A New Type System for JVM Lock Primitives *

Futoshi Iwama
Tohoku University †
iwama@kb.ecei.tohoku.ac.jp

Naoki Kobayashi
Tohoku University †
koba@kb.ecei.tohoku.ac.jp

Abstract

A bytecode verifier for the Java virtual machine language (JVML) checks before the code is executed 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.

1 Introduction

A Java program [1] is usually compiled into a Java bytecode. Before it is interpreted by the Java Virtual Machine (JVM) [13], a bytecode verifier checks properties of the bytecode and rejects it if it violates certain safety requirements. According to the present definition [13], however, the bytecode verifier does not check safe usage of concurrency primitives such as lock primitives.

For this problem, Bigliardi and Laneve [2] proposed a type system for checking that lock primitives are safely used in the sense that each lock operation is followed by one unlock operation. This type system is, however, very restrictive. It essentially checks that each occurrence of the lock primitive (`monitorenter`) is *syntactically* followed by one occurrence of the unlock primitive (`monitorexit`), and it bans intertwined critical sections, jumps into other critical sections, etc.

For example, consider the four pieces of bytecode in Figure 1. *Code 1* locks the object stored in the local variable x at address 1 and then unlocks the object at 5. *Code 2* first locks the objects stored in local variables x and y at addresses 1 and 3, respectively, and then unlocks them at addresses 9 and 6 respectively. *Codes 1* and *2* are accepted by Bigliardi and Laneve's type system [2] since each lock instruction (`monitorenter` x and `monitorenter` y) is syntactically followed by a corresponding unlock instruction (`monitorexit` x and `monitorexit` y).

On the contrary, *codes 3* and *4* should be considered valid but they are rejected by their type system. *Code 3* first locks the object stored in x, loads the value stored in y, and then branches to either 4 or 6. Since the object is unlocked in both branches, the code should be considered valid. However, the code is rejected by their type system since there are *syntactically* two occurrences of the unlock primitive in *code 3* but only the one occurrence of the lock primitive. Similarly, *code 4* is rejected by their type system because the critical sections guarded by different locks are not properly nested.

Since the lock and unlock operations are coupled together in the Java source language, it is not difficult to compile a Java program into bytecode that satisfies the requirement imposed by their type

*This article is an extended version of the paper that appear in the Proceedings of ASIA-PEPM02 [10].

†Graduate School of Information Science, Tohoku University, Aramaki aza Aoba 09, Aoba-ku Sendai-city Miyagi-pref 980-8579, Japan

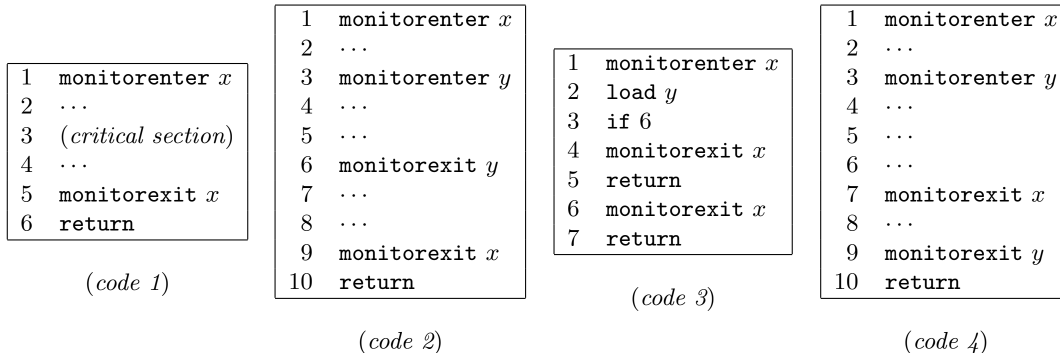

Figure 1: Programs that use lock primitives

system. However, bytecode may be edited by (possibly malicious) programmers, or produced by a compiler of another language that provides lock and unlock primitives separately. It is important to verify the safety at the bytecode level, since Java programs are often downloaded from untrusted cites in the bytecode format. Although verification of low-level code has recently been a hot research topic of programming languages[16, 15, 17], few studies have been done for verification of low-level code using concurrency primitives. Problems may also arise when optimizations are applied to bytecode. For example, a compiler optimizer may move the instruction `monitorexit` x in *code 2* above the instruction `monitorexit` y to minimize the synchronization overhead.

Recently, for such problems, Laneve have proposed a more flexible type system [12]. His type system assigns to each address the set of the types of locked objects. The `monitorenter` instruction adds the type of the locked object to that set, while the `monitorexit` instruction removes the type of the unlocked object from the set. A method is valid when the set of object types is empty at the return address. In order for the above method to work, object types must be singleton types, each of which represents a unique object. To ensure that, his type system annotates the type of an object with the address at which the object is first copied to the stack. Due to this mechanism, Laneve's type system does not suffer from the drawbacks of Bigliardi and Laneve's type system [2]. However, because of the restriction that object types must be singleton types, his type system cannot deal with the case where different objects may flow into the same variable or the same stack position. Laneve informally discuss a solution (introduction of subtyping) to partially overcome the drawback mentioned above, but it is not completely satisfactory. We discuss these issues in Section 7.

In this paper, we take a different approach and propose a new type system for JVM lock primitives. Our type system is more intuitive than the previous type systems [2, 12], and does not suffer form the drawbacks mentioned above. We expect that the idea of our type system is applicable not only to Java bytecode but also to high-level programming languages. Indeed, inspired by the present work, Igarashi and Kobayashi have formalized a resource usage analysis for a functional language [9].

The main idea of our type system is to augment the type of an object with information (which we call *usage*) about in which order the object is locked and unlocked. For example, we express by $L.\widehat{L}.\mathbf{0}$ the usage of an object that is locked, unlocked, and neither locked nor unlocked afterwards, and by $L\&\widehat{L}$ the usage of an object that is either locked or unlocked. Recall *code 3* 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 an ordinary object type (i.e., a class name) 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 [18], so that lock primitives are safely used if a program is well typed. Thus, the problem of verifying safe usage of lock primitives is reduced to the type-checking problem in the extended type system.

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 the type inference algorithm of our verifier and estimates the time complexity of the algorithm. Section 6 reports on experiments with our prototype verifier. Section 7 discusses related work and Section 8 concludes.

2 The Target Language $JVML_L$

In this section, we introduce our target language $JVML_L$ and define its operational semantics. This $JVML_L$ is a subset of the Java bytecode language JVML and is similar to the language $JVML_C$ introduced by Bigliardi and Laneve [2, 12]. 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 $JVML_L$, programs are represented by a set of class definitions:

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

where meta-variable σ denotes a *class name* and each class is defined by a subclass of *Thread* class that has only one method *run*. Class fields *FD* is a series of $a_1 : d_1, \ldots, a_k : d_k$ and meta-variables a, d denote *field name* and *descriptor* (see below about descriptor). A method is defined by a triple consisting of a *method descriptor* D, a *method body* B and an *exception table* E. To define these formally, 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 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 Σ for the set of class names σ. We introduce super- and sub-class relations on the Σ according to the syntax of JVML. When class σ_1 is the super-class of class σ_2, namely class σ_2 is a sub-class of class σ_1, we write $\sigma_2 \sqsubseteq \sigma_1$. Moreover, we introduce the class `Object` (`Object` $\in \mathbf{\Sigma}$) as the the top class on this relation. Formally, the super- and sub-class relations are defined as follows:

Definition 2.1 *The binary relation $\sqsubseteq$ is the least preorder on Σ that is closed under the following rules:*

$$\frac{}{\sigma \sqsubseteq \texttt{Object}} \qquad \frac{\textit{class } \sigma_1 \textit{ is a subclass of class } \sigma_2}{\sigma_1 \sqsubseteq \sigma_2}$$

We also define the set of array class names by $A ::= \mathbf{Int}[\,] \mid \sigma[\,] \mid A[\,]$.

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 2.2 (Instruction) *The set* **Inst** *of $JVML_L$'s* 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 shown 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 stack and local variables, create new threads, etc, by executing the instructions. We describe each instruction briefly. 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 and initializes it and then put a reference to the object on top of the operand stack. If the allocation or initialization fail, then an exception raises. 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 x. [1] As in JVML (and unlike the usual semantics of 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 method body is well-formed, which says 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 $E(l)$. The program is terminated abruptly in the case of $E(l) \notin dom(B)$.

[1] `monitorenter` x corresponds to the sequence of instructions `load` x and `monitorenter` in the actual JVML.

In general, we model abrupt termination of the program with an exception by the case where $l \notin dom(B)$ holds.

A *method descriptor* D is a mapping from a set $\{0, \ldots, n-1\} (\subseteq \mathcal{V})$ to the set $\{\textbf{Int}\} \cup \Sigma \cup A$, 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, $D(x) = \textbf{Int}$ means that the type of x-th argument is integer. We consider an element of the set $\{\textbf{Int}\} \cup \Sigma \cup A$ as a descriptor and use the meta-variable d for it.

A *program* is defined formally by a mapping from a set of class names to a pair consisting of class field FD and methods (B, E, D). we use meta-variable P to denote the program.

Notations: We use some notations. First, we use meta-variable σ_P to identify a class σ, which is declared in a program P. Second, when a class σ is consisted of a class fields $a_1 : d_1, \ldots, a_k : d_k$ and a method (B, E, D) in a program P, we write $P[\sigma] = (a_1 : d_1, \ldots, a_k : d_k, (B, E, D))$ and define $\overline{\sigma_P}$ by $[a_1 : d_1, \ldots, a_k : d_k]$. This $\overline{\sigma_P}$ is a record which specifies σ-classes class fields in a program P and we write $\overline{\sigma_P}.a_i : d_i$ if the record $\overline{\sigma_P}$ has the field a_i with descriptor d_i (namely, $\overline{\sigma_P}$ is the form $[\ldots, a_i : d_i, \ldots]$).

Restriction on our class definition: As shown above, we assume that each class has only one method *run* and it is invoked only by the instruction `start` . (So, a thread is terminated when it executes the instruction `return`.) Note that this assumption does not limit the generality of our type system because, according to the JVML specification [13], all locks acquired in a method must be released in the same method. Therefore, method invocation does not affect the state of locks.

We also assume that only `new` σ and `athrow` may throw exceptions. The `athrow` instruction raises the exception synchronously and instruction `new` σ may throw an exception asynchronously, for example, when allocation or initialization fail. Note that a null pointer exception is not raised in our model because all objects are initialized when the `new` σ instruction creates the object. For the sake of simplicity, we also assume that there is only a single kind of exception. These assumptions also does not limit the generality of our analysis. These reason are explained in section 6.

2.2 The operational semantics of $JVML_L$

We define an operational semantics of the language in a manner similar to [2, 18, 12].

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 $\textbf{VAL}$ 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* and *object, array object.* We write $\textbf{I}$ for the set of integers. We assume that there is a countably infinite set $\textbf{O}$ of *references* (to objects or arrays) and that $\textbf{O}$ contains a special element `null`, denoting the null reference. A *value* is either an integer or a reference. We write $\textbf{VAL}$ for the set $\textbf{I} \cup \textbf{O}$ of values. An *object* is a record of the form $[\texttt{class} = \sigma, \texttt{flag} = b, a_1 = v_1 : d_1, \cdots, a_m = v_m : d_m]$, 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 $\rho = [\texttt{class} = \sigma, \texttt{flag} = b, a_1 = v_1 : d_1, \cdots, a_m = v_m : d_m]$, we write $\rho.class$, $\rho.flag$ and $\rho.a_i$ for σ, b and v_i respectively. We also write $\rho\{a \mapsto v\}$ ($\rho\{flag \mapsto b\}$ reps) for the record given when we replace a value stored in field a (or $flag$) of record ρ (i.e. $\rho.a$ or $\rho.flag$) to v (or b). An *array* is also a record of the form $[\texttt{class} : A, \texttt{flag} : b, \texttt{1} = v_1 : d, \cdots, \texttt{m} = v_m : d]$, where A take the form of the $d[\ ]$ and denotes the array

class name of the array object and m is the length of this array. We write **RCD** for the set of objects and arrays and use meta-variable ρ to denote an element of the set.

As stated in section2.1, a $JVML_L$ program is executed by threads. We represent a *thread state* by a tuple

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

where $l (\in \mathcal{A})$ 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 natural numbers to **T**, and H is a partial mapping from $\mathbf{O} \backslash \{\texttt{null}\}$ 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 one-step reduction relation

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

The relation $P \vdash \langle \Psi, H \rangle \rightarrow \langle \Psi', H' \rangle$ says that a machine state $\langle \Psi, H \rangle$ can change 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 2 and 3. In the figures, $P[\sigma](l)$ denotes the instruction at address l of the method of σ-class thread in P i.e. if $P(\sigma) = (B, E, D)$ then $P[\sigma](l) = B(l)$ and address $1_{\sigma'}$ denotes the first instruction of the method `run` of the σ class. We denote by $\bar{t}$ an element of the set **T** and if $i \notin dom(\Psi)$ then $\Psi \uplus \{i \mapsto \bar{t}\}$ denotes a mapping defined by:

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

Values $v_{init}(d)$ are initial values of descriptor $d (\in \{\mathbf{Int}\} \cup \Sigma \cup A)$ and are defined by $v_{init}(\mathbf{Int})$=0 and $v_{init}(d) = \texttt{null}$ $(d \neq \mathbf{Int})$.

Many rules are straightforward. We explain briefly several rules.

Rules $(ment_1)$, $(ment_2)$: These rule simulate the situation where a thread acquire a lock of an object. The rule $(ment_1)$ states a thread can acquire a lock of an object if the object is not locked and the rule $(ment_2)$ states a thread also can acquire a lock of an object if the object is locked by the same thread.

Rules $(mext_1)$, $(mext_2)$: These rule simulate the situation where a thread releases a lock of an object. The rule $(mext_1)$ states that a thread can release a lock of an object if the thread have acquired the lock one time before and that the object becomes unlocked in this case. The rule $(mext_2)$ states that a thread can release a lock of an object if the thread have acquired the lock more than one time before.

Rules (*new*), (new_{exc}): These rules simulate a situation where a thread try to create a new object. The execution of the *New* instruction nondeterministically either creates a new object or breaks down with an asynchronous exception. The rule (*new*) describes the transition for the successful object creation and the rule (new_{exc}) describes the situation that object allocation fails and the asynchronous exception raise.

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*)).
- *Uncaught exceptions* : An exception is raised by the current instruction, but, there is no handler.
- *Lock error* : The current instruction is `return`, but the thread has not released the lock and the current instruction is `monitorexit` , but the 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.

Restriction on our operational semantics: As stated above, in the operational semantics, the program get stuck in the type mismatch situations For example, for the rule *if*, if the top element on the stack is not integer. the program get stuck. While, in actual JVML, th type mismatches raise an exception. This discrepancy is not important since such programs are rejected by original bytecode verifier as well as by the type system we describe later.

3 Type system

In this section, we give a type system for checking that programs ues 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 construct typing rules for the extended types. It is an extension of Stata and Abadi's type system [18] for JVML.

3.1 Usages and types

As mentioned above, we augment the type of an object with a usage expression, which represents in which order the object is locked and unlocked.

Definition 3.1 (usages) *The set $\mathcal{U}$ of usage expressions (usages, in short) is defined by:*

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

Usage $\mathbf{0}$ describes an object that cannot be locked or unlocked at all. α denotes a usage variable, which is bound by a recursion operator $\mu\alpha$. 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 is either locked and then unlocked, or unlocked and then locked. $U_1 \& U_2$ describes an object that is used according to either U_1 or U_2. Usage $\mu\alpha.U$ describes an object that is 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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c} P[\sigma](l) = \texttt{new}\ \sigma' \quad \overline{\sigma'}_P = [a_1 : d_1, \ldots, a_m : d_m] \\ o \notin dom(H) \quad H' = H\{o \mapsto [\texttt{class} = \sigma', \texttt{flag} = 0, a_1 = v_{init}(d_1) : d_1, \cdots, a_m = v_{init}(d_m) : d_m]\} \end{array}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma \rangle\}, H \rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, o \cdot s, z, \sigma \rangle\}, H' \rangle} \quad (new)$$

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

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

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

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

Figure 2: Operational semantics

$$\frac{\begin{array}{c}P[\sigma](l) = \texttt{monitorenter}\ x \quad f(x) \in dom(H) \quad z^{\#}(f(x)) = 0 \\ H(f(x)).flag = 0 \quad H' = H\{f(x) \mapsto \rho\} \quad \rho = H(o)\{flag \mapsto 1\}\end{array}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z\{f(x) \mapsto 1\}, \sigma\rangle\}, H'\rangle} \quad (ment_1)$$

$$\frac{P[\sigma](l) = \texttt{monitorenter}\ x \quad f(x) \in dom(H) \quad z^{\#}(f(x)) = n \geq 0 \quad H(f(x)).flag = 1}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z\{f(x) \mapsto n+1\}, \sigma\rangle\}, H\rangle} \quad (ment_2)$$

$$\frac{\begin{array}{c}P[\sigma](l) = \texttt{monitorexit}\ x \quad f(x) \in dom(H) \quad z^{\#}(f(x)) = 1 \\ H(f(x)).flag = 1 \quad H' = H\{f(x) \mapsto \rho\} \quad \rho = H(o)\{flag \mapsto 0\}\end{array}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z \setminus f(x), \sigma\rangle\}, H'\rangle} \quad (mext_1)$$

$$\frac{P[\sigma](l) = \texttt{monitorexit}\ x \quad f(x) \in dom(H) \quad z^{\#}(f(x)) = n \geq 2 \quad H(f(x)).flag = 1}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z\{f(x) \mapsto n-1\}, \sigma\rangle\}, H\rangle} \quad (mext_2)$$

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

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

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

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

Figure 3: Operational semantics

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 the usage expression 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 the hole $[\,]$. We use a meta-variable C to denote a usage context. The expression $C[U_1,\dots,U_n]$ denotes the usage obtained by substituting $U_1,\dots,U_n$ 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 $U_1,\dots,U_n$ 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 the associativity and commutativity laws on $\otimes$ and $\&$, and the rules $U\otimes\mathbf{0}\equiv U$ and $\mu\alpha.U\equiv[\mu\alpha.U/\alpha]U$.*

We define types as follows:

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

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

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

Int is the type of integers. **Top** is the type 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 3.4 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 3.4, 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 $rel(U)$ 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 3.5 *The usage pair reduction $\to_{rel}$ is the least binary relation on $\mathcal{U}\times\mathcal{N}$ closed under the following rules.*

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

$$\frac{\langle U_1,n\rangle \to_{rel} \langle U_1',n'\rangle}{\langle U_1\&U_2,n\rangle \to_{rel} \langle U_1',n'\rangle} \quad \frac{\langle U_2,n\rangle \to_{rel} \langle U_2',n'\rangle}{\langle U_1\&U_2,n\rangle \to_{rel} \langle U_2',n'\rangle}$$

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

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

We can now define the reliability of usages as follows:

Definition 3.6 (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 the definition 3.6 states that if an object may not be accessed future by a thread (i.e. if $U' \equiv \mathbf{0}$ or $U' \equiv \mathbf{0}\&U_1$), the object is not locked by the thread (i.e. $n' = 0$). The condition *2* in the definition states that when a lock on 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 object has locked more than once time by the thread (i.e. $n' \geq 1$). These conditions guarantees 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 has acquired the lock previously.

Therefore, the definition of *$rel(U)$* indicates that objects, which are not locked and will be accessed according to reliable usage U by a thread, are properly locked and unlocked by the thread.

Example 3.7
$L.L.(\widehat{L} \otimes \widehat{L})$, $(L.\widehat{L})\&(L.\widehat{L})$, $L.\widehat{L}.\bot_{\mathbf{rel}}$ 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:

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

Definition 3.8 *The* sub-usage relation *$\leq$ is the least preorder on usages that includes the relation $\equiv$ and is closed under the following rules:*

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

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

Definition 3.9 *The binary relation $\equiv$ on types is the least equivalence relation that satisfies the following rules:*

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

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.

Definition 3.10 *The* subtype *relation is the least preorder that includes the relation* $\equiv$ *on types and is closed under the following rules:*

$$\mathbf{Int} \leq \mathbf{Top} \qquad \frac{U \leq \mathbf{0}}{\sigma/U \leq \mathbf{Top}} \qquad \frac{U_1 \leq U_2 \quad \sigma_1 \sqsubseteq \sigma_2}{\sigma_1/U_1 \leq \sigma_2/U_2}$$

$$\frac{U \leq \mathbf{0}}{\xi[\,]/U \leq \mathbf{Top}} \qquad \frac{\xi_1 = \xi_2 \quad U_1 \leq U_2}{\xi_1[\,]/U_1 \leq \xi_2[\,]/U_2} \qquad \frac{U_1 \leq U_2}{\xi[\,]/U_1 \leq \texttt{Object}/U_2}$$

Note that, the last rule follows from the fact that each array class is a direct subclass of the class `Object`.

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 3.11 *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 3.12 *We define* $\tau_1 \leq_{\dot{L}} \tau_2$ *by:*

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

These relation $\tau_1 \leq_L \tau_2$ and $\tau_1 \leq_{\widehat{L}} \tau_2$ means 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 $\texttt{Counter}/\widehat{L}.\mathbf{0} \leq_{\widehat{L}} \mathbf{Top}$ and $Counter[\,]/\widehat{L}.\mathbf{0} \leq_{\widehat{L}} \mathbf{Top}$ hold.

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

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

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 a value stored in the local variable x.

A *stack type*, denoted by a meta-variable S, is a partial mapping form $\mathcal{N}$ to $\mathcal{T}$. $S(n)$ denotes the type of a 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 extend some operations and relations on types to ones on *frame types* or *stack types.*

Definition 3.13 *Suppose that* $dom(F_1) = dom(F_2)$*. Then* $F_1 * F_2$ *is defined by:*

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

Definition 3.14 *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 $\langle \mathcal{F}, \mathcal{S} \rangle \vdash_P (B, E, D)$. It means that the method (B, E, D) 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 $\mathcal{F}, \mathcal{S}, l \vdash_P (B, E, D)$. 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 3.15 $\mathcal{F}, \mathcal{S}, l \vdash_P (B, E, D)$ *is the least relation closed under the rules in Figure 4.*

In Figure 4 and 5 , $\mathcal{F}_l$ 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 a 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 conditions.

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

$$\frac{\begin{array}{c} \forall x \in dom(\mathcal{F}(1)).\ rel_t(\mathcal{F}(1)(x)) \\ Raw(\mathcal{F}(1)(x)) = \begin{cases} D(x) & \text{if } x \in dom(D) \\ \mathbf{Top} & otherwise \end{cases} \\ \mathcal{S}(1) = \epsilon \\ \forall l \in codom(E).(\mathcal{S}(l) = \epsilon) \\ \forall l \in dom(B) \cup codom(E).\ \mathcal{F}, \mathcal{S}, l \vdash_P (B, E, D) \end{array}}{\langle \mathcal{F}, \mathcal{S} \rangle \vdash_P (B, E, D)}$$

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

$$\begin{array}{l} Raw(\mathbf{Int}) = \mathbf{Int} \\ Raw(\mathbf{Top}) = \mathbf{Top} \\ Raw(\sigma/U) = \sigma \\ Raw(\xi[\ ]/U) = \xi[\ ] \end{array}$$

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 states that the operand stack at the beginning of the method or at the beginning of an exception handler is empty. The last line states that the method is well-typed at each address.

Definition 3.17 (Well-typed program) *A program P is well-typed if for each class name $\sigma \in dom(P)$, there exist $\mathcal{F}$ and $\mathcal{S}$ such that $P(\sigma) = (FD, (B, D, E))$ and $\langle \mathcal{F}, \mathcal{S} \rangle \vdash_P (B, E, D)$ holds.*

Example 3.18 *Code 3 in the figure 1 is well-typed as shown in Figure 6.*

Example 3.19 *The method in Figure 7 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.

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

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 3.20 *The method in Figure 8 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.*

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

(PUSH)
$$\frac{B(l) = \texttt{push0} \quad l+1 \in dom(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \mathbf{Int} \cdot \mathcal{S}_l \leq \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

(POP)
$$\frac{B(l) = \texttt{pop} \quad l+1 \in dom(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \quad \mathcal{S}_l \leq \mathbf{Top} \cdot \mathcal{S}_{l+1}}{\mathcal{F}, S, l \ \vdash_P (B, E, D)}$$

(IF)
$$\frac{B(l) = \texttt{if } l' \quad l',\ l+1 \in dom(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1} \& \mathcal{F}_{l'} \quad \mathcal{S}_l \leq \mathbf{Int} \cdot (\mathcal{S}_{l+1} \& \mathcal{S}_{l'})}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

(LOAD)
$$\frac{B(l) = \texttt{load } x \quad l+1 \in dom(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1}\{x \mapsto \mathcal{F}_{l+1}(x) \otimes \mathcal{S}_{l+1}(0)\} \quad \mathcal{S}_{l+1}(0) \cdot \mathcal{S}_l \leq \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

(STORE)
$$\frac{B(l) = \texttt{store } x \quad l+1 \in dom(B) \quad \mathcal{F}_l \leq \mathcal{F}_{l+1}\{x \mapsto \mathbf{Top}\} \quad \mathcal{S}_l \leq \mathcal{F}_{l+1}(x) \cdot \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

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

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

(RETURN)
$$\frac{B(l) = \texttt{return} \quad \mathcal{F}_l \leq \mathbf{Top} \quad \mathcal{S}_l \leq \mathbf{Top}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

(MENTR)
$$\frac{B(l) = \texttt{monitorenter } x \quad l+1 \in dom(B) \quad \mathcal{F}_l \setminus x \leq \mathcal{F}_{l+1} \setminus x \quad \mathcal{F}_l(x) \leq_L \mathcal{F}_{l+1}(x) \quad \mathcal{S}_l \leq \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

(MEXT)
$$\frac{B(l) = \texttt{monitorexit } x \quad l+1 \in dom(B) \quad \mathcal{F}_l \setminus x \leq \mathcal{F}_{l+1} \setminus x \quad \mathcal{F}_l(x) \leq_{\hat{L}} \mathcal{F}_{l+1}(x) \quad \mathcal{S}_l \leq \mathcal{S}_{l+1}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

(ATHROW)
$$\frac{B(l) = \texttt{athrow} \quad \mathcal{F}_l \leq \mathcal{F}_{E(l)} \quad \mathcal{S}_l \leq \mathbf{Top}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

(BREAK)
$$\frac{l \notin dom(B) \quad l \in codom(E) \quad \mathcal{F}_l \leq \mathbf{Top} \quad \mathcal{S}_l \leq \mathbf{Top}}{\mathcal{F}, \mathcal{S}, l \ \vdash_P (B, E, D)}$$

Figure 4: Typing rules

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

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

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

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

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

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

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

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

Figure 5: 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	$\texttt{S}/L.(\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0})$	**Int**	ϵ
2	`load` 1	$\texttt{S}/\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0}$	**Int**	ϵ
3	`if` 6	$\texttt{S}/\widehat{L}.\mathbf{0}\&\widehat{L}.\mathbf{0}$	**Int**	$\textbf{Int} \cdot \epsilon$
4	`monitorexit` 0	$\texttt{S}/\widehat{L}.\mathbf{0}$	**Int**	ϵ
5	`return`	$\texttt{S}/\mathbf{0}$	**Int**	ϵ
6	`monitorexit` 0	$\texttt{S}/\widehat{L}.\mathbf{0}$	**Int**	ϵ
7	`return`	$\texttt{S}/\mathbf{0}$	**Int**	ϵ

Figure 6: Typing for code 3 in the figure 1

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

Figure 7: 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

Figure 8: Typing for an example code

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:

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 $i \in \Psi$, if $\Psi(i) = \langle l, f, s, z, \sigma \rangle$, 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 $z(f(x)) \geq 1$.*

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 3.16).

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.

Remark We do not discuss the usual type safety and focus on the usage of locks. The reason is that if we drop usage expressions from our types system, the resulting type system is almost the same as Stata and Abadi's type system (except some difference of supported instructions), so, usual (no locking-related) soundness of our type system follows immediately. Actually, we can show the following progress theorem

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 $\langle \Psi', H' \rangle$*,* $P \vdash \langle \Psi, H \rangle \to \langle \Psi', H' \rangle$ *holds*

In the theorem, the condition 1 means that all threads terminate normally or abruptly. the condition 2 says evaluation can make another step. We do not prove here the standard progress property which says that evaluation of a well-typed program does not get stuck.

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 (B, E, D) of the program, whether there exist $\mathcal{F}$ and $\mathcal{S}$ such that $\langle \mathcal{F}, \mathcal{S} \rangle \vdash_P (B, E, D)$ 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 [14, 7] except for the last step. We illustrate how type inference works using an example. Consider the third method body 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, by using and Stata and Abadi's type system [18]). The frame type environment $\mathcal{F}$ and the stack type environment $\mathcal{S}$ of the method are given as:

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

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 3.16), we obtain the following constraints:

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

From the constraints except for the first one, we obtain a solution $\alpha_1 = L.((\widehat{L}.\mathbf{0})\&(\widehat{L}.\mathbf{0}))$. 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. As in type systems for deadlock-freedom [11], this problem can be reduced to the reachability problem of Petri nets [4], 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. The size of local variables and stack frames is $O(k)$. Therefore, 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 occurrence of the usage constructor $L.$, namely `monitorenter` instruction in the method, and N is the size of constraints generated in Step 1. Furthermore, l is $O(k)$ and N is $O(k^2)$, therefore, the total time complexity is $O(k^5)$.

However, if we assume that the number of local variables and the size of stack frames is bound by a constant and that the number of instruction `monitorenter` in the method is also a constant, we can expect that 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 language using Objective Caml. The implementation is mostly based on the formal system described so far. There are, however, some differences between the formal system and the implementation, that is due to the difference between $JVMLL$ and *JVML*. We first discuss the main difference between $JVMLL$ and *JVML* and explain how the type system has been extended for deal with the real $JVML$.

Differences between our model and real JVM The language $JVML_L$ is a restriction of JVML and our operational model simulates real $JVML$'s model to a large extent, but, there are some definite differences between both models.

First, we assume that there is only a single kind of exception in our model unlike real *JVML*. This assumptions does not limit the generality of our analysis, because it is sufficient for our analysis to determine target program address to which the control jumps when an exception raised. In real JVM, the target program is able to be decided statically by a type of the exception and an address at which the exception is raised in the same way that usual bytecode verifier decide it. Therefore, if we want to introduce many kinds of exceptions into our model, it is only necessary to model exception table E as a total mapping from pairs consisting of an address $l \in dom(B)$ and a type of exception to $\mathcal{A}$.

Second, in our target language $JVML_L$, only `new` σ and `athrow` instructions raise an exception, but in actual JVML, many other instructions may raise an exception. For example in the JVML, both `getfield` $\sigma.a\ d$ and `putfield` $\sigma.a\ d$ instructions may raise a null pointer exception. Note that the `new` σ instruction in actual JVML only allocates a new memory block to a new object and does not initialized the memory block. Therefore a null pointer exception may be raised, for example, during the

execution of `getfield` $\sigma.a\ d$ instruction if an object on the top of the operand stack is not initialized. To deal with real JVML's instructions that may raise an exception, we have only to modify typing rules for these instructions like the rule (NEW). For example, for the `getfield` $\sigma.a\ d$ instruction, we modify the rules ($\text{PUTFLD}_{\mathbf{Int}}$), (PUTFLD) as the following rules:

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

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

We change the condition $\mathcal{F}_l \leq \mathcal{F}_{l+1}$ to the condition $\mathcal{F}_l \leq \mathcal{F}_{l+1} \& \mathcal{F}_{E(l)}$ and add the condition $\mathcal{S}_l \leq \mathbf{Top}$. The changed condition states that values in local variables may be accessed according to $\mathcal{F}_{E(l)}$ (or may be accessed according to $\mathcal{F}_{l+1}$) and the additional condition states that values in the operand stack may be not accessed later. In cases where an exception is raised during the execution of `getfield` $\sigma.a\ d$ instruction, the values in local variables and the operand stack are accessed according to $\mathcal{F}_{E(l)}$ and **Top** actually. The modification allow for the cases. In this way, we deal with instructions that may raise an exception.

Our verifier Our verifier receives Java class files and checks that all methods in the class use lock primitives safely base on the type system. Moreover, the verifier has two modes. In the first mode, the verifier gives only a yes/no answer on whether each method is well-typed. In the second mode, the verifier pretty-print inferred types.

For example, let us consider the deposit method written by Java language at Account.java in Figure 9. The deposit method is compiled into *code Dep* in the figure.

Given the class file generated from `Account.java` in Figure 9 our prototype verifier displays 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 return value. `Lock Check :` indicates whether verification succeeded or not.

```
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*)

Figure 9: Java source code for an Account class and $JVML_L$ code for the deposit method in the source code: `add` instruction popes two integers from the operand stack, adds the two values and push the integer sum onto the stack. The `goto` l instruction jumps to the program address l.

The message for the deposit method states that the first argument of the method is Account-class object, which is locked (`L.`) and then unlocked(`UL.`) eventually in the method, the second argument of the method is of type **Int** and that the method returns no values. The line "`Lock Check : true`" indicates that all objects in method are properly locked and unlocked. So, the method is well typed with both normal verifier and our (lock-checking) verifier.

If we remove the `monitorexit` 2 instruction at program address 13 in *code Dep* in Figure 9 our verifier displays the following message:

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

This states that the modified code has been rejected by the verifier. The type `Account/L.(UL.0 & 0)` of the first argument indicates 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 checked several class files in Java run time class libraries. All the classes were verified successfully. Figure 10 shows the time spent for the verification of each class. As stated above, our verifier has two modes. In the first mode, the verifier gives only a yes/no answer on whether each method is well-typed. In the second mode, the verifier pretty-print inferred types (as shown above). $Time_v$ shows the execution time for the first mode and $Time_i$ shows one for the second mode.

Class name	*Size* (bytes)	N_{LM}	$Time_v$ (seconds)	$Time_i$ (seconds)
java.lang.Throwable	1559	3	0.003	0.016
java.io.StringReader	1905	6	0.019	0.714
java.lang.ref.ReferenceQueue	2320	6	0.026	1.019
java.lang.Pakage	6490	2	0.009	0.224
java.lang.Thread	7095	1	0.007	0.139
java.net.InetAddress	7647	4	0.018	0.530
java.lang.ThreadGroup	7274	14	0.172	22.041
java.util.ResourceBundle	8655	4	0.053	35.932
java.net.URL	9012	4	0.075	22.138
java.lang.SecurityManager	9128	3	0.037	2.234
java.lang.ClassLoader	14233	6	0.119	24.728

Figure 10: *Size* is the byte size of each class and N_{LM} indicates the number of methods including lock or unlock instructions.

We show more details for each method that includes lock primitives in *java.lang.ThreadGroup* and *java.lang.ClassLoader* classes in Figure 11.

7 Related Work

Our type system was obtained by extending Stata and Abadi's type system for JVML [18] with usages. Bigliardi and Laneve [2] have proposed a type system for checking usage of concurrency primitives including lock primitives. As mentioned in Section 1, the type system is rather complex and it imposes strong restrictions on usage of lock primitives.

Recently, Laneve proposed a more flexible type system [12]. The type system uses *indexed object types*, which are singleton types obtained by annotating a normal object type (i.e. class name) with a program address where each 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 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 *code 3* in Figure 1 as shown in Figure 12.

However, since object types must be singleton types, the type system cannot deal with a case where multiple objects flow to the same variable. For example, consider the *code* 5 in Figure 13. 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. Moreover, even with that extension, subroutines cannot be dealt with properly. On the other hand, the code 5 is well-typed in our type system as shown in Figure 14.

While, there are also some bytecode that can be typed in Laneve's type system but not in our type system.

First, our type system cannot deal with bytecodes that include `load` x instruction and `monitorenter`, `monitorexit` instructions separately. Such kind of bytecodes are rearly generated except for the following left code produced by Microsoft compilers:

java.lang.ClassLoader

Method name	*Insts*	*Maximal frames*	N_{LP}	$Time_v$ (seconds)	$Time_i$ (seconds)
getPackage	44	9	3	0.009	0.210
difinePackage	45	23	3	0.021	0.778
getDefaultDomain	50	9	3	0.004	0.197
findNative	50	16	3	0.010	0.154
getPackages	62	10	4	0.012	0.588
loadLibrary0	193	22	11	0.075	23.762

java.lang.ThreadGroup

Method name	*Insts*	*Maximal frames*	N_{LP}	$Time_v$ (seconds)	$Time_i$ (seconds)
interrupt	58	11	3	0.008	0.271
resume	58	11	3	0.008	0.268
activeGroupCount	58	11	4	0.008	0.156
activeCount	59	12	4	0.008	0.175
setMaxPriority	60	9	3	0.009	0.402
add	63	10	3	0.011	0.572
add	63	10	3	0.011	0.569
remove	78	11	4	0.012	1.743
remove	78	11	4	0.012	1.871
destroy	80	10	3	0.013	1.052
list	83	15	3	0.017	1.109
stopOrSuspend	87	14	3	0.017	2.582
enumerate	92	14	3	0.019	3.368
enumerate	100	15	4	0.022	4.427

Figure 11: *Insts* indicates the number of instructions in each method, *Maximal frames* is the sum of maximal local variables and maximal stack size used during execution of each method and N_{LP} is the number of lock primitives (i.e. `monitorenter` x and `monitorexit` x) in each methods.

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	$\{\}$

Figure 12: Typing for Code 3 in Figure 1 in Laneve's type system

```
load x
dup                      load x
store y                  store y
monitorenter             monitorenter y
...                      ...
load y                   monitorexit y
monitorexit              return
return
```

where, the instruction `dup` makes an extra copy of the top item on the stack and adds it to the operand stack.

For the above code, our verifier replaces the sequence of instructions `dup`, `store` y, `monitorenter` / `monitorexit` with `dup`, `store` y, `monitorenter` y / `monitorexit` y (i.e. the left code is considered as the above right code) and then performs type checking.

Second, 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 *code 7* in Figure 15. 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 code 8: the usage $L.\widehat{L}$ is assigned to object S at address 1.) We think that this kind of code rarely appears in practice. If it is necessary to analyze such code, we can extend the type system by using an idea presented in the generic type system for the π-calculus [8].

Recently, various methods for statically analyzing usage of lock primitives have been proposed for other languages [3, 5]. 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 [11, 19] 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.

Recently, Igarashi and Kobayashi [9] 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 [9]. 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 Igarashi and Kobayashi's paper [9].

```
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
```

(*code 5*)

Figure 13: A program that Laneve's type system does not accept successfully

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

Figure 14: Typing for *Code* 5

```
1  new S               1  new S
2  store x             2  store x
3  load x              3  monitorenter x
4  store y             4  load x
5  monitorenter x      5  store y
6  monitorexit y       6  monitorexit y
7  return              7  return
```

(*code 6*) (*code 7*)

Figure 15: Programs that lock and unlock an object through different variables

Higuchi and Ohori [6] have proposed a type system for Java bytecode that is based on a λ-calculi-like typed term calculus. We consider that it is not difficult to introduce types with *lock usage* in our type system into their framework.

8 Conclusion

We have proposed a type system for checking usage of lock primitives for a subset of JVML [13], 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 language based on the type system. Finally, we have confirmed that the verifier runs properly and efficiently by checking several classes in Java run time library with the verifier.

References

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

[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] 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.

[4] J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Journal of Information Processing and Cybernetics*, 30(3):143–160, 1994.

[5] 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.

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

[7] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. To appear ACM Transactions on Programming Languages and Systems. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 331–342, 2002.

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

[9] 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.

[10] Futoshi Iwama and Naoki Kobayashi. A new type system for JVM lock primitives. In *Proceedings of ASIA-PEPM'02*, pages 156–168. ACM Press, 2002.

[11] 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.

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

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

[14] 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.

[15] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *Proceedings of Workshop on Compiler Support for System Software*, pages 25–35, May 1999.

[16] 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.

[17] 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.

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

[19] 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.

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

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

where $\{r_1, \ldots, r_h\} \subseteq \{1, \ldots, n\}$ and $\alpha_1, \ldots, \alpha_n$ are differnt 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 $rel(\alpha_{r_1}), \ldots, rel(\alpha_{r_h})$.

To check whether $rel(U)$ 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, $Fin_U = -\infty$).

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

By Definition 3.6, $rel(U)$ if and only if $(1) Min_U = 0$ and $(2) Fin_U \leq 0$.

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

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

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

C_{Min} can be expressed in the form

$$\begin{array}{l} \{x_1 = F_1(x_1, \ldots, x_n), \\ \cdots, \\ \ x_n = F_n(x_1, \ldots, x_n)\} \end{array}$$

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

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

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

Because we are only interesting in the value of $Min_{\alpha_{r_i}}$ $(i = 1, \ldots, h)$, we can remove $x_i = F_i(x_1, \ldots, x_n)$ 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 $(u_1^{(j)}, \ldots, u_n^{(j)})$ $(j = 0, 1, \cdots)$ by

$$\begin{array}{l} u_i^{(0)} = 0 \\ u_i^{(j+1)} = F_i(u_1^{(j)}, \ldots, u_n^{(j)}) \end{array}$$

until $j = m$ such that $(u_1^{(m+1)}, \ldots, u_n^{(m+1)}) = (u_1^{(m)}, \ldots, u_n^{(m)})$ or $u_i^{(m)} < 0$ for some $i \in \{r_1, \ldots, r_h\}$. (Note that such m always exists.) and check whether $u_i^{(m)} = 0$ for all $i \in \{r_1, \ldots, r_h\}$. 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 $Fin_{\alpha_{r_i}} \leq 0$ $(i = 1, \ldots, h)$ holds can be checked in a similar manner. Let $y_1, \ldots, y_n$ be variables denoting $Fin_{\alpha_1}, \ldots, Fin_{\alpha_n}$. 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, $\mathit{FinExp}(U)$ is the expression defined by:

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

C_{Fin} can be expressed in the form

$$\begin{array}{l}
\{y_1 = G_1(y_1, \ldots, y_n), \\
\cdots, \\
\ y_n = G_n(y_1, \ldots, y_n)\}.
\end{array}$$

As in the previous case, we assume

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

We first find i such that the least solution of C_{Fin} satisfies $y_i = -\infty$ as follows:
Compute $(z_1^{(j)}, \ldots, z_n^{(j)})$ by

$$\begin{array}{l}
z_i^{(0)} = -\infty \\
z_i^{(j+1)} = \mathit{to_fin}(G_i(z_1^{(j)}, \ldots, z_n^{(j)}))
\end{array}$$

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

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

and operations on *fin* are defined by:

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

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

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 $G_1, \ldots, G_n$ we transform C_{Fin} to C'_{Fin}. To define this C'_{Fin} formally, we define Var^{fin} as $\{y_i \in \{y_1, \ldots, y_n\} \mid z_i^{(m')} = fin\}$ and Var^{inf} as $\{y_i \in \{y_1, \ldots, y_n\} \mid z_i^{(m')} = -\infty\}$. C'_{Fin} is defined by:

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

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

Without loss of generality, we can assume

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

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

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

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

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

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

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

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

until $j = m$ such that $v_i^{(m)} = v_i^{(m+1)}$ for all $i \in \{i_1, \ldots, i_{n'}\}$ or $v_i^{(m)} > 0$ for some $i \in \{r_1, \ldots, r_h\}$. (Such m always exists.) and check whether $v_i^{(m)} \leq 0$ holds for all $i \in \{r_1, \ldots, r_h\}$. 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 $\{U_1, \ldots, U_n\}$ (This l may be 0). Moreover, n (This is the number of constraints) is estimated to $O(N)$. In the rest of this section, we discuss this time complexity.

First, we calculate time complexity of checking whether $Min_{\alpha_{r_i}} = 0$ $(i = 1, \ldots, h)$ 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 $(Min_{U_1}^{(j)}, \ldots, Min_{U_n}^{(j)})$ 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 $u_1^{(j)} \leq 0, \ldots, u_n^{(j)} \leq 0$. 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 $\overline{Var}(F_{r_1}) \cup \cdots \cup \overline{Var}(F_{r_h}) = \{x_1, \cdots, x_n\}$, we have a series of variables $x_i, x_{k_1}, \ldots, x_{k_p}$ $(0 \leq p \leq n)$ 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, $u_i^{(j)}$ $(j = 0, \ldots, m-n-1)$ can range over $\{-l+1, \ldots, -1, 0\}$. Since $\vec{u}_i^{(0)}, \vec{u}_i^{(1)}, \ldots, \vec{u}_i^{(m)}$ decreases monotonically, $m = O(n \times l + n + 1) = O(n(l+1))$, where $\vec{u}_i^{(j)} = (u_1^{(j)}, \ldots, u_n^{(j)})$ (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

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

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

(4) The following fact holds:

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

- The proof is by induction on the strudture 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 $y_{i'}, y_{k_1}, \ldots, y_{k_p}\ (0 < p \leq n)$ such that $k_p = r_k$ and the following conditions hold:

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

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 $u_{i'}^{(j)} \leq 0$ 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 $Min_{U_{r_k}} = 0$, $v_{r_k}^{(j+n)} \geq v_{r_k}^{(j+p)} > u_{r_k}^{(j+p)} \geq Min_{U_{r_k}} = 0$ follows.

From (1),(3) and the assumption that $Min_{\alpha_{r_i}} = 0\ (i = 1, \ldots, h)$ are checked, we have $v_i^{(j)} \neq -\infty\ \Rightarrow\ v_i^{(j)} \geq -l$. 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.

B Proof of Theorem 4.1

We prove the soundness of our type system in this section.

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* $\Gamma \vdash P$ *is defined by:*

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

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 $P \vdash_H \rho$ *well-typed record*. the relation $\vdash_H v : \tau$ says that a value v, that may be a reference to object stored in heap H, has type τ and the relation $P \vdash_H \rho$ *well-typed record* says that all values in record ρ stored in heap H have types specified by class definitions in program P.

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

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

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

Definition B.3 (Well-typed record) $P \vdash_H \rho$ *well-typed record is defined by:*

$$\frac{\begin{array}{c} \rho.\mathit{class} = \sigma \\ \overline{\sigma_P} = a_1 : d_1, \ldots, a_m : d_m \\ \vdash_H \rho.a_1 : \tau_1, \ldots, \vdash_H \rho.a_m : \tau_m \\ \mathit{Raw}(\tau_1) = d_1, \ldots, \mathit{Raw}(\tau_m) = d_m \end{array}}{P \vdash_H \rho \ \textit{well-typed record}} \qquad \frac{\begin{array}{c} \rho.\mathit{class} = A \\ A = d[\,] \\ \vdash_H \rho.1 : \tau_1, \ldots, \vdash_H \rho.m : \tau_m \\ \mathit{Raw}(\tau_1) = d, \ldots, \mathit{Raw}(\tau_m) = d \end{array}}{P \vdash_H \rho \ \textit{well-typed record}}$$

These are type judgment rules for objects and arrays in the heap. In the above definition, the left is the type judgment rule for an object and the right is for an array object.

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:*

$$\frac{\begin{array}{c} \Gamma(\sigma) = \langle \mathcal{F}, \mathcal{S} \rangle \\ P(\sigma) = (FD, (B, D, E)) \quad l \in \mathit{dom}(B) \cup \mathit{codom}(E) \\ \forall x \in \mathit{dom}(\mathcal{F}(l)).(\vdash_H f(x) : \mathcal{F}(l)(x)) \quad \forall n \in \mathit{dom}(\mathcal{S}(l)).(\vdash_H s(n) : \mathcal{S}(l)(n)) \\ \forall o \in \mathit{dom}(H).(P \vdash_H \rho \ \textit{well-typed record}) \\ \forall o \in \mathit{dom}(H).\mathit{rel}_t(\bigotimes(\{\mathcal{F}(l)(x) | f(x) = o\} \cup \{\mathcal{S}(l)(n) | s(n) = o\}), z(o)) \end{array}}{(\Gamma, P) \vdash \langle \langle l, f, s, z, \sigma \rangle, H \rangle}$$

Here, $\bigotimes\{\tau_1, \ldots, \tau_n\}$ 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.) We use a shorthand form $\Theta[\mathcal{F},\mathcal{S},f,s][l](o)$ for the expression $\bigotimes(\{\mathcal{F}(l)(x)|f(x)=o\}\cup\{\mathcal{S}(l)(n)|s(n)=o\}$.

The relation $rel_t(U,n)$ is defined by:

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

The third line of the rule of Definition B.4 states that values in local variables and operand stack are typed correctly. The fourth states that all values in fields are well-typed. The fifth states that all objects will be locked and unlocked safely.

Now, the type judgment relation $\Gamma \vdash \langle\Psi,H\rangle$ is defined by:

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

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

We can show that the following lemmas hold.

Lemma B.1 $U_1\le U_2 \ \wedge\ rel(U_1,n) \ \Rightarrow\ rel(U_2,n)$

Proof Induction on derivation of $U_1\le U_2$. □

Lemma B.2 $\tau_1\le\tau_2 \ \wedge\ \vdash_H v:\tau_1 \ \ \Rightarrow \vdash_H v:\tau_2$

Proof This follows immediately from Definitions B.2 and 3.10. □

Lemma B.3 $\tau_1\le\tau_2 \ \wedge\ rel_t(\tau_1,n) \ \Rightarrow\ rel_t(\tau_2,n)$

Proof This follows directly from Definition 3.10 and Lemma B.1. □

Now we prove Lemmas 4.1 and 4.2.

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 $P(\sigma)=(FD,(B,D,E))$ and $B(l)=\texttt{return}$.

  Because $\Gamma\vdash P$ holds, we obtain the following condition from rule (RETURN):

$$\forall o\in dom(H).(\{\mathcal{F}(l)(x)|f(x)=o\}\le\mathbf{Top}) \tag{1}$$
$$\forall o\in dom(H).(\{\mathcal{S}(l)(n)|s(n)=o\}\le\mathbf{Top}) \tag{2}$$

  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:

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

  From (1),(2) and Difinition of $\Theta[\mathcal{F},\mathcal{S},f,s][l](o)$, $\Theta[\mathcal{F},\mathcal{S},f,s][l](o)\le\mathbf{Top}$ follows.

  Here, we write U_o for a usage $Use(\Theta[\mathcal{F},\mathcal{S},f,s][l](o))$. We obtain the next conditions from (3), $\Theta[\mathcal{F},\mathcal{S},f,s][l](o)\le\mathbf{Top}$, Difinition 3.10 and Difinition of $rel_t(\tau)$:

$$\forall o \in dom(H).rel(U_o, z(o)) \tag{4}$$
$$\forall o \in dom(H).(U_o \leq \mathbf{0}) \tag{5}$$

From (4), (5)and Lemma B.1, we have the following conditions

$$\forall o \in dom(H).rel(\mathbf{0}, z(o)) \tag{6}$$

Therefore, from (6) and the condition *1* in the definition 3.6. the required condition $z(o) = 0$ follows.

- Suppose $P(\sigma) = (FD, (B, D, E))$ and $l \notin dom(B)$.

  From $(\Gamma, P) \vdash \langle \Psi, H \rangle$ and $l \notin dom(B)$, the condition $l \in codom(E)$ follows. Therefore, we obtain the following condition from $\Gamma \vdash P$ and rule (BREAK):

  $$\forall o \in dom(H).(\{\mathcal{F}(l)(x) | f(x) = o\} \leq \mathbf{Top})$$
  $$\forall o \in dom(H).(\{\mathcal{S}(l)(n) | s(n) = o\} \leq \mathbf{Top})$$

  The rest of proof in this case is just as straightforward as the previous case.

- Suppose $P(\sigma) = (FD, (B, D, E))$ 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, the following conditions hold for a class σ' and usages U_x, U.

  $$\mathcal{F}[l](x) = \sigma'/U_x \quad U_x \leq \widehat{L}.U$$

  Let τ be the type:

  $$\bigotimes \left(\begin{array}{l} \{\mathcal{F}(l)(x') | f(x') = f(x) \ \wedge \ x' \neq x\} \\ \cup \ \{\mathcal{S}(l)(n') | s(n') = f(x)\}) \end{array} \right)$$

  and let $U_{(x,i)}$ be the usage $Use(\tau)$. Since $(\Gamma, P) \vdash \langle \Psi, H \rangle$ holds, we have:

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

  from which

  $$rel(U_x \otimes U_{(x,i)}, z(f(x)))$$

  follows.

  By $U_x \leq \widehat{L}.U$ and Lemma B.1, the following condition holds:

  $$rel(\widehat{L}.U \otimes U_{(x,i)}, z(f(x)))$$

  So, we obtain $z(f(x)) \geq 1$ from the condition 2 in Definition 3.6. □

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. We 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

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

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

$$\begin{array}{l} \mathcal{F}_l \leq \mathcal{F}_{l+1} \\ \mathcal{S}_l(0) \leq \textbf{Int} \\ \mathcal{S}_l \leq \mathcal{S}_{l+1} \end{array} \tag{7}$$

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.

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

By (7),(8), the following condition holds:

$$\begin{array}{l} \forall o \in dom(H). \\ (\Theta[\mathcal{F}, \mathcal{S}, f, c \cdot s][l](o) \leq \Theta[\mathcal{F}, \mathcal{S}, f, c+1 \cdot s][l+1](o)) \end{array} \tag{9}$$

Here, $\forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}[l+1](x))$ and $\forall n \in dom(s).(\vdash_H (c+1 \cdot s)(n) : \mathcal{S}[l+1](n))$ follow from (7),(8) and Lemmas B.2.

Moreover, $\forall o \in dom(H).rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, c+1 \cdot s][l+1](o), z(o))$ follows from (9) and Lemma B.3. Therefore, $(\Gamma, P) \vdash \langle\{\langle l+1, f, c+1 \cdot s, z, \sigma\rangle\}, H\rangle$ holds. 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

$$\begin{array}{l} \Psi = \Psi_1 \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle \\ \Psi' = \Psi_1 \uplus \{i \mapsto \langle l+1, f, s, z', \sigma\rangle\} \\ z' = z\{f(x) \mapsto n+1\} \\ f(x) \in dom(H) \quad z^{\#}(f(x)) = n \geq 2 \quad H(f(x)).flag = 1 \\ P(\sigma)(l) = \texttt{monitorenter}\ x \end{array}$$

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

$$\begin{array}{c} y \in dom(\mathcal{F}_l) \setminus \{x\}.(\mathcal{F}_l(y) \leq \mathcal{F}_{l+1}(y)) \\ \mathcal{F}_l(x) \leq_L \mathcal{F}_{l+1}(x) \\ \mathcal{S}_l \leq \mathcal{S}_{l+1} \end{array} \tag{10}$$

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:

$$\begin{array}{l} \forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}(l)(x)) \\ \forall n \in dom(s).(\vdash_H s(n) : \mathcal{S}(l)(n)) \\ \forall o \in dom(H).rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o), z(o)) \end{array} \tag{11}$$

By (10),(11) and $f(x) \in dom(H)$, the following conditions hold for some σ' U, and U'.

$$\sigma'/(U \otimes U') \leq \Theta[\mathcal{F},\mathcal{S},f,s][l+1](f(x)) \tag{12}$$
$$\Theta[\mathcal{F},\mathcal{S},f,s][l](f(x)) \leq \sigma'/(L.U \otimes U') \tag{13}$$

Because $rel_t(\Theta[\mathcal{F},\mathcal{S},f,s][l](f(x)), z(f(x)))$ and (13) hold, the following condition follows from Lemma B.3 and $z^{\#}(f(x)) = z(f(x)) = n$:

$$rel_t(\sigma'/(L.U \otimes U'), n) \tag{14}$$

From Definition 3.6 it follows that:

$$rel_t(\sigma'/(U \otimes U'), n+1) \tag{15}$$

By (13),(14), and (15), we have $rel_t(\Theta[\mathcal{F},\mathcal{S},f,s][l+1](f(x)), z'(f(x))$.

Moreover, $\forall x' \in dom(f).(\vdash_H f(x') : \mathcal{F}(l+1)(x'))$ and $\forall n \in dom(s).(\vdash_H s(n) : \mathcal{S}(l+1)(n))$ follow from (11) and (12). Therefore, $(\Gamma, P) \vdash \langle \Psi_1 \uplus \{i \mapsto \langle l+1, f, s, z\{f(x) \mapsto n+1\}, \sigma\rangle\}, H\rangle$ holds.

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

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

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:

$$\begin{array}{c}
\overline{\sigma'_P}.a : d \\
\mathcal{F}_l \leq \mathcal{F}_{l+1} \\
\mathcal{S}_l \leq (\sigma/\mathbf{0}) \cdot \mathcal{S}' \\
(d/U) \cdot \mathcal{S}' \leq \mathcal{S}_{l+1} \\
rel(U)
\end{array} \tag{16}$$

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:

$$\begin{array}{l}
\forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}(l)(x)) \\
\forall n \in dom(o \cdot s).(\vdash_H (o \cdot s)(n) : \mathcal{S}(l)(n)) \\
\forall o \in dom(H).(P \vdash_H H(o)\ \textit{well-typed record}) \\
\forall o \in dom(H).rel_t(\Theta[\mathcal{F},\mathcal{S},f,o \cdot s][l](o), z(o))
\end{array} \tag{17}$$

By (16) and (17), the following condition hold:

$$\forall n \geq 1 \in dom(v \cdot s).(\vdash_H (v \cdot s)(n) : ((d/U) \cdot \mathcal{S}'(l))(n)),$$

and by $\overline{\sigma'_P}.a : d$, $H(o).\texttt{class} = \sigma'$, $H(o).a = v$ and $P \vdash_H H(o)$ *well-typed record* in (17), we have $\vdash_H v : d/U$.

Therefore, by Lemma B.2, the following condition hold:

$$\forall n \in dom((v \cdot s)).(\vdash_H (v \cdot s)(n) : \mathcal{S}(l+1)(n)). \tag{18}$$

By $\mathcal{S}_l \leq (\sigma/\mathbf{0}) \cdot \mathcal{S}'$, $(d/U) \cdot \mathcal{S}' \leq \mathcal{S}_{l+1}$ in (16) and (17), (18),

$$\Theta[\mathcal{F},\mathcal{S},f,o \cdot s][l](o) \otimes (d/U) \leq \Theta[\mathcal{F},\mathcal{S},f,v \cdot s][l+1](o) \tag{19}$$

hold. By *rel*(U) in (16), (19), Lemma B.3 and definition of reliability of usages in Section 3.2, we have $rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](o), z(o))$.

From this and (16), (17), (18), it follows $\forall o \in dom(H).rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, v \cdot s][l+1](o), z(o))$. Moreover, $\forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}(l+1)(x))$ follows from (16), (17). Therefore, $(\Gamma, P) \vdash \langle \Psi_1 \uplus \{i \mapsto \langle l+1, f, v \cdot s, z, \sigma \rangle\}, H \rangle$ holds. □

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 3.16 and B.4. □

We can now prove the soundness of our type system.

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

Since P is well-typed. Therefore, there is a type environment Γ that satisfies $P \vdash \Gamma$. From this and Lemma 4.3, $(\Gamma, P) \vdash \langle 0 \mapsto \langle 1, \emptyset, \epsilon, main_P \rangle, \emptyset \rangle$ holds. Moreover, from Lemma 4.2, $(\Gamma, P) \vdash \langle \Psi, H \rangle$ holds. Therefore, properties 1,2 and 3 of this lemma follow immediately from the relation $(\Gamma, P) \vdash \langle \Psi, H \rangle$ and Lemma 4.1. □