Java bytecode verification is not possible
(Extended Abstract)

Robert F. Stärk
Computer Science Department
ETH Zurich, CLV C4
CH–8092 Zurich
staerk@inf.ethz.ch

Joachim Schmid
Siemens Corporate Technology
D-81730 Munich
joachim.schmid@mchp.siemens.de

During an attempt to prove that our Java compiler in [6] generates code that is accepted by the Java Bytecode Verifier we found examples of legal Java programs which are rejected by any Bytecode Verifier. The examples show that Java Bytecode Verification as it has been introduced by Sun is not possible. We propose therefore to restrict the so-called rules of definite assignment for the try-finally statement as well as for the labeled statement such that our example programs are no longer allowed. Then we can prove, using the framework of Abstract State Machines (see [3, 1]), that each program from the restricted Java language is accepted by the Java Bytecode Verifier.

The programs `Test1` and `Test2` in Table 1 are legal Java programs. When they are compiled using a standard Java compiler like Sun's JDK 1.2 or 1.3 compiler, then the generated bytecode is rejected by any bytecode verifier we tried including JDK 1.2, JDK 1.3, Netscape 4.73–4.76, Microsoft VM for Java 5.0 and 5.5 and the Kimera Verifier [5]. The problem is that in the eyes of the verifier the variable `i` is unusable at the end of the method at the `return i` instruction, whereas according to the JLS [2, §16.2.14] the variable `i` is definitely assigned after the try-finally statement. The two programs `Test1` and `Test2` cannot be verified, because the class of valid Java programs is too large for verifiers.

Our solution to the problem is to restrict the so-called "rules of definite assignment" for the try-finally and the labeled statement [2, §16.2.14 and §16.2.5]. The two restrictions of the rules of definite assignment are sufficient to prove the following theorem:

Theorem 1 *If a Java compiler satisfies the constraints of Part II in [6], then the bytecode it generates from valid Java programs of the restricted Java language will be accepted by a correct bytecode verifier.*

A compiler satisfying the constraints of Part II in [6] must compile boolean test expressions in a special way taking care of the boolean literals `true` and `false`. It must compile try-finally statements as it is described in [4, §7.13].

<pre>class Test1 {
  int m1(boolean b) {
    int i;
    try {
      if (b) return 1;
      i = 2;
    } finally {
      if (b) i = 3;
    }
    return i;
  }
}</pre>
	<pre>class Test2 {
	  int m2(boolean b) {
	  int i;
	  L: { try {
	         if (b) return 1;
	         i = 2;
	         if (b) break L;
	       } finally {
	         if (b) i = 3;
	       }
	       i = 4;
	     }
	     return i;
	  }
	}</pre>

Table 1: Valid Java programs rejected by any bytecode verifier.

```
int m1(boolean b){       iload_1    ()      {1:int}
  int i;                 ifeq A     (int)   {1:int}
  try {                  iconst_1   ()      {1:int}
    if (b)               istore_3   (int)   {1:int}
      return 1;          jsr S      ()      {1:int,3:int}
    i = 2;               iload_3    ()      {1:int,3:int}
  } finally {            ireturn    (int)   {1:int,3:int}
    if (b)            A: iconst_2   ()      {1:int}
     i = 3;              istore_2   (int)   {1:int}
  }                      jsr S      ()      {1:int,2:int}
  return i;              goto C     ()      {1:int} // 2 mod. by S
}                     S: astore 4   (ra(S)) {1:int}
                         iload_1    ()      {1:int,4:ra(S)}
                         ifeq B     (int)   {1:int,4:ra(S)}
                         iconst_3   ()      {1:int,4:ra(S)}
                         istore_2   (int)   {1:int,4:ra(S)}
                      B: ret 4      ()      {1:int,4:ra(S)}
                      C: iload_2    ()      {1:int}
                         ireturn   // 2 contains wrong type
```

Table 2: Bytecode verification of `Test1` fails.

In the proof of the theorem we use the framework of Abstract State Machines. The compiler as well as the bytecode verifier are specified as an Abstract State Machine. The specifications are executable in the AsmGofer system (on the CD of [6]).

An intermediate notion of *bytecode type assignment* is used. A bytecode type assignment consists of an assignment of type frames (stack maps) to some (but not necessarily all) instructions in the code of a method. The type frames have to satisfy several conditions. A soundness theorem says that bytecode programs with bytecode type assignments satisfy at run-time the so-called structural constraints of [4, §4.8.2]. Hence, they are type safe and do not corrupt the state of the JVM. A completeness theorem says that programs with bytecode type assignments are accepted by the bytecode verifier. In fact, it can be shown that the verifier computes a principal type assignment for each method.

The compiler is then extended such that it generates also type frames (stack maps) for the instructions of the compiled programs. It assigns types to those local registers which correspond to variables that are definitely assigned in the source program. The main theorem then says that the so generated type frames are a bytecode type assignment for the method.

The proof is rather lengthy, since already simple lemmas about reachability properties of the generated code require many cases due to the complexity of the Java programming language.

References

[1] E. Börger and J. Huggins. Abstract State Machines 1988–1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998. Updated bibliography available at `http://www.eecs.umich.edu/gasm`.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(tm) Language Specification*. Addison Wesley, second edition, 2000.

[3] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[4] T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[5] E.G. Sirer, S. McDirmid, and B. Bershad. Kimera: A Java system security architecture. `http://kimera.cs.washington.edu/`, 1997.

[6] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification and Validation*. Springer-Verlag, 2001. to appear.