, v_0 \cdot, \ldots, \cdot v_{n-1} \cdot o \cdot s, z, \sigma\rangle\}, H\rangle \rightarrow \\ \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z, \sigma\rangle\} \uplus \{j \mapsto \langle 1_{\sigma'}, f', \epsilon, \emptyset, \sigma'\rangle\}, H\rangle \end{array}} \quad (\mathit{start})$$

$$\frac{P[\sigma](l) = \mathtt{athrow} \quad P(\sigma) = (\mathit{FD}, D, B, E) \quad E(l) = l'}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l', f, \epsilon, z, \sigma\rangle\}, H\rangle} \quad (\mathit{throw})$$

$$\frac{P[\sigma](l) = \mathtt{return} \quad \forall o \in \mathit{dom}(H).z^{\#}(o) = 0}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi, H\rangle} \quad (\mathit{return})$$

Fig. 3 Operational semantics

$$\frac{\begin{array}{c}P[\sigma](l) = \texttt{monitorenter}\ x \quad f(x) \in \mathit{dom}(H) \quad z^{\#}(f(x)) = 0 \\ H(f(x)).\mathit{flag} = 0 \quad H' = H\{f(x) \mapsto \rho\} \quad \rho = H(o)\{\mathit{flag} \mapsto 1\}\end{array}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \to \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z\{f(x) \mapsto 1\}, \sigma\rangle\}, H'\rangle} \ (\mathit{ment}_1)$$

$$\frac{P[\sigma](l) = \texttt{monitorenter}\ x \quad f(x) \in \mathit{dom}(H) \quad z^{\#}(f(x)) = n \geq 1 \quad H(f(x)).\mathit{flag} = 1}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \to \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z\{f(x) \mapsto n+1\}, \sigma\rangle\}, H\rangle} \ (\mathit{ment}_2)$$

$$\frac{\begin{array}{c}P[\sigma](l) = \texttt{monitorexit}\ x \quad f(x) \in \mathit{dom}(H) \quad z^{\#}(f(x)) = 1 \\ H(f(x)).\mathit{flag} = 1 \quad H' = H\{f(x) \mapsto \rho\} \quad \rho = H(o)\{\mathit{flag} \mapsto 0\}\end{array}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \to \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z \setminus f(x), \sigma\rangle\}, H'\rangle} \ (\mathit{mext}_1)$$

$$\frac{P[\sigma](l) = \texttt{monitorexit}\ x \quad f(x) \in \mathit{dom}(H) \quad z^{\#}(f(x)) = n \geq 2 \quad H(f(x)).\mathit{flag} = 1}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \to \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z\{f(x) \mapsto n-1\}, \sigma\rangle\}, H\rangle} \ (\mathit{mext}_2)$$

$$\frac{P[\sigma](l) = \texttt{getfield}\ \sigma'.a\ d \quad o \in \mathit{dom}(H) \quad H(o).\texttt{class} = \sigma' \quad H(o).a = v}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, o \cdot s, z, \sigma\rangle\}, H\rangle \to \langle \Psi \uplus \{i \mapsto \langle l+1, f, v \cdot s, z, \sigma\rangle\}, H\rangle} \ (\mathit{getfld})$$

$$\frac{\begin{array}{c}P[\sigma](l) = \texttt{putfield}\ \sigma'.a\ d \quad o \in \mathit{dom}(H) \quad H(o).\texttt{class} = \sigma' \\ H' = H\{o \mapsto \rho\} \quad \rho = H(o)\{a \mapsto v\}\end{array}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, v \cdot o \cdot s, z, \sigma\rangle\}, H\rangle \to \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z, \sigma\rangle\}, H'\rangle} \ (\mathit{putfld})$$

$$\frac{P[\sigma](l) = \texttt{aaload} \quad o \in \mathit{dom}(H) \quad H(o).\texttt{class} = d[\,] \quad H(o).c = v}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, c \cdot o \cdot s, z, \sigma\rangle\}, H\rangle \to \langle \Psi \uplus \{i \mapsto \langle l+1, f, v \cdot s, z, \sigma\rangle\}, H\rangle} \ (\mathit{aload})$$

$$\frac{\begin{array}{c}P[\sigma](l) = \texttt{aastore} \quad o \in \mathit{dom}(H) \quad H(o).\texttt{class} = d[\,] \\ H' = H\{o \mapsto \rho\} \quad \rho = H(o)\{c \mapsto v\}\end{array}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, v \cdot c \cdot o \cdot s, z, \sigma\rangle\}, H\rangle \to \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z, \sigma\rangle\}, H'\rangle} \ (\mathit{astore})$$

Fig. 4 Operational semantics

Usage $\mathbf{0}$ describes an object that cannot be locked or unlocked at all. α denotes a usage variable, which is bound by a constructor $\mu\alpha$. Usage $L.U$ describes an object that is first locked and then used according to U. Usage $\widehat{L}.U$ describes an object that is first unlocked and then used according to U. Usage $U_1 \otimes U_2$ describes an object that is used according to U_1 and U_2 in an interleaved manner. For example, $L \otimes \widehat{L}$ describes an object that will be either locked and then unlocked, or unlocked and then locked. The constructor $\otimes$ is used to approximate the whole locking behavior when an object is locked through aliases. For example, if an object is locked and unlocked according to U_1 through variable x, and if the same object is also locked and unlocked according to U_2 through variable y, then, the whole locking behavior is approximated by $U_1 \otimes U_2$. Note that without such approximation, one has to analyze the program for every possible alias pattern. [*3] $U_1 \& U_2$ describes an object that can be used according to either U_1 or U_2. Usage $\mu\alpha.U$ describes an object that can be recursively used according to $[\mu\alpha.U/\alpha]U$ (where $[U_1/\alpha]U_2$ denotes the usage obtained by replacing every free occurrence of α with U_1). For example, $\mu\alpha.(\mathbf{0}\&L.\alpha)$ describes an object that is locked an arbitrary number of times. Usage $\bot_{\mathbf{rel}}$ describes an object that is locked and unlocked properly. We will assign $\bot_{\mathbf{rel}}$ to elements of arrays and object fields to ensure that after they are extracted from objects or arrays, they are properly locked and unlocked. Although the usage expression $\bot_{\mathbf{rel}}$ is equal to $\mu\alpha.(\mathbf{0}\&(L.\widehat{L} \otimes \alpha))$, we introduce $\bot_{\mathbf{rel}}$ for a technical reason.

We often write L and $\widehat{L}$ for $L.\mathbf{0}$ and $\widehat{L}.\mathbf{0}$ respectively. We give higher precedence to unary operators $L.$, $\widehat{L}.$, and $\mu\alpha.$ than to binary operators, so that $L.\widehat{L}\&L.\widehat{L}$ means $(L.\widehat{L})\&(L.\widehat{L})$ rather than $L.(\widehat{L}\&L.\widehat{L})$.

A *usage context* is an expression obtained by replacing some sub-expressions of a usage with holes $[\,]$. We use a meta-variable C to denote a usage context. The expression $C[U_1, \ldots, U_n]$ denotes the usage obtained by substituting $U_1, \ldots, U_n$ for the holes in the context C from left to right. For example, if $C = [\,] \otimes [\,]$, then $C[U_1, U_2] = U_1 \otimes U_2$. We assume that the free usage variables of $U_1, \ldots, U_n$ are different from the bound variables in C. So, if $C = \mu\alpha.[\,]$, then $C[\alpha] = \mu\alpha'.\alpha$.

Definition 3.2
The binary relation $\equiv$ on usages is the least congruence relation that satisfies

[*3] In some cases, alias analysis may be able to determine that a single alias pattern is possible; in that case, the approximation can be avoided. That is not always possible, however: consider a case where a method is called in multiple alias patterns such as $m(o_1, o_2)$ and $m(o_1, o_1)$.

the associativity and commutativity laws on $\otimes$ and &, and the rules $U \otimes \mathbf{0} \equiv U$ and $\mu\alpha.U \equiv [\mu\alpha.U/\alpha]U$.

Remark 3.1
Usage expressions form a small process calculus. In fact, usages can be regarded as *basic parallel processes* [4] with two labels L and $\hat{L}$. The usage constructors $\otimes$ and & correspond to parallel composition and choice. Our usage expressions are more expressive than regular expressions because of the constructor $\otimes$, which corresponds to the shuffle operator in formal languages.

We define types as follows:

Definition 3.3 (Types)
The set $\mathcal{T}$ of *types* is defined by:

$$\text{(types) } \tau ::= \mathbf{Int} \mid \sigma/U \mid \xi[\,]/U \mid \mathbf{Top}$$

$$\text{(element types) } \xi ::= \mathbf{Int} \mid \sigma \mid \xi[\,] \mid \mathbf{Top}$$

Int is the type of integers. **Top** is the type of objects that cannot be used at all. Type σ/U describes an object of class σ that is locked and unlocked according to the usage U. Type $\xi[\,]/U$ describes an array that has elements of type ξ and is locked/unlocked according to U.

Example 3.1
Type `Counter`$/L.\widehat{L}$ describes an object of `Counter` class that is first locked and then unlocked. Type `Account`$/L.(\widehat{L}\&\mathbf{0})$ describes an object of `Account` class that is first locked and then either unlocked or no longer accessed. Type `Counter`$[\,]/L.\widehat{L}$ is the type of an array that is first locked and then unlocked and have `Counter` class objects that are locked and unlocked properly as its elements.

3.2 Reliability of usages and relation on types

As is understood from Example 3.1, the usage of an object expresses whether the object is locked and unlocked properly. The usage of the `Counter` object in the example expresses a proper usage. On the other hand the *usage* of the `Account` object expresses an incorrect usage: The lock of the object may not be released. We say that a *usage* U is *reliable* and write $\mathit{rel}(U)$ if it expresses safe usage of lock primitives, in the sense that each lock operation is followed by an unlock operation and that each unlock operation is preceded by a lock operation.

To formally define $rel(U)$, we consider reduction of pairs $\langle U, n\rangle$ consisting of a usage U and a natural number n. A pair $\langle U, n\rangle$ represents the state of an object that has been locked n times by a thread so far and will be used according to usage U by the thread from now.

Definition 3.4
The usage pair reduction $\rightarrow_{rel}$ is the least binary relation on $\mathcal{U} \times \mathcal{N}$ closed under the following rules.

$$\langle L.\,U, n\rangle \rightarrow_{rel} \langle U, n+1\rangle \quad \langle \widehat{L}.\,U, n\rangle \rightarrow_{rel} \langle U, n-1\rangle \quad \langle \bot_{\mathbf{rel}}, n\rangle \rightarrow_{rel} \langle \mathbf{0}, n\rangle$$

$$\frac{\langle U_1, n\rangle \rightarrow_{rel} \langle U_1', n'\rangle}{\langle U_1 \& U_2, n\rangle \rightarrow_{rel} \langle U_1', n'\rangle} \qquad \frac{\langle U_2, n\rangle \rightarrow_{rel} \langle U_2', n'\rangle}{\langle U_1 \& U_2, n\rangle \rightarrow_{rel} \langle U_2', n'\rangle}$$

$$\frac{\langle U_1, n\rangle \rightarrow_{rel} \langle U_1', n'\rangle}{\langle U_1 \otimes U_2, n\rangle \rightarrow_{rel} \langle U_1' \otimes U_2, n'\rangle} \qquad \frac{U_1 \equiv U_1' \quad \langle U_1', n\rangle \rightarrow_{rel} \langle U_2', n'\rangle \quad U_2' \equiv U_2}{\langle U_1, n\rangle \rightarrow_{rel} \langle U_2, n'\rangle}$$

Let $\rightarrow_{rel}^*$ be the reflexive and transitive closure of $\rightarrow_{rel}$.

We can now define the reliability of usages as follows:

Definition 3.5 (Reliability of usages)
$rel(U, n)$ is defined to hold if the following all conditions hold whenever $\langle U, n\rangle \rightarrow_{rel}^* \langle U', n'\rangle$:

1. if $U' \equiv \mathbf{0}$ or $U' \equiv \mathbf{0}\ \& U_1$ for a usage U_1, then $n' = 0$
2. if $U' \equiv (\widehat{L}.\,U_1 \otimes U_2)$ or $U' \equiv (\widehat{L}.\,U_1 \otimes U_2) \& U_3$ for some usages U_1, U_2, and U_3, then $n' \geq 1$

A usage U is *reliable*, written $rel(U)$, if $rel(U, 0)$ holds.

The condition *1* in Definition 3.5 states that if an object may no longer be locked or unlocked by a thread (i.e. if $U' \equiv \mathbf{0}$ or $U' \equiv \mathbf{0} \& U_1$), the lock of the object is not held by the thread (i.e. $n' = 0$). The condition *2* in the definition states that when the lock of an object may be released by a thread (i.e. $U' \equiv (\widehat{L}.\,U_1 \otimes U_2)$ or $U' \equiv (\widehat{L}.\,U_1 \otimes U_2) \& U_3$), the lock is currently held by the thread (i.e. $n' \geq 1$). These conditions guarantee the proper use of lock primitives: (1) when a thread terminates normally or abruptly, it has released all the locks it has acquired, (2) a thread releases a lock only if it holds the lock.

Example 3.2

$L.L.(\widehat{L} \otimes \widehat{L})$, $(L.\widehat{L})\&(L.\widehat{L})$, $L.\widehat{L}.\bot_{\mathbf{rel}}$ and $L.\mu\alpha.((\widehat{L}.L.\alpha)\&\widehat{L})$ are reliable. Neither $L.(L \otimes \widehat{L})$ nor $L.\widehat{L}.\widehat{L}$ is reliable.

We extend the predicate *rel* to a predicate rel_t on types. It is defined as the least unary relation closed under the following rules:

$$\frac{}{rel_t(\mathbf{Int})} \quad \frac{}{rel_t(\mathbf{Top})} \quad \frac{rel(U)}{rel_t(\sigma/U)} \quad \frac{rel(U)}{rel_t(\xi[\,]/U)}$$

The following relation $U_1 \& U_2$ means that U_1 expresses a more general usage than U_2, so that an object of usage U_1 may be used according to U_2.[*4]

Definition 3.6
The *sub-usage relation* $\leq$ is the least preorder on usages that includes the relation $\equiv$ and is closed under the following rules:

$$U_1 \& U_2 \leq U_1 \quad \frac{rel(U)}{\bot_{\mathbf{rel}} \leq U} \quad \frac{U_i \leq U_i'}{C[U_1, \ldots, U_n] \leq C[U_1', \ldots, U_n']}$$

Here, we define several relations and operations on types to simplify our type system. At first, we extend the congruence relation on usages to the congruence relation $\tau_1 \equiv \tau_2$ on types.

Definition 3.7
The binary relation $\equiv$ on types is the least equivalence relation that satisfies the following rules:

$$\frac{U_1 \equiv U_2}{\sigma/U_1 \equiv \sigma/U_2} \quad \frac{\xi_1 = \xi_2 \quad U_1 \equiv U_2}{\xi_1[\,]/U_1 \equiv \xi_2[\,]/U_2}$$

Similarly, we extend the sub-usage relation to a subtype relation $\tau_1 \leq \tau_2$ on types. By $\tau_1 \leq \tau_2$ we denote that a value of type τ_1 can be used as a value of type τ_2.

[*4] Since usages can be regarded as processes, an alternative approach is to define $U_1 \& U_2$ as a simulation relation or a trace inclusion relation. We preferred the axiomatic definition in this paper for a technical convenience.

Definition 3.8
The *subtype* relation is the least preorder that includes the relation $\equiv$ on types and is closed under the following rules:

$$\mathbf{Int} \leq \mathbf{Top} \qquad \frac{U \leq \mathbf{0}}{\sigma/U \leq \mathbf{Top}} \quad \frac{U_1 \leq U_2}{\sigma/U_1 \leq \sigma/U_2}$$

$$\frac{U \leq \mathbf{0}}{\xi[\,]/U \leq \mathbf{Top}} \quad \frac{U_1 \leq U_2}{\xi[\,]/U_1 \leq \xi[\,]/U_2}$$

Note that by the definition, $\sigma_1/U_1 \leq \sigma_2/U_2$ implies $\sigma_1 = \sigma_2$. This is because we do not have a subclass relation between elements of Σ. If there is a subclass relation, the subtyping relation above should be accordingly refined.

Next, we define several operations on types. To simplify these definitions, we use $*$ to represent a binary operator $\otimes$ or $\&$ and use $\dot{L}.$ to represent a unary operator $L.$ or $\widehat{L}.$.

Definition 3.9
We define $\tau_1 * \tau_2$, $\dot{L}.\tau$ by:

$$\begin{array}{lcl} \mathbf{Top} * \mathbf{Top} & = & \mathbf{Top} \\ \mathbf{Int} * \mathbf{Int} & = & \mathbf{Int} \\ (\sigma/U_1) * (\sigma/U_2) & = & \sigma/(U_1 * U_2) \\ (\xi[\,]/U_1) * (\xi[\,]/U_2) & = & \xi[\,]/(U_1 * U_2) \\ \dot{L}.(\sigma/U) & = & \sigma/(\dot{L}.U) \\ \dot{L}.(\xi[\,]/U) & = & \xi[\,]/(\dot{L}.U) \end{array}$$

The operation is undefined for the arguments that do not match the above definition.

Definition 3.10
We define $\tau_1 \leq_{\dot{L}} \tau_2$ by:

$$\tau_1 \leq_{\dot{L}} \tau_2 \Leftrightarrow \begin{cases} & (\tau_1 \leq \dot{L}.\tau_2) \\ \vee & (\tau_2 = \mathbf{Top} \ \wedge \ \exists\sigma.\tau_1 = \sigma/\dot{L}.\mathbf{0}) \\ \vee & (\tau_2 = \mathbf{Top} \ \wedge \ \exists\xi.\tau_1 = \xi[\,]/\dot{L}.\mathbf{0}) \end{cases}$$

The relations $\tau_1 \leq_L \tau_2$ and $\tau_1 \leq_{\widehat{L}} \tau_2$ above mean that after values of type τ_1 are locked or unlocked respectively, these values will be used as values of type

τ_2. For example, $\texttt{Counter}/L.\widehat{L}.\mathbf{0} \leq_L \texttt{Counter}/\widehat{L}.\mathbf{0}$ and $\texttt{Counter}/\widehat{L}.\mathbf{0} \leq_{\widehat{L}} \textbf{Top}$ and $\texttt{Counter}[\,]/\widehat{L}.\mathbf{0} \leq_{\widehat{L}} \textbf{Top}$ hold.

We also define the function $\mathit{Use}(\tau)$ on types as follows:

$$\mathit{Use}(\tau) = \begin{cases} U & \tau = \sigma/U \\ U & \tau = \xi[\,]/U \\ \mathbf{0} & \tau = \textbf{Top} \\ \text{undefined} & \text{otherwise} \end{cases}$$

3.3 Type environments

A *frame type*, denoted by a meta-variable F, is a partial mapping from $\mathcal{V}$ to $\mathcal{T}$. $F(x)$ denotes the type of the value stored in the local variable x.

A *stack type*, denoted by a meta-variable S, is a partial mapping from $\mathcal{N}$ to $\mathcal{T}$. $S(n)$ denotes the type of the n-th value stored in the operand stack. We write ϵ for the type of the empty stack. A stack type $\tau \cdot S$ is defined by $(\tau \cdot S)(n+1) = S(n)$ and $(\tau \cdot S)(0) = \tau$.

A *frame type environment*, denoted by a meta-variable $\mathcal{F}$, is a mapping from $\mathcal{A}$ to the set of frame types. $\mathcal{F}(l)$ describes the types of values stored in local variables just before the program address l is executed. Similarly, a *stack type environment*, denoted by a meta-variable $\mathcal{S}$, is a mapping form $\mathcal{A}$ to the set of stack types. $\mathcal{S}(l)$ describes the types of values stored in the operand stack just before the program address l is executed. For example, $\mathcal{F}(l)(x) = \sigma/\widehat{L}$ means that σ-class object is stored in the local variable x at program address l, and the lock on the object will be released afterwards. We often write $\mathcal{F}_l$ and $\mathcal{S}_l$ for $\mathcal{F}(l)$ and $\mathcal{S}(l)$ respectively.

We extend some operations and relations on types to those on *frame types* or *stack types.*

Definition 3.11

Suppose that $\mathit{dom}(F_1) = \mathit{dom}(F_2)$. Then $F_1 * F_2$ is defined by:

$$\mathit{dom}(F_1 * F_2) = \mathit{dom}(F_1)$$
$$\forall x \in \mathit{dom}(F_1).(F_1 * F_2)(x) = (F_1(x)) * (F_2(x))$$

Definition 3.12
A frame type F_1 is a subtype of F_2, written $F_1 \leq F_2$, if:

$$dom(F_1) = dom(F_2)$$
$$\forall x \in dom(F_1).(F_1(x) \leq F_2(x))$$

We also write $F \leq \mathbf{Top}$ if $F(x) \leq \mathbf{Top}$ holds for each $x \in dom(F)$.

The operations $S_1 * S_2$ and the relations $S_1 \leq S_2$ and $S \leq \mathbf{Top}$ are defined in a similar manner.

3.4 Typing rules

We consider a judgment of the form $\langle \mathcal{F}, \mathcal{S} \rangle \vdash_P (D, B, E)$. It means that the method (D, B, E) is well-typed under the assumption that the values stored in local variables and the operand stack have the types described by $\mathcal{F}$ and $\mathcal{S}$ and the values in object fields have the types indicated by class definitions in program P.

To define the relation above, we introduce relations $\mathcal{F}, \mathcal{S}, l \vdash_P (D, B, E)$. Intuitively, it means that the instruction at l can be safely executed on the assumption that the values stored in local variables and the operand stack have the types described by $\mathcal{F}$ and $\mathcal{S}$ and the values in object fields have the types indicated by class definitions in program P.

Definition 3.13
$\mathcal{F}, \mathcal{S}, l \vdash_P (D, B, E)$ is the least relation closed under the rules in Figure 5.

In Figure 5 and 6 , $\mathcal{F}_l$ and $\mathcal{S}_l$ are shorthand notations for $\mathcal{F}(l)$ and $\mathcal{S}(l)$ respectively.

We explain several rules below:

Rule (MENTR): The first line states that the instruction at address l is `monitorenter`. The second line states that an instruction exists at the next address $l + 1$. Since the object stored in local variable x is locked at this address and then used according to $\mathcal{F}_{l+1}(x)$, the object is accessed according to $L.\mathcal{F}_{l+1}(x)$ in total. The fourth line expresses this condition. The third line also says that the types of the values stored in the other local variables at address l are subtypes of those at address $l + 1$, since those values are not accessed at l. Similarly, since the stack is not accessed, the stack type at l should be a subtype the stack type at $l + 1$.

Rule (IF): The first line states that the instruction at address l is `if` l'. The second line states that there are instructions at addresses l' and $l + 1$. Since the values stored in local variables are not accessed at l, they are accessed according to either $\mathcal{F}_{l+1}$ or $\mathcal{F}_{l'}$, depending on which branch is taken. The third line expresses this condition. The fourth line expresses the condition that the stack top at address l must be an integer and the condition that the other values stored in the stack are accessed according to either $\mathcal{S}_{l+1}$ or $\mathcal{S}_{l'}$.

Rule (ATHROW): The first line states that the instruction at address l is `athrow`. Since the control jumps to $E(l)$, it must be the case that $E(l) \in dom(B)$, as specified in the second line. The values stored in local variables are not accessed at l and they are accessed according to $E(l)$. This condition is expressed by the third line. The fourth line expresses the condition that all values stored in the stack are not accessed afterwards, since the operand stack becomes empty when the exception is raised.

Rule (PUTFIELD): The first and second lines state that the instruction at address l is `putfield` $\sigma.a$ d and σ-classes definition in program P has field a of descriptor d ($d \neq \mathbf{Int}$). This instruction pops two values form the operand stack and stores the first value into field a of the second value. Here, the first value must be an object or an array that will be locked and unlocked properly, because we assume that elements of objects and arrays are locked and unlocked properly after they are extracted from objects or arrays. The fifth line expresses this condition.

Rule (NEW): The third line states that the values stored in local variables are not accessed at l, they are accessed according to either $\mathcal{F}_{l+1}$ or $\mathcal{F}_{E(l)}$, depending on whether an asynchronous exception is raised or not. The condition $rel(U)$ in forth line states that an object created by the `new` σ instruction have to locked and unlocked properly later. The condition $\mathcal{S}_l \leq \mathbf{Top}$ states that the values stored in the operand stack may be not accessed later. Actually, if an exception is raised, then all values in the operand stack are popped, therefore, this condition is necessary.

Now we define the type judgment relation for methods.

Definition 3.14 (Type judgment for methods)

The relation $\langle \mathcal{F}, \mathcal{S} \rangle \vdash_P (B, E, D)$ is defined by the following rule:

$$\frac{\begin{array}{c} \forall x \in \mathit{dom}(\mathcal{F}_1).\ \mathit{rel}_t(\mathcal{F}_1(x)) \\ \mathit{Raw}(\mathcal{F}_1(x)) = \begin{cases} D(x) & \text{if } x \in \mathit{dom}(D) \\ \mathbf{Top} & \text{otherwise} \end{cases} \\ \mathcal{S}_1 = \epsilon \\ \forall l \in \mathit{codom}(E).(\mathcal{S}(l) = \epsilon) \\ \forall l \in \mathit{dom}(B).\ \mathcal{F}, \mathcal{S}, l \vdash_P (D, B, E) \\ \forall l \notin \mathit{dom}(B).(\mathcal{F}_l \leq \mathbf{Top}\ \wedge\ \mathcal{S}_l \leq \mathbf{Top}) \end{array}}{\langle \mathcal{F}, \mathcal{S} \rangle \vdash_P (D, B, E)}$$

Here, $\mathit{Raw}(\tau)$ is defined by:

$$\begin{aligned} &\mathit{Raw}(\mathbf{Int}) = \mathbf{Int} \\ &\mathit{Raw}(\mathbf{Top}) = \mathbf{Top} \\ &\mathit{Raw}(\sigma/U) = \sigma \\ &\mathit{Raw}(\xi[\,]/U) = \xi[\,] \end{aligned}$$

In the rule above, the first premise enforces that all objects stored in local variables at the beginning of the method are safely used in the sense that a lock that is acquired during execution of the method is always released during the same method execution. The second premise states that the values stored in local variables at the beginning of the method must have the types specified by the method descriptor. The third and fourth premises state that the operand stack at the beginning of the method or at the beginning of an exception handler is empty. The fifth premise states that the method is well-typed at each address. The last premise covers the case when an exception handler is not defined within the method. In that case, the execution abruptly terminates (or in the real JVM, the exception is passed to the callee), so that there should be no locks to be released in the local variables and stack.

Definition 3.15 (Well-typed program)
A program P is well-typed if for each class name $\sigma \in \mathit{dom}(P)$, there exist $\mathcal{F}$ and $\mathcal{S}$ such that $P(\sigma) = (\mathit{FD}, D, B, E)$ and $\langle \mathcal{F}, \mathcal{S} \rangle \vdash_P (D, B, E)$ holds.

(INC)
$$\frac{\begin{array}{c} B(l) = \texttt{inc} \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \le \mathcal{F}_{l+1} \\ \mathcal{S}_l(0) \le \mathbf{Int} \quad \mathcal{S}_l \le \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(PUSH)
$$\frac{\begin{array}{c} B(l) = \texttt{push0} \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \le \mathcal{F}_{l+1} \\ \mathbf{Int} \cdot \mathcal{S}_l \le \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(POP)
$$\frac{\begin{array}{c} B(l) = \texttt{pop} \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \le \mathcal{F}_{l+1} \\ \mathcal{S}_l \le \mathbf{Top} \cdot \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, S, l \ \vdash_P (D, B, E)}$$

(IF)
$$\frac{\begin{array}{c} B(l) = \texttt{if } l' \\ l',\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \le \mathcal{F}_{l+1} \& \mathcal{F}_{l'} \\ \mathcal{S}_l \le \mathbf{Int} \cdot (\mathcal{S}_{l+1} \& \mathcal{S}_{l'}) \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(LOAD)
$$\frac{\begin{array}{c} B(l) = \texttt{load } x \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \le \mathcal{F}_{l+1}\{x \mapsto \mathcal{F}_{l+1}(x) \otimes \mathcal{S}_{l+1}(0)\} \\ \mathcal{S}_{l+1}(0) \cdot \mathcal{S}_l \le \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(STORE)
$$\frac{\begin{array}{c} B(l) = \texttt{store } x \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \le \mathcal{F}_{l+1}\{x \mapsto \mathbf{Top}\} \\ \mathcal{S}_l \le \mathcal{F}_{l+1}(x) \cdot \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(NEW)
$$\frac{\begin{array}{c} B(l) = \texttt{new } \sigma \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \le \mathcal{F}_{l+1} \& \mathcal{F}_{E(l)} \\ (\sigma/U) \cdot \mathcal{S}_l \le \mathcal{S}_{l+1} \quad \mathit{rel}(U) \\ \mathcal{S}_l \le \mathbf{Top} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(START)
$$\frac{\begin{array}{c} B(l) = \texttt{start } \sigma \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \le \mathcal{F}_{l+1} \\ \forall\ i \in \mathit{dom}(D^\sigma). D^\sigma(x) = \tau_i \\ \mathit{dom}(D^\sigma) = \{0, \ldots, n-1\} \\ \mathcal{S}_l \le \tau_0 \cdot, \ldots, \cdot \tau_{n-1} \cdot \sigma/\mathbf{0} \cdot \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(RETURN)
$$\frac{\begin{array}{c} B(l) = \texttt{return} \\ \mathcal{F}_l \le \mathbf{Top} \\ \mathcal{S}_l \le \mathbf{Top} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(MENTR)
$$\frac{\begin{array}{c} B(l) = \texttt{monitorenter } x \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \setminus x \le \mathcal{F}_{l+1} \setminus x \\ \mathcal{F}_l(x) \le_L \mathcal{F}_{l+1}(x) \\ \mathcal{S}_l \le \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(MEXT)
$$\frac{\begin{array}{c} B(l) = \texttt{monitorexit } x \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \setminus x \le \mathcal{F}_{l+1} \setminus x \\ \mathcal{F}_l(x) \le_{\hat{L}} \mathcal{F}_{l+1}(x) \\ \mathcal{S}_l \le \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

(ATHROW)
$$\frac{\begin{array}{c} B(l) = \texttt{athrow} \\ E(l) \in \mathit{dom}(B) \\ \mathcal{F}_l \le \mathcal{F}_{E(l)} \\ \mathcal{S}_l \le \mathbf{Top} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (D, B, E)}$$

Fig. 5 Typing rules

(PUTFLD$_{\mathbf{Int}}$)
$$\frac{B(l) = \texttt{putfield}\ \sigma.a\ \mathbf{Int} \quad \overline{\sigma_P}.a : \mathbf{Int} \quad l+1 \in \mathit{dom}(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \mathcal{S}_l \leq \mathbf{Int} \cdot (\sigma/\mathbf{0}) \cdot \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l\ \vdash_P (D, B, E)}$$

(PUTFLD)
$$\frac{B(l) = \texttt{putfield}\ \sigma.a\ d \quad d \neq \mathbf{Int} \quad \overline{\sigma_P}.a : d \quad l+1 \in \mathit{dom}(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \mathcal{S}_l \leq (d/\bot_{\mathbf{rel}}) \cdot (\sigma/\mathbf{0}) \cdot \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l\ \vdash_P (D, B, E)}$$

(GETFLD$_{\mathbf{Int}}$)
$$\frac{B(l) = \texttt{getfield}\ \sigma.a\ \mathbf{Int} \quad \overline{\sigma_P}.a : \mathbf{Int} \quad l+1 \in \mathit{dom}(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \mathcal{S}_l \leq (\sigma/\mathbf{0}) \cdot \mathcal{S}' \quad \mathbf{Int} \cdot \mathcal{S}' \leq \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l\ \vdash_P (D, B, E)}$$

(GETFLD)
$$\frac{B(l) = \texttt{getfield}\ \sigma.a\ d \quad d \neq \mathbf{Int} \quad \overline{\sigma_P}.a : d \quad l+1 \in \mathit{dom}(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \mathcal{S}_l \leq (\sigma/\mathbf{0}) \cdot \mathcal{S}' \quad (d/U) \cdot \mathcal{S}' \leq \mathcal{S}_{l+1} \quad \mathit{rel}(U)}{\mathcal{F}, \mathcal{S}, l\ \vdash_P (D, B, E)}$$

(ALOAD$_{\mathbf{Int}}$)
$$\frac{B(l) = \texttt{aaload} \quad l+1 \in \mathit{dom}(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \xi = \mathbf{Int}\ \mathit{or}\ \xi = \mathbf{Top} \quad \mathcal{S}_l \leq \mathbf{Int} \cdot (\xi[\,]/\mathbf{0}) \cdot \mathcal{S}' \quad \xi \cdot \mathcal{S}' \leq \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l\ \vdash_P (D, B, E)}$$

(ALOAD)
$$\frac{B(l) = \texttt{aaload} \quad l+1 \in \mathit{dom}(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \xi \neq \mathbf{Int}\ \mathit{and}\ \xi \neq \mathbf{Top} \quad \mathcal{S}_l \leq \mathbf{Int} \cdot (\xi[\,]/\mathbf{0}) \cdot \mathcal{S}' \quad (\xi/U) \cdot \mathcal{S}' \leq \mathcal{S}_{l+1} \quad \mathit{rel}(U)}{\mathcal{F}, \mathcal{S}, l\ \vdash_P (D, B, E)}$$

(ASTORE$_{\mathbf{Int}}$)
$$\frac{B(l) = \texttt{aastore} \quad l+1 \in \mathit{dom}(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \xi = \mathbf{Int}\ \mathit{or}\ \xi = \mathbf{Top} \quad \xi \cdot \mathbf{Int} \cdot (\xi[\,]/\mathbf{0}) \cdot \mathcal{S}_l \leq \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l\ \vdash_P (D, B, E)}$$

(ASTORE)
$$\frac{B(l) = \texttt{aastore} \quad l+1 \in \mathit{dom}(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \xi \neq \mathbf{Int}\ \mathit{and}\ \xi \neq \mathbf{Top} \quad (\xi/\bot_{\mathbf{rel}}) \cdot \mathbf{Int} \cdot (\xi[\,]/\mathbf{0}) \cdot \mathcal{S}_l \leq \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l\ \vdash_P (D, B, E)}$$

Fig. 6 Typing rules for instructions related to the object/array field

l	*instruction*	$\mathcal{F}_l(0)$	$\mathcal{F}_l(1)$	$\mathcal{S}$
1	`monitorenter` 0	$\mathtt{S}/L.(\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0})$	**Int**	ϵ
2	`load` 1	$\mathtt{S}/\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0}$	**Int**	ϵ
3	`if` 6	$\mathtt{S}/\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0}$	**Int**	$\mathbf{Int}\cdot\epsilon$
4	`monitorexit` 0	$\mathtt{S}/\widehat{L}.\mathbf{0}$	**Int**	ϵ
5	`return`	$\mathtt{S}/\mathbf{0}$	**Int**	ϵ
6	`monitorexit` 0	$\mathtt{S}/\widehat{L}.\mathbf{0}$	**Int**	ϵ
7	`return`	$\mathtt{S}/\mathbf{0}$	**Int**	ϵ

Fig. 7 Typing for code 4 in the figure 1

Example 3.3
Code 4 in Figure 1 is well-typed as shown in Figure 7.

Example 3.4
The method in Figure 8 is well-typed. The method first locks the `A`-class object given as the first argument, creates a new `B`-class object, stores it into the `b`-field and then unlocks the `A`-class object. In the code, *Exc* denotes the type of an exception.

The exception table in the method is interpreted as the following function E in our model.

$$E(l) = \begin{cases} 7 & l = 2,3,4,5 \\ 12 & \textit{otherwise} \end{cases}$$

Note that we assume `new B` instruction may raise a exception. For this reason, we must assign type $\mathtt{A}/\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0}$ to $\mathcal{F}_3(0)$ according to typing rule (NEW) so that $\mathcal{F}_3 \leq \mathcal{F}_4\&\mathcal{F}_7$ hold.

Example 3.5
The method in Figure 9 is well-typed. In the code, the `S`-class object in local variable 1 is stored into the array in variable 0. Then an `S`-class object is retrieved from the array and is locked and unlocked.

§4 Soundness of the type system

We have proved that our type system is sound in the sense that if a well-typed program is executed, any thread that has acquired a lock will eventually release the lock (provided that the thread terminates), and any thread that tries to release a lock has previously acquired the lock.

The soundness of our type system is stated formally as follows:

l	*instruction*	$\mathcal{F}_l(0)$	$\mathcal{F}_l(1)$	$\mathcal{S}$
1	`monitorenter` 0	$\mathtt{A}/L.(\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0})$	**Top**	ϵ
2	`load` 0	$\mathtt{A}/(\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0})$	**Top**	ϵ
3	`new B`	$\mathtt{A}/(\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0})$	**Top**	$\mathtt{A}/\mathbf{0}\cdot\epsilon$
4	`putfield A.b B`	$\mathtt{A}/\widehat{L}.\mathbf{0}$	**Top**	$\mathtt{B}/\bot_{\mathbf{rel}}\cdot\mathtt{A}/\mathbf{0}\cdot\epsilon$
5	`monitorexit` 0	$\mathtt{A}/\widehat{L}.\mathbf{0}$	**Top**	ϵ
6	`return`	$\mathtt{A}/\mathbf{0}$	**Top**	ϵ
7	`store` 1	$\mathtt{A}/\widehat{L}.\mathbf{0}$	**Top**	$\mathtt{Exc}/\mathbf{0}\cdot\epsilon$
8	`monitorexit` 0	$\mathtt{A}/\widehat{L}.\mathbf{0}$	$\mathtt{Exc}/\mathbf{0}$	ϵ
9	`load` 1	$\mathtt{A}/\mathbf{0}$	$\mathtt{Exc}/\mathbf{0}$	ϵ
10	`athrow`	$\mathtt{A}/\mathbf{0}$	$\mathtt{Exc}/\mathbf{0}$	$\mathtt{Exc}/\mathbf{0}\cdot\epsilon$
11	`return`	$\mathtt{A}/\mathbf{0}$	$\mathtt{Exc}/\mathbf{0}$	ϵ

*E*xception table: *E*

`from`	`to`	`target`	`type`
2	5	7	`any`
10	10	11	`any`

Fig. 8 Typing for an example code

l	*instruction*	$\mathcal{F}_l(0)$	$\mathcal{F}_l(1)$	$\mathcal{F}_l(2)$	$\mathcal{S}$
1	`monitorenter` 0	$\mathtt{S[\,]}/L.\widehat{L}.\mathbf{0}$	$\mathtt{S}/\bot_{\mathbf{rel}}$	**Int**	ϵ
2	`load` 0	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/\bot_{\mathbf{rel}}$	**Int**	ϵ
3	`load` 2	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/\bot_{\mathbf{rel}}$	**Int**	$\mathtt{S[\,]}/\mathbf{0}\cdot\epsilon$
4	`load` 1	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/\bot_{\mathbf{rel}}$	**Int**	$\mathbf{Int}\cdot\mathtt{S[\,]}/\mathbf{0}\cdot\epsilon$
5	`aastore`	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/\mathbf{0}$	**Int**	$\mathtt{S}/\bot_{\mathbf{rel}}\cdot\mathbf{Int}\cdot\mathtt{S[\,]}/\mathbf{0}\cdot\epsilon$
6	$\cdots$	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/\mathbf{0}$	**Int**	ϵ
7	`load` 0	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/\mathbf{0}$	**Int**	ϵ
8	`load` 2	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/\mathbf{0}$	**Int**	$\mathtt{S[\,]}/\mathbf{0}\cdot\epsilon$
9	`aaload`	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/\mathbf{0}$	**Int**	$\mathbf{Int}\cdot\mathtt{S[\,]}/\mathbf{0}\cdot\epsilon$
10	`store` 1	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/\mathbf{0}$	**Int**	$\mathtt{S}/L.\widehat{L}.\mathbf{0}\cdot\epsilon$
11	`monitorexit` 0	$\mathtt{S[\,]}/\widehat{L}.\mathbf{0}$	$\mathtt{S}/L.\widehat{L}.\mathbf{0}$	**Int**	ϵ
12	`monitorenter` 1	$\mathtt{S[\,]}/\mathbf{0}$	$\mathtt{S}/L.\widehat{L}.\mathbf{0}$	**Int**	ϵ
13	$\cdots$	$\mathtt{S[\,]}/\mathbf{0}$	$\mathtt{S}/\widehat{L}.\mathbf{0}$	**Int**	ϵ
14	`monitorexit` 1	$\mathtt{S[\,]}/\mathbf{0}$	$\mathtt{S}/\widehat{L}.\mathbf{0}$	**Int**	ϵ
15	`return`	$\mathtt{S[\,]}/\mathbf{0}$	$\mathtt{S}/\mathbf{0}$	**Int**	ϵ

Fig. 9 Typing for an example code

Theorem 4.1
Suppose that a program P is well-typed, and that
$P \vdash \langle\{0 \mapsto \langle 1, \emptyset, \epsilon, \emptyset, main_P\rangle\}, \emptyset\rangle \rightarrow^* \langle\Psi, H\rangle$.
For each $i \in dom(\Psi)$, if $\Psi(i) = \langle l, f, s, z, \sigma\rangle$, then the following properties hold:

1. If $P(\sigma) = (FD, D, B, E)$ and $B(l) = \texttt{return}$, then $z(o) = 0$ for all $o \in dom(H)$.
2. If $P(\sigma) = (FD, D, B, E)$ and $l \notin dom(B)$, then $z(o) = 0$ for all $o \in dom(H)$.
3. If $P(\sigma) = (FD, D, B, E)$ and $B(l) = \texttt{monitorexit}\ x$, then $z(f(x)) \geq 1$.

In this theorem, the first and second properties state that when a thread terminates normally or abruptly, it has released all the locks it acquired. The third property states that when a thread tries to release a lock, it has acquired the lock before.

We give an outline of the proof of the theorem below.

First, we introduce a *program type environment*, denoted by Γ, as a mapping from a class name to a pair $\langle\mathcal{F}, \mathcal{S}\rangle$. We write $\Gamma \vdash P$ if the *run* method of each σ-class in program P (i.e. (B, D, E) such that $P(\sigma) = (FD, (B, D, E))$) is well-typed under the type environment $\Gamma(\sigma)$ (in the sense of Definition 3.14).

We also define a type judgment relation $(\Gamma, P) \vdash \langle\Psi, H\rangle$ for machine states. It means that the threads Ψ and the heap H are consistent with the type assumption Γ and the class definition in program P. (These relations are formally defined in appendix.)

We can prove that if a machine state is well typed, invalid usage of a lock does not occur immediately (Lemma 4.1 below), and that the well-typedness of a machine state is preserved during execution of a well-typed program (Lemma 4.2 below). Theorem 4.1 follows immediately from these properties and the fact that the initial machine state is well-typed (Lemma 4.3 below).

Lemma 4.1 (Lack of immediate lock errors)

If $(\Gamma, P) \vdash \langle\Psi, H\rangle$ and $\Psi(i) = \langle l, f, s, z, \sigma\rangle$, then the following properties hold:

1. If $P(\sigma) = (FD, B, D, E)$ and $B(l) = \texttt{return}$, then $z(o) = 0$ for all

$o \in dom(H)$.

2. If $P(\sigma) = (FD, B, D, E)$ and $l \notin dom(B)$, then $z(o) = 0$ for all $o \in dom(H)$.
3. If $P(\sigma) = (FD, B, D, E)$ and $B(l) = \texttt{monitorexit}\ x$, then $z(f(x)) \geq 1$.

Lemma 4.2 (Subject reduction)

Suppose that $\Gamma \vdash P$ and $(\Gamma, P) \vdash \langle \Psi, H \rangle$ hold. If $P \vdash \langle \Psi, H \rangle \rightarrow \langle \Psi', H' \rangle$, then $(\Gamma, P) \vdash \langle \Psi', H' \rangle$ holds.

Lemma 4.3 (Well-typedness of initial state)

If $\Gamma \vdash P$, then $(\Gamma, P) \vdash \{\langle 0 \mapsto \langle 1, \emptyset, \epsilon, \emptyset, main_P \rangle\}, \emptyset \rangle$ holds.

The proofs of the above lemmas and the definitions of the relations $\Gamma \vdash P$ and $(\Gamma, P) \vdash \langle \Psi, H \rangle$ are given in Appendix B.

We can also show the following, usual type safety property (i.e., the progress property, meaning that a well-typed program never gets stuck).

Theorem 4.2 (Progress)
If $(\Gamma, P) \vdash \langle \Psi, H \rangle$, then one of the following conditions holds.

1 For all $i \in dom(\Psi)$, if $\Psi(i) = \langle l, f, s, z, \sigma \rangle$ and $P(\sigma) = (FD, B, D, E)$, then either $B(l) = \texttt{return}$ or $l \notin dom(B)$ holds.

2 For a $\langle \Psi', H' \rangle$, $P \vdash \langle \Psi, H \rangle \rightarrow \langle \Psi', H' \rangle$ holds

The theorem above together with Lemmas 4.2 and 4.3 imply that all threads terminate normally or abruptly. (The latter happens when an exception handler is not defined.) We omit the proof of the above theorem, since it follows immediately from the fact that our type system is essentially the same as Stata and Abadi's type system if we ignore conditions on usages (except some difference of supported instructions), and the fact that Stata and Abadi's type system guarantees the progress property.

§5 Type inference

5.1 Type inference algorithm

Because of the soundness of the type system, we can statically verify that a program properly uses lock primitives by checking that the program is well-typed. To check whether a program P is well-typed, it is sufficient to check, for each method (D, B, E) of the program, whether there exist $\mathcal{F}$ and $\mathcal{S}$ such that $\langle \mathcal{F}, \mathcal{S} \rangle \vdash_P (D, B, E)$ by performing type inference. The type inference proceeds as follows.

1. Step 1: Based on the typing rules, generate constraints on usages and types.
2. Step 2: Reduce the constraints and check whether they are satisfiable.

We do not show details of the algorithm since it is fairly standard [21, 15] except for the last step. We illustrate how type inference works using an example. Consider Code 1 in Figure 1 with an empty exception table and the method descriptor $\{0 \mapsto \sigma, 1 \mapsto \mathbf{Int}\}$. For simplicity, we assume that type information except for *usages* has been already obtained (for example, based on Stata and Abadi's type system [25]). The frame type environment $\mathcal{F}$ and the stack type environment $\mathcal{S}$ of the method are given as:

$$\begin{array}{l}
\mathcal{F}[l](0) = \sigma/\alpha_l \text{ for each } l \in \{1, \ldots, 7\} \\
\mathcal{F}[l](1) = \mathbf{Int} \text{ for each } l \in \{1, \ldots, 7\} \\
\mathcal{S}[l] = \begin{cases} \mathbf{Int} \cdot \epsilon & \text{if } l = 3 \\ \epsilon & \text{otherwise} \end{cases}
\end{array}$$

Here, each α_l is a usage variable to denote unknown usages. It expresses how the object stored in local variable 0 will be locked and unlocked at address l or later.

From the typing rule for the method (Definition 3.14), we obtain the following constraints:

$$\begin{array}{l}
\mathit{rel}(\alpha_1) \\
\alpha_1 \leq L.\alpha_2 \\
\alpha_2 \leq \alpha_3 \\
\alpha_3 \leq \alpha_4 \& \alpha_6 \\
\alpha_4 \leq \widehat{L}.\alpha_5 \\
\alpha_5 \leq \mathbf{0} \\
\alpha_6 \leq \widehat{L}.\alpha_7 \\
\alpha_7 \leq \mathbf{0}
\end{array}$$

From the constraints except for the first one, we obtain a solution (that is maximal with respect to $\leq$) $\alpha_1 = L.((\widehat{L}.\mathbf{0}) \& (\widehat{L}.\mathbf{0}))$. By substituting it for the first constraint, we get the constraint

$$rel(L.((\widehat{L}.\mathbf{0})\&(\widehat{L}.\mathbf{0}))).$$

Since it is satisfied, we know that lock primitives are safely used.

On the other hand, suppose that the instruction at address 3 is `if 7`. Then the constraint $\alpha_3 \leq \alpha_4 \& \alpha_7$ is generated instead of the constraint $\alpha_3 \leq \alpha_4 \& \alpha_6$. In this case, we get the constraint $rel(L.((\widehat{L}.\mathbf{0})\&\mathbf{0}))$. Since it does not hold, we know that lock primitives may be used incorrectly.

As in the above example, the type-checking problem is reduced to the problem of deciding whether constraints of the form $rel(U)$ hold. Deciding whether $rel(U)$ is a kind of model checking problem. As in type systems for deadlock-freedom [16)], the problem can be reduced to the reachability problem of Petri nets [6)], and hence the problem is decidable. A more efficient algorithm for judging the reliability is given in Appendix A.

5.2 Complexity of the inference algorithm

We discuss the complexity of our type inference algorithm. Suppose that the size of the method (i.e. the number of instructions) is k and that the size of local variables and stack frames is $O(k)$. Then, the number of constraints generated in Step 1 is $O(k^2)$ and the time complexity of this step is also $O(k^2)$.

In Step 2, we use the algorithm in Appendix A for checking the satisfiability of the constraints. It takes $O((l+1)\cdot N^2)$ time as discussed in Appendix A, where l is the number of occurrences of the usage constructor L (which corresponds to the number of `monitorenter` instructions in the method) and N is the size of constraints generated in Step 1. Furthermore, since l is $O(k)$ and N is $O(k^2)$, the total time complexity is $O(k^5)$.

However, if we assume that the number of local variables and the stack frame size are bound by a constant and that the number of `monitorenter` instructions in the method is also a constant, the time complexity is $O(k^2)$.

§6 Implementation

Based on the type system in this paper, we have implemented a Java bytecode verifier for the full *JVML*. We first describe the main differences between the formal system presented so far and the actual implementation in Section 6.1, and explain how we have dealt with the differences. We then introduce our verifier and report on experiments in Section 6.2.

6.1 Differences between the formal system and the implementation

Monitorenter/monitorexit instructions In the formal system, we have treated a combination of instructions `load x; monitorenter` as a single instruction `monitorenter x` and `load x; monitorexit` as `monitorexit x`. In the implementation, the verifier first performs these replacements. An occurrence of `monitorenter` which does not match with the above pattern can be expanded to `store z; monitorenter z`, where `z` is a fresh variable name.

Multiple exceptions We have so far considered only a single kind of exception. To deal with multiple kinds of exceptions in the real JVML, it is sufficient to extend an exception table E to a mapping from pairs consisting of an address $l \in dom(B)$ and an exception name to $\mathcal{A}$. We assume that standard bytecode verification is performed to check that exceptions are properly defined.

Other exception-raising instructions In our formal language $JVML_L$, only `new` σ and `athrow` instructions raise an exception. In the actual $JVML$, however, many other instructions may raise an exception. For example, `aaload` and `aastore` instructions may raise an `ArrayIndexOutOfBoundsException` exception, when the index (on the top of the operand stack) is out of the array bounds. The rules for such instructions are similar to (NEW): we enforce the constraints $\mathcal{F}_l \leq \mathcal{F}_{E(l)}$ and $\mathcal{S}_l \leq \textbf{Top}$ on the frame and stack type environments. For example, the rule for `aaload` is refined as follows.

$$\text{(ALOAD}^{\text{EXC}}_{\textbf{Int}}\text{)}\quad \frac{\begin{array}{c} B(l) = \texttt{aaload} \\ l+1 \in dom(B) \\ \mathcal{F}_l \leq \mathcal{F}_{l+1} \& \mathcal{F}_{E(l)} \\ \xi \neq \textbf{Int} \ \textit{and} \ \xi \neq \textbf{Top} \\ \mathcal{S}_l \leq \textbf{Int} \cdot (\xi[\,]/\mathbf{0}) \cdot \mathcal{S}' \quad \mathcal{S}_l \leq \textbf{Top} \\ (\xi/U) \cdot \mathcal{S}' \leq \mathcal{S}_{l+1} \quad \textit{rel}(U) \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

Instructions that may raise null pointer exceptions can be handled in a similar manner.

Subroutines JVML has instructions for subroutines: `jsr` l and `ret` x. Instruction `jsr` l pushes the return address (i.e the address of the next instruction of `jsr` l) of type `returnAddress` on the operand stack, and jumps to the address l. Instruction `ret` x returns to the address stored in the local variable x.

We explain how to deal with subroutines through an example. Consider the bytecode in Figure 10. The instructions from address 5 to 9 is a subroutine. When the subroutine is called, the return address is first stored in variable 2. Then, the object stored in variable 0 is locked and unlocked, and the execution returns from the subroutine.

The code is typed in the usual type system [25] as shown in Figure 10. Here, since variable 1 is not used in the subroutine, it is given type *undef*; it is treated as polymorphic (i.e., may be instantiated to different types for each subroutine call).

In our type system, a subroutine is given a function type of the form

$$\forall\vec{\beta}.(F^s, S^s) \to (F^r, S^r),$$

where F^s and S^s are the frame and stack type environments at the beginning of the subroutine, and F^r and S^r are the type environments at the end of the subroutine. The types are polymorphic on usages variables $\vec{\beta}$. For example, for the subroutine in Figure 10, the following type is assigned.

$$\forall\beta.(F^s, S^s) \to (F^r, S^r)$$

where

$$\begin{array}{lcllcl}
F^s(0) & = & A/L.\widehat{L}.\beta & F^r(0) & = & A/\beta \\
F^s(1) & = & \mathit{undef} & F^r(1) & = & \mathit{undef} \\
F^s(2) & = & \textbf{Top} & F^r(2) & = & \texttt{radd} \\
S^s & = & \texttt{radd} \cdot \epsilon & S^r & = & \epsilon
\end{array}$$

Since β in $F^r(0)$ describes how the object in variable 0 will be used *after* the subroutine call, the type of the subroutine is polymorphic on β. Figure 11 shows the frame and stack environments for each address. Note that those types, including the types of subroutines, are automatically inferred as in the formal system.

6.2 Experiments

Our verifier takes a Java class file as an input, and checks whether each method in the class file uses lock primitives safely base on our type system. The verifier has two modes. In one mode, the verifier gives only a yes/no answer on whether each method is correct. In the other mode, the verifier pretty-prints

l	*instruction*	$\mathcal{F}_l(0)$	$\mathcal{F}_l(1)$	$\mathcal{F}_l(2)$	$\mathcal{S}$
1	`jsr` 6	`B`	`B`	**Top**	ϵ
2	`load` 1	`B`	`B`	`radd`	ϵ
3	`store` 0	`B`	`B`	`radd`	$S \cdot \epsilon$
4	`jsr` 6	`B`	`B`	`radd`	ϵ
5	`return`	`B`	`B`	`radd`	ϵ
6	`store` 2	`B`	*undef*	**Top**	`radd` $\cdot\, \epsilon$
7	`monitorenter` 0	`B`	*undef*	`radd`	ϵ
8	`monitorexit` 0	`B`	*undef*	`radd`	ϵ
9	`ret` 2	`B`	*undef*	`radd`	ϵ

Fig. 10 An example of subroutine

l	*instruction*	$\mathcal{F}_l(0)$	$\mathcal{F}_l(1)$	$\mathcal{F}_l(2)$	$\mathcal{S}$
1	`jsr` 6	$\texttt{B}/L.\widehat{L}.\mathbf{0}$	$\texttt{B}/L.\widehat{L}.\mathbf{0}$	**Top**	ϵ
2	`load` 1	$\texttt{B}/\mathbf{0}$	$\texttt{B}/L.\widehat{L}.\mathbf{0}$	`radd`	ϵ
3	`store` 0	$\texttt{B}/\mathbf{0}$	$\texttt{B}/\mathbf{0}$	`radd`	$\texttt{B}/L.\widehat{L}.\mathbf{0} \cdot \epsilon$
4	`jsr` 6	$\texttt{B}/L.\widehat{L}.\mathbf{0}$	$\texttt{B}/\mathbf{0}$	`radd`	ϵ
5	`return`	$\texttt{B}/\mathbf{0}$	$\texttt{B}/\mathbf{0}$	`radd`	ϵ
6	`store` 2	$\texttt{B}/L.\widehat{L}.\beta$	*undef*	**Top**	`radd` $\cdot\, \epsilon$
7	`monitorenter` 0	$\texttt{B}/L.\widehat{L}.\beta$	*undef*	`radd`	ϵ
8	`monitorexit` 0	$\texttt{B}/\widehat{L}.\beta$	*undef*	`radd`	ϵ
9	`ret` 2	$\texttt{B}/\beta$	*undef*	`radd`	ϵ

Fig. 11 Type assignment for the subroutine

inferred types; that procedure includes simplification of inferred usages, and hence takes longer than the first mode. We use the second mode in describing examples below, and the first mode in measuring the verification time.

For example, let us consider the deposit method written by Java language as Account.java in Figure 12. The deposit method is compiled into *code Dep* in the figure.

Given the class file generated from `Account.java` in Figure 12, our prototype verifier outputs the following message:

```
Class name: Account
Class type: Account

Fileld types:
balance: int

Number: 0
Method name: <init>
Argument types:(Account/0 Int )
Return type: void
Lock Check : true

Number: 1
Method name: deposit
Argument types:(Account/L.UL.0, Int )
Return type: void
Lock Check : true
```

Here, `Argument types:` and `Return type:` indicate the types of arguments and the type of a return value. `Lock Check :` indicates whether the method uses lock primitives properly.

The message for the deposit method (the last five lines) states that the first argument of the method is Account-class object, which is locked (`L.`) and then unlocked(`UL.`), while the second argument of the method is of type **Int** and that the method returns no value. The line “`Lock Check : true`” indicates that every object in method is properly locked and unlocked.

If we remove the `monitorexit` 2 instruction at program address 13 in *code Dep* in Figure 12, our verifier outputs the following message:

```
Method name: deposit
Argument types:(Account/L.(UL.0 & 0), Int)
Return type: void
Lock Check : false
```

The message above says that the modified code does not use lock primitives properly. The type `Account/L.(UL.0 & 0)` of the first argument indicates

```
class Account {
  int balance;

  Account(int n) {
    this.balance = n;
  }

  void deposit(int n) {
    synchronized (this) {
      this.balance
        = this.balance + n;
    }
  }
}
```

(*Account.java*)

```
Method deposit:
 1  load 0
 2  store 2
 3  monitorenter 2
 4  load 0
 5  load 0
 6  getfield Account.balance int
 7  load 1
 8  add
 9  putfield Account.balance int
10  monitorexit 2
11  goto 16
12  store 3
13  monitorexit 2
14  load 3
15  athrow
16  return

Exception table:
from  to  target  type
  4   10    12    any
```

(*code Dep*)

Fig. 12 Java source code for an Account class and $JVML_L$ code for the deposit method

that the argument is an Account-class object, which is first locked (`L.`) and then may *or may not* be unlocked (`UL.0 & 0`).

We have tested our verifier using several class files in Java run time class libraries. All the classes were verified successfully. The column "Size" shows the size of each class file, and the column N_{LM} shows the number of methods that use lock instructions. The rightmost column in Table 1 shows the time spent for the verification of each class. For comparison, we have also measured the running time of our verifier with lock checking turned off; the result is shown in the column $Time_{no}$. As the table shows, the verification time for lock checking is 5-30 times longer than that for ordinary type checking in the present implementation.

Class name	*Size* (bytes)	N_{LM}	$Time_{no}$ (seconds)	$Time$ (seconds)
java.lang.Throwable	1559	3	0.0006	0.003
java.io.StringReader	1905	6	0.0009	0.019
java.lang.ref.ReferenceQueue	2320	6	0.0018	0.026
java.lang.Pakage	6490	2	0.0031	0.009
java.lang.Thread	7095	1	0.0061	0.007
java.net.InetAddress	7647	4	0.0081	0.018
java.lang.ThreadGroup	7274	14	0.0055	0.172
java.util.ResourceBundle	8655	4	0.0028	0.053
java.net.URL	9012	4	0.0066	0.075
java.lang.SecurityManager	9128	3	0.0042	0.037
java.lang.ClassLoader	14233	6	0.0104	0.119

Table 1 Verification time for each class file

We show more details for each method that includes lock primitives in *java.lang.ThreadGroup* and *java.lang.ClassLoader* classes in Table 2. The column "*Insts*" indicates the number of instructions in each method, and the column "*Maximal frames*" is the sum of the number of used local variables and the maximum stack frame size for each method. The column N_{LP} shows the number of lock instructions in each method.

§7 Discussion

Our type system presented so far ensures that a lock that has been acquired in a method execution will be released *within the same method execution.* Although the JVM specification [20] *allows* a verifier to enforce such a restriction, and bytecode programs compiled from Java source programs usually satisfy such constraints, the restriction may sometimes be too restrictive. For example, the bytecode in Figure 13 is rejected by our type system, although the method `delegate` safely locks and unlocks the object stored in variable 1.

Actually, because of the generality of our approach, it is not difficult to extend our type system to allow a bytecode like Figure 13. To do that, we just need to remove the condition $\forall x \in dom(\mathcal{F}_1).\ rel_t(\mathcal{F}_1(x))$ in Definition 3.14, and *java.lang.ClassLoader*

Method name	*Insts*	*Maximal frames*	N_{LP}	*Time* (seconds)
getPackage	44	9	3	0.009
difinePackage	45	23	3	0.021
getDefaultDomain	50	9	3	0.004
findNative	50	16	3	0.010
getPackages	62	10	4	0.012
loadLibrary0	193	22	11	0.075

java.lang.ThreadGroup

Method name	*Insts*	*Maximal frames*	N_{LP}	*Time* (seconds)
interrupt	58	11	3	0.008
resume	58	11	3	0.008
activeGroupCount	58	11	4	0.008
activeCount	59	12	4	0.008
setMaxPriority	60	9	3	0.009
add*2	63	10	3	0.011
add*3	63	10	3	0.011
remove*4	78	11	4	0.012
remove*5	78	11	4	0.012
destroy	80	10	3	0.013
list	83	15	3	0.017
stopOrSuspend	87	14	3	0.017
enumerate	92	14	3	0.019
enumerate	100	15	4	0.022

Table 2 Verification time for each method

*2add(ThreadGroup, Thread), *3add(ThreadGroup, ThreadGroup), *4remove(ThreadGroup, Thread), *5remove(ThreadGroup, ThreadGroup).

```
Method A.delegate         Method B.unlock

1  monitorexit 1          1  monitorexit 1
2  load 0                 2  return
3  load 1
4  invoke B.unlock
5  return
```

Fig. 13 Inter-method locking/unlocking

add the following rule for method invocation:

(INVOKE)

$$\frac{\begin{array}{c} B(l) = \mathtt{invoke}\ \sigma.\mathtt{m} \\ \sigma.m : (\tau_1, \ldots, \tau_n) \to \tau \\ l+1 \in dom(B) \\ \mathcal{F}_l \leq \mathcal{F}_{l+1} \\ \mathcal{S}_l \leq \tau_1 \cdots, \ldots, \cdot \tau_n \cdot \sigma/\mathbf{0} \cdot \mathcal{S}' \quad \tau \cdot \mathcal{S}' \leq \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

Here, $(\tau_1, \ldots, \tau_n) \to \tau$ on the second line includes usage expressions and describes how the invoked method will lock/unlock the argument objects. It is assumed that such method interface (except the interface of the current method) is either inferred by pre-analyzing the invoked method, or by user annotations.[*5] If there is a recursive call to the current method, fresh usage variables are assinged to the usage part of the interface, and constraints on them are generated and solved as in the type inference algorithm discussed in Section 5. Figure 14 shows how the bytecode in Figure 13 is typed in the extended type system.

§8 Related Work

Bigliardi and Laneve [2, 19] have proposed a type system for the same purpose as ours, but their type systems are quite different from ours. The type system [19] uses *indexed object types*, which are singleton types obtained by annotating a normal object type (i.e. class name) with a program address where the object is copied to the operand stack from a local variable. A multiset of the indexed object types is used to express the set of locked objects at each program address. For example, σ_l is a σ-class object that is copied to the operand stack

[*5] The user annotations will not be so heavy a burden if we adopt $\bot_{\mathbf{rel}}$ as default usage annotations, and require user annotations only if default usage annotations are incorrect.

Typing for `A.delegate`

l	$\mathcal{F}_l(1)$	$\mathcal{S}$
1	$\sigma/L.\widehat{L}.\mathbf{0}$	ϵ
2	$\sigma/\widehat{L}.\mathbf{0}$	ϵ
3	$\sigma/\widehat{L}.\mathbf{0}$	$\mathtt{B}/\mathbf{0} \cdot \epsilon$
4	$\sigma/\mathbf{0}$	$\sigma/\widehat{L}.\mathbf{0} \cdot \mathtt{B}/\mathbf{0} \cdot \epsilon$
5	$\sigma/\mathbf{0}$	ϵ

Typing for `B.unlock`

l	$\mathcal{F}_l(1)$	$\mathcal{S}$
1	$\sigma/\widehat{L}.\mathbf{0}$	ϵ
2	$\sigma/\mathbf{0}$	ϵ

Fig. 14 Typing for the code in Figure 13

at program address l and $Z_l = \{\sigma_{l'}\}$ expresses that at address l a σ-class object that has been copied at address l' is locked. The monitorenter instruction adds the type of the locked object to that multiset and the monitorexit instruction removes the type of the unlocked object from the multiset. The type system checks that the multiset of indexed object types is empty at the return address For example the type system assigns types to the first code in Figure 1 as shown in Figure 15.

Since object types must be singleton types, their type system cannot deal with a case where multiple objects flow to the same variable. For example, consider the code in Figure 16. The code is not well-typed in Laneve's type system, since it is not statically known whether the object locked at address 9 has type σ_3 or σ_6. To solve the problem, Laneve informally discusses introduction of a subtype relation, without a formal proof. Even with that extension, their approach does not seem to be able to deal with subroutines in which the same variable is bound to different objects (as in the code in Figure 10.

On the other hand, there are also some bytecode that can be typed in Laneve's type system but not in our type system. Our type system does not keep track of the order of accesses through different local variables or stack locations, which causes some correct programs to be rejected. Consider the code in the lefthand side of Figure 18. It should be considered valid, but it is rejected by our type system. That is because our type system fails to keep track of precise information about the order between accesses through different variables, and assigns $L \otimes \widehat{L}$ to the usage of object S created at address 1. (On the other hand, our type system does accept the code in the righthand side: the usage $L.\widehat{L}$ is assigned to object S at address 1.) In order to analyze such code, we need to extend the type system by using an idea presented in the generic type system for the π-calculus [14].

Address l	Instruction	$\mathcal{F}_l(x)$	$\mathcal{F}_l(y)$	$\mathcal{S}_l$	Z_l
1	`load` x	σ	**Int**	ϵ	$\{\}$
2	`monitorenter`	σ_1	**Int**	$\sigma_1 \cdot \epsilon$	$\{\}$
3	`load` y	σ_1	**Int**	ϵ	$\{\sigma_1\}$
4	`if` 8	σ_1	**Int**	$\mathbf{Int} \cdot \epsilon$	$\{\sigma_1\}$
5	`load` x	σ_1	**Int**	ϵ	$\{\sigma_1\}$
6	`monitorexit`	σ_1	**Int**	$\sigma_1 \cdot \epsilon$	$\{\sigma_1\}$
7	`return`	σ_1	**Int**	ϵ	$\{\}$
8	`load` x	σ_1	**Int**	ϵ	$\{\sigma_1\}$
9	`monitorexit`	σ_1	**Int**	$\sigma_1 \cdot \epsilon$	$\{\sigma_1\}$
10	`return`	σ_1	**Int**	ϵ	$\{\}$

Fig. 15 Typing for Code 1 in Figure 1 in Laneve's type system

```
 1  load 2
 2  if 6
 3  load 0
 4  store 3
 5  goto 8
 6  load 1
 7  store 3
 8  load 3
 9  monitorenter
10  load 3
11  monitorexit
12  return
```

Fig. 16 A program rejected by Laneve's type system

Address l	Instruction	$\mathcal{F}_l(0)$	$\mathcal{F}_l(1)$	$\mathcal{F}_l(2)$	$\mathcal{F}_l(3)$	$\mathcal{S}_l$
1	`load` 2	$\sigma/(L.\widehat{L}.\mathbf{0})\&\mathbf{0}$	$\sigma/(L.\widehat{L}.\mathbf{0})\&\mathbf{0}$	**Int**	**Top**	ϵ
2	`if` 6	$\sigma/(L.\widehat{L}.\mathbf{0})\&\mathbf{0}$	$\sigma/(L.\widehat{L}.\mathbf{0})\&\mathbf{0}$	**Int**	**Top**	$\mathbf{Int} \cdot \epsilon$
3	`load` 0	$\sigma/L.\widehat{L}.\mathbf{0}$	$\sigma/\mathbf{0}$	**Int**	**Top**	ϵ
4	`store` 3	$\sigma/\mathbf{0}$	$\sigma/\mathbf{0}$	**Int**	**Top**	$\sigma/L.\widehat{L}.\mathbf{0} \cdot \epsilon$
5	`goto` 8	$\sigma/\mathbf{0}$	$\sigma/\mathbf{0}$	**Int**	$\sigma/L.\widehat{L}.\mathbf{0}$	ϵ
6	`load` 1	$\sigma/\mathbf{0}$	$\sigma/L.\widehat{L}.\mathbf{0}$	**Int**	**Top**	ϵ
7	`store` 3	$\sigma/\mathbf{0}$	$\sigma/\mathbf{0}$	**Int**	**Top**	$\sigma/L.\widehat{L}.\mathbf{0} \cdot \epsilon$
8	`monitorenter` 3	$\sigma/\mathbf{0}$	$\sigma/\mathbf{0}$	**Int**	$\sigma/L.\widehat{L}.\mathbf{0}$	ϵ
9	`monitorexit` 3	$\sigma/\mathbf{0}$	$\sigma/\mathbf{0}$	**Int**	$\sigma/\widehat{L}.\mathbf{0}$	ϵ
10	`return`	$\sigma/\mathbf{0}$	$\sigma/\mathbf{0}$	**Int**	$\sigma/\mathbf{0}$	ϵ

Fig. 17 Typing for the code in Figure 16

```
new S               new S
store x             store x
load x              monitorenter x
store y             load x
monitorenter x      store y
monitorexit y       monitorexit y
return              return
```

Fig. 18 Programs that lock and unlock an object through different variables

Recently, various methods for statically analyzing usage of lock primitives have been proposed for other languages [5, 10]. However, the semantics of lock primitives treated in those languages are different from the one treated in this paper, and hence it is not clear whether those methods can be applied to our target language. In those languages, each lock has only two states: the locked state and the unlocked state. On the other hand, in our target language, a lock can have infinitely many states (since each lock has a counter expressing how many times it has been acquired).

The idea of adding *usages* to types has its origin in type systems [16, 26] for the π-calculus. In those type systems, usage expressions are used to express in which order communication channels are used for input and output.

Igarashi and Kobayashi [15] developed a general type system for analyzing usage of various resources such as files, memory, and locks. The problem treated in the present paper is an instance of the general problem treated by them [15]. However, the target language of their analysis is a functional language, while our target language is a more low-level language. We also gave a concrete algorithm for checking the reliability of a usage, while the corresponding algorithm is left unspecified in their work [15].

There are a number of type systems for JVML to verify other properties. Stata and Abadi [25] was the first to formalize Java byte code verification in the form of a type system. Our type system has been obtained by extending their type system with usages. Freund and Mitchell [11] developed a type system which guarantees that every object is initialized before it is used. There is some similarity between their soundness proof and ours. Higuchi and Ohori [13] have proposed a type system for Java bytecode which is more closer to the type system of λ-calculus. Our treatment of subroutines is partially inspired from their work.

Our type-based analysis may be viewed as abstract model checking where abstract models (which represent the locking behavior for each object) are first extracted as usage expressions through type inference, and then it is checked whether the abstract models are valid (i.e., whether $rel(U)$ holds). Similar approach has also been taken recently in type-based verification of π-calculus programs [14, 3, 18].

§9 Conclusion

We have proposed a type system for checking usage of lock primitives for a subset of JVML [20], which extends types with information about in which order objects are locked/unlocked. We have proved its correctness and implemented a prototype verifier for the full JVML based on the type system. We have tested the verifier for several classes in Java run-time libraries.

References

1) Gilles Barthe and Tamara Rezk. Non-interference for a jvm-like language. In *Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2005)*, pages 103–112, 2005.
2) Gaetano Bigliardi and Cosimo Laneve. A type system for JVM threads. In *Proceedings of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)*, Montreal, Canada, 2000.
3) Sagar Chaki, Sriram Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 45–57, 2002.
4) S. Christensen. *Decidability and Decomposition in Process Algebras.* PhD thesis, University of Edinburgh, 1993.
5) Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.
6) J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Journal of Information Processing and Cybernetics*, 30(3):143–160, 1994.
7) Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, 1999.
8) Cormac Flanagan and Martín Abadi. Types for safe locking. In *Proceedings of ESOP 1999*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, 1999.
9) Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.
10) Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of ACM SIGPLAN Conference on Programming Language*

Design and Implementation, 2002.

11) Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 310–327, New York, NY, USA, 1998. ACM Press.

12) Stephen N. Freund and John C. Mitchell. A type system for the Java bytecode language and verifier. *J. Autom. Reason.*, 30(3-4):271–321, 2003.

13) Tomoyuki Higuchi and Atsushi Ohori. Java bytecode as a typed term calculus. In *ACM PPDP conference, 2002*, pages 201–211. ACM Press, 2002.

14) Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

15) Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2), 2005. Preliminary summary appeared in Proceedings of POPL 2002.

16) Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag, August 2000. The full version is available as technical report TR00-01, Dept. Info. Sci., Univ. Tokyo.

17) Naoki Kobayashi and Keita Shirane. Type-based information flow analysis for low-level languages. *Computer Software*, 20(2):2–21, 2003. In Japanese. A summary written in English appeared in informal proceedings of 2nd Asian Workshop on Programming Languages and Systems (APLAS'02).

18) Naoki Kobayashi, Kohei Suenaga, and Lucian Wischik. Resource usage analysis for the pi-calculus. *Logical Methods in Computer Science*, 2(3:4):1–42, 2006.

19) Cosimo Laneve. A type system for JVM Threads. *Theoretical Computer Science*, 290(1):241–778, 2003.

20) Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd edition)*. Addison Wesley, 1999.

21) Torben Mogensen. Types for 0, 1 or many uses. In *Implementation of Functional Languages*, volume 1467 of *Lecture Notes in Computer Science*, pages 112–122, 1998.

22) Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

23) George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

24) Pratibha Permandla and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the Java virtual machine language. Tecnical Report.

25) Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

26) Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*, volume 16(3) of *ENTCS*, pages 55–77, 1998.

§Appendix A Algorithm for checking whether usage constraints are satisfiable

In this section, we give an algorithm for checking whether constraints generated in Step 1 of type inference (see 5.1) are satisfiable and estimate time-complexity of the algorithm.

Constraints on usages generated in Step 1 can be reduced to the following set of constraints

$$\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\} \cup \{\mathit{rel}(\alpha_{r_1}), \ldots, \mathit{rel}(\alpha_{r_h})\}$$

where $\{r_1, \ldots, r_h\} \subseteq \{1, \ldots, n\}$ and $\alpha_1, \ldots, \alpha_n$ are different from each other.

We can first solve $\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$ by repeatedly applying the following reduction rules to $(\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}, \emptyset)$:

$$(\{\alpha \leq U\} \cup C\ ,\ S) \rightarrow ([\mu\alpha.U/\alpha]C\ ,\ \{\alpha = \mu\alpha.U\} \cup [\mu\alpha.U/\alpha]S).$$

Here, the first element of the pair is the set of remaining subusage constraints and the second element is the solution. When $(\{\alpha_1 \leq U_1, ..., \alpha_n < U_n\}, \emptyset)$ is reduced to $(\emptyset, S)$, S is the solution for $\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$. So, the problem is reduced to that of checking whether the solution satisfies $\mathit{rel}(\alpha_{r_1}), \ldots, \mathit{rel}(\alpha_{r_h})$.

To check whether $\mathit{rel}(U)$ holds for a usage U, we consider two numbers Min_U and Fin_U for each closed usage U. Min_U is the least n such that $(U, 0) \rightarrow^*_{rel} (U', n)$, while Fin_U is the greatest n such that $(U, 0) \rightarrow^*_{rel} (U', n)$ and $U' \leq \mathbf{0}$ (if no such n exists, $\mathit{Fin}_U = -\infty$).

Intuitively, Min_U expresses the minimum number of locks being acquired while the object is accessed according to U. Similarly, Fin_U expresses the maximum number of locks being held after the access according to U finishes.

Example A.1
$\mathit{Min}_{\widehat{L}.L} = -1,\ \mathit{Min}_{L.\widehat{L}} = 0$ and $\mathit{Fin}_{\widehat{L}.L} = \mathit{Fin}_{L.\widehat{L}} = 0,\ \mathit{Fin}_{\mu\alpha.(L\&L.\widehat{L}.\alpha)} = 1$

By Definition 3.5, $\mathit{rel}(U)$ if and only if (1)$\mathit{Min}_U = \mathit{Fin}_U = 0$. $\mathit{Fin}_U = 0$ corresponds to the first condition of Definition 3.5, and guarantees that the number of locks and unlocks are balanced. $\mathit{Min}_U = 0$ corresponds to the second condition of Definition 3.5, and guarantees that a lock always occurs before an unlock. (Actually, the second condition of Definition 3.5 only requires $\mathit{Min}_U \geq 0$, but it is equivalent to $\mathit{Min}_U = 0$ since $\mathit{Min}_U \leq 0$ by the definition of Min_U.)

Using the above fact, we can check whether $\mathit{rel}(\alpha_{r_i})$ $(i = 1, 2, \cdots, h)$ holds for the solution of $\{\alpha_1 \leq U_1, \ldots, \alpha_n \leq U_n\}$.

First, we check whether $Min_{\alpha_{r_i}} = 0$ $(i = 1, \dots, h)$ holds as follows: Let $x_1, \dots, x_n$ be variables denoting $Min_{\alpha_1}, \dots Min_{\alpha_n}$. Let C_{Min} be the set of equations

$$\{x_i = MinExp(U_i) \mid \alpha_i \leq U_i \text{ is a constraint generated in Step 1}\}$$

where $MinExp(V)$ is an expression defined by:

$$\begin{aligned}
&MinExp(\mathbf{0}) = 0\\
&MinExp(\bot_{\mathbf{rel}}) = 0\\
&MinExp(\alpha_i) = x_i\\
&MinExp(U_1 \otimes U_2) = MinExp(U_1) + MinExp(U_2)\\
&MinExp(U_1 \& U_2) = min(MinExp(U_1), MinExp(U_2))\\
&MinExp(L.U) = min(0, MinExp(U) + 1)\\
&MinExp(\widehat{L}.U) = MinExp(U) - 1.
\end{aligned}$$

C_{Min} can be expressed in the form

$$\begin{aligned}
\{&x_1 = F_1(x_1, \dots, x_n),\\
&\cdots,\\
&x_n = F_n(x_1, \dots, x_n)\}
\end{aligned}$$

where F_i is a monotonic function obtained by composing the operators $+$, constants 0, 1, -1 and min. Here, $min(x, y)$ denotes the minimum of x and y.

We write $Var(F_i)(\subseteq \{x_1, \dots, x_n\})$ for the set of variables that occur in $F_i(x_1, \dots, x_n)$ and define $\overline{Var}(F_i)(\subset \{x_1, \dots, x_n\})$ as the least set that satisfies the following conditions.

$$\overline{Var}(F_i) \supseteq Var(F_i) \cup \{x \in \overline{Var}(F_j) \mid x_j \in Var(F_i)\} \ (i = 1, \dots, n).$$

Intuitively, $\overline{Var}(F_i)$ is the set of variables that affect the value of $F_i(x_1, \dots, x_n)$.

Because we are only interesting in the value of $Min_{\alpha_{r_i}}$ $(i = 1, \dots, h)$, we can remove $x_i = F_i(x_1, \dots, x_n)$ such that $x_i \notin \overline{Var}(F_{r_1}) \cup \cdots \cup \overline{Var}(F_{r_h})$ from C_{Min}. Therefore, in the rest of this section, we assume without loss of generality

$$\overline{Var}(F_{r_1}) \cup \cdots \cup \overline{Var}(F_{r_h}) = \{x_1, \cdots, x_n\}.$$

Compute $(u_1^{(j)}, \dots, u_n^{(j)})$ $(j = 0, 1, \cdots)$ by

$$\begin{aligned}
&u_i^{(0)} = 0\\
&u_i^{(j+1)} = F_i(u_1^{(j)}, \dots, u_n^{(j)})
\end{aligned}$$

until $j = m$ such that $(u_1^{(m+1)}, \dots, u_n^{(m+1)}) = (u_1^{(m)}, \dots, u_n^{(m)})$ or $u_i^{(m)} < 0$ for some $i \in \{r_1, \dots, r_h\}$. (Note that such an m always exists.) and check whether $u_i^{(m)} = 0$ for all $i \in \{r_1, \dots, r_h\}$. If this is true , we have $Min_{\alpha_{r_1}} = 0, \cdots, Min_{\alpha_{r_h}} = 0$ and proceed to the check for Fin_{α_i}. If there exists r_i such that $u_{r_i}^{(m)} < 0$, the constraints are not be satisfiable.

Whether $Fin_{\alpha_{r_i}} = 0$ $(i = 1, \dots, h)$ holds can be checked in a similar manner. First, observe that it is sufficient to check a weaker condition $Fin_{\alpha_{r_i}} = 0$ since we have already $Min_{\alpha_{r_i}} = 0$, hence $Fin_{\alpha_{r_i}} \geq Min_{\alpha_{r_i}} = 0$. Let $y_1, \dots, y_n$ be variables denoting $Fin_{\alpha_1}, \dots, Fin_{\alpha_n}$. Let C_{Fin} be the set of equations

$$\{y_i = FinExp(U_i) \mid \alpha_i \leq U_i \text{ is a constraint generated in Step 1}\}$$

Here, $FinExp(U)$ is the expression defined by:

$$\begin{aligned}
&FinExp(\mathbf{0}) = 0\\
&FinExp(\alpha_i) = y_i\\
&FinExp(\perp_{\mathbf{rel}}) = 0\\
&FinExp(U_1 \otimes U_2) = FinExp(U_1) + FinExp(U_2)\\
&FinExp(U_1 \& U_2) = \max(FinExp(U_1), FinExp(U_2))\\
&FinExp(L.U) = FinExp(U) + 1\\
&FinExp(\widehat{L}.U) = FinExp(U) - 1.
\end{aligned}$$

C_{Fin} can be expressed in the form

$$\begin{aligned}
&\{y_1 = G_1(y_1, \dots, y_n),\\
&\cdots,\\
&\ y_n = G_n(y_1, \dots, y_n)\}.
\end{aligned}$$

As in the previous case, we assume

$$\overline{Var}(G_{r_1}) \cup \cdots \cup \overline{Var}(G_{r_h}) = \{y_1, \dots, y_n\}.$$

We first find i such that the least solution of C_{Fin} satisfies $y_i = -\infty$ as follows:

Compute $(z_1^{(j)}, \dots, z_n^{(j)})$ by

$$\begin{aligned}
&z_i^{(0)} = -\infty\\
&z_i^{(j+1)} = to\_fin(G_i(z_1^{(j)}, \dots, z_n^{(j)}))
\end{aligned}$$

until $j = m'$ such that $(z_1^{(m'+1)}, \dots, z_n^{(m'+1)}) = (z_1^{(m')}, \dots, z_n^{(m')})$ (Such an m' always exists.) , where function to_fin is defined as follows:

$$to\_fin(n) = \begin{cases} -\infty & n = -\infty \\ \mathit{fin} & n = \mathit{fin} \text{ or } n \text{ is an integer.} \end{cases}$$

and operations on *fin* are defined by:

$$\mathit{fin} + 1 = \mathit{fin} \quad \mathit{fin} - 1 = fin$$

$$\max(\mathit{fin}, -\infty) = \max(-\infty, \mathit{fin}) = \max(\mathit{fin}, \mathit{fin}) = \mathit{fin}.$$

When the computation stops, if $v_i^{(m')} = -\infty$ holds, the least solution of C_{Fin} satisfies $y_i = -\infty$. So, assigning $-\infty$ to such variable y_i in $G_1, \dots, G_n$ we transform C_{Fin} to C'_{Fin}. To define this C'_{Fin} formally, we define Var^{fin} as $\{y_i \in \{y_1, \dots, y_n\} \mid z_i^{(m')} = fin\}$ and Var^{inf} as $\{y_i \in \{y_1, \dots, y_n\} \mid z_i^{(m')} = -\infty\}$. C'_{Fin} is defined by:

$$G'_i = G_i[-\infty/y_{f_1}, \dots, -\infty/y_{f_{n''}}] \quad (i = 1, \dots, n)$$

where $\{y_{f_1}, \dots, y_{f_{n''}}\} = Var^{inf}$ $(\subseteq \{y_1, \dots, y_n\})$.

Without loss of generality, we can assume

$$\{y_{r_1}, \dots, y_{r_h}\} \subseteq Var^{fin}$$

$$\overline{Var}(G'_{r_1}) \cup \dots \cup \overline{Var}(G'_{r_h}) = \{y_{i_1}, \cdots, y_{i_{n'}}\} \ (= Var^{fin}).$$

So, C'_{Fin} can be expressed in the form:

$$\begin{array}{l} \{y_{i_1} = G'_{i_1}(y_{i_1}, \dots, y_{i_{n'}}), \\ \cdots, \\ y_{i_{n'}} = G'_{i_{n'}}(y_{i_1}, \dots, y_{i_{n'}})\} \end{array}$$

where $\{y_{i_1}, \dots, y_{i_{n'}}\} = Var^{fin}$ $(\subseteq \{y_1, \dots, y_n\})$.

Using this C'_{Fin}, we compute the value of each y_{r_i} $(i = 1, \dots, h)$.

Compute $(v_{i_1}^{(j)}, \dots, v_{i_{n'}}^{(j)})$ by

$$\begin{array}{l} v_i^{(0)} = -\infty \\ v_i^{(j+1)} = G_i{}'(v_{i_1}^{(j)}, \dots, v_{i_{n'}}^{(j)}) \end{array}$$

until $j = m$ such that $v_i^{(m)} = v_i^{(m+1)}$ for all $i \in \{i_1, \dots, i_{n'}\}$ or $v_i^{(m)} > 0$ for some $i \in \{r_1, \dots, r_h\}$. (Such an m always exists.) and check whether $v_i^{(m)} \le 0$

holds for all $i \in \{r_1, \dots, r_h\}$. If this holds, we have $Fin_{U_{r_1}} \leq 0, \cdots, Fin_{U_{r_h}} \leq 0$ and constraints generated in Step 1 are satisfiable. Otherwise, the constraints are not satisfiable.

Time complexity of the algorithm Time complexity of checking whether the above constrains are satisfiable is $O((l+1) \cdot N^2)$ where N is the size of constrains generated Step 1 and l is the number of occurrences of $L.$ in $\{U_1, \dots, U_n\}$ (This l may be 0). Moreover, n (This is the number of constraints) is estimated to be $O(N)$. In the rest of this section, we discuss this time complexity.

First, we calculate time complexity of checking whether $Min_{\alpha_{r_i}} = 0$ $(i = 1, \dots, h)$ holds. It takes time $O(N)$ to transform sub-usage constraints to the equation system C_{Min} and time $O(n \cdot m)$ to compute $(Min_{U_1}^{(j)}, \dots, Min_{U_n}^{(j)})$ for $j = 0$ to $j = m$. Therefore, The total time complexity is $O(N \cdot m)$.

To estimate the number m , we note that $u_i^{(j)}$ has the following property:

(1) We assume $u_1^{(j)} \leq 0, \dots, u_n^{(j)} \leq 0$. If $u_i^{(j)} < -l$ and $x_i \in \overline{Var}(F_{r_k})$, $u_{r_k}^{(j+n)} < 0$ holds.

– To prove it, we note that if $x_i \in Var(F_k)$ then $u_k^{(j+1)} \leq u_i^{(j)} + \#L(F_k)$ holds, where $\#L(F_k)$ is the number of occurrences of $L.$ in U_k. (This can be proved by induction of the structure on U_k.) Here, let assume $u_i^{(j)} < -l$ and $x_i \in \overline{Var}(F_{r_k})$. Since $\overline{Var}(F_{r_1}) \cup \dots \cup \overline{Var}(F_{r_h}) = \{x_1, \cdots, x_n\}$, we have a series of variables $x_i, x_{k_1}, \dots, x_{k_p}$ $(0 \leq p \leq n)$ such that $k_p = r_k$ and the following conditions hold:

$$\begin{array}{lcl} x_i & \in & Var(F_{k_1}) \\ x_{k_1} & \in & Var(F_{k_2}) \\ & \cdots & \\ x_{k_{p-1}} & \in & Var(F_{k_p}). \end{array}$$

By the above property, we have

$$u_{r_k}^{(j+p)} = u_{r_p}^{(j+p)} \leq u_i^{(j)} + \#L(F_{k_1}) + \cdots + \#L(F_{k_p}) \leq u_i^{(j)} + l.$$

By $u_i^{(j)} < -l$ and $0 \leq p \leq n$, we have $u_{r_k}^{(j+n)} < 0$.

So, $u_i^{(j)}$ $(j = 0, \dots, m-n-1)$ can range over $\{-l+1, \dots, -1, 0\}$. Since $\vec{u}_i^{(0)}, \vec{u}_i^{(1)}, \dots, \vec{u}_i^{(m)}$ decreases monotonically, $m = O(n \times l + n + 1) = O(n(l+1))$, where $\vec{u}_i^{(j)} = (u_1^{(j)}, \dots, u_n^{(j)})$ (note that l may be 0).

Next, we calculate time complexity of checking whether $Fin_{U_{r_i}} \leq 0$ $(i = 1, \ldots, h)$ holds. Obviously the number m' of iterations in the first step is $O(n)$ and it takes $O(N^2)$ to transform C_{Fin} to C'_{Fin}. Therefore, when m is the number of iterations of the second step, the time complexity is $O(m' \cdot n) + O(N^2) + O(m \cdot n) = O(N^2 + m \cdot n)$.

To estimate the number m, we define $w_i^{(j)}$ $(i \in \{1, \ldots, n\},\ j = 0, 1, \ldots)$ by

$$\begin{aligned} w_i^{(0)} &= -\infty \\ w_i^{(j+1)} &= G_i(w_1^{(j)}, \ldots, w_n^{(j)}). \end{aligned}$$

Note that $v_{i'}^{(j)} = w_{i'}^{(j)}$ for all $y_{i'} \in \overline{Var}(G'_{r_1}) \cup \cdots \cup \overline{Var}(G'_{r_h})$.

We note that $v_{i'}^{(j)},\ w_i^{(j)}$ have following the properties:

(2) For all $y_{i'} \in \overline{Var}(G'_{r_1}) \cup \cdots \cup \overline{Var}(G'_{r_h})$, if $j > n$ then $v_{i'}^{(j)} > -\infty$.

– This follows from the definition of $v_{i'}^{(j)}$ and the fact that $z_{i'}^{(j)} = fin$ holds for $j > n$.

(3) For any integer c, $w_i^{(j)} = c\ (\neq -\infty) \ \Rightarrow\ u_i^{(j)} \leq c$

– This follows from the fact that

$$\begin{gathered} (w_i^{(j)} = -\infty \ \vee\ w_i^{(j)} \geq u_i^{(j)}) \ \wedge\ \alpha_i \in FV(U_k) \\ \Rightarrow \\ (w_k^{(j+1)} = -\infty \ \vee\ w_k^{(j+1)} \geq u_k^{(j+1)}) \end{gathered}$$

(The proof is by induction on the structure of U_k.)

(4) The following fact holds:

$$\begin{gathered} (w_i^{(j)} = -\infty \ \vee\ w_i^{(j)} > u_i^{(j)}) \ \wedge\ \alpha_i \in FV(U_k) \\ \Rightarrow \\ (w_k^{(j+1)} = -\infty \ \vee\ w_k^{(j+1)} > u_k^{(j+1)}). \end{gathered}$$

– The proof is by induction on the structure of U_k.

(5) We assume that we have checked $Min_{\alpha_i} = 0$ $(i = r_1, \ldots, r_h)$ holds. If $y_{i'} \in \overline{Var}(G'_{r_k})$ and $v_{i'}^{(j)} > 0$ $(j > n)$, $v_{r_k}^{(j+n)} > 0$ holds.

– Since $y_{i'} \in \overline{Var}(G'_{r_k})$, we have a series of variables $y_{i'}, y_{k_1}, \ldots, y_{k_p}$ $(0 <$

$p \le n$) such that $k_p = r_k$ and the following conditions hold:

$$\begin{array}{lll} y_i & \in & Var(G'_{k_1}) \subseteq Var(G_{k_1}) \\ y_{k_1} & \in & Var(G'_{k_2}) \subseteq Var(G_{k_2}) \\ & \cdots & \\ y_{k_{p-1}} & \in & Var(G'_{k_p}) \subseteq Var(G_{k_p}). \end{array}$$

We note $v_{i'}^{(j')} = w_{i'}^{(j')}$ and $v_{i'}^{(j')} > -\infty$ ($j' > n$) (This follows form (2).) for $i' \in \overline{Var}(G'_{r_1}) \cup \cdots \cup \overline{Var}(G'_{r_h})$. From $u_{i'}^{(j)} \le 0$ and the assumption $v_{i'}^{(j)} > 0$, $v_{i'}^{(j)} > u_{i'}^{(j)}$ follows. So, By repeatedly applying (4), we have $v_{r_k}^{(j+p)} > u_{r_k}^{(j+p)}$. From the assumption $Min_{U_{r_k}} = 0$, $v_{r_k}^{(j+n)} \ge v_{r_k}^{(j+p)} > u_{r_k}^{(j+p)} \ge Min_{U_{r_k}} = 0$ follows.

From (1),(3) and the assumption that $Min_{\alpha_{r_i}} = 0$ ($i = 1, \ldots, h$) are checked, we have $v_i^{(j)} \ne -\infty \Rightarrow v_i^{(j)} \ge -l$. So, from (5), $v_i^{(j)}$ can range over $\{-\infty, -l, \ldots, -1, 0\}$. Since $\vec{v}_i^{(0)}, \vec{v}_i^{(1)}, \ldots, \vec{v}_i^{(m)}$ increases monotonically, $m = O(n \times (l+1) + n) = O(N(l+1))$ (note that l may be 0), where $\vec{v}_i^{(j)} = (v_1^{(j)}, \ldots, v_n^{(j)})$.

Therefore, the time complexity of the algorithm for checking whether above usage constraints are satisfiable is $O((l+1) \cdot N^2)$ as stated at the first statement of this paragraph.

§Appendix B Proofs of Theorem 4.1

This section complements the proof sketch of type soundness in Section 4, by (1) providing concrete definitions of the relations $\Gamma \vdash P$ and $(\Gamma, P) \vdash \langle \Psi, H \rangle$, and (2) proving Lemmas 4.1–4.3.

B.1 Definitions of $\Gamma \vdash P$ and $(\Gamma, P) \vdash \langle \Psi, H \rangle$

First, we define the type judgment relation $\Gamma \vdash P$, which states that program P is well-typed under the program type environment Γ (see Section 4).

Definition B.1
The relation $\Gamma \vdash P$ holds if

$$\forall \sigma \in dom(P).(P(\sigma) = (FD, D, B, E) \Rightarrow \Gamma(\sigma) \vdash_P (D, B, E))$$

Next, we define the type judgment relation $(\Gamma, P) \vdash \langle \Psi, H \rangle$ for machine states.

We first define relations $\vdash_H v : \tau$ and $P \vdash_H \rho\ ok$. The relation $\vdash_H v : \tau$ says that a value v is an integer or an object reference specified by τ. The relation $P \vdash_H \rho\ ok$ says that the run-time representation ρ of an object conforms to the class definition in program P. Those two relations are irrelevant to lock usages, and are used only to ensure that a runtime machine state conforms to the class definitions in P.

Definition B.2 (Typing rules for values)
$\vdash_H v : \tau$ is the least relation closed under the following rules:

$$\frac{v \in \mathbf{VAL}}{\vdash_H v : \mathbf{Top}} \qquad \frac{c \in \mathbf{I}}{\vdash_H c : \mathbf{Int}}$$

$$\frac{o \in \mathbf{O} \quad H(o).\mathtt{class} = \sigma}{\vdash_H o : \sigma/U} \qquad \frac{o \in \mathbf{O} \quad H(o).\mathtt{class} = A}{\vdash_H o : A/U}$$

The following subsumption property follows from the above definition (rather than as an axiom). That is because the above definition does not impose any condition on usages, and $\tau_1 \leq \tau_2$ requires either that τ_2 is **Top** or that τ_1 and τ_2 are identical except usages.

Lemma B.1
If $\tau_1 \leq \tau_2$ and $\vdash_H v : \tau_1$, then $\vdash_H v : \tau_2$.

Definition B.3 (Well-typed record)
The relation $P \vdash_H \rho\ ok$ is defined by:

$$\frac{\begin{array}{c}\overline{\sigma_P} = a_1 : d_1, \ldots, a_m : d_m \\ \vdash_H v_1 : \tau_1, \ldots, \vdash_H v_m : \tau_m \\ \mathit{Raw}(\tau_1) = d_1, \ldots, \mathit{Raw}(\tau_m) = d_m\end{array}}{P \vdash_H [\mathtt{class} = \sigma, \mathtt{flag} = b, a_1 = v_1 : d_1, \cdots, a_m = v_m : d_m]\ \mathit{ok}}$$

$$\frac{\begin{array}{c}\vdash_H v_1 : \tau_1, \ldots, \vdash_H v_m : \tau_m \\ \mathit{Raw}(\tau_1) = d, \ldots, \mathit{Raw}(\tau_m) = d\end{array}}{P \vdash_H [\mathtt{class} = d[\,], \mathtt{flag} = b, \mathit{1} = v_1 : d, \cdots, m = v_m : d]\ \mathit{ok}}$$

The first rule above is for an object, while the second one is for an array.

Next, we define a type judgment relation for thread states.

Definition B.4
The relation $(\Gamma, P) \vdash \langle\langle l, f, s, z, \sigma\rangle, H\rangle$ is defined by:

$$\frac{\begin{array}{c}\Gamma(\sigma) = \langle \mathcal{F}, \mathcal{S}\rangle \\ P(\sigma) = (FD, D, B, E) \quad l \in \mathit{dom}(B) \cup \mathit{codom}(E) \\ \forall x \in \mathit{dom}(\mathcal{F}_l).(\vdash_H f(x) : \mathcal{F}_l(x)) \quad \forall n \in \mathit{dom}(\mathcal{S}_l).(\vdash_H s(n) : \mathcal{S}_l(n)) \\ \forall o \in \mathit{dom}(H).(P \vdash_H H(o)\ ok) \\ \forall o \in \mathit{dom}(H).\mathit{rel}_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o), z(o))\end{array}}{(\Gamma, P) \vdash \langle\langle l, f, s, z, \sigma\rangle, H\rangle}$$

Here, $\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o)$ stands for $\bigotimes(\{\mathcal{F}_l(x) | f(x) = o\} \cup \{\mathcal{S}_l(n) | s(n) = o\})$, where $\bigotimes\{\tau_1, \ldots, \tau_n\}$ is defined by:

$$\bigotimes \emptyset = \mathbf{Top}$$
$$\bigotimes(\varphi \cup \{\tau\}) = \begin{cases} \bigotimes \varphi & \text{if } \tau = \mathbf{Top} \\ (\bigotimes \varphi) \otimes \tau & \text{otherwise} \end{cases}$$

(Strictly speaking, $\bigotimes$ is not a function since the result of the second clause depends on the choice of τ. Nevertheless, the result is unique up to the equivalence relation $\equiv$ on usages in Definition 3.2, hence the choice of τ actually does not matter.)

The relation $\mathit{rel}_t(U, n)$ is defined by:

$$\mathit{rel}_t(\mathbf{Top}, 0) \qquad \frac{\mathit{rel}(U, n)}{\mathit{rel}_t(\sigma / U, n)} \qquad \frac{\mathit{rel}(U, n)}{\mathit{rel}_t(\xi[\ ] / U, n)}$$

The third and fourth lines of the rule of Definition B.4 only state that the shapes of values in the local variables, the stack, and the heap match the types specified by Γ. The most important condition is the last premise, which guarantees that each object is in an intended lock state; The type $\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o)$ specifies how the object o will be locked/unlocked in the rest of the thread execution, and $z(o)$ specifies the current locking state of o. Thus, $\mathit{rel}_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o), z(o))$ ensures that the object o will be locked/unlocked safely, and return to the state where all the locks on o are released when the thread terminates.

Now, the following relation $\Gamma \vdash \langle \Psi, H\rangle$ means that every thread in Ψ is well-typed.

Definition B.5 ($(\Gamma, P) \vdash \langle \Psi, H \rangle$)

$$\frac{\forall i \in \mathit{dom}(\Psi).((\Gamma, P) \vdash \langle \Psi(i), H \rangle)}{(\Gamma, P) \vdash \langle \Psi, H \rangle}$$

B.2 Proofs of the Main Lemmas and Theorem

We first prepare some lemmas.

Lemma B.2
$U_1 \leq U_2 \ \wedge \ \mathit{rel}(U_1, n) \ \Rightarrow \ \mathit{rel}(U_2, n)$

Proof This follows immediately from the fact that $U_1 \leq U_2$ and $\langle U_2, n \rangle \rightarrow_{rel} \langle U_2', n' \rangle$ imply $\langle U_1, n \rangle \rightarrow_{rel} \langle U_1', n' \rangle$ for some U_1' (which can be proved by induction on derivation of $U_1 \leq U_2$). □

Lemma B.3
$\tau_1 \leq \tau_2 \ \wedge \ \mathit{rel}_t(\tau_1, n) \ \Rightarrow \ \mathit{rel}_t(\tau_2, n)$

Proof This follows directly from Definition 3.8 and Lemma B.2. □

The following lemma follows immediately from the definition of *rel* (Definition 3.5).

Lemma B.4
If $\mathit{rel}(U_1, n)$ and $\mathit{rel}(U_2, 0)$, then $\mathit{rel}(U_1 \otimes U_2, n)$.

Now we prove the main lemmas.

Proof of Lemma 4.1 Suppose that $\Gamma \vdash P$, $(\Gamma, P) \vdash \langle \Psi, H \rangle$ and $\Psi(i) = \langle l, f, s, z, \sigma \rangle$ hold.

- Suppose $P(\sigma) = (\mathit{FD}, D, B, E)$ and $B(l) =$ `return`.
  Because $\Gamma \vdash P$ holds, we obtain the following conditions from rule (RETURN):

$$\forall o \in \mathit{dom}(H).(\{\mathcal{F}_l(x) | f(x) = o\} \leq \mathbf{Top}) \qquad (1)$$

$$\forall o \in \mathit{dom}(H).(\{\mathcal{S}_l(n) | s(n) = o\} \leq \mathbf{Top}) \qquad (2)$$

Moreover, $(\Gamma, P) \vdash \langle\langle l, f, s, z, \sigma\rangle, H\rangle$ follows from $(\Gamma, P) \vdash \langle\Psi, H\rangle$. Therefore, the following condition holds:

$$\forall o \in \mathit{dom}(H).\mathit{rel}_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o), z(o)) \qquad (3)$$

From (1), (2) and Definition of $\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o)$, we get $\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o) \leq$ **Top**. By Lemma B.3 and (3), we obtain $\mathit{rel}_t(\mathbf{Top}, z(o))$, which implies $z(o) = 0$ as required.

- Suppose $P(\sigma) = (FD, D, B, E)$ and $l \notin \mathit{dom}(B)$.
  From $(\Gamma, P) \vdash \langle\Psi, H\rangle$ and $l \notin \mathit{dom}(B)$ and $\Gamma \vdash P$, the following condition follows:
  $$\forall o \in \mathit{dom}(H).(\{\mathcal{F}_l(x) | f(x) = o\} \leq \mathbf{Top})$$
  $$\forall o \in \mathit{dom}(H).(\{\mathcal{S}_l(n) | s(n) = o\} \leq \mathbf{Top})$$
  The rest of the proof in this case is similar to the previous case.
- Suppose $P(\sigma) = (FD, D, B, E)$ and $B(l) = \texttt{monitorexit}\ x$.
  Because $\Gamma \vdash P$ holds, we get the following conditions from rule (MEXT):
  $$\mathcal{F}_l(x) \leq_{\widehat{L}} \mathcal{F}_{l+1}(x)$$
  From this, we obtain the following conditions for some class σ' and usages U_x and U.
  $$\mathcal{F}_l(x) = \sigma'/U_x \quad U_x \leq \widehat{L}.U$$
  Since $(\Gamma, P) \vdash \langle\Psi, H\rangle$ holds, we have:
  $$\mathit{rel}_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l](f(x)), z(f(x))).$$
  So, we have
  $$\mathit{rel}(U_x \otimes U', z(f(x)))$$
  for some U_x.
  By $U_x \leq \widehat{L}.U$ and Lemma B.2, the following condition holds:
  $$\mathit{rel}(\widehat{L}.U \otimes U', z(f(x)))$$
  So, we obtain $z(f(x)) \geq 1$ from the second condition of Definition 3.5. □

Proof of Lemma 4.2 We show this by induction on derivation of $P \vdash \langle\Psi, H\rangle \rightarrow \langle\Psi', H'\rangle$, with case analysis on the last rule used. Suppose $\Gamma \vdash P$ and $(\Gamma, P) \vdash \langle\Psi, H\rangle$.

We show only main cases: The other cases are similar.

- **Case** rule (*inc*) : It must be the case that

$$\begin{aligned}&\Psi = \Psi_1 \uplus \{i \mapsto \langle l, f, c \cdot s, z, \sigma\rangle\}\\&\Psi' = \Psi_1 \uplus \{i \mapsto \langle l+1, f, c+1 \cdot s, z, \sigma\rangle\}\\&P[\sigma](l) = \texttt{inc}\end{aligned}$$

Because, $\Gamma \vdash P$ holds, $\mathcal{F}, \mathcal{S}, l \vdash_P (D, B, E)$ holds for $\mathcal{F}, \mathcal{S}, B, E$ and D such that $\Gamma(\sigma) = \langle \mathcal{F}, \mathcal{S}\rangle$, $P(\sigma) = (FD, D, B, E)$. From this, $P[\sigma](l) = B(l) = \texttt{inc}$, and rule (INC), we obtain the following conditions:

$$\begin{aligned}&\mathcal{F}_l \leq \mathcal{F}_{l+1}\\&\mathcal{S}_l(0) \leq \mathbf{Int}\\&\mathcal{S}_l \leq \mathcal{S}_{l+1}\end{aligned} \tag{4}$$

Moreover, $(\Gamma, P) \vdash \langle\langle l, f, c \cdot s, z, \sigma\rangle\}, H\rangle$ follows from the condition $(\Gamma, P) \vdash \langle \Psi, H\rangle$. Therefore, the following conditions follow from Definition B.4.

$$\begin{aligned}&\forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}_l(x))\\&\forall n \in dom(c \cdot s).(\vdash_H (c \cdot s)(n) : \mathcal{S}_l(n))\\&\forall o \in dom(H).rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, c \cdot s][l](o), z(o))\end{aligned} \tag{5}$$

By (4), (5), and Lemma B.1, the following condition holds:

$$\begin{aligned}&\forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}_l(x))\\&\forall n \in dom(c+1 \cdot s).(\vdash_H (c+1 \cdot s)(n) : \mathcal{S}_l(n))\\&\forall o \in dom(H).\\&(\Theta[\mathcal{F}, \mathcal{S}, f, c \cdot s][l](o) \leq \Theta[\mathcal{F}, \mathcal{S}, f, c+1 \cdot s][l+1](o))\end{aligned} \tag{6}$$

Moreover, from the last conditions of (5) and (6), we obtain $\forall o \in dom(H).rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, c+1 \cdot s][l+1](o), z(o))$ by using Lemma B.3. Thus, we have $(\Gamma, P) \vdash \langle\{\langle l+1, f, c+1 \cdot s, z, \sigma\rangle\}, H\rangle$ From this and $(\Gamma, P) \vdash \langle \Psi_1, H\rangle$, the relation $(\Gamma, P) \vdash \langle \Psi_1 \uplus \{i \mapsto \langle l+1, f, c+1 \cdot s, z, \sigma\rangle\}, H\rangle$ follows as required.

- **Case** rule (*ment*$_2$) : It must be the case that

$$\begin{aligned}&\Psi = \Psi_1 \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\\&\Psi' = \Psi_1 \uplus \{i \mapsto \langle l+1, f, s, z', \sigma\rangle\}\\&z' = z\{f(x) \mapsto n+1\}\\&f(x) \in dom(H) \quad z(f(x)) = n \geq 1 \quad H(f(x)).flag = 1\\&P(\sigma)(l) = \texttt{monitorenter}\ x\\&H' = H\end{aligned}$$

By the assumption $\Gamma \vdash P$, the following conditions hold:

$$\begin{gathered} y \in dom(\mathcal{F}_l) \setminus \{x\}.(\mathcal{F}_l(y) \leq \mathcal{F}_{l+1}(y)) \\ \mathcal{F}_l(x) \leq_L \mathcal{F}_{l+1}(x) \\ \mathcal{S}_l \leq \mathcal{S}_{l+1} \end{gathered} \tag{7}$$

where $\Gamma(\sigma) = \langle \mathcal{F}, \mathcal{S} \rangle$. Moreover, by the condition $(\Gamma, P) \vdash \langle \Psi, H \rangle$, the following conditions also hold:

$$\begin{array}{l} \forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}_l(x)) \\ \forall n \in dom(s).(\vdash_H s(n) : \mathcal{S}_l(n)) \\ \forall o \in dom(H).rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o), z(o)) \end{array} \tag{8}$$

From (7) and (8), we obtain the following conditions by using Lemmas B.1 and B.3.

$$\begin{array}{l} \forall y \in dom(f).(\vdash_H f(y) : \mathcal{F}_{l+1}(y)) \\ \forall n \in dom(s).(\vdash_H s(n) : \mathcal{S}_{l+1}(n)) \\ \forall o \in dom(H).(f(x) \neq o \Rightarrow rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l+1](o), z(o))) \end{array} \tag{9}$$

So, it remains to show:

$$rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l+1](f(x)), z'(f(x))).$$

By the condition $\mathcal{F}_l(x) \leq_L \mathcal{F}_{l+1}(x)$, there exist σ' and U such that:

$$\mathcal{F}_l(x) \leq \sigma'/L.U \qquad \sigma'/U \leq \mathcal{F}_{l+1}(x).$$

Hence, by the conditions (7), we have

$$\sigma'/(U \otimes U') \leq \Theta[\mathcal{F}, \mathcal{S}, f, s][l+1](f(x)) \tag{10}$$

$$\Theta[\mathcal{F}, \mathcal{S}, f, s][l](f(x)) \leq \sigma'/(L.U \otimes U') \tag{11}$$

for some U'. By Lemma B.3, we have

$$rel_t(\sigma'/(L.U \otimes U'), n) \tag{12}$$

From Definition 3.5 it follows that:

$$rel_t(\sigma'/(U \otimes U'), n+1) \tag{13}$$

So, by applying Lemma B.3 again, we obtain $rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l+1](f(x)), z'(f(x)))$. From the above conditions, we obtain $(\Gamma, P) \vdash \langle \Psi', H' \rangle$ as required.

- **Case** rule ($getfield$) : It must be the case that

$$\begin{array}{l}
\Psi = \Psi_1 \uplus \{i \mapsto \langle l, f, o \cdot s, z, \sigma\rangle \\
\Psi' = \Psi_1 \uplus \{i \mapsto \langle l+1, f, v \cdot s, z, \sigma\rangle \\
o \in dom(H) \quad H(o).\texttt{class} = \sigma' \quad H(o).a = v \\
P(\sigma)(l) = \texttt{getfield}\ \sigma'.a\ d \\
H' = H
\end{array}$$

Here we assume $d \neq \mathbf{Int}$. The proof for the case of $d = \mathbf{Int}$ is similar. By the assumption $\Gamma \vdash P$ and rule (GETFLD), the following conditions hold:

$$\begin{array}{c}
\overline{\sigma'_P}.a : d \\
\mathcal{F}_l \leq \mathcal{F}_{l+1} \\
\mathcal{S}_l \leq (\sigma'/\mathbf{0}) \cdot \mathcal{S}' \\
(d/U) \cdot \mathcal{S}' \leq \mathcal{S}_{l+1} \\
rel(U)
\end{array} \tag{14}$$

where $\Gamma(\sigma) = \langle \mathcal{F}, \mathcal{S}\rangle$. Moreover, by the condition $(\Gamma, P) \vdash \langle \Psi, H\rangle$, the following conditions also hold:

$$\begin{array}{l}
\forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}_l(x)) \\
\forall n \in dom(o \cdot s).(\vdash_H (o \cdot s)(n) : \mathcal{S}_l(n)) \\
\forall o_1 \in dom(H).(P \vdash_H H(o_1)\ ok) \\
\forall o_1 \in dom(H).rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, o \cdot s][l](o_1), z(o_1))
\end{array} \tag{15}$$

From (14) and (15), we obtain the following conditions by using Lemma B.1 and B.3.

$$\begin{array}{l}
\forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}_{l+1}(x)) \\
\forall n \in dom(v \cdot s).(n \geq 1 \Rightarrow \vdash_H (v \cdot s)(n) : \mathcal{S}_{l+1}(n)) \\
\forall o_1 \in dom(H).(o_1 \notin \{v, o\} \Rightarrow rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](o_1), z(o_1)))
\end{array} \tag{16}$$

So, it remain to show the following three conditions:

$$\begin{array}{l}
\vdash_H (v : \mathcal{S}_{l+1}(0)) \\
rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](o), z(o)) \\
rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](v), z(v))
\end{array} \tag{17}$$

The first condition follows from $\vdash_H (v : d/U)$, $(d/U) \cdot \mathcal{S}' \leq \mathcal{S}_{l+1}$ and Lemma B.1. Here, $\vdash_H (v : d/U)$ follows immediately from $\overline{\sigma'_P}.a : d$ and $P \vdash_H H(o)\ ok$.

We prove the remaining two conditions only for the case when $o \neq v$: the proof for the case when $o = v$ is similar. To check the second condition of (17), observe that the following condition holds:

$$\Theta[\mathcal{F}, \mathcal{S}, f, o \cdot s][l](o) \leq \sigma'/\mathbf{0} \otimes \Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](o).$$

By using Lemma B.3, we get

$$rel_t(\sigma'/\mathbf{0} \otimes \Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](o), z(o)),$$

which implies

$$rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](o), z(o)).$$

To check the third condition of (17), observe that the following condition holds:

$$\Theta[\mathcal{F}, \mathcal{S}, f, o \cdot s][l](v) \otimes d/U \leq \sigma'/\mathbf{0} \otimes \Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](v).$$

By the condition $rel(U)$, $rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, o \cdot s][l](v), z(v))$, and Lemma B.4, we have:

$$rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, o \cdot s][l](v) \otimes d/U, z(v)).$$

By using Lemma B.3, we obtain

$$rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](v), z(v))$$

as required.
Thus, we have $(\Gamma, P) \vdash \langle \Psi', H' \rangle$. □

Lemma 4.3 states that the initial machine state of a well-typed program is also well-typed.

Proof of Lemma 4.3 This lemma follows immediately from Definitions 3.14 and B.4. □

We can now prove the soundness of our type system.

Proof of Theorem 4.1 Suppose that P is well-typed and that $P \vdash \langle 0 \mapsto \langle 1, \emptyset, \epsilon, main_P \rangle, \emptyset \rangle \longrightarrow^* \langle \Psi, H \rangle$ and $\Psi(i) = \langle l, f, s, z, \sigma \rangle$ hold.

Since P is well-typed, there is a type environment Γ that satisfies $P \vdash \Gamma$. From this and Lemma 4.3, we obtain $(\Gamma, P) \vdash \langle 0 \mapsto \langle 1, \emptyset, \epsilon, main_P \rangle, \emptyset \rangle$.

Moreover, by Lemma 4.2, $(\Gamma, P) \vdash \langle \Psi, H \rangle$ holds. Therefore, the three conditions of Theorem 4.1 follow immediately from the relation $(\Gamma, P) \vdash \langle \Psi, H \rangle$ and Lemma 4.1. $\Box$