A Type System for Preventing Data Races and Deadlocks in the Java Virtual Machine Language

Pratibha Permandla   Chandrasekhar Boyapati

Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor, MI 48109
{pratibha,bchandra}@eecs.umich.edu

Abstract

In previous work on *SafeJava* we presented a type system extension to the Java source language that statically prevents data races and deadlocks in multithreaded programs. SafeJava is expressive enough to support common programming patterns, its type checking is fast and scalable, and it requires little programming overhead. SafeJava thus offers a promising approach for making multithreaded programs more reliable. This paper presents a corresponding type system extension for the Java virtual machine language (JVML). We call the resulting language *SafeJVML*. Well-typed SafeJVML programs are guaranteed to be free of data races and deadlocks. Designing a corresponding type system for JVML is important because most Java code is shipped in the JVML format. Designing a corresponding type system for JVML is nontrivial because of important differences between Java and JVML. In particular, the absence of block structure in JVML programs and the fact that they do not use named local variables the way Java programs do make the type systems for Java and JVML significantly different. For example, verifying absence of races and deadlocks in JVML programs requires performing an alias analysis, something that was not necessary for verifying absence of races and deadlocks in Java programs. This paper presents static and dynamic semantics for SafeJVML. It also includes a proof that the SafeJVML type system is sound and that it prevents data races and deadlocks. To the best of our knowledge, this is the first type system for JVML that statically ensures absence of synchronization errors.

Categories and Subject Descriptors
D.3.3 [*Programming Languages*]: Language Constructs
D.2.4 [*Software Engineering*]: Program Verification

General Terms
Languages, Verification

Keywords
SafeJava, Data Races, Deadlocks, Ownership Types

1. Introduction

Multithreaded programming is becoming a mainstream programming practice. But multithreaded programming is difficult and error prone. Multithreaded programs synchronize operations on shared mutable data to ensure that the operations execute atomically. Failure to correctly synchronize such operations can lead to *data races* or *deadlocks*. A data race occurs when two thread concurrently access the same shared data, at least one of the accesses is a write, and there is no synchronization to separate the accesses. A deadlock occurs when there is a set of threads such that every thread in the set is waiting on a lock held by another thread in the set. Synchronization errors in multithreaded programs are timing-dependent, nondeterministic bugs, and are among the most difficult programming errors to detect, reproduce, and eliminate.

In previous work on *SafeJava* [5, 6, 9] we presented a static type system for multithreaded programs. Well-typed SafeJava programs are guaranteed to be free of data races and deadlocks. The basic idea is as follows. When programmers write multithreaded programs, they already have a locking discipline in mind. SafeJava allows programmers to specify this locking discipline in their programs in the form of type declarations. The type checker then statically verifies that a program is consistent with its type declarations.

The SafeJava type system also enforces object encapsulation [5, 7], which is key to enable local reasoning in object oriented programs. Consider, for example, a `Stack` object `s` that is implemented using a linked list. Local reasoning about the correctness of the `Stack` implementation is possible if objects outside `s` do not directly access the list nodes, i.e., the list nodes are *encapsulated* within `s`. SafeJava uses a variant of ownership types [13, 12, 3] to declare that `s` *owns* all the list nodes. The type system then statically ensures that the list nodes are encapsulated within `s`. Object encapsulation is useful for safe multithreading because the lock that protects an object can also protect the objects encapsulated within that object.

Our experience suggests that SafeJava is expressive enough to support common programming patterns, its type checking is fast and scalable, and it requires little programming overhead. In addition, the type declarations in SafeJava programs serve as documentation that lives with the code and is checked throughout the evolution of code. The SafeJava type system thus provides significant software engineering benefits and offers a promising approach for improving the reliability of multithreaded and object-oriented programs.

This paper presents a corresponding type system for (a subset of) the Java virtual machine language (JVML). We call the resulting language *SafeJVML*. Well-typed SafeJVML programs are guaranteed to be free of data races and deadlocks. Well-typed programs are also guaranteed to enforce object encapsulation. This paper presents the static and dynamic semantics of SafeJVML, and includes a proof that the SafeJVML type system is sound and that it prevents data races and deadlocks and enforces encapsulation.

Designing a corresponding type system for JVML is important because it is the format of choice for shipping code. Systems that download untrusted JVML programs first perform bytecode verification to ensure absence of memory errors before running the programs. With our proposed extension to the JVML type sys-

```
class Account {
  private int balance;
  static void transfer(Account from, Account to, int x) {
    synchronized (from) {
      synchronized (to) {
        to.balance   += x;
        from.balance -= x;
    }}
  }
}
```

Figure 1. A `transfer` method in Java

```
static void transfer(Account,Account,int);
 1:  load 0
 2:  store 3
 3:  load 3
 4:  monitorenter
 5:  load 1
 6:  store 4
 7:  load  4
 9:  monitorenter
10:  load 1
11:  load 1
12:  getfield #2; //balance:I
15:  load 2
16:  add
17:  putfield #2; //balance:I
20:  load 0
21:  load 0
22:  getfield #2; //balance:I
25:  load 2
26:  sub
27:  putfield #2; //balance:I
30:  load   4
32:  monitorexit
44:  load 3
45:  monitorexit
56:  return
```

Figure 2. The `transfer` method in Figure 1 in JVML (excluding exception handling)

tem, a bytecode verifier can also statically ensure the absence of data races and deadlocks in a program before running it. Moreover, many code bases have a combination of Java source code and JVML code. Verifying the absence of races and deadlocks in such a code base requires corresponding race-free and deadlock-free type systems for both Java and JVML.

Designing a corresponding type system and a *syntax-directed* type checker for JVML is nontrivial because of important differences between Java and JVML. In particular, the absence of block structure in JVML programs and that they do not use named local variables like Java programs make the type systems for Java and JVML significantly different. For example, verifying absence of races and deadlocks in JVML programs involves performing an alias analysis, something we did not have to do for verifying Java programs.

Consider the `transfer` method in Figure 1. Suppose there are type annotations (not shown in the figure) that declare that every `Account` object is protected by its own lock. A type checker can then statically verify that the `transfer` method is race-free because the accesses to the `balance` field of the `to` and `from Account` objects happen within the block of code where the locks on the `to` and `from Account` objects are held. Now consider the corresponding JVML code in Figure 2. To check that the JVML code is race-free, one must use alias analysis to statically ensure that the `getfield` and `putfield` instructions operate on the same objects on which the locks are obtained using `monitorenter`. Moreover, one must statically ensure that `getfield` and `putfield` accesses happen after the corresponding `monitorenter` instructions and before the corresponding `monitorexit` instructions, something that is nontrivial in general if the code is not block structured and uses gotos.

To the best of our knowledge, this is the first type system for JVML that statically prevents data races, deadlocks and encapsulation violations. This paper combines ideas from four different systems—i) formalization of the JVML type system [4, 20, 29], ii) type systems for preventing data races and deadlocks in Java programs [5, 9, 6, 16]) iii) ownership types for enforcing object encapsulation [1, 3, 7, 13], and iv) type systems for JVML for statically ensuring that

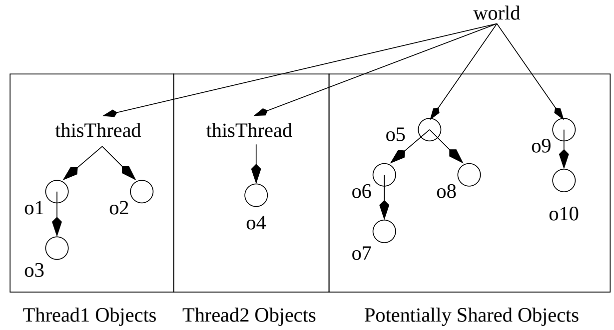

Figure 3. An ownership relation

E1. The owner of an object does not change over time.

E2. The ownership relation forms a tree rooted at `world`.

E3. If object z owns y but $z \not\succeq x$, then x cannot access y.

R4. To safely access an object, a thread must hold the lock on the *root owner* of that object. (r is the root owner of an object o iff $r \succeq o$ and `world` directly owns r.)

R5. Every thread implicitly holds the lock on its corresponding `thisThread` owner. A thread can access objects owned by its `thisThread` without synchronization.

D6. Every lock belongs to some lock level. The lock level of a lock does not change over time. The lock levels form a partial order.

D7. To acquire a new lock of lock level l, the levels of all the locks held by the thread must be greater than l.

Figure 4. SafeJava properties

methods release all locks they acquire and no other lock [26, 23]—and achieves what neither of these systems individually achieves.

We also note in passing that any type system that guarantees race freedom also eliminates all the complex issues associated with the use of weak memory consistency models [28]. A detailed explanation of this issue can be found in [2]. The rest of the paper is organized as follows. Section 2 presents an overview of SafeJava. Section 3 presents SafeJVML, including its dynamic and static semantics and a soundness proof. Section 4 describes related work.

2. Overview of SafeJava

This section presents an overview of a core subset of SafeJava for preventing data races [9] and deadlocks [6] and encapsulation violations [7]. The key to the type system is the concept of object ownership. Every object has an owner. An object can be owned by another object, by a special per-thread owner called `thisThread`, or a global owner called `world`. We use the notation $o_1 \succeq o_2$ to denote that o_1 directly or transitively owns o_2 or o_1 is the same as o_2. The relation $\succeq$ is thus the reflexive transitive closure of the *owns* relation. If `thisThread` $\succeq o$, then o is local to the corresponding thread and cannot be accessed by any other thread. All other objects are potentially shared between multiple threads. Figure 3 presents an example ownership relation. We draw an arrow from object x to object y if x owns y. Our type system statically verifies that a program respects the properties shown in Figure 4. Properties E1-E3 ensure encapsulation. Properties R4 and R5 prevent races.

Figure 5 shows a `TStack` program in SafeJava. A `TStack` is a stack of `T` objects. A `TStack` is implemented using a linked list. A class definition in SafeJava is parameterized by a list of owners.

```
class TStack<thisOwner, TOwner> {
    TNode<this, TOwner> head = null;

    T<TOwner> pop() requires(this) {
        if (head == null) return null;
        T<TOwner> value = head.value;  head = head.next;
        return value;
    }
}
class TNode<thisOwner, TOwner> {
    T<TOwner>                value;
    TNode<thisOwner, TOwner> next;
}

TStack<thisThread, thisThread> s1;
TStack<thisThread, world>      s2;
TStack<world,      world>      s3;
```

Figure 5. Stack of T objects in SafeJava

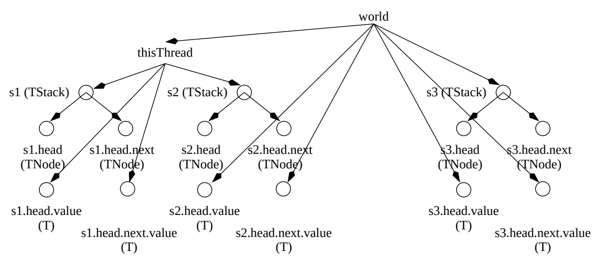

Figure 6. Ownership relation for TStacks s1,s2,s3

This parameterization helps programmers write generic code to implement a class, then create different objects of the class that have different protection mechanisms. In Figure 5, the `TStack` class is parameterized by `thisOwner` and `TOwner`. `thisOwner` owns the `this TStack` object and `TOwner` owns the `T` objects contained in the `TStack`. In general, the first formal parameter of a class always owns the `this` object. In case of `s1`, the owner `thisThread` is used for both the parameters to instantiate the `TStack` class. It means that `TStack s1` as well as all the `T` objects contained in the `TStack` are local to the main thread. In case of `s2`, the `TStack` is local to the main thread but the `T` objects contained in the `TStack` are potentially shared between multiple threads. In case of `s3`, both the `TStack` and the `T` objects contained in the `TStack` are potentially shared between multiple threads. The ownership relation for the `TStack` objects `s1`, `s2`, and `s3` is depicted in Figure 6 (assuming the stacks contain two elements each). In SafeJava, a method can contain a `requires` clause that specifies the objects the method accesses that must be protected by externally acquired locks. Callers are required to hold the locks on the *root owners* (see Figure 4) of the objects specified in the `requires` clause before they invoke a method to avoid data races. The `pop` method assumes that the callers hold the lock on the root owner of the `TStack` object.

To prevent deadlocks, programmers partition all the locks in our system into a fixed number of lock levels and specify a partial order among the lock levels. The type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order. A lock level is like a static field in Java; a lock level is a per-class entity rather than a per-object entity. But unlike static fields in Java, lock levels are used only for compile-time type checking and are not preserved at runtime. Programmers can specify a partial order among the lock levels using the $<$ and $>$ syntax in the lock level declarations. Since a program has a fixed number of lock levels, our type checker can statically verify that the lock levels do indeed form a partial order. Properties D6 and D7 in Figure 4 prevent deadlocks.

P	$::=$	*defn**
defn	$::=$	`class` *cn*$\langle$*formal+*$\rangle$ `extends` *c* `where` *constr** {*level* field* meth**}
c	$::=$	*cn*$\langle$*owner+*$\rangle$ | `Object`$\langle$*owner*$\rangle$
owner	$::=$	*formal* | `this` | `world`:*cn.l* | `thisThread`
constr	$::=$	(*owner* $\succeq$ *owner*) | (*owner* $\not\succeq$ *owner*)
level	$::=$	`LockLevel` *l* = `new` | `LockLevel` *l* $<$ *cn.l** $>$ *cn.l**
meth	$::=$	*t mn*(*t**) `requires` ($x_{0..k}$) `locks` (*cn.l**) {***inst****}
field	$::=$	*t fd*
methodref	$::=$	$\|$*c*, *mn*$\langle$*owner+*$\rangle$, (*t**), *t*, `requires`($x_{0..k}$)$\|_M$
fieldref	$::=$	$\|$*c*, *fd*, *t*$\|_F$
t	$::=$	*c* | `int` | c_i , i is an integer
formal	$::=$	*f*
inst	$::=$	`push` *v* | `pop` | `store` *x* | `load` *x* | `add` | `ifeq` *L* | `new` *c* | `invokevirtual` *methodref* | `returnval` | `start` | `getfield` *fieldref* | `putfield` *fieldref* | `monitorenter` | `monitorexit`

fd $\in$ field names, *mn* $\in$ method names, cn $\in$ class names, f $\in$ owner names, *x* $\in$ variable names, *M* $\in$ methodref names, *F* $\in$ fieldref names, *v* $\in$ Integers

Figure 7. SafeJVML grammar

Note that the complete SafeJava language is more expressive than the core subset presented here, and supports most of the commonly used synchronization patterns. It also supports safe region-based memory management [10] and safe software upgrades [8]. A detailed description can be found in [5] and [6, 7, 8, 9, 10].

3. SafeJVML

This section presents SafeJVML, an extension to the Java virtual machine language (JVML) for statically preventing data races and deadlocks as well as for statically enforcing object encapsulation. To simplify the presentation of key ideas behind our approach, we describe our system formally in the context of a core subset of JVML. In particular, we avoid subroutines and object initialization because they are orthogonal to preventing synchronization and encapsulation errors. However, they can be easily added to our system using previous work on formalization of subroutines [11, 24, 30] and object initialization [19] for JVML.

Figure 7 shows the grammar for SafeJVML. The grammar is similar to the grammar for SafeJava [5] with respect to the class and method signatures. But the SafeJVML instructions are different from those of SafeJava. In particular, JVML programs are not block structured and do not use named local variables. This makes it difficult to design a syntax-directed type checker for JVML that tracks the relation between locks acquired and the objects they protect. To address this problem we use indexed types [26], which statically guarantee that all variables with the same indexed type c_i are aliases. Indexed types [26] were previously used to statically ensure that the `monitorenter` and `monitorexit` instructions are matched along every program path. In this paper, we adopt the idea to ensure absence of races and deadlocks in JVML programs.

The SafeJVML instruction set closely resembles the JVML instruction set. The only difference is the format of *methodref* shown in Figure 7. In SafeJVML, *methodref* also includes a `requires` clause which specifies the objects the method accesses that must be protected by externally acquired locks. Each x_i in `requires`($x_{0..k}$) denotes the i^{th} argument passed to the method. For example, `requires`(x_0, x_1) specifies that the `this` object (x_0) and the first argument (x_1) must be protected by externally acquired locks.

3.1 Dynamic Semantics

This section presents a small step operational semantics for SafeJVML. This is necessary to formally define the semantics of SafeJVML programs, and well as to state and prove the type soundness

$$\begin{array}{lcl} C & := & \Phi; h \\ \Phi & := & T\ \Phi \mid \epsilon \\ T & := & (A) \\ A & := & \langle M, pc, f, s, ls\rangle\ A \mid \epsilon \\ h & : & location \rightarrow \langle fd_i{=}v_i, g_j{=}w_j, level{=}cn'.l\rangle^{i\in\{1..m\}, j\in\{1..n\}}_{cn\langle w_{1..n}\rangle} \end{array}$$

Figure 8. SafeJVML execution state

theorems. We call the corresponding virtual machine SafeJVM. The SafeJVM execution state is a configuration $C := \Phi; h$, where Φ is a set of threads and h is a memory heap. Each thread T in the thread set Φ has a stack of activation records. Each activation record A consists of the *methodref* M of the method, the address pc of the next instruction in the code array, a map f from the set of local variables to values, the operand stack s, and a set of locks ls. The heap is modeled as a partial function h mapping locations to records. The definitions are shown in Figure 8. $fd_{1..m}$ denote the fields in an object of type $cn\langle w_{1..n}\rangle$. The special fields $g_{1..n}$ track the runtime owners $w_{i..n}$ of the object. These fields are named after the static formal owner parameters $g_{1..n}$ of the corresponding class. The special field $level$ stores the lock level of the lock associated with this object. We use the notation $h[o].fd$ to access the value of field fd from the instance at location o in heap h. To create a new heap with a modified value for that field, we use the notation $h[o.fd \rightarrow v]$. The special field l represents the lock associated with each object. We assume that every object's record has this field.

Note that we include ls for each method frame that belongs to a thread instead of having a global lock set for a thread. The reason is to simplify the type soundness proofs by maintaining close correspondence with static semantics where we have a separate static lock set for each method to enable modular checking of methods. ls for a method frame contains the locks that are acquired within the method and the locks that are specified in its `requires` clause. The locks specified in the `requires` clause are externally acquired locks; we check that these locks are indeed held by the thread before adding them to the current frame's lock set. Therefore the lock set held by a thread T is the union of the lock sets held in the activation records of the thread. That is, $Locks(T) = \cup_{\langle A\rangle \in T}(ls \in A)$. We also maintain the lock levels even though they are unnecessary, to maintain close correspondence with static semantics.

Figure 10 presents the dynamic semantics for SafeJVML and Figure 9 presents some auxiliary definition. The rules in Figure 10 only include the components that participate in the transition. The transition however applies to every configuration that contains the components using the following congruence rule. Below, Φ, Φ_1, and Φ_2 are sets of threads, and $Locks[\Phi]$ is the collection of objects locked by threads in Φ. That is, $Locks[\Phi] = \cup_{(T\in\Phi)} Locks[T]$.

$$\frac{\Phi_1; h \rightarrow \Phi_2; h' \qquad (Locks[\Phi_1] \cup Locks[\Phi_2]) \cap Locks[\Phi] = \phi}{\Phi_1 \cup \Phi; h \rightarrow \Phi_2 \cup \Phi; h'}$$

The `new` instruction creates a new object and initializes its fields to default values. It also initializes fields $g_{1..n}$ with runtime owners of the object. To access the runtime owners, it uses a map *RO* shown in Figure 9, which takes an object and a static owner parameter and returns the corresponding runtime owner. The `start` instruction starts a new thread with the lock set that contains only `thisThread`, because the new thread does not inherit any locks from its parent thread. The control of the new thread is transferred to its `start` method. Figure 10 presents these and other rules formally.

3.2 Static Semantics

This section describes the static semantics of SafeJVML. Following standard practice in JVML type system formalizations [20, 26, 30],

$$type(h,v) = \left\{ \begin{array}{l} \text{int, if } v \text{ is an } integer \\ t\text{, if } v \in location \text{ and } h[v] = \langle..\rangle_t \end{array} \right\}$$

$$lock(h,v) = \left\{ \begin{array}{l} \texttt{thisThread}\text{, if } type(h,v) = cn\langle\texttt{thisThread}..\rangle \\ v\text{, if } type(h,v) = cn\langle \text{ world:cn'.l..}\rangle \\ lock(h,v')\text{, if } type(h,v) = cn\langle v', ..\rangle \end{array} \right\}$$

$$level(h,g) = \left\{ \begin{array}{l} cn.l\text{, if } g = \text{world:}cn.l \\ \infty\text{, otherwise} \end{array} \right\}$$

$$RO(o,g) = \left\{ \begin{array}{l} \texttt{thisThread}\text{, if } g = \texttt{thisThread} \\ \texttt{world}\text{, if } g = \texttt{world:}cn.l \\ o\text{, if } g = \texttt{this} \\ \text{h[o].}g\text{, otherwise} \end{array} \right\}$$

f_0 : function mapping local variables to arbitrary values

Figure 9. Auxiliary definitions for dynamic semantics

we assume there is a separate intraprocedural type inference phase that infers the types of local variables at every program point. This paper only describes the type checking rules. Type inference can be performed by solving the constraints generated by the type checking rules. The advantage of separating type inference from type checking is that it reduces the size of the trusted computing base; a bug in type inference cannot compromise a JVM, only a bug in type checking can. Moreover, type checking becomes syntax directed. We also assume the SafeJava to SafeJVML compiler generates programs according to the grammar in Figure 7. That is, the compiler preserves the type annotations on class and method signatures.

The core of our type system is a set of rules for reasoning about the typing judgment: $P, E, F, S, LS, L_{min}, i \vdash M$. P denotes the program that is being checked. It contains the information about class definitions. The typing environment E tracks the owners and constraints which are in scope. The typing environment contains the declared owner parameters, the declared constraints among owners, and the declared `locks` clause in scope:

$$E ::= \emptyset \mid E, \text{owner } f \mid E, constr \mid E, \texttt{locks}(cn.l*)$$

F, S, LS, and L_{min} provide respectively the types of local variables, the types of stack slots, the locks that are statically known to be held, and the sequence of minimum lock levels at every program point. That is, F_i is the map from local variables to types at i^{th} instruction. S_i is a sequence of types of the operand stack at i^{th} instruction. LS_i is a multi-set of indexed object types denoting the locks held at instruction i. L_{min_i} is a sequence of l_{min}'s. Recall lock levels from Section 2. The definition of l_{min} is as follows:

$$l_{min} ::= \infty \mid cn.l \mid \text{LUB}(cn_1.l_1 \ ...\ cn_k.l_k)$$

By definition, $\text{LUB}(cn_1.l_1 \ ...\ cn_k.l_k) > cn_i.l_i\ \forall_{i=1..k}$. LUB(...) is not computed—it is an expression used as such for type checking. The lock level ∞ denotes that the thread currently holds no locks.

Figures 11 presents the static semantics for the instructions in SafeJVML. The full set of typing rules are in the appendix. The judgment $P, E, F, S, LS, L_{min}, i \vdash M$ denotes that instruction i satisfies all type constraints. $M[i]$ denotes the i^{th} instruction of the method with *methodref* M. We use the notation that for any type t, t.owners = $o_{1..n}$ if $t = cn\langle o_{1..n}\rangle$. Also, $t[o_1/f_1][o_2/f_2]..[o_m/f_m]$ denotes the type t in which the formal owner parameters are replaced with actual owner parameters.

Figure 12 illustrates the types at every program point for the `transfer` method shown in Figure 2. We use this example to explain few of our typing rules. Like we mentioned before, we use indexed types to keep track of aliases. The indexed type c_i is the type of

$$\frac{\text{M[pc] = }\texttt{push}\ v}{(\langle M, pc, f, s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f, v.s, ls\rangle.A); h} \qquad \frac{\text{M[pc] = }\texttt{pop}}{(\langle M, pc, f, v.s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f, s, ls\rangle.A); h}$$

$$\frac{\text{M[pc]= }\texttt{ifeq}\ L \quad v_1 = v_2}{(\langle M, pc, f, v_1.v_2.s, ls\rangle.A); h \rightarrow (\langle M, L, f, s, ls\rangle.A); h} \qquad \frac{\text{M[pc]= }\texttt{ifeq}\ L \quad v_1 \neq v_2}{(\langle M, pc, f, v_1.v_2.s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f, s, ls\rangle.A); h}$$

$$\frac{\text{M[pc]= }\texttt{add}}{(\langle M, pc, f, v_1.v_2.s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f, (v_1{+}v_2).s, ls\rangle.A); h}$$

$$\frac{\text{M[pc] = }\texttt{getfield}\ \|cn\langle f_{1..n}\rangle, fd, t\|_F}{(\langle M, pc, f, o.s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f, (h[o].\|cn\langle f_{1..n}\rangle, fd, t\|_F).s, ls\rangle.A); h}$$

$$\frac{\text{M[pc] = }\texttt{putfield}\ \|cn\langle f_{1..n}\rangle, fd, t\|_F}{(\langle M, pc, f, v.o.s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f, s, ls\rangle.A); h[o.\|cn\langle f_{1..n}\rangle, fd, t\|_F \mapsto v]}$$

$$\frac{\text{M[pc] = }\texttt{start} \quad o \in Dom(h)}{(\langle M, pc, f, o.s, ls\rangle A); h \rightarrow (\langle M_{start}, 1, f_0[0 \rightarrow o], \epsilon, \texttt{thisThread}\rangle).(\langle M, pc{+}1, f, s, ls\rangle.A); h}$$

$$\frac{\text{M[pc]= }\texttt{load}\ x}{(\langle M, pc, f, s, ls\rangle A); h \rightarrow (\langle M, pc{+}1, f, f[x].s, ls\rangle.A); h} \qquad \frac{\text{M[pc] = }\texttt{monitorenter} \quad h[o].l = 0}{(\langle M, pc, f, o.s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f, s, ls \cup \{o\}\rangle.A); h[o.l \mapsto 1]}$$

$$\frac{\text{M[pc] = }\texttt{returnval} \quad M = \|cn\langle o_{1..n}\rangle, mn\langle o_{n+1..m}\rangle, \alpha, \gamma, requires(x_{0..k})\|_M}{(\langle M, pc, f, v.s, ls\rangle.\langle M', pc', f', s', ls'\rangle.A); h \rightarrow (\langle M', pc', f', v.s', ls'\rangle.A); h}$$

$$\frac{\text{M[pc] = }\texttt{store}\ x}{(\langle M, pc, f, v.s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f[x \rightarrow v], s, ls\rangle.A); h} \qquad \frac{\text{M[pc] = }\texttt{monitorenter} \quad h[o].l = n, (n > 0)}{(\langle M, pc, f, o.s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f, s, ls\rangle.A); h[o.l \mapsto n{+}1]}$$

$$\frac{\text{M[pc] = }\texttt{new}\ cn\langle o_{1..n}\rangle \quad o \notin Dom(h) \quad class\ cn\langle g_{1..n}\rangle \ldots \in P}{(\langle M, pc, f, s, ls\rangle.A); h \rightarrow (\langle M, pc{+}1, f, o.s, ls\rangle.A); h[o \mapsto Defaults(cn)][o.level \mapsto level(h, o_1)][o.g_i \mapsto RO(f[0], o_i)]_{\forall i \in 1..n}}$$

$$\frac{\text{M[pc] = }\texttt{monitorexit} \quad h[o].l = 1}{(\langle M, pc, f, o.s, ls \cup \{o\}\rangle A); h \rightarrow (\langle M, pc{+}1, f, s, ls\rangle.A); h[o.l \mapsto 0]}$$

$$\frac{\text{M[pc] = }\texttt{invokevirtual}\ \|cn\langle f_{1..n}\rangle, mn, \alpha, \gamma, \texttt{requires}(x_{0..k})\|_N \quad |\alpha| = |s_1| \quad \forall i \in \{0..k\}, lock(h, (o.s_1)[i]) \in ls}{(\langle M, pc, f, s_1.(o.s), ls\rangle.A); h \rightarrow (\langle N, 1, f_0[0 \rightarrow o, 1..|\alpha| \rightarrow s_1], \epsilon, \{\texttt{thisThread} \cup (\cup_{i \in 0..k} lock(h, (o.s_1)[i]))\}\rangle.\langle M, pc, f, s, ls\rangle.A); h}$$

$$\frac{\text{M[pc] = }\texttt{monitorexit} \quad h[o].l = n, (n > 1)}{(\langle M, pc, f, o.s, ls\rangle A); h \rightarrow (\langle M, pc{+}1, f, s, ls\rangle.A); h[o.l \mapsto n-1]}$$

Figure 10. Dynamic semantics for SafeJVML

variables whose value of type c was first copied at i^{th} instruction. In a well-typed program, all variables that have the same type c_i are guaranteed to be aliases. Consider the rule for `load` in Figure 11. If it is the first copy of the value, then its type is changed; the type of the object is tagged with the program point at which the `load` is performed. For example, in Figure 12, one of the `Account` object's type is tagged with `PC` 1, the instruction at which the object is first copied on to the stack. The other `Account` object's type is tagged with `PC` 5. Successive copies preserve the type of the first copy.

We define an indexing operation over types t to mark types when variables are copied. Let t_i, where i is an integer, be the following:

$$\begin{array}{lcll} t_i & = & t, & \text{if } t\text{=int} \\ t_i & = & c_i, & \text{if } t = c \text{ (first copy of the variable changes the type)} \\ t_i & = & \hat{c}, & \text{if } t = \hat{c} \text{ (successive copies keep the type of the first copy)} \end{array}$$

We also define *Index* and *Type* as partial functions from types to integers and types respectively. The notation $\hat{t}$ denotes indexed types.

$$\begin{array}{rcll} Index[c_i] & = & i, & \text{and } Index \text{ is undefined otherwise} \\ Type[c_i] & = & c, & \text{and } Type[t] = t \text{ otherwise} \end{array}$$

Before we proceed, we explain the auxiliary function $Lock(\hat{t})$ (formally defined in the appendix). A lock is an object directly owned by world. $Lock(\hat{t})$ denotes the lock that protects an object with indexed type $\hat{t}$. If the owner of an object is a formal owner parameter, then we cannot determine the root owner of the object from within the static scope of the enclosing class. In that case, we define the root owner of the object with indexed type $\hat{t}$ to be $L(\hat{t})$. Note that $L(\hat{t})$ is not computed—it is used as such for type checking.

The rule for acquiring a new lock using `monitorenter` in Figure 11 checks that the top of the stack is a lock of some lock level $cn'.l$ that is less than l_{min}. The rule also ensures that after the instruction, $cn'.l$ is stored on the top of $L_{min_{i+1}}$ sequence. The rule for `getfield` in Figure 11 checks that the class declares or inherits the field and that the type on the top of the stack matches the type of the class in which the field is declared. It also checks that the thread holds the lock on the *root owner* (see Figure 4) of the object.

Going back to our example in Figure 12, the thread acquires the lock on $\texttt{Account}_1$ and $\texttt{Account}_5$ objects before accessing their `balance` fields. By consistently acquiring the lock on an object before accessing its fields, the potential for data races is avoided. There are two points to note in this example. One is that the type system statically tracks that the `monitorenter` operations are performed on the same objects whose fields are accessed by the `getfield` instructions. The second point is to note that the type system statically tracks that each `getfield` operation is performed after the corresponding `monitorenter` operation and before the corresponding `monitorexit` operation on the same object.

The rule for invoking a method using `invokevirtual` in Figure 11 checks that the arguments are of right type and that the thread holds the locks on the root owners of all the expressions in the requires clause. The rule ensures that l_{min}, which is the topmost value in the L_{min_i} sequence is greater than all the levels specified in the locks clause of the method. The rule appropriately renames the expressions and types used outside their declared context. Figure 11 presents the rules for these and other instructions formally. The appendix contains the rest of the type checking rules.

3.3 Soundness

This section provides a proof that the SafeJVML type system is sound and that well-typed SafeJVML programs do not have data races or deadlocks or encapsulation errors. We first define a good machine state configuration. We use the notation $P \vdash h\ wt$ to denote that the heap h is well-typed. The rules for mapping run-

M[i]= **push** v

$$\frac{\begin{array}{c} P,E \vdash int.S_i <: S_{i+1} \\ P,E \vdash F_i <: F_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **pop**

$$\frac{\begin{array}{c} S_i = t.\beta \\ P,E \vdash \beta <: S_{i+1} \\ P,E \vdash F_i <: F_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **add**

$$\frac{\begin{array}{c} P,E \vdash S_i <: int.int.\beta \\ P,E \vdash int.\beta <: S_{i+1} \\ P,E \vdash F_i <: F_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i]= **Load** x

$$\frac{\begin{array}{c} x \in Dom(F_i) \\ P,E \vdash F_i[x] = t \\ P,E \vdash F_i[x \rightarrow t_i] <: F_{i+1} \\ P,E \vdash t_i.S_i <: S_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ \forall \hat{c} \in S_i.\, i \neq Index[\hat{c}] \\ \forall y \in Dom[F_i].\, i \neq Index[F_i[y]] \\ \forall \hat{c} \in LS_i.\, i \neq Index[\hat{c}] \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **Store** x

$$\frac{\begin{array}{c} x \in Dom(F_i) \\ S_i = t.\beta \\ P,E \vdash \beta <: S_{i+1} \\ P,E \vdash F_i[x \rightarrow t] <: F_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **ifeq L**

$$\frac{\begin{array}{c} P,E \vdash S_i < t.t.\beta \\ P,E \vdash \beta <: S_{i+1} \\ P,E \vdash F_i <: F_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ P,E \vdash \beta <: S_L \\ P,E \vdash F_i <: F_L \\ P,E \vdash LS_i = LS_L \\ P \vdash L_{min_i} = L_{min_L} \\ i+1, L \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **monitorenter**

$$\frac{\begin{array}{c} P,E \vdash F_i <: F_{i+1} \\ P,E \vdash S_i <: \hat{cn}\langle world : cn'.l, ..\rangle.S_{i+1} \\ P,E \vdash \{\hat{cn}\langle world : cn'.l..\rangle\} \notin LS_i \\ P,E \vdash LS_{i+1} = LS_i \cup \{\hat{cn}\langle world : cn'.l..\rangle\} \\ L_{min_i} = l_{min}.\beta' \\ P \vdash cn'.l < l_{min} \\ P \vdash L_{min_{i+1}} = (cn'.l).l_{min}.\beta' \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **monitorexit**

$$\frac{\begin{array}{c} P,E \vdash F_i <: F_{i+1} \\ P,E \vdash S_i <: \hat{cn}\langle world : cn'.l, ..\rangle.S_{i+1} \\ P,E \vdash \{\hat{cn}\langle world : cn'.l, ..\rangle\} \in LS_i \\ P,E \vdash LS_{i+1} = LS_i - \{\hat{cn}\langle world : cn'.l, ..\rangle\} \\ P \vdash L_{min_i} = (cn'.l).L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **start**

$$\frac{\begin{array}{c} P,E \vdash Type[\hat{t}] : Thread \\ P,E \vdash S_i <: \hat{t}.S_{i+1} \\ P,E \vdash F_i <: F_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **new** $cn\langle o_{1..n}\rangle$

$$\frac{\begin{array}{c} P \vdash cn\langle f_{1..n}\rangle \text{ where } constr* \\ P,E \vdash o_i \succeq o_1 \\ P,E \vdash_{owner} o_i \\ P,E \vdash constr[o_1/f_1]..[o_n/f_n] \\ P,E \vdash cn\langle [o_1/f_1]..[o_n/f_n]\rangle.S_i <: S_{i+1} \\ P,E \vdash F_i <: F_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **returnval**

$$\frac{\begin{array}{c} \gamma \neq void \\ P,E \vdash S_i <: \gamma.\beta \\ P,E \vdash LS_i = LS_1 \\ P,E \vdash L_{min_i} = L_{min_1} \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **getfield** $\|cn\langle f_{1..n}\rangle, fd, t\|_F$

$$\frac{\begin{array}{c} P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \\ P,E \vdash S_i <: \hat{cn}\langle o_{1..n}\rangle.\beta \\ Type[\hat{cn}\langle o_{1..n}\rangle] = cn\langle o_{1..n}\rangle \\ P,E \vdash Lock(\hat{cn}\langle o_{1..n}\rangle) \in LS_i \\ P,E \vdash t[o_1/f_1]..[o_n/f_n][\hat{cn}\langle o_{1..n}\rangle/\mathtt{this}].\beta <: S_{i+1} \\ P,E \vdash F_i <: F_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **putfield** $\|cn\langle f_{1..n}\rangle, fd, t\|_F$

$$\frac{\begin{array}{c} P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \\ P,E \vdash S_i <: t[o_1/f_1]..[o_n/f_n][\hat{cn}\langle o_{1..n}\rangle/\mathtt{this}].\hat{cn}\langle o_{1..n}\rangle.\beta \\ Type[\hat{cn}\langle o_{1..n}\rangle] = cn\langle o_{1..n}\rangle \\ P,E \vdash Lock(\hat{cn}\langle o_{1..n}\rangle) \in LS_i \\ P,E \vdash \beta <: S_{i+1} \\ P,E \vdash F_i <: F_{i+1} \\ P,E \vdash LS_i = LS_{i+1} \\ P \vdash L_{min_i} = L_{min_{i+1}} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash M}$$

M[i] = **invokevirtual** $\|cn\langle f_{1..n}\rangle, mn, \alpha, \gamma, requires(x_{0..k})\|_M$

$$\frac{\begin{array}{c} P \vdash (mn, \alpha, \gamma, requires(x_{0..k}), locks(cn'.l^*)) \in cn\langle f_{1..n}\rangle \\ P,E \vdash F_i <: F_{i+1} \\ \mathtt{Renamed}(\alpha) \stackrel{def}{=} \alpha[o_1/f_1]...[o_n/f_n][\hat{cn}\langle o_{1..n}\rangle/\mathtt{this}] \\ P,E \vdash S_i <: \text{Renamed}(\hat{\alpha}).\hat{cn}\langle o_{1..n}\rangle.\beta \qquad P,E \vdash \text{Renamed}(\hat{\gamma}).\beta <: S_{i+1} \\ \forall j \in [0..k].\ P,E \vdash Lock((\hat{cn}\langle o_{1..n}\rangle.\text{Renamed}(\hat{\alpha}))[j]) \in LS_i \qquad P \vdash LS_{i+1} = LS_i \\ L_{min_i} = l_{min}.\beta' \qquad \forall cn_i.l_i \in cn'.l^*.\ P \vdash cn_i.l_i < l_{min} \qquad P \vdash L_{min_{i+1}} = L_{min_i} \\ i+1 \in Dom(M) \end{array}}{P,E,F,S,LS,L_{min},i \vdash P : M}$$

Figure 11. Static semantics for SafeJVML instructions

PC	Instruction	$F_i[0]$	$F_i[1]$	$F_i[2]$	$F_i[3]$	$F_i[4]$	S_i	LS_i
1	`load 0`	Account	Account	int	–	–	ϵ	ϕ
2	`store 3`	$\mathtt{Account}_1$	Account	int	–	–	$\mathtt{Account}_1$	ϕ
3	`load 3`	$\mathtt{Account}_1$	Account	int	$\mathtt{Account}_1$	–	ϵ	ϕ
4	`monitorenter`	$\mathtt{Account}_1$	Account	int	$\mathtt{Account}_1$	–	$\mathtt{Account}_1$	ϕ
5	`load 1`	$\mathtt{Account}_1$	Account	int	$\mathtt{Account}_1$	–	ϵ	$\mathtt{Account}_1$
6	`store 4`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	–	$\mathtt{Account}_5$	$\mathtt{Account}_1$
7	`load 4`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	ϵ	$\mathtt{Account}_1$
9	`monitorenter`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	$\mathtt{Account}_5$	$\mathtt{Account}_1$
10	`load 1`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	ϵ	$\mathtt{Account}_1$,$\mathtt{Account}_5$
11	`load 1`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	$\mathtt{Account}_5$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
12	`getfield`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	$\mathtt{Account}_5$.$\mathtt{Account}_5$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
15	`load 2`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int.$\mathtt{Account}_5$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
16	`add`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int.int.$\mathtt{Account}_5$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
17	`putfield`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int.$\mathtt{Account}_5$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
20	`load 0`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	ϵ	$\mathtt{Account}_1$,$\mathtt{Account}_5$
21	`load 0`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	$\mathtt{Account}_1$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
22	`getfield`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	$\mathtt{Account}_1$.$\mathtt{Account}_1$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
25	`load 2`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int.$\mathtt{Account}_1$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
26	`sub`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int.int.$\mathtt{Account}_1$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
27	`putfield`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int.$\mathtt{Account}_1$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
30	`load 4`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	ϵ	$\mathtt{Account}_1$,$\mathtt{Account}_5$
32	`monitorexit`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	$\mathtt{Account}_5$	$\mathtt{Account}_1$,$\mathtt{Account}_5$
44	`load 3`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	ϵ	$\mathtt{Account}_1$
45	`monitorexit`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	$\mathtt{Account}_1$	$\mathtt{Account}_1$
56	`return`	$\mathtt{Account}_1$	$\mathtt{Account}_5$	int	$\mathtt{Account}_1$	$\mathtt{Account}_5$	ϵ	ϕ

Figure 12. Static types for the `transfer` method in Figure 2

time values in the heap to types are given at the end of appendix. *GoodConfiguration*(P, Φ, h) states that given program P and heap h, the thread set Φ is well-typed iff for every activation record $\langle M, pc, f, s, ls\rangle \in$ thread T included in the thread set Φ, the conditions in Figure 13 hold with respect to the static type information F, S, and LS for the method M, and where E provides the information regarding owners, constraints, and locks clause in scope.

We now formally state and prove the theorems.

Theorem 1 (SafeJVML Preservation)
Suppose $P \vdash wt$. Then, $\forall\ \Phi, \Phi', h, h'$, **if** $P \vdash h\ wt$ **and** *GoodConfiguration*(P, Φ, h) **and** $P \vdash \Phi; h \rightarrow \Phi'; h'$, **then** $P \vdash h'\ wt$ **and** *GoodConfiguration*(P, Φ', h').

Proof: We sketch the proof for `getfield` $\|cn\langle f_{1..n}\rangle, fd, t\|_F$ instruction to motivate the structure of the invariants. First, we show that the execution of `getfield` $\|cn\langle f_{1..n}\rangle, fd, t\|_F$ instruction in a machine configuration $\Phi; h$, where $P \vdash h\ wt$ and *GoodConfiguration*(P, Φ, h), yields a new well-typed heap. For `getfield`, this is trivial to show since h is not modified. In general, the heap updates respect three properties: the types of records never change, values written into heap records have the same types as the overwritten values, and any new records introduced by allocation are well-typed records. If an instruction changes a heap h to h' according to these rules, then h' will be well-typed.

Next we show that the execution of `getfield` instruction preserves all the *GoodConfiguration* invariants listed in Figure 13. Suppose a `getfield` instruction moves the virtual machine from $(\langle M, pc, f, o.s, ls\rangle.A); h$ to $(\langle M, pc+1, f, v.s, ls\rangle.A); h$ where $v = h[o].\|cn\langle f_{1..n}\rangle, fd, t\|_F$. Further suppose that E, F, S and LS comprise the type information used to show that $P, E, F, S, LS, L_{min}, pc \vdash M$. We proceed by showing that all the conditions in Figure 13 hold. All conditions except C4 hold trivially since the `getfield` instruction does not affect these invariants. In fact, the only instruction that affects Condition C1 is the `new` instruction. It is easy to show that `new` preserves this invariant—every object has a unique owner and the ownership relation forms a tree before the execution of `new`, therefore adding a child to one of the nodes of the tree during `new`'s execution preserves the tree structure.

Coming back to the `getfield` instruction, the only condition that is affected by it is Condition 4 which states that $P, h \vdash s$: *RunTimeType*(s, S_{pc+1}). From the static type checking rule for `getfield` instruction, we have $P, E \vdash t.S_{pc} <: \hat{cn}\langle o_{1..n}\rangle.S_{pc+1}$ for some t and $\hat{cn}\langle o_{1..n}\rangle$. Since $P, h \vdash o.s'$: *RunTimeType*$(o.s', S_{pc})$, $P, h \vdash o$: *RunTimeType*$(o, \hat{cn}\langle o_{1..n}\rangle)$, and $P, h \vdash v$: *RunTimeType*(v, t), we can conclude that $P, h \vdash v.s'$: *RunTimeType*$(v.s', S_{pc+1})$. Thus $P, h \vdash s$: *RunTimeType*(s, S_{pc+1}). Therefore the execution of `getfield` preserves all the invariants in Figure 13.

Theorem 2 (SafeJVML Progress)
Let $P \vdash wt$ and $\forall\ \Phi, h$, $P \vdash h\ wt$ and *GoodConfiguration*(P, Φ, h). Then either:
i) $\exists \Phi', h'$. $P \vdash \Phi; h \rightarrow \Phi' : h'$ (progress), **or**
ii) $(\forall T \in \Phi).(T = \langle A\rangle \wedge A = \epsilon)$ (normal termination), **or**
iii) $\exists T \in \Phi$, s.t. T's next instruction is a null pointer dereference.

Proof: We prove this by showing that if $P \vdash h\ wt$ and *GoodConfiguration*(P, Φ, h) hold, then either the program is in stuck a deadlock state, or at least one thread is stuck attempting to dereference a null pointer, or at least one thread can make progress, or the activation record stack for every thread is empty. We later prove in Theorem 5 that a deadlock state is not possible because well-typed programs in SafeJVML are free of deadlocks. Thus, the above theorem holds. The details of the proof are similar to the details of the proof of Theorem 1 presented above.

Theorem 3 (SafeJVML Encapsulation)
An object x can access an object owned by o only if $(o \succeq x)$.

Proof: Recall that the notation $(o \succeq x)$ denotes that o directly or transitively owns x or o is same as x. Also, note that the owner of an object does not change over time and *GoodConfiguration* judgment holds before every instruction. Consider the code: *class* $C\langle f, ...\rangle\{...\ T\langle o, ...\rangle\ y\ ...\}$. Variable y of type $T\langle o, ...\rangle$ is declared within the static scope of C. Owner o can therefore be either 1) `this`, or 2) `world`, or a 3) a formal class parameter. We show that in each case, the constraint ($o \succeq$ `this`) holds. In the first two cases, the constraint holds trivially. In the last case, $(o \succeq f)$ and ($f \succeq$ `this`), so the constraint holds. Therefore an object x of a class C can access an object y owned by o only if $(o \succeq x)$.

C1. The ownership relation in the program forms a tree.

Recall that the owner of an object o is stored in $h[o].g_0$, where g_0 denotes the first formal parameter of o's class.

C2. The owners of every object satisfy the owner constraints specified in object's class.

That is, the runtime owners $h[o].g_1$, ..,$h[o].g_n$ of an object o satisfy the constraints declared in its class definition. Note that $cn\langle h[o].g_1, .., h[o].g_n\rangle$ gives the runtime type of an object o whose static type is $cn\langle g_{1..n}\rangle$. We use *RunTimeType*$(v, t)$ to denote the runtime type of a value v whose static type is t. *RunTimeType*$(v, t) = t$ if t is an integer. *RunTimeType*$(v, t) = cn\langle h[v].g_1, .., h[v].g_n\rangle)$ if $t = cn\langle g_{1..n}\rangle$.

C3. $pc \in Dom(M)$

C4. The stacks have values of the expected types.

That is, $s{=}v_1..v_k$ implies $S_{pc}{=}t_1..t_k$, and $P, h \vdash v_i$: *RunTimeType*(v_i, t_i). In short, $P, h \vdash s$: *RunTimeType*(s, S_{pc}).

C5. The local variables contain values of the expected types.

That is, $\forall y{\in}Dom(F_{pc})$, $(F_{pc}[y]{=}t) \Rightarrow P, h \vdash f[y]$: *RunTimeType*$(f[y], t)$.

C6. The static and dynamic lock sets are consistent.

(a) $\{o_j\} \in ls_0$ implies $P, h \vdash o_j$: *RunTimeType*$(o_j, \hat{t})$ and $\{\hat{t}\} \in LS_0$.

(b) $\{o_j\} \in ls$, implies $P, h \vdash o_j$: *RunTimeType*$(o_j, \hat{t})$ and $\{(k - k').\hat{t}\} \in LS_{pc}$, where $k = h[o_j].l$ at program point pc and $k' = h[o_j].l$ at program point 0.

ls_0 denotes the locks that are specified in the requires clause.

C7. Two variables with the same indexed type must be aliases.

Let,

(a) $o = f(x)$, when $F_{pc}[x] = \hat{t}$ or
$o = v_j$, when $s = v_1..v_k$, $S_{pc} = t_1..t_k$ and $t_j = \hat{t}$

(b) $o' = f(y)$, when $F_{pc}[y] = \hat{t}'$ or
$o' = v_{j'}$, when $s = v_1..v_k$, $S_{pc} = t_1..t_k$ and $t_{j'} = \hat{t}'$

If $\hat{t} = \hat{t}'$, then $o = o'$. Furthermore, if $\hat{t} \in LS_{pc}$, then $o \in ls$.

C8. The static and dynamic lock levels of the locks are consistent.

If $P, h \vdash o : cn\langle o_1 : l, o_{2..n}\rangle$, then $h[o].level = l$

Figure 13. Properties of a good machine state configuration

Theorem 4 (SafeJVML DataRaceFreedom)
Well-typed programs in SafeJVML are free of data races.

Proof: The type checking rules for SafeJVML ensure that every thread holds the lock protecting an object in its static lock set LS before accessing the object. The *GoodConfiguration* judgment ensures consistency between dynamic and static entities. Together, they ensure that every thread holds the lock protecting an object in its dynamic lock set ls before accessing the object. Well-typed SafeJVML programs are thus race free.

Theorem 5 (SafeJVML DeadLockFreedom)
Well-typed programs in SafeJVML are free of deadlocks.

Proof: The typing rules for SafeJVML ensure that the lock levels in the program form a partial order and that the locks are acquired in the decreasing order of their lock levels. The type checking rule for acquiring a new lock checks that the level of the lock being acquired is less than l_{min}, which is the topmost value in the L_{min} sequence; the type checking rules guarantee that l_{min} is the minimum level among the levels of all the locks already held in the static lock set LS. The typing rules along with *GoodConfiguration* judgment, which ensures the consistency between dynamic and static lock sets held by the thread, prove that the level of the lock acquired is less than the levels of the locks in dynamic lock set ls. Thus, well-typed programs in SafeJVML are free of deadlocks.

4. Related Work

This section presents work on related type systems. Our type system for checking JVML instructions is based on a formalization of the JVML type system developed in [20]. Their work covers a large subset of JVML but does not handle multithreaded programs. [4] and [29] provide detailed semantics for JVML but also do not handle multithreaded programs. Other formalizations of the JVML type system have focused on subroutines [24, 30, 11] and object initialization[19]. The type systems in [26, 23] statically verify that every method releases all the locks it acquires and no other locks. Currently, while this property holds for all well-typed Java programs, it does not hold for all well-typed JVML programs that pass the bytecode verification. JVMs use runtime checking to ensure this property. The type system in [26] is designed for JVML programs that are compiled from Java source programs, whereas the type system in [23] is more general and supports JVML programs produced from other sources well. We used ideas from [26] to track aliases in our system.

None of the previously proposed type systems for JVML handle data races, deadlocks, or encapsulation. The main contribution of our paper is that, to the best of our knowledge, this is the first type system for JVML that statically prevents data races, deadlocks, and encapsulation violations.

Our type system extension to JVML is based on a corresponding type system extension to Java that we previously developed called SafeJava [5, 6, 7, 9]. The SafeJava type system for preventing data races is most closely related to [2, 16, 22]. The SafeJava type system for enforcing object encapsulation uses a variant of ownership types [1, 12, 13, 25]. A detailed comparison of the SafeJava type system with related type systems and other approaches for preventing synchronization errors and encapsulation errors can be found in [5] and [6, 7, 9].

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[2] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

[3] Anindya Banerjee and David A. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages (POPL)*, January 2002.

[4] Peter Bertelsen. Dynamic semantics of Java bytecode. In *Workshop on Principles of Abstract Machines*, 1998.

[5] Chandrasekhar Boyapati. SafeJava: A unified type system for safe programming. Ph.D. thesis, Massachusetts Institute of Technology, February 2004.

[6] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[7] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, January 2003.

[8] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003.

[9] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[10] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[11] Robert O' Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Principles of Programming Languages (POPL)*, 1999.

[12] David G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[13] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[14] Cormac Flanagan and Martin Abadi. Object types against races. In *Conference on Concurrent Theory (CONCUR)*, August 1999.

[15] Cormac Flanagan and Martin Abadi. Types for safe locking. In *European Symposium on Programming (ESOP)*, March 1999.

[16] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)*, June 2000.

[17] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Principles of Programming Languages (POPL)*, January 1998.

[18] Stephen N. Freund. Type systems for object-oriented intermediate languages. Ph.D. thesis, Stanford University, 2000.

[19] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In *ACM Transactions on Programming Languages and Systems*, November 1999.

[20] Stephen N. Freund and John C. Mitchell. A formal framework for Java bytecode language and verifier. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[21] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[22] Dan Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation (TLDI)*, January 2003.

[23] Futoshi Iwama and Naoki Kobayashi. A new type system for JVM lock primitives. In *ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation(ASIA-PEPM)*, May 2002.

[24] Gerwin Klein and Martin Wildmoser. Verified bytecode subroutines. *Journal of Automated Reasoning*, 2003.

[25] Neel Krishnaswamy and Jonathan Aldrich. Permission based ownership: Encapsulating state in higher order typed languages. In *Programming Language Design and Implementation (PLDI)*, June 2005.

[26] Cosimo Laneve and Gaetano Bigliardi. A type system for JVM threads. In *The Third ACM SIGPLAN Workshop on Types in Compilation (TIC)*, September 2000.

[27] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[28] William Pugh. Fixing the Java memory model. In *ACM Java Grande Conference*, June 1999.

[29] Zhenyu Qian. A formal specification of Java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.

[30] Raymie Stata and Martin Abadi. A type system for Java bytecode subroutines. In *Principles of Programming Languages (POPL)*, January 1998.

Appendix

A. Rules for Type Checking

This section presents the SafeJVML type system. The SafeJVML grammar is shown in Figure 7. We first define a number of predicates used in the type system. These are based on similar predicates from [17, 16, 5]. We refer the reader to those papers for their precise formulation.

Predicate	Meaning
WFClasses(P)	There are no cycles in the class hierarchy
ClassOnce(P)	No class is declared twice in P
FieldsOnce(P)	No class contains two fields, declared or inherited, with same name
MethodsOncePerClass(P)	No class contains two methods with same name
OverridesOK(P)	Overriding methods have the same return type and parameter types as the methods being overridden. The `requires` and `locks` clauses of an overriding method must be superseded by those of the overridden methods
LockLevelsOK(P)	There are no cycles in the lock levels

We define the typing environment as follows. The typing environment contains the declared types of variables, the declared owner parameters, the declared constraints among owners, and the declared locks clause.

E ::= $\emptyset$ | E, owner f | E, *constr* | E, *locksclause*

We define a minimum lock level as follows:

l_{min} ::= ∞ | $cn.l$ | LUB($cn_1.l_1$ ... $cn_k.l_k$)

By definition, LUB($cn_1.l_1$ ... $cn_k.l_k$) $> cn_i.l_i \ \forall_{i=1..k}$. LUB(...) is not computed—it is an expressions used as such for type checking. The lock level ∞ denotes that the thread currently holds no locks.

We define the type system using the following judgments. We present the typing rules for these judgments after that, except that the typing rules for instructions in methods are given in Figure 11.

Judgment	Meaning
$\vdash P$	Program P is well defined
$P \vdash$ *defn*	*defn* is a well-formed class
$P; E \vdash_{\text{owner}} o$	o is an owner
$P; E \vdash$ *constr*	constraint *constr* is satisfied
$P; E \vdash t$	t is a well-formed type
$P; E \vdash t_1 <: t_2$	t_1 is a subtype of t_2
$P; E \vdash wf$	typing environment E is well-formed
$P \vdash$ *field* $\in c$	class c declares/inherits *field*
$P \vdash$ *meth* $\in c$	class c declares/inherits *meth*
$P \vdash_{\text{level}} cn.l$	$cn.l$ is a well-formed lock level
$P \vdash cn_1.l_1 < cn_2.l_2$	$cn_1.l_1$ is less than $cn_2.l_2$ in the partial order formed by lock levels
$P \vdash cn.l < l_{min}$	$cn.l$ is less than l_{min} in the partial order formed by lock levels
$P; E \vdash Lock(\hat{t}) = r$	r is the lock that protects an object of type $\hat{t}$
$P, E, F, S, LS, L_{min} \vdash M$	Given program P, environment E, types of local variables F, types of stack slots S, static lock set LS, and sequence of minimum lock levels L_{min}, executing the instructions in M does not cause a type error

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\textit{WFClasses(P)}\quad \textit{ClassOnce(P)}\quad \textit{FieldsOnce(P)}\quad \textit{MethodsOncePerClass(P)}\quad \textit{OverridesOK(P)}\quad \textit{LockLevelsOK(P)} \qquad P = defn_{1..n} \qquad P \vdash defn_i}{\vdash P}$$

$\boxed{P \vdash defn \in c}$

[CLASS]

$$\frac{E = \text{owner } f_{1..n}, f_i \succeq f_1, constr* \qquad P;E \vdash wf \qquad P;E \vdash c' \qquad P;E \vdash field_i \qquad P;E \vdash meth_i}{P \vdash \texttt{class}\ cn\langle f_{1..n}\rangle\ \texttt{extends}\ c'\ \texttt{where}\ constr*\ \{level*\ field*\ meth*\}}$$

$\boxed{P;E \vdash constr}$

[CONSTR ENV]

$$\frac{E = E_1, constr, E_2}{P;E \vdash constr}$$

[OWNER $\succeq$]

$$\frac{P;E \vdash e : cn\langle o_{1..n}\rangle}{P;E \vdash (o_1 \succeq e)}$$

[WORLD $\succeq$]

$$\frac{P;E \vdash_{\text{owner}} o \qquad P \vdash_{\text{level}} cn.l}{P;E \vdash (\texttt{world}{:}cn.l \succeq o)}$$

[REFL $\succeq$]

$$\frac{P;E \vdash_{\text{owner}} o}{P;E \vdash (o \succeq o)}$$

[TRANS $\succeq$]

$$\frac{P;E \vdash (o_3 \succeq o_2) \qquad P;E \vdash (o_2 \succeq o_1)}{P;E \vdash (o_3 \succeq o_1)}$$

$\boxed{P;E \vdash_{\text{owner}} o}$

[OWNER WORLD]

$$\frac{P \vdash_{\text{level}} cn.l}{P;E \vdash_{\text{owner}} \texttt{world}{:}cn.l}$$

[OWNER THIS]

$$\frac{E = E_1, c\ \texttt{this}, E_2}{P;E \vdash_{\text{owner}} \texttt{this}}$$

[OWNER THREAD]

$$\frac{}{P;E \vdash_{\text{owner}} \texttt{thisThread} \qquad P;E \vdash_{\text{owner}} \texttt{otherThread}}$$

[OWNER FORMAL]

$$\frac{E = E_1, \text{owner } f, E_2}{P;E \vdash_{\text{owner}} f}$$

$\boxed{P;E \vdash wf}$

[ENV $\emptyset$]

$$\frac{}{P;\emptyset \vdash wf}$$

[ENV OWNER]

$$\frac{f \notin \text{Dom}(E) \qquad P;E \vdash wf}{P;E, \text{owner } f \vdash wf}$$

[ENV CONSTR]

$$\frac{constr = (o' \succeq o) \lor constr = (o' \not\succeq o) \qquad P;E \vdash wf \qquad P;E \vdash_{\text{owner}} o, o' \qquad E' = E, constr \qquad \nexists x,y\ (P;E' \vdash y \succeq x) \land (P;E' \vdash y \not\succeq x)}{P;E, constr \vdash wf}$$

[ENV LOCKSCLAUSE]

$$\frac{locksclause = \texttt{locks}(\ldots) \qquad P;E \vdash wf}{P;E, locksclause \vdash wf}$$

$\boxed{P;E \vdash t}$

[TYPE INT]

$$\frac{}{P;E \vdash \texttt{int}}$$

[TYPE OBJ]

$$\frac{P;E \vdash_{\text{owner}} o}{P;E \vdash \texttt{Object}\langle o\rangle}$$

[TYPE C]

$$\frac{P \vdash \texttt{class}\ cn\langle f_{1..n}\rangle \ldots \texttt{where}\ constr* \ldots \qquad P;E \vdash_{\text{owner}} o_i \qquad P;E \vdash o_i \succeq o_1 \qquad P;E \vdash constr\,[o_1/f_1]..[o_n/f_n]}{P;E \vdash cn\langle o_{1..n}\rangle}$$

$\boxed{P;E \vdash t_1 <: t_2}$

[SUBTYPE C]

$$\frac{P;E \vdash cn\langle o_{1..n}\rangle \qquad P \vdash \texttt{class}\ cn\langle f_{1..n}\rangle\ \texttt{extends}\ cn'\langle f_1\ o*\rangle \ldots}{P;E \vdash cn\langle o_{1..n}\rangle <: cn'\langle f_1\ o*\rangle[o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE TRANS]

$$\frac{P;E \vdash t_1 <: t_2 \qquad P;E \vdash t_2 <: t_3}{P;E \vdash t_1 <: t_3}$$

[SUBTYPE REFL]

$$\frac{P;E \vdash t}{P;E \vdash t <: t}$$

$\boxed{P \vdash_{\text{level}} cn.l}$

[LEVEL]

$$\frac{P \vdash \texttt{class}\ cn\ldots\ \{\ldots \text{LockLevel } l \ldots\}}{P \vdash_{\text{level}} cn.l}$$

$\boxed{P \vdash cn_1.l_1 < cn_2.l_2}$

[LEVEL $<$]

$$\frac{P \vdash \texttt{class}\ cn_1 \ldots\ \{\ldots \text{LockLevel } l_1 < ..cn_2.l_2..\}}{P \vdash cn_1.l_1 < cn_2.l_2}$$

[LEVEL $>$]

$$\frac{P \vdash \texttt{class}\ cn_2 \ldots\ \{\ldots \text{LockLevel } l_2 > ..cn_1.l_1..\}}{P \vdash cn_1.l_1 < cn_2.l_2}$$

$\boxed{P \vdash cn.l < l_{min}}$

[LEVEL $<$ INFTY]

$$\frac{l_{min} = \infty \qquad P \vdash_{\text{level}} cn.l}{P \vdash cn.l < l_{min}}$$

[LEVEL $<$ LUB]

$$\frac{l_{min} = \text{LUB}(\ldots\ cn.l\ \ldots) \qquad P \vdash_{\text{level}} cn.l}{P \vdash cn.l < l_{min}}$$

[LEVEL $<$ CN.L]

$$\frac{l_{min} = cn'.l' \qquad P \vdash cn.l < cn'.l'}{P \vdash cn.l < l_{min}}$$

[LEVEL TRANS]

$$\frac{P \vdash cn'.l' < l_{min} \qquad P \vdash cn.l < cn'.l'}{P \vdash cn.l < l_{min}}$$

$\boxed{P;E \vdash Lock(\hat{t}) = r}$

[LOCK THISTHREAD]

$$\frac{P;E \vdash \textit{Type}[\hat{t}] = cn\langle \texttt{thisThread}\ o*\rangle}{P;E \vdash Lock(\hat{t}) = \texttt{thisThread}}$$

[LOCK OTHERTHREAD]

$$\frac{P;E \vdash \textit{Type}[\hat{t}] = cn\langle \texttt{otherThread}\ o*\rangle}{P;E \vdash Lock(\hat{t}) = \texttt{otherThread}}$$

[LOCK WORLD]

$$\frac{P;E \vdash \textit{Type}[\hat{t}] = cn\langle \texttt{world}{:}cn'.l'\ o*\rangle}{P;E \vdash Lock(\hat{t}) = \hat{t}}$$

[LOCK FORMAL]

$$\frac{P;E \vdash \textit{Type}[\hat{t}] = cn\langle o_{1..n}\rangle \qquad E = E_1, \text{owner } o_1, E_2}{P;E \vdash Lock(\hat{t}) = \text{L}(\hat{t})}$$

[LOCK THIS]

$$\frac{P;E \vdash \textit{Type}[\hat{t}] = cn\langle \texttt{this}\ o_{2..n}\rangle}{P;E \vdash Lock(\hat{t}) = Lock(F_1[0]}$$

$\boxed{P \vdash field \in c}$

[FIELD DECLARED]

$$\frac{P \vdash \texttt{class}\ cn\langle f_{1..n}\rangle \ldots \{\ldots\ field \ldots\}}{P \vdash field \in cn\langle f_{1..n}\rangle}$$

[FIELD INHERITED]

$$\frac{P \vdash field \in cn\langle f_{1..n}\rangle \qquad P \vdash \texttt{class}\ cn'\langle g_{1..m}\rangle\ \texttt{extends}\ cn\langle o_{1..n}\rangle \ldots}{P \vdash field\,[o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m}\rangle}$$

$\boxed{P \vdash meth \in c}$

[METHOD DECLARED]

$$\frac{P \vdash \texttt{class}\ cn\langle f_{1..n}\rangle \ldots \{\ldots\ meth \ldots\}}{P \vdash meth \in cn\langle f_{1..n}\rangle}$$

[METHOD INHERITED]

$$\frac{P \vdash meth \in cn\langle f_{1..n}\rangle \qquad P \vdash \texttt{class}\ cn'\langle g_{1..m}\rangle\ \texttt{extends}\ cn\langle o_{1..n}\rangle \ldots}{P \vdash meth\,[o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m}\rangle}$$

$\boxed{P, E, F, S, LS, L_{min} \vdash M}$

[VERIFICATION]

$$\frac{\begin{array}{c} M = \|\ cn\langle f_{1..n}\rangle, mn, \alpha, \gamma, \texttt{requires}(x_{0..k})\ \|_M \\ P \vdash \texttt{class}\ cn\langle f_{1..n}\rangle \ldots constr* \ldots \qquad P \vdash \gamma\ mn(\alpha)\ \texttt{requires}\,(x_{0..k})\ \texttt{locks}\,(cn'.l*) \ldots \in cn\langle f_{1..n}\rangle \\ E = E_1, \text{owner } f_{1..n}, constr*, \texttt{locks}(cn'.l*) \qquad E \vdash wf \\ P;E \vdash F_0[0 \to cn\langle f_{1..n}\rangle, 1..|\alpha| \to \alpha] <: F_1 \qquad S_1 = \epsilon \qquad \forall x_i \in x_{0..k}\ P;E \vdash x_i : t' \\ \forall (x_i : t') \in x_{0..k}\ (F_1[i] = t'_{-i} \land Lock(t'_{-i}) \in LS_1) \qquad L_{min_1} = \text{LUB}(cn'.l*) \\ \forall i \in Dom(M)\, P, E, F, S, LS, L_{min}, i \vdash M \end{array}}{P, E, F, S, LS, L_{min}, i \vdash M}$$

[INFERENCE]

$$\frac{\exists F, S, LS, L_{min}.\ \ P, E, F, S, LS, L_{min} \vdash M}{P, E \vdash M}$$

The above rules define when a JVML program is well-typed. We also define that a heap is well-typed if every record in the heap is well-typed and the runtime state is consistent with the static type information. The function $type(h, v)$ used below is defined in Figure 9.

$\boxed{[P, h \vdash o : t]}$

[HEAP]

$$\frac{h[o] = \langle\ fd_i = v_i, g_j = w_j, \text{level}{=}cn'.l\rangle^{i \in \{1..m\}, j \in \{1..n\}}_{cn\langle w_{1..n}\rangle} \qquad \forall i \in \{1..m\}.P \vdash (t_i\ fd_i) \in cn\langle f_{1..n}\rangle \qquad \forall i \in \{1..m\}.P \vdash type(h, v_i) <: t_i}{P, h \vdash o : cn\langle w_{1..n}\rangle}$$