Type Safety in the JVM: Some Problems in JDK 1.2.2 and Proposed Solutions

Alessandro Coglio and Allen Goldberg

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304
{coglio, goldberg}@kestrel.edu

1 Introduction

We are currently developing mathematical specifications for various components of the JVM, including the bytecode verifier [2, 4, 7], the class loading mechanism [8], and the Java 2 security mechanisms. We are also deriving a complete implementation of the bytecode verifier [2] through Specware [10], a system developed at Kestrel Institute that supports provably correct, compositional development of software from formal specifications.

In the course of our formalization efforts, we have uncovered subtle bugs in Sun JDK 1.2.2 that lead to type safety violations. These bugs are in the bytecode verifier and relate to the naming of reference types. We found that in certain circumstances these names can be spoofed by suitable use of delegating class loaders. Since the JVM specification [6] is informal English prose, we cannot crisply characterize these bugs as errors in the specification or just in one or more implementations. However, some of these bugs are consistent with a reasonable interpretation of the specification. We have verified that the bugs exist in Sun JDK 1.2.2 (on both Solaris and Windows NT). Some are fixed in Sun JDK 1.3 Beta by restricting access to system packages.

Overall, these flaws raise the issue of a more precise specification of the bytecode verifier and the loading mechanisms, and increased assurance of type safety properties. Besides fixes for all these bugs, we have devised a more general approach to insuring type safety that has additional advantages, including lazier class loading.

2 Types in the JVM

According to [6], a class in the JVM is identified by its fully qualified name (FQN) plus its defining loader. In fact, classes are in correspondence with instances of class `java.lang.Class`, and the only way to create new `Class` instances is through the `defineClass` methods of class `java.lang.ClassLoader`. These methods call internal JVM code that carries out actual class creation from byte arrays in `classfile` format. This code enforces the constraint that a class loader cannot create two classes with a same FQN. Thus it is possible to identify a class by its FQN plus its defining loader.

However, the bytecode verifier, when verifying a class, essentially uses just FQNs. In a few cases, it actually resolves class names and makes use of the `Class` instances they resolve to. In particular, the bytecode verifier sometimes needs to *merge*, that is find the first common superclass of two class names. The two class names are resolved, thus loading the classes and their superclasses, and their ancestry searched to find their first common superclass. The bytecode verifier also resolves names to check assignment compatibility (i.e., subtype relationship) between two class names.

The use of FQNs and occasional use of actual classes guarantee type safety only under certain assumptions. Examples of these assumptions are the loading constraints introduced in the Java 2 Platform [6, 5] to avoid the type safety problems exactly arising from the violation of the assumptions they enforce [9]. Simply stated, loading constraints ensure that classes exchanging objects (through their methods and fields) agree on the actual types (and not only on the FQNs) of such objects.

As it turns out, loading constraints do not cover all the assumptions needed to guarantee type safety. An example occurs when checking a stack position that contains the type (FQN) that results from merging two classes: the bytecode verifier assumes that the FQN in the stack position resolves (using the defining loader of the analyzed class as the initiating loader) to the actual first common superclass. Another example occurs when checking type constraints for the `invokespecial` instruction: the bytecode verifier assumes that an FQN of any superclass of the current class resolves to the actual superclass. Furthermore the bytecode verifier assumes that the FQNs `java.lang.Object` and `java.lang.String` resolve to the "usual" system classes. It is in fact possible to construct programs where these assumptions are violated, thus causing name spoofing and type safety failures. For reasons of space, here we only describe one of the bugs. Further details, including runnable programs, can be found in [1].

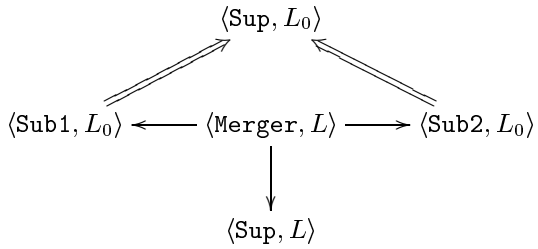
As in [6] and [5] we use the notation N^L to denote the class associated to name N and loader L when loading of N is initiated by L, i.e., L is an initiating loader of N. Furthermore $\langle N, L\rangle$ denotes the (unique) class with FQN N and defining loader L.

3 The merging bug

[6, Sect. 4.9.2] describes how, during data flow analysis of a method's code, the types assigned to stack positions and local variables along different control paths are merged. In order to merge two distinct class names `Sub1` and `Sub2`, the corresponding classes are loaded by invoking the `loadClass` method of the defining loader L of the class whose method is being verified, with argument `Sub1` first, then `Sub2`. The ancestry of the classes $\texttt{Sub1}^L$ and $\texttt{Sub2}^L$ is then searched to find the first common superclass. If the first common superclass found is $\langle \texttt{Sup}, L_0\rangle$ then the bytecode verifier writes the FQN `Sup` in the merged stack position. Suppose that, after the merging point, an instruction accesses a field or method of a class named `Sup` in the field or method reference. Since

the bytecode verifier has deduced that the stack position indeed contains a class with FQN `Sup`, the check for assignment compatibility will succeed, as described in [6, Sect. 4.9.1]. This is correct only assuming that $\texttt{Sup}^L = \langle \texttt{Sup}, L_0 \rangle$, i.e., that loading of FQN `Sup` initiated by L results in the actual superclass of $\texttt{Sub1}^L$ and $\texttt{Sub2}^L$.

However, such an assumption can be violated if for example L_0 is the system class loader and L is a user-defined class loader that delegates to L_0 (by invoking `findSystemClass`) the loading of FQNs `Sub1` and `Sub2` but not of `Sup`. The following situation is arranged (`Merger` is the class being verified):



A thin arrow from a class (identified by its name and defining loader) to another indicates that the source of the arrow resolves the FQN of the target to the target. A double arrow indicates that the source is a subclass of the target. Suppose that class `Merger` contains the following code:

```
Sub1 s1 = new Sub1();
Sub2 s2 = new Sub2();
Sup s;
if (s1 != null)  s = s1;  // this test just serves to
else  s = s2;             // create two merging paths
s.m();  // type unsafe!
```

This code passes verification for the reason described above. If $\langle \texttt{Sup}, L \rangle$ has a method `m` with the right descriptor, at runtime the method call goes through because `Sup` resolves to $\langle \texttt{Sup}, L \rangle$. However, the object stored in `s` has class $\langle \texttt{Sup}, L_0 \rangle$ (as well as class $\langle \texttt{Sub1}, L_0 \rangle$). If $\langle \texttt{Sup}, L_0 \rangle$ is different from $\langle \texttt{Sup}, L \rangle$, the effect of the method call is undefined. In typical implementations, it will probably call some unrelated method that happens to have the same index, thus causing type unsafety.

This may be interpreted as a bug in the JVM specification, rather than the implementation. Although [6] does not crisply state that types are denoted by FQNs in the bytecode verifier (it typically just talks about "reference types"), that seems to be the intended meaning, or at least the most reasonable interpretation. In any case, future editions of [6] should clarify this point. This bug also exists in JDK 1.3 Beta.

A possible solution to the problem is to keep information, when merging two FQNs `Sub1` and `Sub2`, about the actual first common superclass $\langle \texttt{Sup}, L_0 \rangle$ (not only its FQN `Sup`). When checking assignment compatibility with the FQN `Sup`

(referenced in the runtime constant pool), the FQN is resolved and the resulting `Class` instance is compared with the one obtained from merging. In this way there can be no confusion. Interestingly, inspection of the bytecode verifier code in JDK 1.2.2 shows that information about the actual first common superclass is indeed maintained and accessible. However, it is not used to prevent this problem. Alternatively, to avoid early loading of `Sup` by L, a loading constraint $\texttt{Sup}^{L} = \texttt{Sup}^{L_0}$ can be added by the bytecode verifier to the set of globally maintained loading constraints.

4 A general solution

As previously mentioned, the bytecode verifier makes use of FQNs, occasionally resolving them to actual classes. This resolution results in premature loading of classes. We now propose a design for the bytecode verifier (and related parts of the JVM) that (1) avoids premature loading and (2) allows a cleaner separation between bytecode verification and loading. This cleaner separation also promotes a better understanding of how bytecode verification and other mechanisms (such as loading constraints) cooperate to insure type safety in the JVM.

In the design we propose, the bytecode verifier uniformly uses FQNs, never actual classes. The intended disambiguation is that FQN N stands for class N^L, where L is the defining loader of the class under verification (note that, at verification time, class N^L might not be present in the JVM yet). The bytecode verifier never causes resolution (and loading) of any class.

The result of merging two FQNs is a set containing the two FQNs. More precisely, the bytecode verifier uses (finite) sets of FQNs (and not just FQNs) to type stack positions and local variables containing reference types [2, 4, 7]. Initially (e.g., in the local variables containing method invocation arguments) sets are singletons. Merging is set union. The meaning of a set of FQNs typing a local memory is that the local memory may contain an instance of a class whose FQN is in the set. No relationship among the elements of the set is intended.

When a set of FQNs is checked for assignment compatibility with a given FQN N, for each element M of the set different from N, a subtype loading constraint $M^L < N^L$ is generated. The meaning of such constraint is that class M^L must be a subclass of class N^L. The constraint is added to the global state of the JVM, and checked for consistency with the loaded class cache. If either class has not been loaded yet, the constraint is just recorded. Whenever the loaded class cache is updated, it is checked for consistency with the current subtype loading constraints. This is very similar to the equality loading constraints of the form $N^L = N^{L'}$ introduced in the Java 2 Platform. In fact, subtype constraints complement equality constraints.

Checking the consistency of the loaded class cache and loading constraints that include both subtype constraints and equality constraints is neither difficult nor inefficient. A naïve algorithm will transitively close both subtype and equality constraints and then check that when the loaded class cache is updated none of the constraints in the transitive closure is violated. An efficient algorithm will

use a union-find data structure to store equivalence classes of classes asserted to be the same and track the asserted subtype dependencies of the classes.

In this design, the result of bytecode verification of a class is therefore not just a yes/no answer, but also a set of subtype constraints that explicitly and clearly express the assumptions made by the bytecode verifier to certify the class. Furthermore, the bytecode verifier is a well-defined, purely functional piece of the JVM that does not depend on the current state of JVM data structures.

Let us now see how this approach avoids the merging bug. When verifying the code in `Merger`, the creation (and initialization) of the two instances of class `Sub1` and `Sub2` has the effect of typing the local variables as $\{\texttt{Sub1}\}$ and $\{\texttt{Sub2}\}$. After the merging point, the type on top of the stack is $\{\texttt{Sub1}, \texttt{Sub2}\}$. Since the call of method `m` references class `Sup` (through the costant pool), subtype constraints $\texttt{Sub1}^L < \texttt{Sup}^L$ and $\texttt{Sub2}^L < \texttt{Sup}^L$ are generated. When the code is eventually executed, before the method is called all of $\texttt{Sub1}^L$, $\texttt{Sub2}^L$, and $\texttt{Sup}^L$ will have been loaded. Since subtype constraints are violated, the JVM will throw an exception preventing resolution of the method (and therefore its invocation).

Our approach also allows a cleaner treatment of interface types in the bytecode verifier. Since an interface can have more than one superinterface, two given interfaces may not have a unique first common superinterface. According to [6], the result of merging two interface FQNs is therefore `java.lang.Object`, which is indeed a superclass of any interface. However, this requires a special treatment of `java.lang.Object` when checking its assignment compatibility with an interface FQN: the bytecode verifier just passes the check because `java.lang.Object` might derive from merging interfaces, even though `java.lang.Object` itself is not assignment-compatible with an interface. This "looseness" does not cause type unsafety because the `invokeinterface` instruction performs a search of the methods declared in the runtime class of the object on which it is executed. If no method matching the referenced descriptor is found, an exception is thrown. This runtime check does not impose any additional runtime penalty. Our scheme is cleaner in that it provides a uniform treatment of classes and interfaces.

In [8] we provide formal arguments that this design of the bytecode verifier, together with (subtype and equality) loading constraints, guarantees type safety in the JVM. In that paper we formalize the operational semantics of a simplified JVM that includes class loading, resolution, bytecode verification, and execution of some instructions, and we prove type safety results about it.

Our approach of having a self-contained bytecode verifier that generates constraints is similar in spirit to [3]. However, they do not consider multiple class loaders. Their bytecode verifier generates, besides subtype constraints, several other kinds of constraints, e.g. for fields and methods referenced in the code being verified. We only generate subtype constraints because the others can be checked at runtime (as specified in [6]) without performance penalty or premature loading.

References

1. Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in JDK 1.2.2 and proposed solutions. `http://www.kestrel.edu/java`, 2000.
2. Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Towards a provably-correct implementation of the JVM bytecode verifier. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.
3. Philip W. L. Fong and Robert D. Cameron. Proof linking: An architecture for modular verification of dynamically-linked mobile code. *ACM SIGSOFT Software Engineering Notes*, 23(6):222–230, November 1998. Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering.
4. Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security*, 1998.
5. Sheng Liang and Gilad Bracha. Dynamic class loading in the Java™ virtual machine. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44. ACM Press, 1998.
6. Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
7. Zhenyu Qian. A formal specification of Java™ virtual machine instructions for objects, methods and subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java™*. LNCS 1523, Springer-Verlag, 1998.
8. Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. `http://www.kestrel.edu/java`, 2000.
9. Vijay Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. `http://www.research.att.com/vj/bug.html`.
10. Yellamraju Srinivas and Richard Jüllig. Specware: Formal support for composing software. In B. Moeller, editor, *Proceedings of the Conference on Mathematics of Program Construction*, pages 399–422. LNCS 947, Springer-Verlag, Berlin, 1995.