A New Type System for JVM Lock Primitives

Futoshi Iwama
University of Tokyo
iwama@kb.cs.titech.ac.jp

Naoki Kobayashi
Tokyo Institute of Technology
kobayasi@cs.titech.ac.jp

A bytecode verifier for the Java virtual machine language (JVML) checks that bytecode does not cause any fatal error before the code is executed. 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 can guarantee that a thread that has acquired a lock will eventually release the lock and that a thread releases a lock only if it has acquired the lock previously.

1 Introduction

A Java program [1] is usually compiled into a Java bytecode. Before it is interpreted by the Java Virtual Machine(JVM) [9], a bytecode verifier checks properties of the bytecode and rejects it if it violates certain safety policies. According to the present definition [9], 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 three bytecode programs in Figure 1. The leftmost code locks the object stored in local variable x at the address 1 and then unlocks the object at 5. It is accepted by their type system since the lock instruction `monitorenter` x is

```
1 monitorenter x
2 ...
3 (critical section)
4 ...
5 monitorexit x
6 return
```

(*code 1*)

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

(*code 2*)

```
1 monitorenter x
2 ...
3 monitorenter y
4 ...
5 monitorexit x
6 ...
7 monitorexit y
8 return
```

(*code 3*)

Figure 1: Programs that use lock primitives

syntactically matched by `monitorexit` x. However, their type system rejects the other programs. The code in the middle 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. It is, however, rejected by their type system since there are *syntactically* two occurrences of the unlock primitive as to the one occurrence of the lock primitive. The rightmost code is also valid but it is rejected by their type system since their type system requires that the critical sections guarded by different locks are nested. It is not difficult to compile a Java program into bytecode that satisfies the requirement imposed by their type system, but problems may arise when optimizations are applied to the bytecode or the bytecode is produced from a source program in other languages.

To solve the above problem, we propose a new type system for verifying that a thread that has acquired a lock in a method will release the lock during the same method execution, and that when a thread tries to unlock an object in a method, the thread has previously locked in the same execution of the method. Our type system is more concise and expressive than Bigliardi and Laneve's type system: Unlike their type system, our type system can accept jumps between critical sections (as in the second code in Figure 1), and intertwined critical sections (as in the third code in Figure 1).

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.

The rest of this paper Section 2 introduces our target language. In section 3, we define our type system and show the correctness of the type system. In section 4, related works are discussed; In section 5, we conclude this paper.

2 Target language $JVML_T$

In this section, we introduce our target language $JVML_T$. It is a subset of Java bytecode language *JVML* and similar to the language $JVML_C$ introduced by Bigliardi and Laneve [2].

2.1 The syntax of $JVML_T$

A program in $JVML_T$ is executed by threads. Each thread has its own operand stack and local variables. A thread manipulates its stack and local variables, creates a new thread, etc. We write $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{V}$ for the set of natural numbers, the set of 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, and use a meta-variable σ to denote a class name.

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

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

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 local variable x onto the operand stack, and `store` x removes the top value form the operand stack and stores the value into local variable x. Instruction `if` l pops the top value from the operand stack and jumps to the address l

if the value is 0, and otherwise proceeds to the next address. Instruction `new` σ creates a new σ-class object and pushes a reference to the object onto the operand stack. 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). Instructions `monitorenter` x and `monitorexit` x respectively locks and unlocks the object pointed to by the reference stored at the top of the operand stack.[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 store 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. Instruction `athrow` raises an exception and jumps to the address specified by the exception table (see below). Instruction `return` returns from the current method.

For simplicity, 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`.) We have also omitted instructions to access state variables of objects. We assume that only `new` σ and `athrow` may throw exceptions. Instruction `new` σ may throw an exception, for example, when initialization fails. For the sake of simplicity, we assume that there is only a single kind of exceptions. We think that it is not difficult to extend our type system to deal with omitted features.

A *method body* is a mapping from a finite subset of $\mathcal{A}$ to **Inst** and represented by the meta-variable B. Examples of method bodies are shown in Figure 1. An *exception table* is a mapping from a finite subset of $\mathcal{A}$ to $\mathcal{A}$, and denoted by a meta-variable E. If an exception is raised at address l, the control jumps to address $E(l)$. A *method descriptor*, denoted by a meta-variable D, is a mapping from a set $\{0, \ldots, n-1\} (\subseteq \mathcal{V})$ to the set $\{\mathbf{Int}, \mathbf{Top}\} \cup \Sigma$. $D(x)$ denotes the type of the x-th argument of a method.

A *method* is a triple consisting of a method body, a descriptor and a exception table. A *program* is a mapping from a set of class

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

names to methods and denoted by a meta-variable P.

2.2 The operational semantics of $JVML_T$

We define an operational semantics of the language in a manner similar to [11, 2].

We write $dom(f)$ and $codom(f)$ for the domain and the co-domain of function f respectively. $f\{x \mapsto v\}$ 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 $y \in dom(f \setminus x)$.

We write $\mathbf{I}$ for the set of integers. We assume that there is a countably infinite set $\mathbf{O}$ of *references* (to objects). A *value* is either an integer or a reference. We write $\mathbf{VAL}$ for the set $\mathbf{I} \cup \mathbf{O}$ of values. An *object* is a record $[\mathtt{class} : \sigma, \mathtt{flag} : d]$, where σ denotes the class name of the object, and d is either 0, indicating that the object is not locked, or 1, indicating that the object is locked. If $\rho = [\mathtt{class} : \sigma, \mathtt{flag} : d]$, we write $\rho.class$ and $\rho.flag$ for σ and d respectively. We write $\mathbf{RCD}$ the set of objects.

A *stack* modeled by a partial mapping from $\mathcal{N}$ to $\mathbf{VAL}$ whose domain is of the form $\{i \in \mathcal{N} \mid 0 \leq i < n\}$. 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.

A *thread state* is 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, z maps each heap address 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 $\mathbf{T}$ for the set of thread states. We extend a partial mapping z to a total mapping $z^{\#}$ by:

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

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 form natural numbers to $\mathbf{T}$, and H is a partial mapping from $\mathbf{O}$

to the set **RCD** of objects. $\Psi(i)$ represents the state of the thread whose identifier is i. $H(o)$ denotes the object stored at o.

We define the operational semantics of $JVML_T$ using a one-step reduction relation $P \vdash \langle \Psi, H \rangle \rightarrow \langle \Psi', H' \rangle$. It 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: if $P(\sigma) = (B, E, D)$ then $P[\sigma](l) = B(l)$. $\bar{t}$ denotes an element of set $\mathbf{T}$ and if $i \notin dom(\Psi)$ then $\Psi \uplus \{i \mapsto \bar{t}\}$ denotes a mapping defined by:

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

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: $\langle 0 \mapsto \langle 1, \emptyset, \epsilon, \emptyset, main \rangle, \emptyset \rangle$.

3 Type system

In this section, we give a type system for checking safe usage of lock primitives. It is an extension of Stata and Abadi's type system [11] for JVML.

3.1 Usages and types

As mentioned in section 1, 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 of usage expressions (usages, in short) is defined by:*

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

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. $\widehat{L}.U$ 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.

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

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

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

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

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

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

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

$$\frac{P[\sigma](l) = \texttt{new } \sigma' \quad o \notin dom(H) \quad H' = H\{o \mapsto \rho\} \quad \rho.class = \sigma' \quad \rho.flag = 0}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \rightarrow \langle \Psi \uplus \{i \mapsto \langle l+1, f, o \cdot s, z, \sigma\rangle\}, H'\rangle} \quad (new)$$

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

$$\frac{\begin{array}{c} P[\sigma](l) = \texttt{start } \sigma' \quad j \notin dom(\Psi) \cup \{i\} \\ o \in dom(H) \quad H(o).class = \sigma' \quad dom(D^{\sigma'}) = \{0, \ldots, n-1\} \quad f' = f_0\{0 \mapsto v_0, \ldots, n-1 \mapsto v_{n-1}\} \end{array}}{P \vdash \begin{array}{c} \langle \Psi \uplus \{i \mapsto \langle l, f, v_0 \cdot, \ldots, \cdot v_{n-1} \cdot o \cdot s, z, \sigma\rangle\}, H\rangle \ \rightarrow \\ \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z, \sigma\rangle\} \uplus \{j \mapsto \langle 1, f_0\{0 \mapsto v_0, \ldots, n-1 \mapsto v_{n-1}\}, \epsilon, \emptyset, \sigma'\rangle\}, H\rangle \end{array}} \quad (start)$$

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

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

Figure 2: operational semantics

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

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

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

$$\frac{P[\sigma](l) = \texttt{return}}{P \vdash \langle \Psi \uplus \{i \mapsto \langle l, f, s, z, \sigma\rangle\}, H\rangle \to \langle \Psi, H\rangle} \quad (return)$$

Figure 3: operational semantics

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 free occurrences of α with U_1). For example, $\mu\alpha.(\mathbf{0} \& L.\alpha)$ describes an object that is locked an arbitrary number of times.

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

We define two binary relations $\equiv$ and $\leq$ on usages below.

Definition 3.2 *The binary relation $\equiv$ on usages is the least congru-*

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

Definition 3.3 *The* sub-usage relation $\leq$ *is the least preorder on usages that includes the relation $\equiv$ and is closed under the following rules:*

$$U_1 \& U_2 \leq U_1 \qquad \frac{U_i \leq U_i'}{C[U_1, \ldots, U_n] \leq C[U_1', \ldots, U_n']}$$

Definition 3.4 (Types) *The set $\mathcal{T}$ of* types *is defined by:*

$$\tau \quad ::= \quad \mathbf{Int} \mid \mathbf{Top} \mid \sigma/U$$

Int is the type of integers. **Top** is the type of values 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.

Example 3.5 Type $\texttt{Counter}/L.\widehat{L}$ describes an object of `Counter` class that is first locked and then unlocked. Type $\texttt{Account}/L.(\widehat{L}\&\mathbf{0})$ describes an object of `Account` class that is first locked and then either unlocked or no longer accessed.

We extend the sub-usage relation to a subtype relation $\tau_1 \leq \tau_2$ on types. $\tau_1 \leq \tau_2$ means that a value of type τ_1 can be used as a value of type τ_2.

Definition 3.6 *The* subtype *relation is the least preorder closed under the following rules:*

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

3.2 Reliability of usages

As is understood from Example 3.5, 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. The pair $\langle U, n\rangle$ represents the state of an object that has been locked n times so far and will be used according to usage U from now. A change of the state is represented by the usage pair reduction $\rightarrow_{rel}$ defined below.

Definition 3.7 *The usage preorder $\succeq$ is the least binary relation on usages closed under the rules:*

$$U \otimes \mathbf{0} \succeq U \quad U_1 \& U_2 \succeq U_1 \quad U_1 \& U_2 \succeq U_2$$

$$U_1 \otimes U_2 \succeq U_2 \otimes U_1 \quad (U_1 \otimes U_2) \otimes U_3 \succeq U_1 \otimes (U_2 \otimes U_3)$$

$$\mu\ \alpha. U \succeq [\mu\ \alpha. U/\alpha]U \quad \frac{U_1 \succeq U_1' \quad U_2 \succeq U_2'}{U_1 \otimes U_2 \succeq U_1' \otimes U_2'}$$

Definition 3.8 *The usage pair reduction $\rightarrow_{rel}$ is the least relation on pairs consisting of an usage and a natural number closed under the rules:*

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

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

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

We can now define the reliability of a usage.

Definition 3.9

$$rel(\langle U, n\rangle) \overset{def}{\longleftrightarrow} \forall U'.\forall n'.\langle U, n\rangle \rightarrow_{rel}^* \langle U', n'\rangle \Rightarrow \begin{cases} U' \succeq \mathbf{0} & \Rightarrow \quad n' = 0 \\ U' \succeq \widehat{L}.U_1 \otimes U_2 & \Rightarrow \quad n' \geq 1 \end{cases}$$

Definition 3.10 (Reliability of *usages*)

$$rel(U) \overset{def}{\longleftrightarrow} rel(\langle U, 0\rangle)$$

Example 3.11 $L.L.(\widehat{L} \otimes \widehat{L})$, $(L.\widehat{L})\&(L.\widehat{L})$, and $L.\mu\alpha.((\widehat{L}.L.\alpha)\&\widehat{L})$ are reliable. Neither $L.(L \otimes \widehat{L})$ nor $L.\widehat{L}.\widehat{L}$ is reliable .

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

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

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 introduced as 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. Similar, 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. a the type of the operand stack just before the execution of a instruction at address l. For example, $\mathcal{F}(l)(x) = \sigma/\widehat{L}$ means that σ-class object is stored in the local variable x at address l, and the lock on the object will be released afterwards.

We define several operations and relations on types, frame types, and stack types. We use $*$ to represent a binary operator $\otimes$ or $\&$ and use $\dot{L}.$ to represent a unary operator $L.$ or $\widehat{L}.$

Definition 3.12 *We define $\tau_1 * \tau_2$, $\dot{L}.\tau$ by:*

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

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

Definition 3.13 (binary operations on frame types) *Suppose that $dom(F_1) = dom(F_2)$. $F_1 * F_2$ is defined by:*

$$\begin{array}{l} dom(F_1 * F_2) = dom(F_1) \\ \forall x \in dom(F_1).(F_1 * F_2)(x) = (F_1(x)) * (F_2(x)) \end{array}$$

Definition 3.14 (subtype relation on frame types) *A frame type F_1 is a subtype of F_2, written $F_1 \leq F_2$, if:*

$$\begin{array}{l} dom(F_1) = dom(F_2) \\ \forall x \in dom(F_1).(F_1(x) \leq F_2(x)) \end{array}$$

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

Definition 3.15 *We define $\tau_1 \leq_{\dot{L}} \tau_2$ by:*

$$\tau_1 \leq_{\dot{L}} \tau_2 \Leftrightarrow (\tau_1 \leq \dot{L}.\tau_2) \vee (\tau_2 = \mathbf{Top} \wedge \exists \sigma.\tau_1 = \sigma/\dot{L}.\mathbf{0})$$

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.

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

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

3.4 Typing rules

We consider a type judgment of the form $\langle \mathcal{F}, \mathcal{S} \rangle \vdash (B, E, D)$. 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 $\mathcal{S}$.

To define the relation above, we introduce the relation $\mathcal{F}, \mathcal{S}, l \vdash (B, E, D)$. Intuitively, it means that the instruction at j can be correctly executed under the assumption that the values stored in local variables and the operand stack have the types described by $\mathcal{F}$ and $\mathcal{S}$.

Definition 3.16 *$\mathcal{F}, \mathcal{S}, l \vdash (B, E, D)$ is the least relation closed rules in Figure 4.*

In Figure 4, $\mathcal{F}_l$ and $\mathcal{S}_l$ are shorthand notations for $\mathcal{F}(l)$ and $\mathcal{S}(l)$ respectively. D^σ denotes the method descriptor of class σ.

We explain several rules below:

Rule (MENTR): The first line states that the instruction at the address l is `monitorenter` in the method (B, E, D). 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 third line expresses this condition. The third line also says that the types of 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 the address l is `if` . 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 the 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 afterword, since the operand stack becomes empty when the exception is raised.

Now we define the type judgment relation for methods.

Definition 3.17 (Type judgment for methods) *The relation* $\mathcal{F}, \mathcal{S} \vdash (B, E, D)$ *is defined by the following rule:*

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

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

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

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 3.18 (Well-typed program) *A program P is well-typed if for each method (B, E, D), there exist $\mathcal{F}$ and $\mathcal{S}$ such that $\mathcal{F}, \mathcal{S} \vdash (B, E, D)$ holds.*

3.5 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. We show the full proof in Appendix B and here show an outline of the proof.

The soundness of our type system is stated as follow.

Theorem 3.1 *Suppose that a program P is well-typed, and that $P \vdash \langle 0 \mapsto \langle 1, \emptyset, \epsilon, \emptyset, main \rangle, \emptyset \rangle \rightarrow^* \langle \Psi, H \rangle$. If $\Psi(i) = \langle l, f, s, z, \sigma \rangle$, then the following properties hold:*

1. *If $P[\sigma](l) =$ `return`, then $z(o) = 0$ for all $o \in dom(H)$.*
2. *If $P[\sigma](l) =$ `monitorexit` x, then $z(f(x)) \geq 1$.*

Here, $P[\sigma](l)$ denotes the instruction at address l of the *run* method of the σ-class. The first property states that when a thread terminates, it has released all the locks it acquired. The second 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 is well-typed under the type environment $\Gamma(\sigma)$ (in the sense of Definition 3.17). We also define a type judgment relation $\Gamma \vdash \langle \Psi, H \rangle$ for machine states. It means that the threads Ψ and the heap H is consistent with the type assumption Γ. (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 3.1 below), and that the well-typedness of a machine state is preserved during execution of a well-typed program (Lemma 3.2 below). Theorem 3.1 follows immediately from these properties and the fact that the initial machine state is well-typed.

Lemma 3.1 (lack of immediate lock errors) *If* $\Gamma \vdash \langle \Psi, H \rangle$ *and* $\Psi(i) = \langle l, f, s, z, \sigma \rangle$, *then the following properties hold:*

1. *If* $P[\sigma](l) = \texttt{return}$, *then* $z(o) = 0$ *for all* $o \in dom(H)$.
2. *If* $P[\sigma](l) = \texttt{monitorexit}\ x$, *then* $z(f(x)) \geq 1$.

Lemma 3.2 (subject reduction) *Suppose that* $\Gamma \vdash P$ *and* $\Gamma \vdash \langle \Psi, H \rangle$ *hold. If* $P \vdash \langle \Psi, H \rangle \rightarrow \langle \Psi', H' \rangle$, *then* $\Gamma \vdash \langle \Psi', H' \rangle$ *holds.*

3.6 Type inference algorithm

By the soundness of the type system, we can statically verify safe usage of lock primitives in a program by checking that the program is well-typed. To check whether a program 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 $\mathcal{F}, \mathcal{S} \vdash (B, E, D)$ 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 [10, 7], We illustrate how type inference works using an example. Consider the second 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 Abadi and Stata's type system [11]). The frame type environment $\mathcal{F}$ and the stack type environment $\mathcal{S}$ of the method are given as:

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

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

From the typing rule for the method (Definition 3.17), we obtain the following constraints:

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

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

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

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 $\mathit{rel}(L.((\widehat{L}.\mathbf{0})\&\mathbf{0}))$. 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 [8], this problem can be reduced to the reachability problem of Petri net [4], and hence the problem is decidable. A more efficient algorithm for judging the reliability is given in Appendix A.

4 Related work

Our type system in this paper was obtained by extending Stata and Abadi's type system for JVML [11] with usages. Their type system deals with subroutines. We think that it is not difficult to extend our type system to deal with subroutines. 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, 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 is 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 [8, 12] 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 [7] 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 [7]. 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 [7].

5 Conclusion

We proposed a new type system for checking usage of lock primitives for a subset of JVML [9], and proved its correctness. We have already implemented a core part of the type inference algorithm.

It is left for future work to extend our type system to deal with object fields, method invocations, and other concurrency primitives. Future work also includes improvement of the accuracy of the analysis. Currently, our type system does not keep track of the order of accesses through different local variables or stack locations, which cause some correct programs to be rejected. We can solve this problem by using an idea used in a type system for the π-calculus [6].

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 *Proc. 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 *Proc. of PLDI*, 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.

[6] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proc. of POPL*, pages 128–141, January 2001.

[7] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proc. of POPL*, pages 331–342, 2002.

(INC)

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

(PUSH)

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

(POP)

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

(IF)

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

(LOAD)

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

(STORE)

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

(NEW)

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

(START)

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

(RETURN)

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

(MENTR)

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

(MEXT)

$$\frac{\begin{array}{c} B(l) = \texttt{monitorexit } x \\ l+1 \in \mathit{dom}(B) \\ \mathcal{F}_l \setminus x \leq \mathcal{F}_{l+1} \setminus x \\ \mathcal{F}_l(x) \leq_{\widehat{L}} \mathcal{F}_{l+1}(x) \\ \mathcal{S}_l \leq \mathcal{S}_{l+1} \end{array}}{\mathcal{F}, \mathcal{S}, l \vdash (B, E, D)}$$

(ATHROW)

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

Figure 4: typing rules

[8] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *Proc. of CONCUR2000*, volume 1877 of *LNCS*, pages 489–503. Springer-Verlag, August 2000. The full version is available as technical report TR00-01, Dept. Info. Sci., Univ. Tokyo.

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

[10] Torben Mogensen. Types for 0, 1 or many uses. In *Implementation of Functional Languages*, volume 1467 of *LNCS*, pages 112–122, 1998.

[11] Raymie Stata and Martín Abadi. A type system for java bytecode subroutines. *ACM Trans. Prog. Lang. Syst.*, 21(1):90–137, 1999.

[12] 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 $rel(U)$

In this section, we give an algorithm for checking $rel(U)$ for a closed (i.e., not containing free usage variables) usage U.

To check whether $rel(U)$ holds, we consider two numbers Min_U and Fin_U for each closed usage U. Min_U is the least n such that $(U, 0) \rightarrow^*_{rel} (U', 0)$, while Fin_U is the greatest n such that $(U, 0) \rightarrow^*_{rel} (U', n)$ and $U' \succeq \mathbf{0}$ (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 3.10, $rel(U)$ if and only if (1)$Min_U = 0$ and (2)$Fin_U \leq 0$.

We need some definitions to present an algorithm to check the above conditions.

We say that a usage U is *recursive* if it is of the form $\mu\alpha.U$. Let U be a closed usage. We define the set $Rec(U)$ as the least set satisfying the following rules:

$$
\begin{array}{l}
Rec(U_1 \otimes U_2) \supseteq \{U_1\} \cup \{U_2\} \\
Rec(U_1 \& U_2) \supseteq \{U_1\} \cup \{U_2\} \\
Rec(L.U) \supseteq Rec(U) \\
Rec(\widehat{L}.U) \supseteq Rec(U) \\
Rec(\mu\alpha.U) \supseteq \{\mu\alpha.U\} \cup Rec([\mu\alpha.U/\alpha]U)
\end{array}
$$

Intuitively, $Rec(U)$ is the set of recursive usages that may appear when U is reduced. Note that it is a finite set.

Suppose that U is a closed, recursive usage. Let C_{Min_U} be the set of equations

$$\{Min_{\mu\alpha.V} = MinExp([\mu\alpha.V/\alpha]V) \mid \mu\alpha.V \in Rec(U)\},$$

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

$$
\begin{array}{l}
MinExp(\mathbf{0}) = 0 \\
MinExp(U_1 \otimes U_2) = MinExp(U_1) + MinExp(U_2) \\
MinExp(U_1 \& U_2) = min(MinExp(U_1), MinExp(U_2)) \\
MinExp(L.U) = min(0, MinExp(U) + 1) \\
MinExp(\widehat{L}.U) = MinExp(U) - 1 \\
MinExp(\mu\alpha.U) = Min_{\mu\alpha.U}
\end{array}
$$

C_{Min_U} can be expressed in the form

$$
\begin{array}{l}
\{Min_{U_1} = F_1(Min_{U_1}, \ldots, Min_{U_n}), \\
\ \cdots, \\
\ Min_{U_n} = F_n(Min_{U_1}, \ldots, Min_{U_n})\}
\end{array}
$$

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

Suppose that U is closed. We can check whether $Min_U = 0$ as follows. (Here, we can assume without loss of generality that U is recursive, since otherwise we can replace U with $\mu\alpha.U$.) Compute $(Min^{(j)}_{U_1}, \ldots, Min^{(j)}_{U_n})$ by

$$
\begin{array}{l}
Min^{(0)}_{U_i} = 0 \\
Min^{(j+1)}_{U_i} = F_i(Min^{(j)}_{U_1}, \ldots, Min^{(j)}_{U_n})
\end{array}
$$

from $j = 0$ to m such that $(Min^{(m+1)}_{U_1}, \ldots, Min^{(m+1)}_{U_n}) = (Min^{(m)}_{U_1}, \ldots, Min^{(m)}_{U_n})$ or $Min^{(m)}_U < 0$. (Note that such m always exists.) Then, $Min_U = 0$ if and only if $Min^{(m)}_U < 0$.

Whether $Fin_U \leq 0$ holds can be checked in a similar manner. Let C_{Fin_U} be the set of equations:

$$\{Fin_{\mu\alpha.V} = FinExp([\mu\alpha.V/\alpha]V) \mid \mu\alpha.V \in Rec(U)\}$$

Here, $FinExp(V)$ is an expression defined by:

$$\begin{array}{l}
FinExp(\mathbf{0}) = 0 \\
FinExp(U_1 \otimes U_2) = FinExp(U_1) + FinExp(U_2) \\
FinExp(U_1 \& U_2) = \max(FinExp(U_1), FinExp(U_2)) \\
FinExp(L.U) = FinExp(U) + 1 \\
FinExp(\widehat{L}.U) = FinExp(U) - 1 \\
FinExp(\mu\alpha.U) = Fin_{\mu\alpha.U}
\end{array}$$

Suppose that U is closed and recursive, and that $Min_U = 0$ holds. Let C_{Fin_U} be the set:

$$\begin{array}{l}
\{Fin_{U_1} = G_1(Fin_{U_1}, \ldots, Fin_{U_n}), \\
\cdots, \\
\ Fin_{U_n} = G_n(Fin_{U_1}, \ldots, Fin_{U_n})\}.
\end{array}$$

Compute $(Fin_{U_1}^{(j)}, \ldots, Fin_{U_n}^{(j)})$ by

$$\begin{array}{l}
Fin_{U_i}^{(0)} = -\infty \\
Fin_{U_i}^{(j+1)} = G_i(Fin_{U_1}^{(j)}, \ldots, Fin_{U_n}^{(j)})
\end{array}$$

from $j = 0$ to m such that $(Fin_{U_1}^{(m+1)}, \ldots, Fin_{U_n}^{(m+1)}) = (Fin_{U_1}^{(m)}, \ldots, Fin_{U_n}^{(m)})$ or $Fin_U^{(m)} > 0$. (Such m always exists.) Then, $Fin_U \leq 0$ holds if and only if $Fin_U^{(m)} \leq 0$.

Example A.2 Let U be $\mu\alpha.(\mathbf{0}\&(L.\widehat{L} \otimes \alpha))$. Then,

$$\begin{array}{rcl}
MinExp([U/\alpha](\mathbf{0}\&(L.\widehat{L} \otimes \alpha))) & = & min(MinExp(\mathbf{0}), MinExp(L.\widehat{L} \otimes U)) \\
& = & min(0, MinExp(L.\widehat{L}) + MinExp(U)) \\
& = & min(0, 0 + Min_U)
\end{array}$$

Therefore, $C_{Min_U} = \{Min_U = min(0, Min_U)\}$. Since $Min_U^{(1)} = Min_U^{(0)} = 0$, we know $Min_U = 0$. Similarly, C_{Fin_U} is the set $\{Fin_U = \max(0, Fin_U)\}$. Since $Fin_U^{(1)} = Fin_U^{(0)} = 0$, we know that $Fin_U = 0$.

Example A.3 Let U be $\mu\alpha.(\mathbf{0}\&(\widehat{L}.L \otimes \alpha))$. Then, $C_{Min_U} = \{Min_U = min(0, Min_U - 1)\}$. Since $Min_U^{(1)} = -1$, we know $Min_U = 0$ does not hold.

Example A.4 Let U be $\mu\alpha.(L\&(L.\widehat{L} \otimes \alpha))$. Then, $C_{Min_U} = \{Min_U = min(1, Min_U)\}$. Since $Min_U^{(1)} = Min_U^{(0)} = 0$, we know $Min_U = 0$. Similarly, C_{Fin_U} is the set $\{Fin_U = \max(1, Fin_U)\}$. Since $Fin_U^{(1)} = 1$, we know that $Fin_U = 0$ does not hold.

B Proof of Theorem 3.1

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 3.5).

Definition B.1 *The relation $\Gamma \vdash P$ is defined by:*

$$\Gamma \vdash P \Leftrightarrow \forall \sigma \in dom(P).(\Gamma(\sigma) \vdash P(\sigma))$$

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

We first define a type judgment relation $\vdash_H v : \tau$.

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

$$\frac{v \in \mathbf{VAL}}{\vdash_H v : \mathbf{Top}} \qquad \frac{c \in \mathbf{I}}{\vdash_H c : \mathbf{Int}} \qquad \frac{o \in \mathbf{O} \quad H(o).class = \sigma}{\vdash_H o : \sigma/U}$$

Next, we define a type judgment relation for threads.

Definition B.3 (type judgment for threads) *The relation $\Gamma \vdash \langle\langle l, f, s, z, \sigma\rangle, H\rangle$ is defined by:*

$$\frac{\begin{array}{c}\Gamma(\sigma) = \langle \mathcal{F}, \mathcal{S} \rangle \\ \forall x \in dom(\mathcal{F}(l)).(\vdash_H f(x) : \mathcal{F}(l)(x)) \quad \forall n \in dom(\mathcal{S}(l)).(\vdash_H s(n) : \mathcal{S}(l)(n)) \\ \forall o \in dom(H).rel_t(\bigotimes(\{\mathcal{F}(l)(x) | f(x) = o\} \cup \{\mathcal{S}(l)(n) | s(n) = o\}), z(o))\end{array}}{\Gamma \vdash \langle\langle l, f, s, z, \sigma\rangle, H\rangle}$$

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.)

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

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

The second line of the rule of Definition B.3 states that values in local variables and operand stack are typed correctly. The third 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.4 ($\Gamma \vdash \langle \Psi, H \rangle$)

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

We can show that the following lemmas hold.

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

Proof (lemma B.1) Induction on derivation of $U_1 \leq U_2$ $\Box$

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

Proof (lemma B.2) This is shown directly from Definitions B.2 and 3.6. $\Box$

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

Proof (lemma B.3) This follows directly from Definition 3.6 and Lemma B.1.$\Box$

Now, we prove the following lemma, which states that if a machine state is well typed, invalid usage of a lock does not occur immediately.

Lemma B.4 (lack of immediate lock errors)
Suppose $\Gamma \vdash P$ *and* $\Gamma \vdash \langle \Psi, H \rangle$*. If* $\Psi(i) = \langle l, f, s, z, \sigma \rangle$*, then the following conditions hold:*

- *If* $P[\sigma](l) = \texttt{return}$*, then* $\forall o \in dom(H).z(o) = 0$
- *If* $P[\sigma](l) = \texttt{monitorexit}\ x$*, then* $z(f(x)) \geq 1$

Proof (Lemma B.4) Suppose that $\Gamma \vdash P$, $\Gamma \vdash \langle \Psi, H \rangle$ and $\Psi(i) = \langle l, f, s, z, \sigma \rangle$ hold.

- Suppose $P[\sigma](l) = \texttt{return}$.

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

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

  Moreover, from $\Gamma \vdash \langle \Psi, H \rangle$, we obtain $\Gamma \vdash \langle \langle l, f, s, z, \sigma \rangle, H \rangle$. Therefore, the following relation holds:

  $$\forall o \in dom(H).rel_t(\bigotimes(\{\mathcal{F}(l)(x) | f(x) = o\} \cup \{\mathcal{S}(l)(n) | s(n) = o\}), z(o))$$

  From these conditions and Definition 3.10, the required condition z(o)=0 follows.

- Suppose $P[\sigma](l) = \texttt{monitorexit}\ x$.

  Because $\Gamma \vdash P$ holds, we get the following conditions from rule (MEXT):

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

  Let τ be the type:

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

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

  $$rel_t(\bigotimes(\{\mathcal{F}(l)(x') | f(x') = f(x)\} \cup \{\mathcal{S}(l)(n') | s(n) = f(x)\}, z(f(x)))),$$

  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 Definition 3.9. $\square$

We can also prove the following lemma. This states that the well-typedness of a machine state is preserved during execution of a well-typed program.

Lemma B.5 (subject reduction) *Suppose that $\Gamma \vdash P$ holds. If $\Gamma \vdash \langle \Psi, H \rangle$ and $P \vdash \langle \Psi, H \rangle \to \langle \Psi', H' \rangle$, then $\Gamma \vdash \langle \Psi', H' \rangle$ holds.*

Proof (Lemma B.5)

We show this by induction on derivation of $P \vdash \langle \Psi, H \rangle \to \langle \Psi', H' \rangle$ with case analysis on the last rule used. We suppose $\Gamma \vdash P$ and $\Gamma \vdash \langle \Psi, H \rangle$. We write $\Theta[\mathcal{F}, \mathcal{S}, f, s][l](o)$ for $(\bigotimes\{\mathcal{F}(l)(x) | f(x) = o\} \cup \{\mathcal{S}(l)(n) | s(n) = o\})$.

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

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

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

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

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

Moreover, $\Gamma \vdash \ (\langle \langle l, f, c \cdot s, z, \sigma \rangle \}, H \rangle$ follows from the condition $\Gamma \vdash \ (\langle \Psi, H \rangle$. Therefore the following conditions follow from Definition B.3.

$$\begin{array}{l} \forall x \in dom(f).(\vdash_H f(x) : \mathcal{F}(l)(x)) \\ \forall n \in dom(s).(\vdash_H (c \cdot s)(n) : \mathcal{S}(l)(n)) \\ \forall o \in dom(H).rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, c \cdot s][l](o), z(o)) \end{array} \qquad (2)$$

By (1),(2) and Lemma B.3, the following condition holds:

$$\begin{array}{l}\forall o \in dom(H). \\ (\Theta[\mathcal{F}, \mathcal{S}, f, c \cdot s][l](o) \leq \Theta[\mathcal{F}, \mathcal{S}, f, c+1 \cdot s][l+1](o))\end{array} \tag{3}$$

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 (1),(2) 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 (3) and Lemma B.3.

Therefore, $\Gamma \vdash \langle \{\langle l+1, f, c+1 \cdot s, z, \sigma\rangle\}, H\rangle$ holds. From this and $\Gamma \vdash \langle \Psi_1, H\rangle$, the relation $\Gamma \vdash \langle \Psi_1 \uplus \{i \mapsto \langle l+1, f, c+1 \cdot s, z, \sigma\rangle\}, H\rangle$ follows as required.

Case rule $(ment_2)$: It must be the case that

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

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

$$\begin{array}{c}y \in dom(\mathcal{F}_l) \setminus \{x\}.(\mathcal{F}_l(y) \leq \mathcal{F}_{l+1}(y)) \\ \mathcal{F}_l(x) \leq_L \mathcal{F}_{l+1}(x) \\ \mathcal{S}_l \leq \mathcal{S}_{l+1}\end{array} \tag{4}$$

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

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

By (4),(5) and $f(x) \in dom(H)$, the following conditions hold for some class σ' and usages $U \otimes U'$.

$$\sigma'/(U \otimes U') \quad \leq \quad \Theta[\mathcal{F}, \mathcal{S}, f, s][l+1](f(x)) \tag{6}$$

$$\Theta[\mathcal{F}, \mathcal{S}, f, s][l](f(x)) \quad \leq \quad \sigma'/(L.U \otimes U') \tag{7}$$

Because $rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l](f(x)), z(f(x)))$ and (7) hold, the following conditions follow from Lemma B.3 and $z^{\#}(f(x)) = z(f(x)) = n$:

$$rel_t(\sigma'/(L.U \otimes U'), n) \tag{8}$$

By Definition 3.9, the following condition holds:

$$rel_t(\sigma'/(U \otimes U'), n+1) \tag{9}$$

From (7),(8),(9), we obtain $rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l+1](f(x)), z'(f(x))$.

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 (5) and (6).

Therefore, $\Gamma \vdash \langle \Psi_1 \uplus \{i \mapsto \langle l+1, f, s, z\{f(x) \mapsto n+1\}, \sigma\rangle\}, H\rangle$ holds.

Case rule $(mext_2)$: It must be the case that

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

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

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

By the relation $\Gamma \vdash \langle \Psi, H\rangle$, the following conditions also hold:

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

In (10) and (11), $\mathcal{F}, \mathcal{S}, B, E, D$ satisfy $\Gamma(\sigma) = \langle \mathcal{F}, \mathcal{S}\rangle$ and $P(\sigma) = (B, E, D)$. By (10),(11) and $f(x) \in dom(H)$, the following conditions hold for some class σ' and usages $U \otimes U'$.

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

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

Because $rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l](f(x)), z(f(x)))$ and (13) hold, the following conditions follow from Lemma B.3 and $z^{\#}(f(x)) = z(f(x)) \geq 2$:

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

From Definition 3.9, it follows that the following condition holds:

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

Here, it follows $rel_t(\Theta[\mathcal{F}, \mathcal{S}, f, s][l{+}1](f(x)), z'(f(x))$ from (13),(14),(15).

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 (10) and (11).

Therefore, $\Gamma \vdash \langle \Psi \uplus \{i \mapsto \langle l+1, f, s, z\{f(x) \mapsto n-1\}, \sigma\rangle\} H\rangle$ holds. $\square$

The following lemma states that the initial machine state of a well-typed program is also well-typed.

Lemma B.6 (well-typedness of initial machine state)

$$\Gamma \vdash P \;\Rightarrow\; \Gamma \vdash \langle 0 \mapsto \langle 1, \emptyset, \epsilon, \emptyset, main\rangle, \emptyset\rangle$$

Proof (lemma B.6) This lemma follows immediately from Definitions 3.17 and B.3. $\square$

We can now prove the soundness of our type system.

Proof (theorem 3.1)

Suppose that P is well-typed and that $P \vdash \langle 0 \mapsto \langle 1, \emptyset, \epsilon, main\rangle, \emptyset\rangle \rightarrow^* \langle \Psi, H\rangle$ 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 B.6, $\Gamma \vdash \langle 0 \mapsto \langle 1, \emptyset, \epsilon, main\rangle, \emptyset\rangle$ holds. Moreover, from Lemma B.5, $\Gamma \vdash \langle \Psi, H\rangle$ holds. Therefore, properties 1 and 2 of this theorem follow immediately from the relation $\Gamma \vdash \langle \Psi, H\rangle$ and Lemma B.4. $\square$