# A Formal Specification of Java™ Class Loading

Zhenyu Qian, Allen Goldberg*, and Alessandro Coglio

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304

July 21, 2000

**Abstract.** The Java Virtual Machine (JVM) has a novel and powerful mechanism to support lazy, dynamic class loading according to user-definable policies. Class loading directly impacts type safety, on which the security of Java applications is based. Conceptual bugs in the loading mechanism were found in earlier versions of the JVM that lead to type violations. A deeper understanding of the class loading mechanism, through such means as formal analysis, will improve our confidence that no additional bugs are present.

The work presented in this paper provides a formal specification of (the relevant aspects of) class loading in the JVM and proves its type safety. Our approach to proving type safety is different from the usual ones since classes are dynamically loaded and full type information may not be statically available. In addition, we propose an improvement in the interaction between class loading and bytecode verification, which is cleaner and enables lazier loading.

## 1 Introduction

The Java Virtual Machine (JVM) has a novel and powerful class loading mechanism. Class loading is the process of obtaining a representation of a class (declaration), called a *class file*, and installing that representation within the JVM. The JVM allows lazy, dynamic loading of classes, user-definable loading policies, and a form of name space separation using loaders. According to both the Java and JVM specifications [12,15] distinct loaded classes may have the same name, and within an executing JVM each loaded class is identified by its name plus the class loader that has loaded it.

One of the key properties of the JVM, from both a security and software engineering perspective, is *type safety*. If the JVM confuses distinct classes, type safety problems can result. Ensuring type safety requires a fairly sophisticated mechanism. The reason is that on one hand it is impossible to determine, prior to execution, the actual loader of a class because loaders can delegate class loading to each other according to the program logic of user-written code. On the other hand, loading a class before it is required for execution is undesirable. Saraswat first publicly reported [17] name spoofing problems due to deficiencies in the mechanisms employed by earlier versions of the JVM.

* A. Goldberg's current affiliation and address: Shoulders Corp., 800 West El Camino Real, Mountain View, CA 94040.

The Sun solution [14] to such problems employs a constraint mechanism. Constraints on the disambiguation of names to ensure type safety are posted when necessary. As classes are loaded the constraints are checked.

The main result of this paper is to provide formal arguments that the Sun approach is sufficient to prove type safety. We formalize the operational semantics of a simplified JVM that uses this approach. The operational semantics models class loading, resolution, bytecode verification, and execution of some selected instructions.

Our model of the JVM departs from Sun's regarding how the bytecode verifier checks subtype relationships. Sun's bytecode verifier performs the checks by loading certain referenced classes, while our verifier uniformly posts constraints. These subtype constraints are checked when classes are loaded, analogously to the constraints mentioned above. The advantage of our approach is lazier class loading and clearer interaction between verification and class loading.

Because a class loader is a runtime object, a reference to a class (in a class file) is just a name and cannot include the loader. Disambiguation of such a name is needed, but can only occur when the referenced class has been loaded. This implies that traditional approaches to proving type safety, in which a static type semantics is extracted from the source code and related to the dynamic semantics, cannot be applied. Instead, our type safety proof relates the dynamic semantics with static type information, currently loaded classes, and currently posted constraints (which express requirements on not-yet-loaded classes).

Our analysis led us to identify bugs in the current Sun implementation of the JVM, some of which relate to inadequacies in the JVM specification [15][1]. The bugs result from the failure of the bytecode verifier to properly disambiguate names. For a full description of these bugs see [4], where it is also shown how the bugs are avoided by having the verifier post subtype constraints.

The remainder of the paper is organized as follows. In the next section we present relevant JVM concepts. Section 3 gives an overview of our approach. Section 4 describes our formalization. Section 5 states the safety properties and gives proofs. Section 6 discusses related work, and section 7 states conclusions.

The current paper is a full version of the paper [16].

## 2 Background

### 2.1 Class objects

A class file is typically produced by compiling a Java class (declaration): the class file contains essentially the same information, except that the code of

[1] Some of these bugs were independently discovered and reported in [18].

each of its methods is compiled to *bytecode*, i.e. an assembly-like language that uses an operand stack and a register array, both local to the method. The registers are also referred to as *local variables*. The JVM executes bytecode.

The internal representations of classes in an executing JVM are objects of the system class named `Class`, and they are called *class objects*. We will use *Obj*, *ClLd* and *Cls* to denote the class objects for the system classes named `Object`, `ClassLoader` and `Class`. For convenience, we will not distinguish between a class and its corresponding class object in the informal discussion below.

### 2.2 Resolution and loading

Bytecode instructions use (fully qualified) names to reference classes. *Class resolution* is the process of replacing these symbolic references with (pointers to) actual class objects in the executing JVM. Resolution causes loading of the class and checking of the access control modifiers of the loaded class.

Bytecode instructions reference fields and methods by name, including the name of the class where they are expected to be declared. *Field* and *method resolution* is the process of replacing these symbolic references with (pointers to) actual fields and methods. It requires first resolving the class in which the field or method is declared, then checking the presence of the field or method in the resolved class, and finally checking its access control modifiers.

*(Class) loaders* are objects of subclasses of class *ClLd*. By overriding certain methods of class *ClLd*, user-defined class loaders can implement arbitrary loading policies.

Class *ClLd* contains a `defineClass` method. It is a `final` method and thus cannot be overridden in subclasses of class *ClLd*. This method takes a class file (in the form of a byte array) as argument, and returns a newly created class object, unless the class file has an invalid format – in which case an exception is thrown. The creation of the new class requires resolution of all its superclasses. If the `defineClass` method is invoked on a loader and a class object is returned, then the loader is called the *defining loader* of the resulting class.

Class *ClLd* contains a `loadClass` method, which may be overridden in subclasses. This method takes a class name (in the form of a string) as argument, and returns a class object (or throws an exception). The user's code in this method can implement arbitrary loading policies. Typically the code will fetch a class file in some way (e.g., from the local file system, or a network connection) and then invoke `defineClass` with the resulting class file as argument. However, user's code can delegate loading by calling `loadClass` upon another loader. If the `loadClass` method is invoked by the JVM on a loader and a class object is returned, then the loader is called an *initiating loader* of the resulting class. The defining loader of a loaded class is also regarded as an initiating loader of the class.

Therefore, a loaded class may have many initiating loaders but only one defining loader.

When a class name needs to be resolved within an executing class, the defining loader of the executing class is used as initiating loader for the class (name) to be resolved.

### 2.3 A simple example

Let us use an example to explain some basic concepts and issues regarding classes, loaders, and type safety.

```
public class C {               // Class $C$ with defining loader $l_1$.
  public void m() {
    (new D()).n(new T());      // Load classes $D$ and $T_1$ and
  }                            // pass a $T_1$ object as argument.
}
public class D {               // Class $D$ with defining loader $l_2$.
  public void n(T t) {         // Receive a $T_1$ object as argument.
    (new E()).k(t);            // Load class $E$ and
                               // pass the $T_1$ object as argument.
    int i = t.f;               // Load class $T_2$ and fail since $T_1 \neq T_2$!
  }
}
public class E {               // Class $E$ with defining loader $l_1$.
  public void k(T t) {         // Receive a $T_1$ object as argument.
    Object o = t.f;            // Operation succeeds.
  }
}
public class T {               // Class $T_1$ with defining loader $l_1$.
  Object f;
}
public class T {               // Class $T_2$ with defining loader $l_2$.
  int f;
}
```

Figure 1: A simple example

Assume that we have a subclass *MyLd* of class *ClLd*, two distinct objects $l_1$ and $l_2$ of class *MyLd*, and five classes $C$, $D$, $E$, $T_1$ and $T_2$, satisfying the following:

- Classes $C$, $D$, $E$, $T_1$ and $T_2$ have names `C`, `D`, `E`, `T` and `T`, respectively.
- Classes $C$, $E$ and $T_1$ have defining loader $l_1$, and classes $D$ and $T_2$ have defining loader $l_2$.

– Using loader $l_1$ as initiating loader for names `D` and `T` yields the classes $D$ and $T_1$, respectively, while using loader $l_2$ as initiating loader for names `E` and `T` yields classes $E$ and $T_2$, respectively. This means that $l_1$ delegates to $l_2$ the loading of a class named `D`, and that $l_2$ delegates to $l_1$ the loading of a class named `E`.

Figure 1 shows classes $C$, $D$, $E$, $T_1$ and $T_2$. We consider execution starting with the `m()` method. Initially, only class $C$ is loaded. The others are loaded when they are needed, i.e. at their first active use (see below)[2].

Starting at the `m()` method, the JVM first resolves names `D` and `T`, loading classes $D$ and $T_1$ using $l_1$ as initiating loader for names `D` and `T`, respectively. Then it creates objects of these classes. The `n(T t)` method is then invoked with an object of class $T_1$ as argument.

Class $D$ has defining loader $l_2$. Within the `n(T t)` method, name `E` is resolved, loading class $E$ using $l_2$ as initiating loader for name `E`. Then the `k(T t)` method is invoked with the object of class $T_1$ as argument.

Class $E$ has defining loader $l_1$. Within the `k(T t)` method, the execution of the field access `t.f` causes resolution of class name `T`, using $l_1$ as initiating loader. The resulting class is $T_1$. The field access succeeds because the resolved class of the formal parameter `t` is the same as that of the actual value it holds.

Returning back to the `n(T t)` method, the execution of the field access `t.f` causes resolution of class name `T`, using $l_2$ as initiating loader. The resulting class is $T_2$. Now execution will throw an exception (and prevent field access) because the JVM detects that the class $T_1$ of the actual value in the formal parameter `t` does not match the class $T_2$ of the formal parameter `t`.

We observe that in the execution just described no exception is thrown until the field access `t.f` in the `n(T t)` method is attempted. The reason is that the JVM cannot check whether the class $T_2$ of the formal parameter `t` matches the class $T_1$ of the value it holds until the class $T_2$ is loaded. In fact, the JVM would not throw an exception if the field access `t.f` in the `n(T t)` method were not present.

The usual approach to proving type safety is to show that during execution values stored in variables always conform to the types statically assigned to these variables. In our example, this means showing that, after the `n(T t)` method is invoked, the $T_1$ object passed as argument conforms to "the type statically assigned to formal parameter `t` of `n(T t)`". However, the JVM cannot determine what the class the name `T` denotes, until the name `T` is actually resolved in class $D$. In fact, if the class were loaded at the time when the `n(T t)` method is invoked, the JVM would be able to detect that the value does not conform to the loaded class.

The bug reported by Saraswat [17] led to type violations in earlier versions of the JVM. In reference to the example in Figure 1, the earlier versions of

[2] The JVM specification allows loading of a class to happen at any time no later than its first active use. In this example, we assume that loading of a class happens exactly at its first active use.

the JVM did not check the equality of classes $T_1$ and $T_2$ when executing the field access `t.f` in the `n(T t)` method. The result of such field access was unpredictable, since it operated on an object of the wrong type.

The Sun solution to Saraswat's bug, which checks the equality of classes $T_1$ and $T_2$ when executing the field access `t.f` in the `n(T t)` method in Figure 1, introduce two internal data structures into the JVM called *loaded class cache* and *loading constraints* [14].

## 2.4 Loaded class caches and loading constraints

The JVM resolves referenced class names to loaded classes. The results of the loading process are stored in a JVM data structure called the *loaded class cache*. The loaded class cache maps an initiating loader and a class name to a loaded class. Recall that a reference to a class is resolved by using the defining loader of the current class as the initiating loader of the referenced class; so the loaded class cache records how a class name is disambiguated.

When a class name needs to be resolved, the JVM, before loading with a given initiating loader, checks the loaded class cache. If a class has been loaded for that initiating loader and that class name, the JVM returns the already loaded class as the result. Thus resolution will always give consistent results. If the loaded class cache has no entry for the initiating loader and class name then loading is carried out. If a class is returned, the loaded class cache is updated with a new entry for the just loaded class. If instead an exception is thrown, resolution fails.

In the sequel, we call a pair $\langle l, cn\rangle$ of an initiating loader $l$ and a class name $cn$ a *loading request*.

A loaded class cache is a finite map from loading requests to loaded classes:

$$\{\langle l_1, cn_1\rangle \mapsto c_1, \cdots, \langle l_n, cn_n\rangle \mapsto c_n\}.$$

A *loading constraint* is a triple $\langle l, l', cn\rangle$ consisting of two loaders $l$ and $l'$ and a class name $cn$. A loading constraint as above expresses the requirement that using $l$ and $l'$ as the initiating loaders for name $cn$, must yield the same (loaded) class if they both succeed. Loading constraints are generated by the JVM when fields and methods are resolved. Such constraints enforce that classes exchanging objects (through field access and method invocation) agree on the actual classes (and not only the names) of the exchanged objects.

An executing JVM maintains a loaded class cache and a set of loading constraints. These two data structures are checked for mutual consistency whenever either one is updated. Whenever an update causes a violation of a loading constraint w.r.t. the loaded class cache, an exception is thrown, thus causing a failure of the operation that triggered the update.

In the example in Figure 1, a loading constraint $\langle l_1, l_2, \mathtt{T}\rangle$ is generated when the `n(T t)` method is resolved (from $C$) during execution of the expression `(new D()).n(new T())`. The loader $l_1$ is the defining loader for class

$C$, which is used as an initiating loader for name `T` within the `m()` method. The loader $l_2$ is the defining loader for class $D$, which will be used as an initiating loader for name `T` within the `n(T t)` method, when `T` is resolved from $D$. Class $T_1$ is loaded for the loading request $\langle l_1, \texttt{T}\rangle$ in the execution of the instruction `new T()`. The loading constraint is checked only when class $T_2$ is loaded for the loading request $\langle l_2, \texttt{T}\rangle$, during the execution of the field access `t.f` within the `n(T t)` method.

## 2.5 Bytecode verification

Type-safe execution requires that each instruction operates on data with appropriate types. In order to reduce the number of runtime checks and thus to improve efficiency, most type-safety requirements are checked statically. In the JVM, the code of each loaded class is verified prior to execution of any of its methods, by a JVM component called the *bytecode verifier*. Bytecode verification should assign types to operand stack elements and local variables for each instruction, consistently with the types required by the instructions in the method. However, in order to avoid premature loading, the bytecode verifier only assigns class names, instead of class types, to local variables and operand stack elements. Therefore, the bytecode verifier only makes "partial" checks on class types; such checks are in fact complemented by the loading constraint mechanisms at runtime.

The following is the bytecode of the `m()`, `n(T t)` and `k(T t)` methods in Figure 1. The comments on the right give the class names assigned by the bytecode verifier to some elements in the operand stack at some program points.

```
method m()
 p1: new(D)
 p2: new(T)                    //  [···,D]
 p3: invokevirtual(D,n,T,void) //  [···,D,T]
    ...
method n(T)
    ...
 r1: getfield(T,f,int)         //  [···,T]
 r2: ...                       //  [···,int]
    ...
method k(T)
    ...
 q1: getfield(T,f,Object)      //  [···,T]
 q2: ...                       //  [···,Object]
    ...
```

Intuitively, the instruction `new(D)` creates a new object of class $D$ and pushes (a reference to) it onto the operand stack[3]. Thus, bytecode verification assigns the name `D` to the top element in the operand stack at program point `p2` after that instruction. The instruction `new(T)` at program point `p2` works in a similar way.

The instruction `invokevirtual(D,n,T,void)` dynamically selects the `n(T)` method based on the class of the second top object on the operand stack, and invokes the method with the top element in the operand stack as argument. The instruction contains a class name `D` indicating the class in which the method is to be found, and a class name `T` indicating the class of the argument. Bytecode verification uses this information to check the consistency of the class names assigned to the top stack positions. In this case, it checks whether the class names `D` and `T` are the ones assigned to the top two elements in the operand stack. Note that this static consistency check is not sufficient to guarantee type safety: this is the reason why the executing JVM will generate a loading constraint $\langle l_1, l_2, \mathtt{T}\rangle$, as explained in Section 2.4.

The instruction `getfield(T,f,int)` fetches the `int` value of the field `f` in the top object of the operand stack and pushes it onto the operand stack. The instruction contains a class name `T` indicating the class in which the field is to be found. Again bytecode verification uses this information to check the consistency of the class name assigned to the top stack position. In this case, it checks whether class name `T` is the one assigned to the top element in the operand stack. At the next program point `r2`, bytecode verification assigns type `int` to the top element of the operand stack.

Bytecode verification analyzes `getfield(T,f,Object)` in a similar way as `getfield(T,f,int)`. The only difference is that the class name `Object` is assigned to the top element of the operand stack at program point `q2`.

### 2.6 Another example

Let us now consider an example involving subtypes. This example provides some motivation for the subtype constraints that will be introduced and explained in the next sections.

We assume that we have a subclass *MyLd* of class *ClLd*, two distinct objects $l_1$ and $l_2$ of class *MyLd*, and five classes $C$, $D$, $T_1$, $T_2$ and $S$, satisfying the following:

- Classes $C$, $D$, $T_1$, $T_2$ and $S$ have names `C`, `D`, `T`, `T` and `S` respectively.
- Classes $C$, $D$ and $T_1$ have defining loader $l_1$, and classes $T_2$ and $S$ have defining loader $l_2$.
- Using loader $l_1$ as initiating loader for names `D`, `T` and `S` yields the classes $D$, $T_1$ and $S$, respectively, while using loader $l_2$ as initiating loader for

[3] For simplicity, we ignore the issue of object initialization in this example. In other words, we assume that the bytecode instruction `new` creates a fully initialized object.

name `T` yields class $T_2$, respectively. This means that $l_1$ delegates to $l_2$ the loading of a class named `S`.

Note that $S$ is a subclass of $T_2$, and not of $T_1$, because resolving name `T` with $l_2$ (which is the defining loader of $S$) as initiating loader yields $T_2$.

```
public class C {              // Class $C$ with defining loader $l_1$.
  public void m() {
    (new D()).n(new S());
  }
}
public class D {              // Class $D$ with defining loader $l_1$.
  public void n(T t) {
    Object o = t.f;
  }
}
public class T {              // Class $T_1$ with defining loader $l_1$.
  Object f;
}
public class T {              // Class $T_2$ with defining loader $l_2$.
  int f;
}
public class S extends T {    // Class $S$ with defining loader $l_2$.
}
```

Figure 2: Another example

Figure 2 shows classes $C$, $D$, $T_1$, $T_2$ and $S$. We assume that initially, only class $C$ is loaded, and consider execution starting with the `m()` method.

Based on analogies with the previous example, one might expect execution to happen as follows. First, classes $D$ and $S$ are loaded. According to the JVM specification, when a class is loaded, all its superclasses are also loaded. Therefore, when $S$ is loaded $T_2$ is also loaded.

Next, an object of class $D$ and one of class $S$ are created, and method `n(T t)` is invoked upon the $D$ object, passing the $S$ object as argument. The field access `t.f` causes resolution of name `T` from class $D$, which results in class $T_1$. At this point, field access should fail because $S$ is not a subclass of $T_1$.

However, that is not quite what happens in the JVM. Before method `m()` is executed, it must go through bytecode verification. Let us examine the bytecode for this method, together with the class names assigned by the bytecode verifier to some elements of the operand stack:

```
method m()
```

```
p1: new(D)
p2: new(S)                    //  [···,D]
p3: invokevirtual(D,n,T,void) //  [···,D,S]
    ...
```

The difference between this and the `m()` method in the previous example is that the name `T` assigned to the top element in the operand stack at program point `p3` is now replaced with the name `S`. Since the `invokevirtual` instruction references name `T`, the JVM specification actually requires bytecode verification to resolve both names `T` and `S` from class $C$ (i.e., using $l_1$ as initiating loader), and to check that the appropriate subclass relationship holds.

The check fails, because the loading request $\langle l_1, \mathtt{T}\rangle$ yields $T_1$, which is not a superclass of $S$. Therefore, bytecode verification throws an exception, and method `m()` is not executed at all.

Of course, the same would happen even if the field access `t.f` were not present in method `n(T t)`. In fact, when class $C$ is being verified, class $D$ has not been loaded.

This suggests that resolving class names during bytecode verification in order to check subclass relationships, does not lead to the laziest possible loading strategy. In the next sections, we will show how subtype constraints allow lazier loading, and how the behavior of the JVM in this example would match what one would probably expect.

## 3 Overview of Our Approach

The main objective of our work is to raise the assurance that the JVM as specified and implemented by Sun is safe. Because the mechanisms that enforce safety in the JVM are inherently complex, it is not straightforward to assess their correctness, as demonstrated by the discovery of bugs that lead to type safety violations [17,18,4]. In order to raise assurance, we construct a formal specification, which is consistent with the JVM English specification and the Sun implementation, and we prove results about that specification. Actually, our specification intentionally differs from Sun's in one aspect, namely the use of subtype constraints for bytecode verification, which enable a cleaner design and lazier loading policy.

We present an operational semantics for an abstraction of the JVM and prove a type safety result. The operational semantics differs from usual ones (e.g. [1]) that just specify state transitions for the JVM bytecode instructions. Our semantics includes transitions for "macro operations," e.g. resolution and class loading, that are performed by the "core" of the JVM. The core of the JVM maintains data structures, such as a loaded class cache, which are updated by these macro operations. The interaction between the core of the machine and user code is complex: there is a mutual recursion between user code executing in a thread and the core machine. The state transition formalism we use reflects this with the use of "nested" rules.

We start by introducing the formal entities (sets, operations, etc.) used in our formalization. We then define states and transitions for a state machine representing an abstraction of the JVM. Transitions fire when certain conditions are satisfied. Some conditions correspond to runtime checks actually performed by the executing JVM; their failure raises exceptions in the JVM. Other conditions serve to ensure type-safe operations in our formalization; there are no corresponding runtime checks in the JVM. If these type safety conditions are not satisfied, then the behavior of the JVM is undefined[4]. We finally introduce a notion of *validity* of states, and present theorems stating that transitions from valid states lead to valid states, and that transitions from valid states are always possible unless either there is no more code to execute or a condition corresponding to an actual check in the JVM is violated. In other words, in a valid state, the failure of a type safety condition alone never causes the machine to halt. This means that if the JVM starts in a state corresponding to a valid state in our formalization, each operation in the JVM will be always type-safe, and no checks corresponding to the type safety conditions are needed.

Our formalization makes many simplifications to the JVM. Most important is that we ignore concurrency and exceptions. One could easily imagine a concurrency bug leading to inconsistent data structures; such bugs are not addressed by our specification. The JVM, when presented with a program that is inconsistent, throws an exception to report the error. For example, if a loading constraint is violated, a linking exception is thrown. Since we do not model exceptions, in our formalization the machine simply halts. We also do not treat static fields or methods, primitive types, interfaces, arrays, object initialization, subroutines, and field or method modifiers. We believe these features are orthogonal to the issues raised by class loading. We treat a few bytecode instructions, namely those to access fields, call methods, return results, create new objects, and load/store from/to local variables. In a strong sense the essence of the Sun approach is captured by our formalization.

Our formalization includes bytecode verification. Bytecode verification assigns class names to memory locations (in the operand stack and for all local variables) at each program point of a method based on the instructions in the method, as shown in Section 2.5. For example, a `getfield` instruction requires a certain class name (for the target object) to be assigned to the top of the stack at the program point where the instruction is, and requires

[4] In practice, the behavior of an actual JVM when such conditions are not satisfied is determined by the implementation of the machine. Knowledge of the implementation can be maliciously exploited to compromise the security of the JVM, if type safety can be circumvented. For example, typical implementations access fields through offsets added to the address of an object. In such implementations, a type-unsafe operation could access arbitrary data within objects, regardless of field layout (see example in the previous section).

another class name (for the field value) to be assigned at the top of the stack at the next program point.

In our formalization the state of an executing JVM contains a global state consisting of a loaded class cache, a set of loading and subtype constraints and a heap for storing objects. In addition, the state includes a component that models the execution stack of a single thread (as per the restriction above) storing frames. Each frame is a tuple consisting of a class, a method of the class, a program point within the method, and a state of the local memory (the operand stack and all local variables).

We identify a subset of execution states called *valid states* that satisfy certain constraints. Many of the constraints are straightforward. For example, one of such constraints requires that all classes in a frame be in (the range of) the loaded class cache. However, the key constraint is the *conformance* condition, which relies on loading and subtype constraints. The problem is that some classes may never get loaded or only get loaded at run time and that the bytecode verifier assigns only type names, not full type information, to memory locations, to avoid premature loading. Loading and subtype constraints are used to state requirements for not-yet-loaded classes denoted by class names. The conformance relation takes into account not only loaded classes, but also loading and subtype constraints.

To further explain the above point, consider again the example in Figure 1. The type of the object passed to the `n(T t)` method is $T_1$. Until name `T` is resolved from $D$, the formal parameter `t` of method `n(T t)` has no full type assigned to it (only name `T`). Therefore, we cannot state that the passed object matches the type of the formal parameter, simply because there is no full type information. However, upon resolution of method `n(T t)` from $C$, the loading constraint $\langle l_1, l_2, \mathtt{T}\rangle$ is introduced. Such a constraint requires equality of the class loaded by $l_1$ for name `T` (i.e., $T_1$) and the class that will be possibly loaded by $l_2$ for the same name `T`. By taking this constraint into account, the conformance relation captures the fact that the already loaded type $T_1$ "matches" the not-yet-loaded type for the formal parameter of the method. If $T_2$ were never loaded (e.g., if the assignment `i = t.f` were not present in the method), the constraint would never be violated and the conformance relation would still hold.

Let us now explain subtype constraints. Suppose that, during bytecode verification, a class name $cn$ is required (e.g., by a $\mathtt{getfield}(cn, \ldots)$ instruction) at the top of the stack, where a name $cn'$, with $cn' \neq cn$, has instead been assigned. The field access is correct as long as $cn'$ is a subclass of $cn$. More precisely, since $cn$ and $cn'$ are just names, the requirement is that if and when the loading requests $\langle l, cn'\rangle$ and $\langle l, cn\rangle$, where $l$ is the defining loader of the class whose method is being verified, yield two loaded classes $C'$ and $C$ (respectively), then $C'$ must be a subclass of $C$. According to Sun's specification and implementation, and as described in the previous section for the example in Figure 2, the bytecode verifier checks this subtype relation eagerly, by resolving names $cn$ and $cn'$.

In our formalization, we check subtype relations lazily: the bytecode verifier just posts subtype constraints of the form $\langle l, cn, cn' \rangle$. Such a constraint expresses exactly the requirement that if and when the loading requests $\langle l, cn' \rangle$ and $\langle l, cn \rangle$ yield two classes $C'$ and $C$, then $C'$ must be a subclass of $C$. These subtype constraints are handled analogously to the loading constraints. Each time a class is loaded, subtype constraints are checked for violation. Each time a new subtype constraint is introduced, it is checked for violation, too. In other words, loading constraints, subtype constraints, and the loaded class cache are constantly maintained in a mutually consistent state. The primary advantage of this approach is lazier loading, because no class needs to be loaded for verification purposes. Another advantage is that the interaction between the bytecode verifier and the rest of the core JVM is simpler and clearer: the verifier is in fact just a functional component that takes a class as argument and returns a yes/no answer plus a set of subtype constraints as result.

To further illustrate subtype constraints, let us consider the example in Figure 2 using subtype constraints. Again, we start with only class $C$ loaded. Before method `m()` is executed, it must be verified. Bytecode verification successfully verifies the method and posts the subtype constraint $\langle l_1, \mathtt{S}, \mathtt{T} \rangle$ (without loading any class). Method `m()` starts executing. Classes $D$ and $S$ are loaded. Since $T_1$ has not been loaded yet, the subtype constraint $\langle l_1, \mathtt{S}, \mathtt{T} \rangle$ is not violated yet. The newly created $S$ object is passed as an argument to method `n(T t)` invoked upon the newly created $D$ object. When the field access `t.f` is about to be executed, class name `T` is now resolved with $l_1$ (which is the defining loader of $D$) as initiating loader. This yields class $T_1$; since it is not a superclass of $S$, now the subtype constraint $\langle l_1, \mathtt{S}, \mathtt{T} \rangle$ is violated. Therefore, an exception is thrown and field access is prevented.

## 4 Formalization

### 4.1 Mathematical notations

We use $\mathbf{B}$ to denote the set of Boolean values, $\mathbf{N}$ the set of natural numbers starting from 0, and $\mathbf{C}$ the set of all (Unicode) characters.

We use the usual notations for sets, in particular, $\uplus$ for disjoint union and $\mathcal{P}_\omega(\_)$ for the set of all finite subsets. A set $A$ is called *countable* iff there is a bijective function mapping $A$ onto the set $\mathbf{N}$ of natural numbers.

For a set $A$, we define

$$\begin{aligned}
&A^* = \{[a_0, \ldots, a_{k-1}] \mid k \geq 0, a_0, \ldots, a_{k-1} \in A\},\\
&A^+ = A^* - \{[\,]\},\\
&|[a_0, \ldots, a_{k-1}]| = k,\\
&[a_0, \ldots, a_{k-1}] + a = [a_0, \ldots, a_{k-1}, a],\\
&[a_0, \ldots, a_{k-1}]|_j = a_j \text{ for } 0 \leq j \leq k-1,\\
&[a_0, \ldots, a_{k-1}][j \mapsto a] = [a_0, \ldots, a_{j-1}, a, a_{j+1}, \ldots, a_{k-1}] \text{ for } 0 \leq j \leq k-1.
\end{aligned}$$

We write $[a]^k$ for the sequence consisting of $k \geq 0$ occurrences of $a$.

For sets $A$ and $B$, $A \to B$ denotes the set of all (total) functions from $A$ to $B$, and $A \xrightarrow{\mathrm{f}} B$ the set of all finite functions from $A$ to $B$. We use $\mathcal{D}(f)$ and $\mathcal{R}(f)$ to denote the domain and range of a function $f$. Following the tradition, we also write $f : A \to B$ and $f : A \xrightarrow{\mathrm{f}} B$ for $f \in A \to B$ and $f \in A \xrightarrow{\mathrm{f}} B$, respectively.

We use the notation $\{a \mapsto b \mid \alpha(a, b)\}$ to denote the function $f$ defined by $f(a) = b$ for all $a, b$ such that $\alpha(a, b)$.

Let $f : A \xrightarrow{\mathrm{f}} B$. The function $f[a \mapsto b] : A \cup \{a\} \xrightarrow{\mathrm{f}} B$ is defined by

$$(f[a \mapsto b])(a') = f(a')$$

for $a' \in A - \{a\}$ and $(f[a \mapsto b])(a) = b$. The function $f\{a \mapsto b\} : A \cup \{a\} \xrightarrow{\mathrm{f}} B$ is defined by $(f\{a \mapsto b\})(a') = f(a')$ for $a' \in A$ and $(f\{a \mapsto b\})(a) = b$ if $a \notin A$. These two operations differ only in the case $a \in A$: the former updates the given finite function $f$ for the input $a$, whereas the latter does not.

### 4.2 Names and instructions

Class, field, and method names are formalized by three sets,

$$\mathit{CN} \subset \mathbf{C}^+, \qquad \mathit{FN} \subset \mathbf{C}^+, \qquad \mathit{MN} \subset \mathbf{C}^+.$$

We will make explicit use of the following system class and method names:

$$\mathtt{Obj}, \mathtt{Cls}, \mathtt{ClLd}, \mathtt{Str} \in \mathit{CN}, \qquad \mathtt{ldCl}, \mathtt{dfCl} \in \mathit{MN},$$

standing for `java.lang.Object`, `java.lang.Class`, `java.lang.ClassLoader`, `java.lang.String`, `loadClass`, and `defineClass`, respectively.

The JVM instructions we consider are captured by the set

$$\begin{aligned} I = \; & \{\mathsf{getfield}(cn, fn, cn_0) \mid \langle cn, fn, cn_0\rangle \in \mathit{CN} \times \mathit{FN} \times \mathit{CN}\} \cup \\ & \{\mathsf{putfield}(cn, fn, cn_0) \mid \langle cn, fn, cn_0\rangle \in \mathit{CN} \times \mathit{FN} \times \mathit{CN}\} \cup \\ & \{\mathsf{invokevirtual}(cn, mn, cn_1, cn_0) \mid \langle cn, mn, cn_1, cn_0\rangle \in \mathit{CN} \times \mathit{MN} \times \mathit{CN} \times \mathit{CN}\} \cup \\ & \{\mathsf{new}(cn) \mid cn \in \mathit{CN}\} \cup \\ & \{\mathsf{areturn}\}. \end{aligned}$$

### 4.3 Fields and methods

We introduce a set of fields with two selector functions; one yields the field name and the second the descriptor (just a class name since the only types we consider are classes):

$$F, \qquad \mathit{nm} : F \to \mathit{FN}, \qquad \mathit{ds} : F \to \mathit{CN}.$$

We introduce a set of methods with four selector functions; one yields the method name, one its argument type, one its result type names and one its code (i.e., a finite, non-empty sequence of instructions):

$$M, \quad nm : M \to MN, \quad at : M \to CN, \quad rt : M \to CN, \quad cd : M \to I^{+},$$

Each $m \in M$ satisfies that

$$\forall j \in \mathbf{N}.\ 0 \leq j < |cd(m)| \ \Rightarrow\ (cd(m)|_j = \mathsf{areturn} \ \Leftrightarrow\ j = |cd(m)| - 1).$$

Because there are no branch instructions defined in our simplified instruction set, code is straight line. The conditions ensure that there is no unreachable code, and that the last instruction is the unique `areturn` instruction of the method.

Program points are indices of instructions within code. They are elements of the set

$$P = \mathbf{N}.$$

## 4.4 Objects and heaps

In the JVM, each object is identified by a reference (i.e., pointer), which dereferences to the state of the object. Such a state consists of an immutable and a mutable part. The immutable part is specific to the execution of a JVM but once an object is introduced into the runtime state of a JVM its immutable properties are invariant over subsequent execution states. On the other hand, the values of fields can change during execution.

We introduce a countable set

$$O$$

consisting of all possible objects. Each element of $O$ is an object reference plus its immutable state.[5]

Since $O$ does not include the (special) `null` reference, we introduce a set for values in fields and operand stacks:

$$V = O \uplus \{\mathsf{null}\}.$$

In the JVM each loaded class is represented by an instance of a (meta) class `Class`. All possible instances of class `Class` are collected in a countable set

$$C \subset O.$$

Every object has a type, i.e., an element of $C$, given by the selector function

$$cl : O \to C.$$

[5] Collapsing immutable states into references allows a lighter notation.

We assume that there are four system classes

$$Obj, Cls, ClLd, Str \in C$$

that satisfy:

1. $C = \{o \in O \mid cl(o) = Cls\}$;
2. $\{o \in O \mid cl(o) = c\}$ is a countable set for each $c \in C$.

The elements $Obj$, $Cls$, $ClLd$ and $Str$ represent the class objects of the system classes named `Obj`, `Cls`, `ClLd` and `Str`, respectively. Condition 1 means that class objects are instances of the class represented by the class object $Cls$. The condition implies that $cl(Cls) = Cls$, meaning that the (class) object $Cls$ represents the class of itself. Condition 2 ensures that there are always enough instances of each class available in the formalization.

We introduce the countable set

$$S = \{o \in O \mid cl(o) = Str\}$$

which consists of all possible string objects, and a function

$$str : \mathbf{C}^* \to S,$$

yielding a `String` object for each sequence of characters.

The above definitions and properties are universal in the sense that they hold for any model of JVM execution. The subtype relation between classes is not universal in this sense. It depends on the supertype declarations that appear in the classes that are loaded and are recorded in the runtime state of a JVM execution.

A class loader is any object of a subclass of class $ClLd$. We cannot specify a universal predicate on $O$ that characterize class loaders because of the dependence on subclassing. At this stage, we only reserve the set

$$L = O - (C \cup S)$$

as the space of all potential loaders. The state transitions will ensure that every object from $L$ that is used as a loader is indeed a loader.

The immutable state of classes includes

$$nm : C \to CN, \quad fld : C \to \mathcal{P}_\omega(F), \quad mth : C \to \mathcal{P}_\omega(M),$$
$$sup : C \to CN \uplus \{\mathsf{nil}\}, \quad ld : C \to L,$$

giving the name of the class, the set of all fields declared in it, the set of all methods declared in it, its superclass, and its defining loader. We assume that they satisfy

1. $\forall o \in O.\ sup(o) = \mathsf{nil} \Leftrightarrow o = Obj$;
2. $\forall c \in C.\ \forall f_1, f_2 \in fld(c).\ nm(f_1) = nm(f_2) \ \Rightarrow\ f_1 = f_2$;

3. $\forall c \in C.\ \forall m_1, m_2 \in mth(c).$
   $\langle nm(m_1), at(m_1), rt(m_1)\rangle = \langle nm(m_2), at(m_2), rt(m_2)\rangle \ \Rightarrow\ m_1 = m_2.$

Condition (1) means that object $Obj$ is the only class that has no superclass. Condition (2) ensures that no two distinct fields have the same name. Condition (3) ensures that no two distinct methods have the same name and types. For convenience, we define a function $fnm : C \to \mathcal{P}_\omega(FN)$ by

$$fnm(c) = \{nm(f) \mid f \in fld(c)\}.$$

The mutable states of objects are records, which are formally modeled by finite functions from field names to values:

$$Rc = FN \xrightarrow{f} V.$$

The mutable states of objects are stored in heaps, which are formalized as finite functions from objects to records:

$$Hp = O \xrightarrow{f} Rc.$$

### 4.5 Loaded class cache

A loaded class cache is a finite function from $L \times CN$ to $C$:

$$LC = L \times CN \xrightarrow{f} C.$$

For $lc \in LC$, we call elements of $\mathcal{D}(lc)$ *loading requests*, and elements of $\mathcal{R}(lc)$ *loaded classes*.

Given $lc \in LC$, we define the *subclass* relation $\preceq_{lc} : C \times C \to \mathbf{B}$ as the least relation satisfying

$$c \preceq_{lc} c' \ \Leftrightarrow\ c = c' \ \vee\ (\langle ld(c), sup(c)\rangle \in \mathcal{D}(lc) \ \wedge\ lc(ld(c), sup(c)) \preceq_{lc} c')$$

Note that if $\langle \cdots, sup(c)\rangle \in \mathcal{D}(lc)$ then $sup(c) \neq \mathsf{nil}$, since $\mathsf{nil} \notin CN$. We write

$$c \prec_{lc} c' \ \Leftrightarrow\ c \neq c' \ \wedge\ c \preceq_{lc} c'.$$

The intuition behind the above definition is that if $\langle ld(c), sup(c)\rangle \in \mathcal{D}(lc)$, then $lc(ld(c), sup(c))$ is the direct superclass of class $c$, and that the defining loader $ld(c)$ of class $c$ is an initiating loader of the superclass.

The above definition immediately implies that given $lc \in LC$,

$$c \preceq_{lc} c' \ \wedge\ c \preceq_{lc} c'' \ \Rightarrow\ c' \preceq_{lc} c''.$$

We define an *upward closure* relation $clos : LC \to \mathbf{B}$ by

$$clos(lc) \ \Leftrightarrow\ (\forall c \in \mathcal{R}(lc).\ sup(c) \neq \mathsf{nil} \ \Rightarrow\ \langle ld(c), sup(c)\rangle \in \mathcal{D}(lc))$$

Intuitively, if $clos(lc)$ and $c \in \mathcal{R}(lc)$, then $\mathcal{R}(lc)$ contains all direct and indirect superclasses of $c$.

We define a *subclass circularity* relation $circ : LC \to \mathbf{B}$ by

$$circ(lc) \;\Leftrightarrow\; \exists c, c' \in C.\; c \preceq_{lc} c' \;\wedge\; c' \preceq_{lc} c \;\wedge\; c \neq c'.$$

Given $lc \in LC$, we define a function $allfnm_{lc} : C \to \mathcal{P}_\omega(FN)$ returning all field names (declared or inherited) of a class, by

$$allfnm_{lc}(c) = \bigcup_{c \preceq_{lc} c'} fnm(c')$$

Given $lc \in LC$, we define a function

$$fldSel_{lc} : \mathcal{R}(lc) \times FN \times CN \to (\mathcal{R}(lc) \times F) \uplus \{fail\}$$

by

$$\begin{array}{l}
c \preceq_{lc} c' \;\wedge\; f \in fld(c') \;\wedge\; \langle nm(f), ds(f)\rangle = \langle fn, cn_0\rangle \;\wedge \\
(\nexists c'' \in C.\; \exists f' \in fld(c'').\; c \preceq_{lc} c'' \prec_{lc} c' \wedge \langle nm(f), ds(f)\rangle = \langle nm(f'), ds(f')\rangle) \;\Rightarrow \\
\qquad fldSel_{lc}(c, fn, cn_0) = \langle c', f\rangle, \\
(\nexists c' \in C, f \in fld(c').\; c \preceq_{lc} c' \;\wedge\; \langle nm(f), ds(f)\rangle = \langle fn, cn_0\rangle) \;\Rightarrow \\
\qquad fldSel_{lc}(c, fn, cn_0) = fail.
\end{array}$$

If there is a superclass of class $c$ that declares the field $f$ with the given name $fn$ and descriptor $cn_0$, then $fldSel_{lc}(c, fn, cn_0) = \langle c', f\rangle$, where $c'$ is the least one among such superclasses.

Given $lc \in LC$, we define a function

$$mthSel_{lc} : C \times MN \times CN \times CN \to (\mathcal{R}(lc) \times M) \uplus \{fail\}$$

by

$$\begin{array}{l}
c \preceq_{lc} c' \;\wedge\; m \in mth(c') \;\wedge\; \langle nm(m), at(m), rt(m)\rangle = \langle mn, cn_1, cn_0\rangle \;\wedge \\
(\nexists c'' \in C.\; \exists m' \in mth(c''). \\
\quad c \preceq_{lc} c'' \prec_{lc} c' \;\wedge\; \langle nm(m), at(m), rt(m)\rangle = \langle nm(m'), at(m'), rt(m')\rangle) \;\Rightarrow \\
\qquad mthSel_{lc}(c, mn, cn_1, cn_0) = \langle c', m\rangle, \\
(\nexists c' \in C, m \in mth(c').\; c \preceq_{lc} c' \wedge \langle nm(m), at(m), rt(m)\rangle = \langle mn, cn_1, cn_0\rangle) \;\Rightarrow \\
\qquad mthSel_{lc}(c, mn, c_1, cn_0) = fail.
\end{array}$$

If there is a superclass of class $c$ that declares the method $m$ with the given signature name $mn$ and types $cn_1$ and $cn_0$, then $mthSel_{lc}(c, mn, cn_1, cn_0) = \langle c', m\rangle$, where $c'$ is the least one among such superclasses.

## 4.6 Bytecode verifier

For the restricted set of JVM instructions used in this paper, bytecode verification is very simple. In particular, since there are no branches, there is no need to merge types at a program point where control flow merges. Each local memory can be assigned with a single class name, and bytecode verification can be computed through the straight line of code.

The results of bytecode verification are elements in the following two sets:

$$\begin{aligned} ST &= (CN^*)^*, \\ SR &= \mathcal{P}_\omega(CN \times CN). \end{aligned}$$

The first result assigns a type (which is just a class names) to each stack position at each program point of the method. The second result is a set of pairs of the form $\langle cn, cn' \rangle$, called *subtype requirements.* These pairs are interpreted to mean that the class represented by the name $cn$ must be a subclass of that represented by the name $cn'$ for the typing of stack positions to be valid.

Instead of directly defining a bytecode verifier that computes stack entry types assignable to the operand stack and the subtype requirements, we give equations a bytecode verifier must satisfy. Figure 3 defines a relation $bcv : C \times M \times ST \times SR \to \mathbf{B}$ capturing equations on assignable stack entry types and corresponding subtype requirements for a given method of a class. The sequence $psr \in SR^*$ consists of sets of subtype requirements, each set for a program point. The auxiliary function $mksr : CN \times CN \to \mathcal{P}_\omega(CN \times CN)$ is used to construct singleton subtype requirement sets, as defined by

$$\begin{aligned} cn = cn' &\Rightarrow mksr(cn, cn') = \{\}, \\ cn \neq cn' &\Rightarrow mksr(cn, cn') = \{\langle cn, cn' \rangle\}. \end{aligned}$$

At program point 0, the equation $st|_0 = [nm(c), at(m)]$ requires that the operand stack should consist of two elements: the first is the `this` object, whose class name is $nm(c)$, and the second is an argument, whose class name is $at(m)$; the equation $psr|_0 = \{\}$ means that there are no subtype requirements here.

If program point $p$ is a getfield instruction, the equation $st|_p = s + t$ requires that that the operand stack should have an object at the top, from which a field value is obtained; the equation $st|_{p+1} = s + cn_0$ requires that the top of the stack hold a type of the field value; the equation $psr|_{p+1} = psr|_p \cup mksr(t, cn)$ requires that the class of the object at the stack top should be a subclass of the class required by the instruction. Note that if $t = cn$, then no subtype assumption is created here.

The situation for a putfield or invokevirtual instruction is analogous to the above. The operand stack for a putfield instruction is required to contain at least two objects; one is to be assigned to a field of the other. The operand stack for an invokevirtual instruction should contain at least two objects; one is the `this` object of the method and the other is the actual argument.

The equation for a new instruction is straigtforward.

The operand stack for an areturn instruction should contain at least one object as the result of the current method. The equation $sr = psr|_p \cup$

$$
\begin{array}{l}
bcv(c, m, st, sr) \;\Leftrightarrow \\
(\exists psr \in SR^*.\; |cd(m)| = |st| = |psr| \;\wedge\; st|_0 = [nm(c), at(m)] \;\wedge\; psr|_0 = \{\} \;\wedge \\
\quad (\forall p \in P.\; 0 \le p < |cd(m)| \;\Rightarrow \\
\qquad (cd(m)|_p = \mathsf{getfield}(cn, fn, cn_0) \Rightarrow \\
\qquad\quad \exists s \in CN^*.\; \exists t \in CN. \\
\qquad\quad st|_p = s + t \;\wedge\; st|_{p+1} = s + cn_0 \;\wedge\; psr|_{p+1} = psr|_p \cup mksr(t, cn)) \;\wedge \\
\qquad (cd(m)|_p = \mathsf{putfield}(cn, fn, cn_0) \Rightarrow \\
\qquad\quad \exists s \in CN^*.\; \exists t, t' \in CN. \\
\qquad\quad st|_p = s + t + t' \;\wedge\; st|_{p+1} = s \;\wedge \\
\qquad\quad psr|_{p+1} = psr|_p \cup mksr(t, cn) \cup mksr(t', cn_0)) \;\wedge \\
\qquad (cd(m)|_p = \mathsf{invokevirtual}(cn, mn, cn_1, cn_0) \Rightarrow \\
\qquad\quad \exists s \in CN^*.\; \exists t, t' \in CN. \\
\qquad\quad st|_p = s + t + t' \;\wedge\; st|_{p+1} = s + cn_0 \;\wedge \\
\qquad\quad psr|_{p+1} = psr|_p \cup mksr(t, cn) \cup mksr(t', cn_1)) \;\wedge \\
\qquad (cd(m)|_p = \mathsf{new}(cn) \Rightarrow \\
\qquad\quad \exists s \in CN^*.\; st|_p = s \;\wedge\; st|_{p+1} = s + cn \;\wedge\; psr|_{p+1} = psr|_p) \;\wedge \\
\qquad (cd(m)|_p = \mathsf{areturn} \Rightarrow \\
\qquad\quad \exists s \in CN^*.\; \exists t \in CN.\; st|_p = s + t \;\wedge\; sr = psr|_p \cup mksr(t, rt(m)))))
\end{array}
$$

Figure 3: Equations of bytecode verification

$mksr(t, rt(m))$ requires that the set $sr$ is obtained from the value of $psr$ at the areturn instruction, where $mksr(t, rt(m))$ says that the class of the result is required to be a subclass of the return class of the method. Note that if $t = rt(m)$, then no subtype assumption is created here.

We assume that all methods in the set $M$ pass bytecode verification. In this case, we use two functions

$$bcvst : C \times M \to ST, \quad bcvsr : C \times M \to SR,$$

satisfying that

$$bcv(c, m, bcvst(c, m), bcvsr(c, m))$$

to denote the results of bytecode verification.

## 4.7 Loading and subtype constraints

*Loading* and *subtype constraints* are formalized as finite sets in

$$Ct = \mathcal{P}_\omega((L \times L \times CN) \cup (L \times CN \times CN)).$$

A *loading constraint* is a triple $\langle l, l', cn \rangle \in L \times L \times CN$, expressing that if loaders $l$ and $l'$ load classes for class name $cn$, then the loaded classes must be the same. The loading constraints here are essentially those defined in [14].

Using loading constraints, we formally define an equivalence relation

$$\simeq_{ct} : (L \times CN) \times (L \times CN) \to \mathbf{B}$$

as the smallest equivalence relation satisfying

$$\langle l, l', cn\rangle \in ct \;\Rightarrow\; \langle l, cn\rangle \;\simeq_{ct}\; \langle l', cn\rangle.$$

A *subtype constraint* is a triple $\langle l, cn, cn'\rangle \in L \times CN \times CN$, expressing that if loader $l$ loads a class for class name $cn$, and another class for class name $cn'$, then the loaded class for class name $cn$ must be a subclass of loaded class for class name $cn'$. A subtype constraint is actually a subtype requirement associated with a loader.

To model the subtype requirements modulo $\simeq_{ct}$, we define for each $ct \in Ct$ a relation

$$\leq_{ct} : (L \times CN) \times (L \times CN) \to \mathbf{B}$$

as the smallest transitive relation satisfying

$$\begin{array}{l}(\langle l, cn\rangle \;\simeq_{ct}\; \langle l', cn'\rangle \;\Rightarrow\; \langle l, cn\rangle \;\leq_{ct}\; \langle l', cn'\rangle) \wedge \\ (\langle l, cn, cn'\rangle \in ct \;\Rightarrow\; \langle l, cn\rangle \;\leq_{ct}\; \langle l, cn'\rangle).\end{array}$$

For each $lc \in LC$, we introduce a consistency-check relation $css_{lc} : Ct \to \mathbf{B}$, checking if the relations $\simeq_{ct}$ and $\leq_{ct}$ are satisfied or not on all loaded requests in $lc$. Formally the relation is defined by

$$\begin{array}{l}css_{lc}(ct) \;\Leftrightarrow \\ \forall \langle l, cn\rangle, \langle l', cn'\rangle \in \mathcal{D}(lc). \\ \quad (\langle l, cn\rangle \;\simeq_{ct}\; \langle l', cn'\rangle \;\Rightarrow\; lc(l, cn) = lc(l', cn')) \wedge \\ \quad (\langle l, cn\rangle \;\leq_{ct}\; \langle l', cn'\rangle \;\Rightarrow\; lc(l, cn) \preceq_{lc} lc(l', cn'))\end{array}$$

The key of this consistency-checking relation is that it does not check those requirements on classes that have not been loaded. This means that a subtype requirement generated by bytecode verification are not checked, until the referenced classes get loaded during execution.

## 4.8 States

Our type safety proof is based on a formalization using a state transition formalism. In this section we describe the state space of the transition system. We leave some details of the state abstract (i.e., we only state the properties we make use of), in order to make the formalization simpler and more general.

The state has a global component consisting of the heap and internal data structures regarding loaded classes, namely the loaded class cache and loading constraints. The state also has a component modeling the execution stack of a single thread. In our simplified JVM we only consider a single thread of execution.

The global states are of the form

$$GStt = Hp \times LC \times Ct.$$

We formulate a set of all operand stacks as

$$OS = V^*.$$

Furthermore, we introduce a set for frames as

$$\mathit{Frm} = C \times M \times P \times OS.$$

A frame $\langle c, m, p, os \rangle$ indicates that the $p$-th instruction in the code of method $m$ of class $c$ is about to be executed, and that $os$ is the current state of the operand stack. Recall local variables are not part of our formalization.

We introduce (method call) stacks as sequences of frames:

$$\mathit{Stk} = \mathit{Frm}^*.$$

Finally (method call) stacks and global states constitute (execution) states.

$$\mathit{Stt} = \mathit{Stk} \times \mathit{GStt}.$$

## 4.9 State transitions

A transition system is a pair $\langle X, \tau \rangle$ where $X$ is a set of states and $\tau \subseteq X \times X$ is a transition relation between states ($x \tau x'$ means that from state $x$ we can move to state $x'$). In our formalization we make use of a transition system $\langle \mathit{Stt}, \Longrightarrow \rangle$ and a family of labeled transition systems $\langle \mathit{GStt}, \overset{lab}{\Longrightarrow} \rangle$ with $lab \in \mathit{Lab}$. In substance, the unlabeled transition system captures the execution of JVM instructions, while the labeled ones capture activities taking place in the core of the machine.

The labeled transitions are

$$\overset{\mathrm{LD}(l,cn)}{\Longrightarrow}, \overset{\mathrm{RC}(c,cnn)}{\Longrightarrow}, \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow}, \overset{\mathrm{RM}(c,cn,mn,cn_1,cn_0)}{\Longrightarrow} \subseteq \mathit{GStt} \times \mathit{GStt},$$

where
$\langle l, cn \rangle \in L \times CN$,
$\langle c, cnn \rangle \in C \times CNN$ for $CNN = CN \uplus \{\mathsf{nil}\}$,
$\langle c, cn, fn, cn_0 \rangle \in C \times CN \times FN \times CN$, and
$\langle c, cn, mn, cn_1, cn_0 \rangle \in C \times CN \times MN \times CN \times CN$.

The LD transitions formalize class loading, the RC transitions class resolution, the RF transitions field resolution and the RM transitions method resolution.

Transitions are defined by means of schematic rules of the form $\frac{\alpha}{\beta}$. Each rule contains a set $\alpha$ of premises and a conclusion $\beta$, expressing that if all premises hold, then the conclusion holds. Conclusions assert transitions between states. The unlabeled and labeled transition relations are defined as the smallest relations that satisfy all the rules.

For the sake of readability, when we write $f(x)$ in a rule, where $f$ is a partial function, we regard the condition $x \in \mathrm{Dom}(f)$ as being implicitly present as a premise in the rule. Later in the paper we will prove that these definedness premises are actually unnecessary, since they hold whenever other premises all hold.

This paper follows the convention that the variables are written using lower-case letters and stand for elements of the sets denoted by their upper-case counterparts. For example, $o$ always stands for an element in $O$, and $v$ in $V$. Note that $v$ may be equal to null but $o$ may not.

Figure 4 describes state transitions for the JVM instructions except invokevirtual. Note that these rules have a nested form in which labeled rules are "invoked" as preconditions that bind primed variables which are then used to define the new state in a semantically obvious and consistent way. Also note that the value of the parameters for labeled transitions are determined from the state of the JVM in the originating state.

$$\frac{\begin{array}{l} cd(m)|_p = \mathsf{getfield}(cn, fn, cn_0) \\ \langle hp, lc, ct\rangle \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle \\ cl(o) \preceq_{lc'} lc'(ld(c), cn) \end{array}}{\begin{array}{l} \langle stk + \langle c, m, p, os + o\rangle, hp, lc, ct\rangle \Longrightarrow \\ \langle stk + \langle c, m, p+1, os + hp'(o)(fn)\rangle, hp', lc', ct'\rangle \end{array}} \quad \text{(GF)}$$

$$\frac{\begin{array}{l} cd(m)|_p = \mathsf{putfield}(cn, fn, cn_0) \\ \langle hp, lc, ct\rangle \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle \\ cl(o) \preceq_{lc'} lc'(ld(c), cn) \end{array}}{\begin{array}{l} \langle stk + \langle c, m, p, os + o + v\rangle, hp, lc, ct\rangle \Longrightarrow \\ \langle stk + \langle c, m, p+1, os\rangle, hp'[o \mapsto hp'(o)[fn \mapsto v]], lc', ct'\rangle \end{array}} \quad \text{(PF)}$$

$$\frac{\begin{array}{l} cd(m)|_p = \mathsf{new}(cn) \\ \langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle \\ o \in O - (C \cup \mathcal{D}(hp')) \\ cl(o) = lc'(ld(c), cn) \end{array}}{\begin{array}{l} \langle stk + \langle c, m, p, os\rangle, hp, lc, ct\rangle \Longrightarrow \\ \langle stk + \langle c, m, p+1, os + o\rangle, hp'[o \mapsto crRc_{lc'}(cl(o))], lc', ct'\rangle \end{array}} \quad \text{(NE)}$$

$$\frac{cd(m)|_p = \mathsf{areturn}}{\begin{array}{l} \langle stk + \langle c', m', p', os'\rangle + \langle c, m, p, os + v\rangle, hp, lc, ct\rangle \Longrightarrow \\ \langle stk + \langle c', m', p', os' + v\rangle, hp, lc, ct\rangle \end{array}} \quad \text{(RE)}$$

Figure 4: Transition rules for the JVM instructions except invokevirtual

Rule (GF) is for a getfield instruction. Intuitively, the JVM first resolves the field name referenced in the operand of the getfield by "invoking" the labeled transition RF to (if necessary) update the global state. Then the execution updates the current program counter and operand stack. The premise

$cl(o) \preceq_{lc'} lc'(ld(c), cn)$ is a type-safety check, requiring that the class of the object $o$ in which the field should be found, is a subclass of the class resulting from resolving name $cn$ from the defining loader $ld(c)$ of the current class $c$ – such class name is resolved as part of field resolution, so it is present in the loaded class cache $lc'$ (see below). The subclass relation $\preceq_{lc'}$ depends on the currently loaded classes, i.e. on $lc'$. In the conclusion, $hp'(o)$ is the mutable state of the object $o$ in the heap $hp'$, and $hp'(o)(fn)$ the value of the field with the name $fn$ in the mutable state.

Rule (PF) is for a putfield instruction. It is similar to rule (GF) except that it takes both an object $o$ and a value $v$ as arguments, and puts the value $v$ in the field with the name $fn$ in the mutable state of the object $o$ in the heap $hp'$.

Rule (NE) is for a new instruction. Intuitively, it first resolves (if necessary) the class name referenced by the new by the calls the procedure of class resolution represented by the labeled state transition, and then selects a new object $o$ with the required class $lc'(ld(c), cn)$, and finally extends the heap $hp'$ with an initial mutable state for the new object. To describe the initial mutable state, an auxiliary function $crRc_{lc} : C \to Rc$ is defined for each $lc \in LC$ by

$$crRc_{lc}(c) \;=\; \{fn \mapsto \mathsf{null} | fn \in allfnm_{lc}(c)\}.$$

Note that $o \notin C$. Since $cl(o) = lc'(ld(c), cn)$, $lc'(ld(c), cn) \neq Cls$.

Rule (RE) is trivial.

The rules in Figure 5 describe state transitions for invokevirtual instructions. Rule (IV) is for all invokevirtual instructions that do not invoke the `dfCl` method in the system class $ClLd$. Intuitively, the transition first calls the procedure of method resolution, and then dynamically selects a method for the execution. The premise $cl(o) \preceq_{lc'} lc'(ld(c), cn)$ is a type-safety check, requiring that the class of the object $o$ upon which the method is invoked, is a subclass of the class resulting from resolving name $cn$ from the defining loader $ld(c)$ of the current class $c$ – such a class name is resolved as part of method resolution, so it is present in the loaded class cache $lc'$ (see below). Note that in reality the `this`-object and the actual parameter should be put in some local variables. Here we put them in the operand stack, since our formalization does not consider local variables.

In reality, the `dfCl` method in the system class $ClLd$ is a final method. The argument type of the `dfCl` method is a byte array. Since we do not consider modifiers of methods, we simply assume that the code in consideration does not contain any overriding methods for the `dfCl` method. Since we do not consider byte arrays, we assume that the argument type of the `dfCl` method has the name `Obj`.

Rule (DC) describes the invocation of the `dfCl` method in the system class $ClLd$. Since the `dfCl` method is native, we ignore any actual code for it, and use a function and a relation to explicitly model the aspects of the object creation. The function is the previously defined function $crRc$, describing the

$$\frac{\begin{array}{l} cd(m)|_p = \mathsf{invokevirtual}(cn, mn, cn_1, cn_0) \\ \langle hp, lc, ct\rangle \overset{\mathrm{RM}(c,cn,mn,cn_1,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle \\ lc'(ld(c), cn) \not\preceq_{lc'} ClLd \;\vee\; \langle mn, cn_1, cn_0\rangle \neq \langle \texttt{dfCl}, \texttt{Obj}, \texttt{Cls}\rangle \\ cl(o) \preceq_{lc'} lc'(ld(c), cn) \\ mthSel_{lc'}(cl(o), mn, cn_1, cn_0) = \langle c', m'\rangle \end{array}}{\begin{array}{l} \langle stk + \langle c, m, p, os + o + v\rangle, hp, lc, ct\rangle \Longrightarrow \\ \langle stk + \langle c, m, p+1, os\rangle + \langle c', m', 0, [o, v]\rangle, hp', lc', ct'\rangle \end{array}} \quad \text{(IV)}$$

$$\frac{\begin{array}{l} cd(m)|_p = \mathsf{invokevirtual}(cn, \texttt{dfCl}, \texttt{Obj}, \texttt{Cls}) \\ \langle hp, lc, ct\rangle \overset{\mathrm{RM}(c,cn,\texttt{dfCl},\texttt{Obj},\texttt{Cls})}{\Longrightarrow} \overset{\mathrm{RC}(c,\texttt{Cls})}{\Longrightarrow} \langle hp', lc', ct'\rangle \\ lc'(ld(c), cn) \preceq_{lc'} ClLd \\ lc'(ld(c), \texttt{Cls}) = Cls \\ crCl_{hp'}(l, c') \\ \langle hp', lc', ct'\rangle \overset{\mathrm{RC}(c', sup(c'))}{\Longrightarrow} \langle hp'', lc'', ct''\rangle \\ addLc(lc'', l, nm(c'), c') = lc''' \\ \neg circ(lc''') \\ css_{lc'''}(addCt_{lc'''}(ct'', c')) \end{array}}{\begin{array}{l} \langle stk + \langle c, m, p, os + l + o\rangle, hp, lc, ct\rangle \Longrightarrow \\ \langle stk + \langle c, m, p+1, os + c'\rangle, hp''[c' \mapsto crRc_{lc'''}(Cls)], lc''', addCt_{lc'''}(ct'', c')\rangle \end{array}} \quad \text{(DC)}$$

Figure 5: Transition rules for the invokevirtual instruction

creation of an initial mutable state for an object of class $Cls$. The relation is $crCl_{hp} : L \times C \to \mathbf{B}$ for each $hp \in Hp$ defined by

$$crCl_{hp}(l, c) \;\Leftrightarrow\; c \in (C - \mathcal{D}(hp)) \;\wedge\; l = ld(c).$$

The relation describes the creation of a class object with a given loader. Note that since $c \in C$, $cl(c) = Cls$.

For convenience, we introduce more functions for the description of rule (DC). A function $addLc : LC \times L \times CN \times C \to LC \uplus \{fail\}$ is defined by

$$\begin{array}{l} \langle l, cn\rangle \notin \mathcal{D}(lc) \;\wedge \\ (\langle ld(c), sup(c)\rangle \in \mathcal{D}(lc) \;\Rightarrow\; allfnm_{lc}(lc(ld(c), sup(c))) \cap fnm(c) = \emptyset) \Rightarrow \\ \qquad addLc(lc, l, cn, c) = lc[\langle l, cn\rangle \mapsto c] \\ \langle l, cn\rangle \in \mathcal{D}(lc) \;\vee \\ (\langle ld(c), sup(c)\rangle \in \mathcal{D}(lc) \;\Rightarrow\; allfnm_{lc}(lc(ld(c), sup(c))) \cap fnm(c) \neq \emptyset) \Rightarrow \\ \qquad addLc(lc, l, cn, c) = fail \end{array}$$

Intuitively, the function extends a loaded class cache only when the added loading request does not already exist and causes no field hiding; otherwise it yields a failure. Note that if we allowed field hiding, we would not be able to model the mutable states of objects as finite functions from field names to values.

A function $bcvCt : C \to Ct$, which transforms all subtype requirements of a class generated by bytecode verification into a set of loading constraints, is defined by

$$bcvCt(c) = \{\langle ld(c), cn, cn' \rangle \mid m \in mth(c), \langle cn, cn' \rangle \in bcvsr(c, m)\}.$$

In Java, corresponding argument and return types of an overriding and overridden method always have identical names. But identical class names referenced in two classes do not necessarily imply identical loaded classes. To solve the problem, loading constraints on names of argument and return classes of overriding and overridden methods with respect to the corresponding initiating loaders are introduced [14]. Formally we define a function $ovrCt_{lc} : C \to \mathcal{P}_\omega(L \times L \times CN)$ for each $lc \in LC$, which generates a set of loading constraints for a given class as follows:

$$\begin{aligned} ovrCt_{lc}(c) = \{&\langle ld(c), ld(c'), at(m) \rangle, \langle ld(c), ld(c'), rt(m) \rangle \mid m \in mth(c) \wedge \\ &mthSel_{lc}(lc(ld(c), sup(c)), nm(m), at(m), rt(m)) = \langle c', m' \rangle\} \end{aligned}$$

Finally, we define a function $addCt_{lc} : Ct \times C \to Ct$ for each $lc \in LC$ by

$$addCt_{lc}(ct, c) = ct \cup bcvCt(c) \cup ovrCt_{lc}(c)$$

Rule (DC) first calls the procedure of method resolution represented by $\overset{\mathrm{RM}(c,cn,\mathtt{dfCl},\mathtt{Obj},\mathtt{Cls})}{\Longrightarrow}$ and that of class resolution represented by $\overset{\mathrm{RC}(c,\mathtt{Cls})}{\Longrightarrow}$. The condition $lc'(ld(c), cn) \preceq_{lc'} ClLd$ ensures that the invocation is performed on a real loader. The condition $lc'(ld(c), \mathtt{Cls}) = Cls$ ensures that the return type of the `dfCl` method is $Cls$, as expected. The condition $crCl_{hp'}(l, c')$ describes the selection of an object reference $c'$. The expression $crRc_{lc'''}(Cls)$ in the resulting state represents the initial mutable state for the new class object.

Note that the relation $crCl_{hp'}(l, c')$ is a simplification of reality, since it does not cover the fact that the class object $c'$ should actually be built from the actual argument $o$. In reality, the fields, methods and superclass of the class object $c'$ should actually be given by the argument $o$. Although there are no problems to formalize them, these details are irrelevant to the properties we are interested in this paper.

The creation of a class object includes resolution of the superclass of the object class. The labeled transition $\overset{\mathrm{RC}(c',sup(c'))}{\Longrightarrow}$ models this. This step is crucial in ensuring that the loaded class cache in the resulting state is upward closed.

The fact that the expression $addLc(lc'', l, nm(c'), c')$ does not yield a failure implies a requirement in the JVMS (Section 5.3.5) that if the creation process starts with a given defining loader and a class name, then no class objects can be loaded for the loading request consisting of the defining loader and the class name. It also formally rules out field hiding, which we do not consider in the paper.

The meaning of the conditions $\neg circ(lc''')$ and $css_{lc'''}(addCt_{lc'''}(ct'', c'))$ is straightforward.

The rules in Figure 6 define labeled transitions. Rules (RC1) and (RC2) describe the procedure of class resolution. Rule (RC1) is trivial. Rule (RC2) intuitively says that resolution of a class name *cn* from a class *c* first calls the loading of a class for the name *cn* using the defining loader $ld(c)$ of $c$ as the initiating loader.

Rules (LD1) and (LD2) describe the procedure of loading. Rule (LD1) says that no new loading will be started if the loaded class cache records the loading request.

$$\frac{}{\langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,\mathsf{nil})}{\Longrightarrow} \langle hp, lc, ct\rangle} \ (\mathrm{RC1})$$

$$\frac{\langle hp, lc, ct\rangle \overset{\mathrm{LD}(ld(c),cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle}{\langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle} \ (\mathrm{RC2})$$

$$\frac{\langle l, cn\rangle \in \mathcal{D}(lc)}{\langle hp, lc, ct\rangle \overset{\mathrm{LD}(l,cn)}{\Longrightarrow} \langle hp, lc, ct\rangle} \ (\mathrm{LD1})$$

$$\frac{\begin{array}{l}\langle l, cn\rangle \notin \mathcal{D}(lc)\\ mthSel_{lc}(cl(l), \mathtt{ldCl}, \mathtt{Str}, \mathtt{Cls}) = \langle c, m\rangle\\ \langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,\mathtt{Str})}{\Longrightarrow} \overset{\mathrm{RC}(c,\mathtt{Cls})}{\Longrightarrow} \langle hp', lc', ct'\rangle\\ \langle lc'(ld(c), \mathtt{Str}), lc'(ld(c), \mathtt{Cls})\rangle = \langle Str, Cls\rangle\\ \langle [\langle c, m, 0, [l, str(cn)]\rangle], hp'[str(cn) \mapsto \{\}], lc', ct'\rangle \Longrightarrow^{*}\\ \quad \langle [\langle c, m, p, os + c'\rangle], hp'', lc'', ct''\rangle\\ cd(m)|_p = \mathsf{areturn}\\ css_{lc''\{\langle l,cn\rangle \mapsto c'\}}(ct'')\end{array}}{\langle hp, lc, ct\rangle \overset{\mathrm{LD}(l,cn)}{\Longrightarrow} \langle hp'', lc''\{\langle l, cn\rangle \mapsto c'\}, ct''\rangle} \ (\mathrm{LD2})$$

$$\frac{\begin{array}{l}\langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle\\ fldSel_{lc'}(lc'(ld(c), cn), fn, cn_0) = \langle c', f\rangle\\ css_{lc'}(ct' \cup \{\langle ld(c), ld(c'), cn_0\rangle\})\end{array}}{\langle hp, lc, ct\rangle \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \langle hp', lc', ct' \cup \{\langle ld(c), ld(c'), cn_0\rangle\}\rangle} \ (\mathrm{RF})$$

$$\frac{\begin{array}{l}\langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle\\ mthSel_{lc'}(lc'(ld(c), cn), mn, cn_1, cn_0) = \langle c', m\rangle\\ css_{lc'}(ct' \cup \{\langle ld(c), ld(c'), cn_j\rangle \mid 0 \leq j \leq 1\})\end{array}}{\langle hp, lc, ct\rangle \overset{\mathrm{RM}(c,cn,mn,cn_1,cn_0)}{\Longrightarrow} \langle hp', lc', ct' \cup \{\langle ld(c), ld(c'), cn_j\rangle \mid 0 \leq j \leq 1\}\rangle} \ (\mathrm{RM})$$

Figure 6: Transition rules for operations in the JVM core

Rule (LD2) says that the loading is realized as an execution of the `ldCl` method of the loader if the loaded class cache does not record the loading request, provided that such a `ldCl` method can be found. Intuitively one may expect the following additional premises in the rule to ensure that $\mathit{mthSel}_{lc}(\mathit{cl}(l), \mathtt{ldCl}, \mathtt{Str}, \mathtt{Cls})$ yields a `ldCl` method of a loader.

- $\exists m \in \mathit{mth}(\mathit{ClLd}).\ \mathit{nm}(m) = \mathtt{ldCl}\ \wedge\ \mathit{at}(c)\ \wedge\ \mathit{rt}(c)$, i.e. that class $\mathit{ClLd}$ declares a `ldCl` method.
- $\mathit{cl}(l) \preceq_{lc} \mathit{ClLd}$, i.e. that the object $l$ is really a loader.

The first condition is an assumption about the system class. Although the second is important in ensuring that our transition has the desired semantics, it can be derived in the context we are interested anyway (see Section 5.2). In any case, both conditions are not essential to type-safety.

The condition $\langle lc'(\mathit{ld}(c), \mathtt{Str}), lc'(\mathit{ld}(c), \mathtt{Cls})\rangle = \langle \mathit{Str}, \mathit{Cls}\rangle$ in rule (LD2) ensures that the `ldCl` method has the correct argument and return type. In the transition $\Longrightarrow^*$, the expression $\mathit{str}(cn)$ represents a string object for the string $cn$, the class object $c'$ the loaded class object. The empty record $\{\}$ in the heap extension $\mathit{str}(cn) \mapsto \{\}$ corresponds to the fact that a string has no mutable state. In reality a `ldCl` method typically calls the `dfCl` method to create a new class object.

The expression $lc''\{\langle l, cn\rangle \mapsto c'\}$ in rule (LD2) formalizes a tricky point: if the loaded class cache $lc''$ records the loading request $\langle l, cn\rangle$, then it remains unchanged. In other words, if it is detected at the end of the loading that the loading request has been loaded and thus recorded by the loaded class cache, the loading will ignore the class object $c'$ and yield the recorded loaded class as the result.

Rules (RF) and (RM) describe the procedures of field and method resolution. They are straightforward.

Since the transition relations are in fact the smallest relations generated by finitely many applications of the rules, a transition holds if and only if a finite proof tree can be built where the transition is the root and all leaves are non-transition conditions. Infinite proof trees correspond to non-terminating execution in practice.

## 4.10 An example

Consider the bytecode in Section 2.5. Assume that execution reaches a state $\langle \cdots + \langle C, m, p1, \cdots\rangle, hp, lc, ct\rangle$ at program point $p_1$, where $\langle l_1, \mathtt{D}\rangle$, $\langle l_1, \mathtt{T}\rangle$, $\langle l_2, \mathtt{T}\rangle \notin \mathcal{D}(lc)$. Then a transition for the $\mathsf{new(D)}$ is determined by the proof tree in Figure 7, where ellipses denote omitted details. In the figure, since $\langle l_1, \mathtt{D}\rangle \notin \mathcal{D}(lc)$, $\langle l_1, \mathtt{D}\rangle \notin \mathcal{D}(lc'')$ and thus rule (LD2) is applied.

By a similar proof tree, we can determine the transition for $\mathsf{new(T)}$ at program point $p_2$. Since $\langle l_1, \mathtt{T}\rangle$ is not in the domain of the loaded class cache at program point $p_2$, the proof tree contains an application of rule (LD2),

$$\dfrac{\cdots \quad \dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \cdots\quad \begin{array}{l}\langle[\langle \mathit{MyLd}, \mathit{ldCl}, 0, [l_1, \mathit{str}(\mathtt{D})]\rangle], \mathit{hp}'[\mathit{str}(\mathtt{D}) \mapsto \{\}], \mathit{lc}', \mathit{ct}'\rangle \Longrightarrow^{*}\\ \langle[\langle \mathit{MyLd}, \mathit{ldCl}, \cdots, \cdots + D\rangle], \mathit{hp}'', \mathit{lc}'', \mathit{ct}''\rangle\end{array}\end{array}}{\langle \mathit{hp}, \mathit{lc}, \mathit{ct}\rangle \overset{\mathrm{LD}(l_1,\mathtt{D})}{\Longrightarrow} \langle \mathit{hp}'', \mathit{lc}''\{\langle l_1, \mathtt{D}\rangle \mapsto D\}, \mathit{ct}''\rangle}\;(LD2)}{\langle \mathit{hp}, \mathit{lc}, \mathit{ct}\rangle \overset{\mathrm{RC}(C,\mathtt{D})}{\Longrightarrow} \langle \mathit{hp}'', \mathit{lc}''\{\langle l_1, \mathtt{D}\rangle \mapsto D\}, \mathit{ct}''\rangle}\;(RC2)}{}}{\begin{array}{l}\langle \cdots + \langle C, m, p1, \cdots\rangle, \mathit{hp}, \mathit{lc}, \mathit{ct}\rangle \Longrightarrow\\ \langle \cdots + \langle C, m, p_2, \cdots + o\rangle, \mathit{hp}''[o \mapsto \cdots], \mathit{lc}''\{\langle l_1, \mathtt{D}\rangle \mapsto D\}, \mathit{ct}''\rangle\end{array}}\;(NE)$$

Figure 7: Proof tree for new(D) at $p_1$

which causes the item $\langle l_1, \mathtt{T}\rangle \mapsto T_1$ to be added to the loaded class cache at program point $p_3$.

Now assume that execution has finished the invokevirtual(D, n, T, void) at program point $p_3$ and reaches a state

$$\langle \cdots + \langle C, m, p_3 + 1, \cdots\rangle + \langle D, n, r_1, [o, o_1]\rangle, hp, lc, ct\rangle.$$

It is easy to see that the proof tree for the transition contains an application of rule (RM), which introduces the loading constraint $\langle l_1, l_2, \mathtt{T}\rangle$ in the loading constraint set $ct$. By our assumption above at program point $p_1$, $\langle l_2, \mathtt{T}\rangle \notin \mathcal{D}(lc)$. This means that the loading constraint $\langle l_1, l_2, \mathtt{T}\rangle$ is not violated at this moment.

Assume execution reaches the getfield at program point $r_3$. Since $\langle l_2, \mathtt{T}\rangle$ is not in the domain of the loaded class cache, the proof tree for this instruction would include an application of rule (LD2), which on one hand would require the item $\langle l_2, \mathtt{T}\rangle \mapsto T_2$ to be added to the loaded class cache, but on the other hand check the *css* relation ensuring that no loading constraints are violated with respect to the new loaded class cache. Since $T_1 \neq T_2$, the loading constraint $\langle l_1, l_2, \mathtt{T}\rangle$ is violated. Thus such a proof tree does not exist. That is, no transition fires for this instruction.

## 5 Safety properties

### 5.1 Valid states and real loaders

We are not interested in all possible states and global states, but only those that satisfy some constraints, which we call *valid* states and *valid* global states. In this section, we will formally define what valid states and valid global states are.

First of all, we are interested only in those states where the heap is *well-formed* with respect to the loaded class cache in the following sense:

– The class of each object in the heap is a loaded class in the loaded class cache.

– Each object in the heap has the required set of field names.
– Each object in the heap contains only field values that are either $\mathsf{null}$ or objects in the heap.
– Each class object in the heap is in the loaded class cache.

Formally, we define a relation $wfm_{lc} : Hp \to \mathbf{B}$ for each given $lc \in LC$ by

$$\begin{array}{l} wfm_{lc}(hp) \;\Leftrightarrow \\ (\forall o \in \mathcal{D}(hp). \\ \quad cl(o) \in \mathcal{R}(lc) \wedge \\ \quad \mathcal{D}(hp(o)) = allfnm_{lc}(cl(o)) \wedge \\ \quad (\forall fn \in \mathcal{D}(hp(o)).\ hp(o)(fn) = \mathsf{null} \;\vee\; hp(o)(fn) \in \mathcal{D}(hp)) \wedge \\ \quad (o \in C \;\Rightarrow\; o \in \mathcal{R}(lc))). \end{array}$$

Second, we are interested only in those loaded class caches that are *well-formed* in the following sense:

– They are upward-closed.
– They are not subclass-circular.
– They record the defining loaders of all loaded classes as initiating loaders.
– They contain only those loaded classes on which the function $ld$ yields real loaders. (Recall that for a given class object $c \in C$, the value $ld(c)$ is not necessarily a real loader with respect to the loaded class cache in the execution.)
– They do not allow field hiding.

Formally, we define a relation $wfm : LC \to \mathbf{B}$ by

$$\begin{array}{l} wfm(lc) \;\Leftrightarrow \\ clos(lc) \;\wedge\; \neg circ(lc) \wedge \\ (\forall c \in \mathcal{R}(lc). \\ \quad cl(ld(c)) \preceq_{lc} ClLd \wedge \\ \quad \langle ld(c), nm(c)\rangle \in \mathcal{D}(lc) \;\wedge\; lc(ld(c), nm(c)) = c \wedge \\ \quad (sup(c) \neq \mathsf{nil} \;\Rightarrow\; allfnm_{lc}(lc(ld(c), sup(c))) \cap fnm(c) = \emptyset)) \end{array}$$

Third, we are interested only in those loading constraint sets that are consistent, and contain all loading constraints for argument and return classes of overriding and overridden methods, and all loading constraints generated from the subtype requirements of all loaded classes. To express this, we define a relation $wfm_{lc} : Ct \to \mathbf{B}$ for each $lc \in LC$ by

$$wfm_{lc}(ct) \;\Leftrightarrow\; css_{lc}(ct) \;\wedge\; (\forall c \in \mathcal{R}(lc).\ ovrCt_{lc}(c) \subseteq ct \;\wedge\; bcvCt(c) \subseteq ct)$$

We are interested only in those states where the (method call) stack is *well-formed* with respect to the global state in the following sense:

– In each frame of the (method call) stack,
  • the current class has been loaded into the loaded class cache,

  - the method is declared in the current class,
  - the program point is valid in the method, and
  - the operand stack contains only null or objects in the heap.
- For any two successive frames of the (method call) stack,
  - the program point in the first frame is the next program point of an invokevirtual instruction invoking the method in the second frame, and
  - the loading constraint set contains a constraint ensuring that the loaders in both frames will load the same return class for the invoked method in the second frame.

The second group of conditions mean that method calls are the only reason that causes a method call stack to grow, and that the corresponding caller and callee always load the same return class for the method. Actually the corresponding caller and callee also always load the same argument class for the method, but we do not use that invariant in the proof.

Formally we define a relation $wfm_{hp,lc,ct} : Stk \to \mathbf{B}$ for each $\langle hp, lc, ct\rangle \in GStt$, covering the above points:

$$\begin{array}{l}
wfm_{hp,lc,ct}(stk) \;\Leftrightarrow \\
(\forall \langle c, m, p, os\rangle \in stk. \\
\quad c \in \mathcal{R}(lc) \;\wedge \\
\quad m \in mth(c) \;\wedge \\
\quad 0 \le p < |cd(m)| \;\wedge \\
\quad (v \in os \;\Rightarrow\; (v \ne \mathsf{null} \;\Rightarrow\; v \in \mathcal{D}(hp)))) \;\wedge \\
(\forall \langle c', m', p', os'\rangle, \langle c, m, p, os\rangle \in Frm. \\
\quad \cdots + \langle c', m', p', os'\rangle + \langle c, m, p, os\rangle + \cdots = stk \;\Rightarrow \\
\qquad p' > 0 \;\wedge \\
\qquad (\exists cn \in CN.\; cd(m')|_{p'-1} = \mathsf{invokevirtual}(cn, nm(m), at(m), rt(m))) \;\wedge \\
\qquad \langle ld(c'), ld(c), rt(m)\rangle \in ct)
\end{array}$$

For each $lc \in LC$ and $ct \in Ct$, we define a relation

$$\lll_{lc,ct} : (L \times CN) \times (L \times CN) \to \mathbf{B}$$

covering $\le_{ct}$ and the order induced by $\preceq_{lc}$, formally as the smallest transitive relation satisfying

$$\begin{array}{l}
(\langle l, cn\rangle \;\le_{ct}\; \langle l', cn'\rangle \;\Rightarrow\; \langle l, cn\rangle \lll_{lc,ct} \langle l', cn'\rangle) \;\wedge \\
(\langle l, cn\rangle, \langle l', cn'\rangle \in \mathcal{D}(lc) \;\wedge\; lc(l, cn) \preceq_{lc} lc(l', cn') \;\Rightarrow\; \langle l, cn\rangle \lll_{lc,ct} \langle l', cn'\rangle).
\end{array}$$

We define a *conformance* relation $cfm_{lc,ct} : V \times L \times CN \to \mathbf{B}$ for each $lc \in LC$ and $ct \in Ct$ by

$$cfm_{lc,ct}(v, l, cn) \;\Leftrightarrow\; (v \ne \mathsf{null} \;\Rightarrow\; \langle ld(cl(v)), nm(cl(v))\rangle \lll_{lc,ct} \langle l, cn\rangle)$$

Intuitively, the conformance relation ensures that if a value $v \in V$ is not null, then its class is or will be a subclass of the loaded class for name $cn$ and loader $l$ after the classes are loaded.

To understand the motivation behind the conformance relation, let us consider a state $(stk, hp, lc, ct)$ such that some operand stack in $stk$ contains a value $v$. Assume that $wfm_{lc}(hp)$, $wfm(lc)$, $wfm_{lc}(ct)$ and $wfm_{hp,lc,ct}(stk)$, which imply that

$$\langle ld(cl(v)), nm(cl(v))\rangle \in \mathcal{D}(lc) \wedge lc(ld(cl(v)), nm(cl(v))) = cl(v) \wedge css_{lc}(ct).$$

In this case the relation

$$\langle ld(cl(v)), nm(cl(v))\rangle \lll_{lc,ct} \langle l, cn\rangle$$

actually means that if $\langle l, cn\rangle \in \mathcal{D}(lc)$, then

$$lc(ld(cl(v)), nm(cl(v))) \preceq_{lc} lc(l, cn)$$

The tricky point here is the condition $\langle l, cn\rangle \in \mathcal{D}(lc)$. In other words, if no class has been loaded for the loading request $\langle l, cn\rangle$, the value $v$ can be of any class.

Now we define a relation $cfm_{lc,ct} : Hp \to \mathbf{B}$ for each $lc \in LC$ and $ct \in Ct$ to check the conformance of heaps:

$$\begin{array}{l} cfm_{lc,ct}(hp) \;\Leftrightarrow \\ \forall o \in \mathcal{D}(hp).\ \forall fn \in \mathcal{D}(hp(o)).\ \forall cn_0 \in CN.\ \forall c \in C.\ \forall f \in F. \\ fldSel_{lc}(cl(o), fn, cn_0) = \langle c, f\rangle \;\Rightarrow\; cfm_{lc,ct}(hp(o)(fn), ld(c), cn_0) \end{array}$$

The relation ensures that the content $hp(o)(fn)$ of a field of an object in a heap always conforms to the name $cn_0$ of the class of the field with respect to the loader $ld(c)$ of the class where the field is actually declared. Note that $c$ is either $cl(o)$ or a superclass of it.

We define *valid global states* using a relation $vld : GStt \to \mathbf{B}$ defined by

$$vld(hp, lc, ct) \;\Leftrightarrow\; wfm_{lc}(hp) \;\wedge\; wfm(lc) \;\wedge\; wfm_{lc}(ct) \;\wedge\; cfm_{lc,ct}(hp)$$

For convenience, we lift the first relation $cfm$ above to sequences of values and class names as follows:

$$cfm_{lc,ct}([v_1, \ldots, v_n], l, [cn_1, \ldots, cn_k]) \;\Leftrightarrow\; n = k \;\wedge\; (\bigwedge_{1\le i\le n} cfm_{lc,ct}(v_i, l, cn_i))$$

Furthermore, we lift the last relation $cfm$ to (method call) stacks, and define $cfm_{lc,ct} : Stk \to \mathbf{B}$ by

$$cfm_{lc,ct}(stk) \;\Leftrightarrow\; \forall\langle c, m, p, os\rangle \in stk.\ cfm_{lc,ct}(os, ld(c), bcvst(c, m)|_p)$$

Now we introduce *valid states* via a relation $vld : Stt \to \mathbf{B}$ defined by

$$vld(stk, hp, lc, ct) \;\Leftrightarrow\; vld(hp, lc, ct) \;\wedge\; wfm_{hp,lc,ct}(stk) \;\wedge\; cfm_{lc,ct}(stk)$$

The transition step $\Longrightarrow$ *preserves validity* if whenever

$$\langle stk, hp, lc, ct\rangle \Longrightarrow \langle stk', hp', lc', ct'\rangle$$

and $vld(stk, hp, lc, ct)$, $vld(stk', hp', lc', ct')$; the transition step $\overset{lab}{\Longrightarrow}$ *preserves validity* if whenever

$$\langle hp, lc, ct\rangle \overset{lab}{\Longrightarrow} \langle hp', lc', ct'\rangle$$

and $vld(hp, lc, ct)$, $vld(hp', lc', ct')$.

A labeled transition step of the form

$$\langle hp, lc, ct\rangle \overset{\mathrm{LD}(l,cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle$$

is said to be *associated with a real loader* if $cl(l) \preceq_{lc} ClLd$. A labeled transition step of the form

$$\langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} (\text{or} \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \text{or} \overset{\mathrm{RM}(c,cn,mn,cn_1,cn_0)}{\Longrightarrow}) \langle hp', lc', ct'\rangle$$

is *associated with a real loader* if $cl(ld(c)) \preceq_{lc} ClLd$.

## 5.2 The properties

This section formulates three theorems. The first is simple.

**Theorem 1.** *If $stt \Longrightarrow stt'$, and $stt$ is valid, (or $gstt \overset{lab}{\Longrightarrow} gstt'$, and $gstt$ is valid,) then for all $stt'' \Longrightarrow stt'''$ and $gstt'' \overset{lab'}{\Longrightarrow} gstt'''$ in the proof tree of $stt \Longrightarrow stt'$ (or $gstt \overset{lab}{\Longrightarrow} gstt'$, respectively), $stt''$, $stt'''$, $gstt''$ and $gstt'''$ are all valid.*

The third theorem states that all those objects used as loaders in the state machine are really loaders. For example, a direct consequence of the theorem is that in the applications of rule (LD2), the condition $cl(l) \preceq_{lc} ClLd$ holds.

**Theorem 2.** *In any proof for $stt \Longrightarrow stt'$, if $vld(stt)$ then all labeled transitions in the proof are associated with a real loader.*

As mentioned before, our transition system formalizes a defensive JVM, which performs runtime type safety checks to ensure type safety. Formally, the premises and the pattern of the input state in the consequence of a transition rule together determine when the transition can fire. But the pattern and individual premises play different roles in modeling the execution. Concretely, a group of premises and the pattern determine the selection of transition rules; a second group of premises rule out possible runtime errors; a third group of premises express some "purely defensive" conditions, which always hold in the context we are interested (i.e. in valid states), provided that all other premises hold; a fourth group of premises formulates the selection of some objects, fields or methods, which always hold for some objects, fields or methods. These groups of premises are not necessarily disjoint.

For example, in rule (GF) the top frame in the input state must be of the form

$$stk + \langle c, m, p, os + o\rangle,$$

which implies, among others, that the transition cannot fire on an operand stack with $\mathsf{null}$ on the top. The premise

$$cd(m)|_p = \mathsf{getfield}(cn, fn, cn_0)$$

requires that the rule can be selected only when method $m$ has a $\mathsf{getfield}$ instruction at program point $p$. The premise

$$\langle hp, lc, ct\rangle \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle$$

ensures the success of field resolution, and thus rules out runtime errors. The premise $cl(o) \preceq_{lc'} lc'(ld(c), cn)$ is a purely defensive condition ensuring type safety. The condition always holds for a valid input state in the consequence, provided that all other premises hold (see below).

In rule (NE), the premise

$$o \in O - (C \cup \mathcal{D}(hp'))$$

expresses a selection of a non-class object $o$. Such an object always exists, since the set $O - (C \cup \mathcal{D}(hp'))$ is a countable set.

In rule (IV), the premise $cl(o) \preceq_{lc'} lc'(ld(c), cn)$ is a purely defensive condition ensuring type safety. The premise

$$mthSel_{lc'}(lc'(cl(o), mn, cn_1, cn_0) = \langle c', m'\rangle.$$

expresses the selection of a method. The selection actually never fails at this stage if all other premises (including the transition premises) of the rule hold.

Recall that we assume that the definedness premises for all applications of partial functions are always present in the rules. All these premises are "purely defensive" conditions ensuring type safety.

The second theorem explicitly gives all conditions the JVM need to check in order to move from a valid state in executing a JVM instruction. It includes a form of type-safety. From it, one can realize that it is unnecessary for the JVM to explicitly check the following pure defensive conditions for type safety.

– The definedness conditions for all applications of partial functions in each rule.
– The premise $cl(o) \preceq_{lc'} lc'(ld(c), cn)$ in rules (GF), (PF) and (IV).

**Theorem 3.** *Let* $stt = \langle stk + \langle c, m, p, os\rangle, hp, lc, ct\rangle$ *be a valid state. Let any of the following conditions be true:*

*1.* $cd(m)|_p = \mathsf{getfield}(cn, fn, cn_0)\ \wedge$
$\exists\langle hp', lc', ct'\rangle.\ \langle hp, lc, ct\rangle \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle$*;*
*2.* $cd(m)|_p = \mathsf{putfield}(cn, fn, cn_0)\ \wedge$
$\exists\langle hp', lc', ct'\rangle.\ \langle hp, lc, ct\rangle \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle$*;*
*3.* $cd(m)|_p = \mathsf{new}(cn)\ \wedge\ \exists\langle hp', lc', ct'\rangle.\ \langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle$*;*
*4.* $cd(m)|_p = \mathsf{areturn}\ \wedge\ stk \neq [\,]$*;*
*5.* $cd(m)|_p = \mathsf{invokevirtual}(cn, mn, cn_1, cn_0)\ \wedge$
$\langle mn, cn_1, cn_0\rangle \neq \langle \mathtt{dfCl}, \mathtt{Obj}, \mathtt{Cls}\rangle\ \wedge$
$\exists\langle hp', lc', ct'\rangle.\ (\langle hp, lc, ct\rangle \overset{\mathrm{RM}(c,cn,mn,cn_1,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle\ \wedge$
$\qquad lc'(ld(c), cn)\ \not\preceq_{lc'} ClLd)$;
*6.* $cd(m)|_p = \mathsf{invokevirtual}(cn, \mathtt{dfCl}, \mathtt{Obj}, \mathtt{Cls})\ \wedge$
$\exists\langle hp', lc', ct'\rangle, \langle hp'', lc'', ct''\rangle, c', lc''', l, o.$
$\quad (\langle hp, lc, ct\rangle \overset{\mathrm{RM}(c,cn,\mathtt{dfCl},\mathtt{Obj},\mathtt{Cls})}{\Longrightarrow} \overset{\mathrm{RC}(c,\mathtt{Cls})}{\Longrightarrow} \langle hp', lc', ct'\rangle\ \wedge$
$\quad lc'(ld(c), cn) \preceq_{lc'} ClLd\ \wedge$
$\quad lc'(ld(c), \mathtt{Cls}) = Cls\ \wedge$
$\quad crCl_{hp'}(l, c')\ \wedge$
$\quad \langle hp', lc', ct'\rangle \overset{\mathrm{RC}(c',sup(c'))}{\Longrightarrow} \langle hp'', lc'', ct''\rangle\ \wedge$
$\quad addLc(lc'', l, nm(c'), c') = lc'''\ \wedge$
$\quad \neg circ(lc''')\ \wedge$
$\quad css_{lc'''}(addCt_{lc'''}(ct'', c')))\ \wedge$
$\quad stk = stk' + l + o.$

*Then there exists* $stt' \in Stt$ *such that* $stt \Longrightarrow stt'$*.*

## 5.3 Some useful lemmas

**Lemma 4.** – *If* $fldSel_{lc}(c, fn, cn_0) = \langle c', f\rangle$*, then* $c \preceq_{lc} c'$ *and* $fn \in allfnm_{lc}(c)$*.*
– *If* $mthSel_{lc}(c, mn, cn_1, cn_0) = \langle c', m\rangle$*, then* $c \preceq_{lc} c'$*.*
– *In both cases above, if* $c \neq c'$*, then* $c' \in \mathcal{D}(lc)$*.*

*Proof.* Directly follows from the definitions of $\preceq_{lc}$, $fldSel_{lc}$, $allfnm_{lc}$ and $mthSel_{lc}$. □

**Lemma 5.** *If* $\langle hp, lc, ct\rangle \overset{\mathrm{LD}(l,cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle$*, then* $\langle l, cn\rangle \in \mathcal{D}(lc')$*.*

*Proof.* Directly follows from rules (LD1) and (LD2). □

**Lemma 6.** *If* $\langle hp, lc, ct\rangle \overset{\mathrm{RC}(c,cn)}{\Longrightarrow} \langle hp', lc', ct'\rangle$*, then* $\langle ld(c), cn\rangle \in \mathcal{D}(lc')$*.*

*Proof.* Directly follows from rule (RC2) and Lemma 5. □

**Lemma 7.** *If* $\langle hp, lc, ct\rangle \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle$*, then*

$\langle ld(c), cn\rangle \in \mathcal{D}(lc')\ \wedge$
$(\exists c' \in C.\ \exists f \in F.\ fldSel_{lc'}(lc'(ld(c), cn), fn, cn_0) = \langle c', f\rangle\ \wedge\ \langle ld(c), ld(c'), cn_0\rangle \in ct')\ \wedge$
$fn \in allfnm_{lc'}(lc'(ld(c), cn)))$

*Proof.* Directly follows from rule (RF) and Lemmas 4 and 6. □

Our simplification that no field hiding is present has a consequence as stated in the following lemma.

**Lemma 8.** *Assume that a loaded class cahce lc satisfies that*

$$clos(lc) \ \wedge \ (\forall c \in \mathcal{R}(lc).\ allfnm_{lc}(lc(ld(c), sup(c))) \cap fnm(c) = \emptyset).$$

*If there are $c, c', c''$, $fn$, $cn$ and $f$ such that*

$$c \preceq_{lc} c' \ \wedge \ fldSel_{lc}(c', fn, cn) = \langle c'', f\rangle$$

*then $fldSel_{lc}(c, fn, cn) = \langle c'', f\rangle$.*

*Proof.* The case where $c = c'$ is trivial. In the case where $c \neq c'$, since $c \preceq_{lc} c'$, $c, c' \in \mathcal{R}(lc)$, and the proof follows from the given conditions. □

**Lemma 9.** *If $\langle hp, lc, ct\rangle \overset{\mathrm{RM}(c,cn,mn,cn_1,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle$, then*

$$\begin{array}{l} \langle ld(c), cn\rangle \in \mathcal{D}(lc') \ \wedge \\ (\exists c' \in C.\ \exists m' \in M.\ mthSel_{lc'}(lc'(ld(c), cn), mn, cn_1, cn_0) = \langle c', m'\rangle \ \wedge \\ \qquad\qquad\qquad \langle ld(c), ld(c'), cn_1\rangle, \langle ld(c), ld(c'), cn_0\rangle \in ct') \end{array}$$

*Proof.* Directly follows from rule (RM) and Lemma 6. □

The next lemma says that the global state expands only along with the execution in a certain sense. Note that we do not consider garbage collection in this paper.

**Lemma 10.** *If*

$$\begin{array}{l} \langle stk, hp, lc, ct\rangle \Longrightarrow \langle stk', hp', lc', ct'\rangle \ \text{or} \\ \langle hp, lc, ct\rangle \overset{lab}{\Longrightarrow} \langle hp', lc', ct'\rangle \ \text{for } lab \in Lab, \end{array}$$

*then $\mathcal{D}(hp) \subseteq \mathcal{D}(hp') \ \wedge \ lc \subseteq lc' \ \wedge \ ct \subseteq ct'$.*

*Proof.* Follows by induction on the proof trees of applications of transition rules. □

The following lemma formally states how loading constraints ensure that overriding methods always have the same argument and return classes.

**Lemma 11.** *If $wfm_{lc}(ct)$, then*

$$\begin{array}{l} \forall c, c' \in \mathcal{R}(lc).\ \forall m \in mth(c).\ \forall m' \in mth(c'). \\ c \preceq_{lc} c' \ \wedge \ nm(m) = nm(m') \ \wedge \ at(m) = at(m') \ \wedge \ rt(m) = rt(m') \ \Rightarrow \\ \quad \langle ld(c), at(m)\rangle \ \simeq_{ct} \ \langle ld(c'), at(m')\rangle \ \wedge \ \langle ld(c), rt(m)\rangle \ \simeq_{ct} \ \langle ld(c'), rt(m')\rangle \end{array}$$

*Proof.* Follows from an induction on the length of the relation $c \preceq_{lc} c'$, using the definition of $wfm_{lc}(ct)$. □

The following lemma formally states how the consistency and conformance relations work together to ensure a subclass relation.

**Lemma 12.** *If*

$$\begin{aligned}&cfm_{lc,ct}(o,l,cn) \ \wedge\ \langle l,cn\rangle \in \mathcal{D}(lc) \ \wedge\ css_{lc}(ct) \ \wedge \\ &\langle ld(cl(o)), nm(cl(o))\rangle \in \mathcal{D}(lc) \ \wedge\ lc(ld(cl(o)), nm(cl(o))) = cl(o),\end{aligned}$$

*then* $cl(o) \preceq_{lc} lc(l,cn)$.

*Proof.* Follows directly from the definitions of $cfm_{lc,ct}$, $\lll_{lc,ct}$, $\preceq_{lc}$ and $css_{lc}$. □

## 5.4 The premises for type safety and definedness

The condition $cl(o) \preceq_{lc'} lc'(ld(c), cn)$ in rules (GF), (PF) and (IV) is implied by other conditions for valid states. Now we state a lemma for rule (GF), which implies this. We omit the lemmas and proofs for rules (PF) and (IV), since they are completely similar.

**Lemma 13.** *For rule (GF), if* $vld(\langle stk + \langle c,m,p,os+o\rangle, hp, lc, ct\rangle)$ *and the transition step* $\langle hp, lc, ct\rangle \overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow} \langle hp', lc', ct'\rangle$ *holds and preserves validity, then*

$$\begin{aligned}&cfm_{lc',ct'}(o, ld(c), cn) \ \wedge \\ &\langle ld(c), cn\rangle \in \mathcal{D}(lc') \ \wedge \\ &(\exists c' \in C.\ \exists f \in F.\ fldSel_{lc'}(lc'(ld(c),cn), fn, cn_0) = \langle c', f\rangle \ \wedge \\ &\qquad\qquad \langle ld(c), ld(c'), cn_0\rangle \in ct') \ \wedge \\ &fn \in allfnm_{lc'}(lc'(ld(c),cn)) \ \wedge \\ &o \in \mathcal{D}(hp') \ \wedge\ fn \in \mathcal{D}(hp'(o)) \ \wedge\ cl(o) \preceq_{lc'} lc'(ld(c),cn).\end{aligned}$$

*Proof.* Since $vld(stk + \langle c,m,p,os+o\rangle, hp, lc, ct)$, there are $s,t$ such that

$$bcvst(c,m)|_p = s + t \ \wedge\ cfm_{lc,ct}(o, ld(c), t).$$

By Lemma 10, $cfm_{lc',ct'}(o, ld(c), t)$. We distinguish between two cases:

- If $t = cn$, then $cfm_{lc',ct'}(o, ld(c), cn)$.
- If $t \neq cn$, by the definition of $bcvsr(c,m)$, $\langle t, cn\rangle \in bcvsr(c,m)$. Since $wfm_{lc}(ct)$, $\langle ld(c), t, cn\rangle \in ct$. By Lemma 10, $\langle ld(c), t, cn\rangle \in ct'$. Thus $cfm_{lc',ct'}(o, ld(c), cn)$.

Thus we always have that $\mathit{cfm}_{lc',ct'}(o, ld(c), cn)$.

Since $\mathit{vld}(hp, lc, ct)$ and $\overset{\mathrm{RF}(c,cn,fn,cn_0)}{\Longrightarrow}$ preserves validity, $\mathit{vld}(hp', lc', ct')$. By Lemma 7,

$$\begin{array}{l}\langle ld(c), cn\rangle \in \mathcal{D}(lc') \wedge \\ (\exists c' \in C.\ \exists f \in F.\ \mathit{fldSel}_{lc'}(lc'(ld(c), cn), fn, cn_0) = \langle c', f\rangle \wedge \\ \qquad\qquad \langle ld(c), ld(c'), cn_0\rangle \in ct') \wedge \\ fn \in \mathit{allfnm}_{lc'}(lc'(ld(c), cn))\end{array}$$

Since $\mathit{wfm}_{hp,lc,ct}(stk + \langle c, m, p, os + o\rangle)$, $o \in \mathcal{D}(hp)$. By Lemma 10, $o \in \mathcal{D}(hp')$. Since $\mathit{wfm}_{lc}(hp)$ and $\mathit{wfm}(lc)$, we have that

$cl(o) \in \mathcal{R}(lc)$, $\langle ld(cl(o)), nm(cl(o))\rangle \in \mathcal{D}(lc)$ and $lc(ld(cl(o)), nm(cl(o))) = cl(o)$.

By Lemmas 12 and 10, $cl(o) \preceq_{lc'} lc'(ld(c), cn)$ and thus $fn \in \mathcal{D}(hp'(o))$. □

We consider the property that although some expressions in transition rules contain uses of partial functions and relations, they are always well-defined in the context we are considering.

The property ensures that no implicitly abnormal behavior is possible in the application of a transition rule, even if the well-definedness of uses of partial functions and relations is not checked.

**Lemma 14.** *(Definedness) For each transition rule, if*

- *the input state of the transition in the consequence is a valid state,*
- *all well-defined premises (those that consist of well-defined uses of functions and relations, including all transition premises) are true, and*
- *all transition premises preserve validity,*

*then all uses of functions and relations in the rule are well-defined.*

*Proof.*
- Consider rule (GF). The key is to prove that $hp'(o)(fn)$ is well defined. We need only to prove that $o \in \mathcal{D}(hp') \wedge fn \in \mathcal{D}(hp'(o))$, which follows from Lemma 13.
- For rule (PF), the key is to prove that $hp'(o)$ is well-defined, whose proof is similar to that for Lemma 13.
- The well-definedness of $lc'(ld(c), cn)$ in rules (NE), (IV), (DC), (RC2), (RF) and (RM), and $lc'(ld(c), \mathtt{Cls})$ and $lc'(ld(c), \mathit{Str})$ in rules (DC) and (LD2) follows from Lemmas 5, 6, 7 and 9.
- The proofs for rules (RE), (RC1) and (LD1) are trivial. □

## 5.5 Validity preservation

We give a lemma as a preparation for the theorem on validity preservation.

**Lemma 15.** *(Validity) For each transition rule, if*

*1. the transition in the consequence takes a valid input state,*
*2. all well-defined premises hold, and*
*3. all transition premises preserve validity,*

*then the transition in the consequence produces a valid output state.*

*Proof.* Consider each transition rule. First, by Lemma 14, all applications of functions and relations in the rule are well-defined. Thus all premises are true.

– Consider rule (GF). In order to prove that

$$vld(stk + \langle c, m, p+1, os + hp'(o)(fn)\rangle, hp', lc', ct'),$$

the key is to prove $cfm_{lc',ct'}(hp'(o)(fn), ld(c), cn_0)$.
By Lemma 13, $o \in \mathcal{D}(hp')$, $fn \in \mathcal{D}(hp'(o))$, $cl(o) \preceq_{lc'} lc'(ld(c), cn)$, and there are $c'$ and $f$ such that

$$fldSel_{lc'}(lc'(ld(c), cn), fn, cn_0) = \langle c', f\rangle \;\wedge\; \langle ld(c), ld(c'), cn_0\rangle \in ct'.$$

Since $cl(o) \preceq_{lc'} lc'(ld(c), cn)$, by Lemma 8, $fldSel_{lc'}(cl(o), fn, cn_0) = \langle c', f\rangle$.
Since $cfm_{lc',ct'}(hp')$,

$$fldSel_{lc'}(cl(o), fn, cn_0) = \langle c', f\rangle \;\Rightarrow\; cfm_{lc',ct'}(hp'(o)(fn), ld(c'), cn_0).$$

Thus $cfm_{lc',ct'}(hp'(o)(fn), ld(c), cn_0)$.
– Consider rule (PF). By a proof similar to that for Lemma 13, $o \in \mathcal{D}(hp')$ and $fn \in \mathcal{D}(hp'(o))$. Thus

$$\mathcal{D}(hp'(o)) = \mathcal{D}(hp'(o)[fn \mapsto v]).$$

Hence it is straightforward to check that $wfm_{lc'}(hp'[o \mapsto hp'(o)[fn \mapsto v]])$.
Now to prove that $vld(hp'[o \mapsto hp'(o)[fn \mapsto v]], lc', ct')$, the key is to prove

$$cfm_{lc',ct'}(hp'[o \mapsto hp'(o)[fn \mapsto v]]).$$

The case where $v = \mathsf{null}$ is trivial. We assume that $v \neq \mathsf{null}$.
By a proof similar to that for Lemma 13, there are $c' \in C$ and $f \in F$ such that

$$fldSel_{lc'}(lc'(ld(c), cn), fn, cn_0) = \langle c', f\rangle \;\wedge\; \langle ld(c), ld(c'), cn_0\rangle \in ct'.$$

By anohter proof similar to that for Lemma 13, $cfm_{lc',ct'}(v, ld(c), cn_0)$ and $cl(v) \preceq_{lc'} lc'(ld(c), cn_0)$.
Since $cfm_{lc',ct'}(v, ld(c), cn_0)$ and $\langle ld(c), ld(c'), cn_0\rangle \in ct'$, $cfm_{lc',ct'}(v, ld(c'), cn_0)$.
Since $cl(o) \preceq_{lc'} lc'(ld(c), cn)$ and $fldSel_{lc'}(lc'(ld(c), cn), fn, cn_0) = \langle c', f\rangle$, by Lemma 8, $fldSel_{lc'}(cl(o), fn, cn_0) = \langle c', f\rangle$. Since $cfm_{lc',ct'}(hp')$, we have that $cfm_{lc',ct'}(hp'[o \mapsto hp'(o)[fn \mapsto v]])$.

– The proof for rule (NE) is straightforward. Note that $o \notin C$ and $cl(o) = lc'(ld(c), cn)$ imply that $lc'(ld(c), cn) \neq Cls$.

– Consider rule (IV). Recall that $bcvst(c', m')|_0 = [nm(c'), cn_1]$. To prove that $vld(stk + \langle c, m, p+1, os\rangle + \langle c', m', 0, [o, v]\rangle, hp', lc', ct')$, the key is to prove that

$$\begin{array}{l} c' \in \mathcal{R}(lc') \wedge \\ cfm_{lc',ct'}(o, ld(c'), nm(c')) \wedge \\ cfm_{lc',ct'}(v, ld(c'), cn_1) \wedge \\ \langle ld(c), cn_0\rangle \ \simeq_{ct'} \ \langle ld(c'), cn_0\rangle. \end{array}$$

Since $wfm_{hp,lc,ct}(stk + \langle c, m, p, os + o + v\rangle)$ and $wfm_{lc}(hp)$, $cl(o) \in \mathcal{R}(lc)$. By Lemma 10, $cl(o) \in \mathcal{R}(lc')$. Since $mthSel_{lc'}(cl(o), mn, cn_1, cn_0) = \langle c', m'\rangle$, by Lemma 4, $c' \in \mathcal{R}(lc')$ and $cl(o) \preceq_{lc'} c'$.
Since $wfm(lc')$, $(\forall c \in \mathcal{R}(lc').\ \langle ld(c), nm(c)\rangle \in \mathcal{D}(lc') \wedge lc'(ld(c), nm(c)) = c$. Since $cl(o), c' \in \mathcal{R}(lc')$, $\langle ld(cl(o)), nm(cl(o))\rangle \lll_{lc',ct'} \langle ld(c'), nm(c')\rangle$, and thus $cfm_{lc',ct'}(o, ld(c'), nm(c'))$.
Since $vld(stk + \langle c, m, p, os + o + v\rangle, hp, lc, ct)$, there are $s, t, t_1$ such that

$$bcvst(c, m)|_p = s + t + t_1 \ \wedge \ cfm_{lc,ct}(o, ld(c), t) \ \wedge \ cfm_{lc,ct}(v, ld(c), t_1).$$

- If $t = cn$, then $cfm_{lc,ct}(o, ld(c), cn)$.
- If $t \neq cn$, $\langle t, cn\rangle \in bcvsr(c, m)$. Since $wfm_{lc}(ct)$, $\langle ld(c), t, cn\rangle \in ct$. Thus $cfm_{lc,ct}(o, ld(c), cn)$.

By a similar analysis, $cfm_{lc,ct}(v, ld(c), cn_1)$. By Lemma 10, $cfm_{lc',ct'}(o, ld(c), cn)$ and $cfm_{lc',ct'}(v, ld(c), cn_1)$.
By Lemma 9, $\langle ld(c), cn\rangle \in \mathcal{D}(lc')$ and there are $c''$ and $m''$ such that

$$\begin{array}{l} mthSel_{lc'}(lc'(ld(c), cn), mn, cn_1, cn_0) = \langle c'', m''\rangle \wedge \\ \langle ld(c), ld(c''), cn_1\rangle, \langle ld(c), ld(c''), cn_0\rangle \in ct'. \end{array}$$

Since $cfm_{lc',ct'}(o, ld(c), cn) \ \wedge \ \langle ld(c), cn\rangle \in \mathcal{D}(lc') \ \wedge \ css_{lc'}(ct')$, by Lemma 12, $cl(o) \preceq_{lc'} lc'(ld(c), cn)$. Since

$$\begin{array}{l} clos(lc') \wedge \\ mthSel_{lc'}(cl(o), mn, cn_1, cn_0) = \langle c', m'\rangle \wedge \\ mthSel_{lc'}(lc'(ld(c), cn), mn, cn_1, cn_0) = \langle c'', m''\rangle, \end{array}$$

$c' \preceq_{lc'} c''$.
By Lemma 11,

$$\langle ld(c'), cn_1\rangle \ \simeq_{ct'} \ \langle ld(c''), cn_1\rangle \text{ and } \langle ld(c'), cn_0\rangle \ \simeq_{ct'} \ \langle ld(c''), cn_0\rangle.$$

Since $\langle ld(c), ld(c''), cn_1\rangle \in ct'$, we have that $\langle ld(c'), cn_1\rangle \ \simeq_{ct'} \ \langle ld(c), cn_1\rangle$.
Since $cfm_{lc',ct'}(v, ld(c), cn_1)$, we have that $cfm_{lc',ct'}(v, ld(c'), cn_1)$.
Finally, since $\langle ld(c), ld(c''), cn_0\rangle \in ct'$, $\langle ld(c), cn_0\rangle \ \simeq_{ct'} \ \langle ld(c'), cn_0\rangle$.

– Consider rule (RE). By definition, there is $s'$ such that $bcvst(c', m')|_{p'} = s' + rt(m)$. The key is to prove

$$cfm_{lc,ct}(v, ld(c'), rt(m)).$$

The case where $v = \mathsf{null}$ is trivial. We assume that $v \neq \mathsf{null}$.
Since $vld(stk + \langle c', m', p', os' \rangle + \langle c, m, p, os + v \rangle, hp, lc, ct)$, there are $s, t$ such that

$$\begin{array}{l} bcvst(c,m)|_p = s + t \wedge \\ cfm_{lc,ct}(v, ld(c), t) \wedge \\ (t \neq rt(m) \Rightarrow \langle ld(c), t, rt(m) \rangle \in ct), \end{array}$$

and there is $cn$ such that

$$\begin{array}{l} cd(m')|_{p'-1} = \mathsf{invokevirtual}(cn, nm(m), at(m), rt(m)) \wedge \\ \langle ld(c), rt(m) \rangle \simeq_{ct} \langle ld(c'), rt(m) \rangle. \end{array}$$

Now it is straightforward to see that no matter $t = rt(m)$ or not, we have that $cfm_{lc,ct}(v, ld(c), rt(m))$, thus $cfm_{lc,ct}(v, ld(c'), rt(m))$.
– Consider rule (DC). A large part of the proof is similar to that in the above for rule (IV). The rest is straightforward.
Note that the transition $\overset{\mathrm{RC}(c', sup(c'))}{\Longrightarrow}$ ensures that $clos(lc''[\langle l, nm(c') \rangle \mapsto c'])$.
The definition of $addLc(lc'', l, nm(c'), c')$ ensures that

$$\forall c \in \mathcal{R}(lc'''). \; allfnm_{lc'''}(lc'''(ld(c), sup(c))) \cap fnm(c) = \emptyset.$$

The proof of $cl(ld(c')) \preceq_{lc'''} ClLd$ in the proof of $wfm(lc''')$ needs some explanation. First, $vld(\langle stk + \langle c, m, p, os + l + o \rangle, hp, lc, ct \rangle)$, implies that $cfm_{lc,ct}(l, ld(c), cn)$. Since

$$\langle ld(c), cn \rangle \in \mathcal{D}(lc') \;\wedge\; lc'(ld(c), cn) \preceq_{lc'} ClLd,$$

$cl(l) \preceq_{lc'} ClLd$. Since $l = ld(c')$, $cl(ld(c')) \preceq_{lc'} ClLd$. By Lemma 10, we have that $cl(ld(c')) \preceq_{lc'''} ClLd$.
– The proofs for rules (RC1), (RC2) and (LD1) are trivial.
– Consider rule (LD2). Since $mthSel_{lc}(cl(l), \mathtt{ldCl}, \mathtt{Str}, \mathtt{Cls}) = \langle c, m \rangle$, by Lemma 4, $cl(l) \preceq_{lc} c$ and thus $cfm_{lc,ct}(l, ld(c), nm(c))$. By Lemma 10, we have that $cfm_{lc',ct'}(l, ld(c), nm(c))$.
By Lemma 6, $\langle ld(c), \mathtt{Str} \rangle \in \mathcal{D}(lc')$. Since $lc'(ld(c), \mathtt{Str}) = Str$, we have that $cfm_{lc',ct'}(cn, ld(c), \mathtt{Str})$. Hence

$$vld(\langle [\langle c, m, 0, [l, cn] \rangle], hp'[cn \mapsto \{\}], lc', ct' \rangle).$$

Since the transitions steps in the premise preserve validity,

$$vld(\langle [\langle c, m, p, os + c' \rangle], hp'', lc'', ct'' \rangle).$$

Since $wfm_{hp'',lc'',ct''}([\langle c, m, p, os + c' \rangle])$, $c' \in \mathcal{D}(hp'')$. Since $wfm_{lc''}(hp'')$, $c' \in \mathcal{R}(lc'')$. Since $css_{lc''\{\langle l, cn \rangle \mapsto c'\}}(ct'')$,

$$vld(\langle hp'', lc''\{\langle l, cn \rangle \mapsto c'\}, ct'' \rangle).$$

– The proof for rules (RF) and (RM) are straightforward. □

Now we can prove the theorem on validity preservation.

**Proof of Theorem 1**. By definition, if $stt \Longrightarrow stt'$, then there is a finite (proof) tree of applications of the rules with $stt \Longrightarrow stt'$ as the root. The assertion follows by applying Lemma 15 to each application of a rule in each path from a leaf to the root, in that order. □

## 5.6 Labeled transitions with real loaders

As a consequence of Theorem 1, we show that in building a proof tree, it is sufficient to consider only those labeled transitions that are associated with a real loader, in particular, those transitions $\overset{\mathrm{LD}(l,cn)}{\Longrightarrow}$ where $l$ is a real loader.

First, we consider individual transition rules.

**Lemma 16.** *For each transition rule, if*

- *the input state of the transition in the consequence is a valid state,*
- *whenever the transition in the consequence is a labeled transition, it is associated with a real loader,*
- *all premises are true, and*
- *all transition premises preserve validity,*

*then all labeled transitions premises are associated with a real loader.*

*Proof.* – The labeled transition premises of rules (GF), (PF), (NE) and (IV), as well as the labeled transitions $\overset{\mathrm{RM}(c,cn,\mathtt{dfCl},\mathtt{Obj},\mathtt{Cls})}{\Longrightarrow}$ and $\overset{\mathrm{RC}(c,\mathtt{Cls})}{\Longrightarrow}$ in rule (DC), are associated with a real loader, since the valid input state of the transition in the consequence implies that

$$cl(ld(c)) \preceq_{lc} ClLd.$$

– Consider $\overset{\mathrm{RC}(c',sup(c'))}{\Longrightarrow}$ in rule (DC). Since the input state of the transition in the consequence is valid, $cfm_{lc,ct}(l, ld(c), cn)$. By Lemma 10, $cfm_{lc',ct'}(l, ld(c), cn)$. By Lemma 6, $\langle ld(c), cn\rangle \in \mathcal{D}(lc')$. Since the transitions $\overset{\mathrm{RM}(c,cn,\mathtt{dfCl},\mathtt{Obj},\mathtt{Cls})}{\Longrightarrow}$ and $\overset{\mathrm{RC}(c,\mathtt{Cls})}{\Longrightarrow}$ preserve validity,

$$cl(l) \preceq_{lc'} lc'(ld(c), cn).$$

Since $lc'(ld(c), cn) \preceq_{lc'} ClLd$ and $l = ld(c')$ are premises, (the latter is in the definition of $crCl1$,) $cl(ld(c')) \preceq_{lc'} ClLd$. Thus the transition $\overset{\mathrm{RC}(c',sup(c'))}{\Longrightarrow}$ is associated with a real loader.

– No proofs are needed for rules (RE), (RC1) and (LD1).
– The proofs for rules (RC2), (RF) and (RM) are trivial.

– The labeled transitions in rule (LD2) are associated with a real loader, since the input state of the transition in the consequence is valid, and the premise $\mathit{mthSel}_{lc}(\mathit{cl}(l), \mathtt{ldCl}, \mathtt{Str}, \mathtt{Cls}) = \langle c, m\rangle$ implies that $c \in \mathcal{R}(lc)$. $\square$

**Proof of Theorem 2**. Consider a finite proof tree for $stt \Longrightarrow stt'$ with $stt \Longrightarrow^* stt'$ as the root. By Theorem 3, all transitions preserve validity. Now the assertion follows by applying Lemma 16 to each application of a rule in each path from the root to a leaf, in that order. $\square$

## 5.7 Safety

The next lemma below implies that in rule (NE), if the condition

$$o \in O - (C \cup \mathcal{D}(hp')) \ \wedge \ cl(o) = lc'(ld(c), cn)$$

does not hold, then $lc'(ld(c), cn) = \mathit{Cls}$. In reality, the case would correspond to the creation of a class object using a `new` instruction, something Java disallows.

**Lemma 17.** *For any $hp' \in Hp$, there is always $o \in O - (C \cup \mathcal{D}(hp'))$. For $c \in C$, if there is no $o \in O - (C \cup \mathcal{D}(hp'))$ with $cl(o) = c$, then $c = \mathit{Cls}$.*

The next lemma implies that in rule (DC), there is always $c' \in C$ satisfying the condition $\mathit{crCl}_{hp'}(l, c')$.

**Lemma 18.** *For any $l \in L$ and $hp \in Hp$, there is always $c \in (C - \mathcal{D}(hp))$ with $l = ld(c)$.*

**Proof of Theorem 3**. In cases 1 and 2, by Lemma 13, rule (GF) or (PF) can be applied, and the assertion of the theorem holds. In case 3, by Lemma 17, rule (NE) can be applied, and the assertion of the theorem follows. In case 4, rule (RE) can be applied, and the assertion of the theorem trivially follows. In case 5, by Lemma 9 and a lemma for rule (IV) similar to Lemma 13, rule (IV) can be applied, and the assertion of the theorem holds. In case 6, by Lemma 18, rule (DC) can be applied, and the assertion of the theorem holds. $\square$

To conclude this section, we emphasize that a successful transition step as defined by our rules corresponds to a terminating execution. Non-terminating execution corresponds to infinite proof trees. An infinite proof tree may have an infinite width or height. Infinite width corresponds to the case where the transition sequence premise of rule (LD2) is infinite. Infinite height corresponds to the case where there are infinitely nested applications of transition rules, each for a transition premise of a previous transition rule.

## 6 Related Work

The requirement that loaders in addition to class names are needed to uniquely identify class objects is mentioned in the Java language specification [12]. But it was not completely understood what the concrete mechanism should be. The bugs reported by Saraswat [17], Tozawa and Hagiya [18] and ourselves [4] are evidence of this. Saraswat proposed a solution where method overriding is based on full types instead of names only. However his solution may cause counter-intuitive dynamic dispatch and requires a modification to the class loaders' API to avoid premature class loading.

Dean [5] presented probably the first formal model for Java class loading, focusing on the static typing, using the PVS verification system [8]. The model is clean and abstract, but does not consider loading and subtype constraints.

Jensen, LeMetayer and Thorn [13] proposed an abstract formalization of Java class loading, and showed how the Saraswat bug is disallowed by the formalization. Their formalization pre-dates Sun's response to the Saraswat bug and differs in some aspects from the official semantics of the JVM [2].

Goldberg [10] formalized a way to integrate some aspects of class loading into bytecode verification. The idea is that the bytecode verifier does not load classes to insure type safety, but generates constraints that are checked when the referenced classes are loaded. He did not consider multiple loaders. Our approach includes and generalizes this idea.

Liang and Bracha [14] introduced the concepts of loading constraints and loaded class cache. Their setting is informal, and thus it is hard to verify their claims, or see where the problems are and how they are solved. This solution is also documented in [15]. A good informal account of class loading can be found in [11]. However, that account often does not distinguish between the JDK 1.2 implementation and the JVM specification.

Fong and Cameron [7] proposed a general, modular architecture for mobile-code loading and verification, and discussed a possible instantiation for Java loading and bytecode verification. In particular, their concept of proof obligations corresponds to our concepts of constraints. However, they do not consider multiple loaders.

Börger and Schulte [1] presented a fairly complete operational semantics of the JVM which includes loading. The work follows the well-established approach of Abstract State Machines and can take advantage of existing machine-supported simulation tools. Their model of loading considers exceptions, but does not consider the loaded class cache and loading and subtype constraints.

Tozawa and Hagiya's formal model for a type safe JVM [19] is closely related to ours, although their model and ours were developed independently. Some of the components considered in their model are also considered in ours. For example, their loaded class caches and loading constraints are very similar to ours, and their concept of widening conversion roughly corresponds to our concept of subtype constraints. There are two main conceptual differences between their model and ours. The first is that they do not consider subtype constraints: their widening conversion is checked at verification time, which means that the involved classes must be loaded. This corresponds to Sun's eager loading strategy for verification. The second is that they do not consider execution of code in user-defined loaders. Furthermore, they take a different overall approach. They define an operational semantics that depends on an "environment", which consists of loaded classes, objects in the heap, etc. They define judgements (such as widening conversion) holding in an environment, and they prove some monotonicity properties for judgements w.r.t. extending environments (e.g., as per loading of new classes). In our approach, instead, the information about loaded classes, heap, etc. is part of the state, and there are explicit state transitions that load new classes, create new objects, and so on. Therefore, we believe our model is more intuitive and closer to the JVM specification and implementations.

Drossopoulou [6] developed a model for Java class loading, focusing at an abstract level on the interface between execution, loading and verification. Her work introduces and analyzes a high-level language, which is closer to Java than the JVM. Her model does not consider multiple loaders, while ours does, but her work handles exceptions, while ours does not.

Type-safe loading has also been studied in static settings both for high-level modules (e.g. [3]) and for assembly languages (e.g. [9]). In these settings, the major difference between the dynamic and static loading is that the final steps of loading and the formation of the "real" executable are delayed until loading-time [9]. The mechanism for dynamic loading in Java is more complex, since it is possible that a class that is used in the static type inference never gets loaded, and that a loader itself is a user-defined program and needs to be loaded at run time.

## 7 Conclusion

The mechanisms for Java class loading and bytecode verification are complex. It is hard to ascertain properties such as type safety using an informal specification. We have presented a formalization of class loading and formally proved type safety.

Our formalization addresses most key issues mentioned in [12,15,14] and in Sun's current implementation. In addition, we have introduced subtype constraints so that the bytecode verifier does not need to resolve any class, thus avoiding premature loading and having a cleaner interaction with the rest of the machine.

Our formalization models a simplified JVM. It includes essential internal data structures such as the loaded class cache, loading and subtype constraints. It supports selected language features such as classes, objects, methods, and dynamic and lazy loading, but not interfaces, arrays, primitive types, access control, exception handling, garbage collection and multi-threading. So far we have not seen any fundamental problems to extending our formalization to address the missing data structures and features.

**Acknowledgements**

The research has been partially supported by DARPA contracts F30602-96-C0363 and F30602-99-C-0091. We sincerely thank Gilad Bracha for numerous discussions, clarifications and comments on early versions of the paper. We also thank Sheng Liang for several useful discussions, and Stephen Fitzpatrick for comments on a short version of the paper.

## References

1. E. Börger and W. Schulte. Modular design for the Java virtual machine architecture. `ftp://ftp.di.unipi.it/pub/Papers/boerger/jvmarch.ps`, 1999.
2. G. Bracha. A critique of 'Security and dynamic loading in Java: A formalisation'. `http://java.sun.com/people/gbracha /critique-jmt.html`, 1999.
3. L. Cardelli. Program fragments, linking, and modularization. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 266–277, 1997.
4. A. Coglio and A. Goldberg. Type safety in the JVM: Some problems in JDK 1.2.2 and proposed solutions. In *Proc. ECOOP Workshop on Formal Techniques for Java Programs*, 2000. Long version available at `http://www.kestrel.edu/java`.
5. D. Dean. The security of static typing with dynamic linking. In *Proc. 4th ACM Conf. on Computer and Communications Security*. ACM Press, 1997.
6. S. Drossopoulou. Towards an abstract model of Java dynamic linking and verification. Department of Computing, Imperial College, London, UK.
7. P. Fong and R. Cameron. Proof linking: An architecture for modular verification of dynamically-linked mobile code. In *Proc. 6th ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering (FSE'98)*, 1998.
8. Formal Methods Program - SRI Computer Science Laboratory. The PVS specification and verification system. `http://pvs.csl.sri.com/`, 1999.
9. N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proc. 26th ACM Symp. Principles of Programming Languages*, 1999.
10. A. Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security*, 1998.
11. L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley, 1999.
12. J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification.* Addison-Wesley, 1996.
13. T. Jensen, D. LeMetayer, and T. Thorn. Security and dynamic class loading in Java: a formalisation. In *Proc. IEEE Int. Conference on Computer Languages*, 1998.
14. S. Liang and G. Bracha. Dynamic class loading in the Java™ virtual machine. In *Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44. ACM Press, 1998.

15. T. Lindholm and F. Yellin. *The Java$^{TM}$ Virtual Machine Specification - 2nd edition*. Addison-Wesley, 1999.
16. Z. Qian, A. Goldberg, and A. Coglio. A formal specification of Java™ class loading. In *Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2000. To appear.
17. V. Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. `http://www.research.att.com/~vj/bug.html`.
18. A. Tozawa and M. Hagiya. Careful analysis of type spoofing. In *JIT'99 Java-Informations-Tage 1999, Clemens H. Cap, Hrsg., Informatik aktuell*, pages 290–296. Springer Verlag, 1999.
19. A. Tozawa and M. Hagiya. New formalization of the JVM. `http://nicosia.is.s.u-tokyo.ac.jp/members/miles/papers/cl-99.ps`, 1999.