The Problem of Bytecode Verification in Current Implementations of the JVM

Robert F. Stärk[1] and Joachim Schmid[2]

[1] Computer Science Department, ETH Zürich, CH–8092 Zürich
staerk@inf.ethz.ch
[2] Siemens Corporate Technology, D–81730 Munich
Joachim.schmid@mchp.siemens.de

Abstract. The main problems of Java Bytecode Verification are embedded subroutines and multiple inheritance via interfaces. The problem with interfaces has been solved (by using sets of reference types or by introducing a run-time check for the invokeinterface instruction). It is widely believed that also the problem with subroutines has been solved. This is not true. Most research has been focussed on soundness of Bytecode Verification, i.e., that bytecode accepted by the verifier is type-safe at run-time and cannot corrupt the memory. The problem of completeness has not been addressed. During an attempt to prove that our Java compiler 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 for the full Java programming language. We propose therefore to restrict the so-called rules of definite assignment for the try-finally statement and for the labeled statement such that the example programs are no longer allowed. Then we can prove, using the framework of Abstract State Machines, that each program from the slightly restricted Java language is accepted by the Java Bytecode Verifier.

1 Introduction

The Java programming language [4] is a strongly typed general-purpose language. Java programs are compiled to bytecode instructions (class files) according to the Java Virtual Machine Specification [11]. Class files can be executed by the Java Virtual Machine (JVM). A correct Java compiler produces bytecode instructions which can be trustfully executed on a JVM. A JVM, however, can not distinguish between bytecode generated by a correct compiler and bytecode produced by a different source to exploit the JVM. This is the reason why a JVM has to verify bytecode programs before executing them.

Usually, a JVM does not check single bytecode instructions each time it executes them, because such a virtual machine would be too slow. Instead, a JVM tries to verify a class file in advance, at link-time, without run-time information. If this verification is successful, then the class file can be trustfully executed without violating any type constraints at run-time; otherwise the bytecode is rejected.

The Java language was designed in a way that every legal Java program compiled with a correct compiler should be accepted by the verifier. The Java Virtual Machine Specification [11] states this explicitly:

> *Sun's class file verifier is independent of any compiler. It should certify all code generated by Sun's compiler for the Java programming language; it should also certify code that other compilers can generate, as well as code that the current compiler could not possibly generate.*

When we tried to prove that the Java compiler described in [16] generates verifiable code, we found example programs which show that Sun's intention is not fulfilled. There are valid Java programs, compiled with Sun's compiler, which are not accepted by Sun's class file 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 [15]. 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 [4, §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. Programs `Test1` and `Test2` are the reason why we claim that bytecode verification as described in [11] is not possible for the full Java language. We will analyze the two programs in detail below and propose a solution to the problem.

There are other reasons for the rejection of valid Java programs by the bytecode verifier. Table 2 contains some example programs which are rejected by certain bytecode verifiers when compiled with specific versions of Java compilers. The programs in Table 2 are rejected due to bugs in the specific compiler or programming errors in the bytecode verifier.

Program `Test3` is rejected when compiled with Sun's JDK 1.2.2 compiler due to a well-known bug in this version of the compiler [12]. The problem is that the compiler erroneously uses the same register to store the string `s` before executing the finally block at the `return s` statement, and to store the variable `i` in the finally-block. The bytecode verifier detects this error and rejects the incorrect code reporting that "register 4 contains a wrong type".

Program `Test4` is rejected by the JDK 1.2 and the JDK 1.3 bytecode verifier. It seems that these versions of the verifier are not prepared to handle nested try-finally statements in the way they are used in the example. The verifier reports an "illegal return from subroutine" although the bytecode generated by the compiler seems to be correct. Fortunately nested try-finally statements are almost never used in practice.

Program `Test5` is rejected by the JDK 1.3 bytecode verifier although it is a valid Java program. The verifier rejects the compiled program reporting that

```java
class Test1 {
  int m1(boolean b) {
    int i;
    try {
      if (b) return 1;
      i = 2;
    } finally {
      if (b) i = 3;
    }
    return i;
  }
}
```

```java
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;
  }
}
```

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

```java
class Test3 {
  String m3() {
    String s = "hello";
    try {
      synchronized(s) {
        return s;
      }
    }
    finally { int i = 0; }
  }
}
```

```java
class Test4 {
  void m4(boolean b) {
    try {
      try { if (b) return; }
      finally {
        try { if (b) return; }
        finally {
          if (b) return;
        }
      }
    }
    finally { if (b) return; }
  }
}
```

```java
class Test5 {
  void m5() {
    int[][] a = null;
    a[0] = new int[0];
  }
}
```

```java
class Test6 {
  void m6(boolean b) {
    boolean z;
    while (b ? true: true);
    b = z;
  }
}
```

Table 2. Valid Java programs rejected due to compiler or verifier bugs.

there are “incompatible types for storing into array of arrays”. The problem is that the verifier does not know that the variable `a` is an array of arrays of `int`. When the value `null` is assigned to `a`, the verifier concludes that the type of `a` is the null type. When the new array `new int[0]` is assigned to `a[0]`, the verifier is not able to conclude that the component type of the null type (when considered as an array) is again the null type. The program should be accepted by the verifier. Only at run-time a `NullPointerException` should be thrown, since the newly created array is assigned to the non-existing array element `a[0]`.

Program `Test6` is a valid Java program according to the JLS [4]. The program is rejected by most Java compilers. They report that the variable `z` may not have been initialized when it is used in the assignment `b = z`. These compilers, however, are wrong. According to §16.2.9 of the JLS [4], a variable is definitely assigned after a while statement, if it is definitely assigned after the test expressions when the expressions evaluates to false. According to §16.1.5, a variable is definitely assigned after a boolean conditional expression when it evaluates to false, if it is definitely assigned after both branches of the expression when they evaluate to false. Finally, according to §16.1.1, any variable is definitely assigned after the boolean literal `true` when it evaluates to false (because this is not possible). Hence, according to the JLS [4], the variable `z` is definitely assigned after the while-statement and the program `Test6` is legal.

The JDK 1.3beta compiler is correct and accepts the program, but the code it generates is rejected by the bytecode verifier. The problem is that the 1.3beta compiler does not optimize the boolean expression `b?true:true` and therefore the verifier thinks that the assignment `b = z` is reachable. The example `Test6` shows that a compiler has to optimize boolean expressions containing the literals `true` and `false`. Otherwise, it generates unverifiable code.

The example `Test6` shows in addition a not so obvious inconsistency in the design of the Java programming language. In the rules of definite assignment (§16.1.1 of the JLS [4]), the expression `b?true:true` is treated like the constant `true`. In the reachability analysis of §14.20, however, the expression `b?true:true` is treated like an arbitrary boolean expression which can also have the value false. Since the expression `b?true:true` is not a constant expression with value true, the while statement can complete normally according to §14.20 and the assignment `b = z` has to be regarded as reachable by a Java compiler. Our compiler described in Part II of [16] compiles the program `Test6` correctly and the generated bytecode is accepted by the verifier.

2 Related work

The programs `Test1` and `Test2` in Table 1 cannot by typed in many type systems that have been introduced. For example, Stata and Abadi treat subroutines in a simplistic way in [17]. They do not consider the possibility to jump out of a subroutine to an enclosing subroutine via a `break`, `continue` or `throw`. Thus, there are, in addition to `Test1` and `Test2`, many legal Java programs that cannot be typed in their system.

The bytecode of `Test1` and `Test2` still cannot be typed in the extended and refined system of Freund and Mitchell in [3,2]. The two examples can also not be typed in the system of O'Callahan in [13] which is based on ideas of type systems for continuations and polymorphic recursion. It seems that any type system (or bytecode verifier) that checks each subroutine only once will reject legal Java programs like `Test1` and `Test2`. This includes also various systems by Qian (e.g. [14]) and other systems (see [7]).

There exists type systems and bytecode verifiers that assign more than one stack map to instructions in subroutines (e.g. Haase [6] or Henrio and Serpette [8]). Such bytecode verifiers accept `Test1` and `Test2`, since a subroutine can then be typed (or verified) differently for each call of the subroutine. An equivalent approach would be to inline finally blocks and embedded subroutines. Both approaches, however, lead to an exponential behavior in the worst case (for nested subroutines).

Another solution proposed by Leroy in [10] is that the same local variable can no longer be used with different types for different purposes in a method body and that the virtual machine initializes local variables with default values. Then the problems described in this paper disappear and bytecode verification becomes much simpler and is even feasible on smart cards.

3 Compiling try-finally statements

The problem with the programs `Test1` and `Test2` in Table 1 is the try-finally statement. The try-finally statement of Java is used for error recovering. The syntax is as follows (where $block_1$ and $block_2$ are block statements):

`try` $block_1$ `finally` $block_2$

There are several ways for leaving a block. A block can be left by reaching the end of the block, or via an exception thrown in the block, or by a `return`, `break`, or `continue` statement inside the block. The semantics of the try-finally statement is to execute the finally block $block_2$ after the try block $block_1$, no matter how control leaves the try block. That is, the finally block is guaranteed to be executed whether its try block completes normally or abruptly (via a break, continue, return or an exception).

It is the task of the compiler to ensure, that the finally block will be executed after the try block (no matter how control leaves it). One solution would be to copy the code for the finally block before each possible position where the try block could be left. This code duplication, however, would blow up the compiled code and therefore the designers of the Java Virtual Machine decided to use embedded subroutines to compile the try-finally statement.

An embedded subroutine is almost like a method invocation. It can be invoked from any point in the program. When returning from the subroutine, the program counter is positioned at the instruction following the instruction where the subroutine was invoked from. A call of a subroutine does not create a new frame and therefore the local environment of the caller of the subroutine is fully available in the subroutine.

There are two special bytecode instructions, `jsr`(s) and `ret`(x), which are used to implement subroutines. The instruction `jsr`(s) takes the current program counter pc, pushes $pc+1$ on the operand stack, and sets the program counter to s. At position s, the `astore`(x) instruction stores the top value of the operand stack (the return address $pc+1$) into the register x. The code for the finally block follows after the `astore`(x) instruction. At the end of the finally block the `ret`(x) instruction jumps back to the calling position by updating the program counter pc to the return address stored in register x.

The compilation scheme for the try-finally statement is illustrated for the body of method `m1` of the program `Test1`. Table 3 contains on the left-hand side the Java source code of the method `m1`. The next column contains the bytecode instructions of the compiled method. Columns 3 and 4 contain the type frames (stack maps) generated by the verifier in the attempt to verify the program.

The subroutine calls '`jsr S`' are inserted before the `ireturn` instruction and and at the end of the code for the try block. For the `ireturn` instruction, the evaluated expression is stored in the register `3`, the subroutine is called, and then the content of the register is returned. The so called default handler for the try block is not shown in Table 3. The default handler catches any exception thrown in the try block, saves it into a temporary register, calls the subroutine and re-throws the saved exception after the return from the subroutine.

There seems to be no error in the bytecode in Table 3, and in fact, there is no error, because on each path leading to the instruction `iload_2` at label `C`, the variable `2` is assigned. Nevertheless, the bytecode is rejected due to verification of subroutines which we describe in the next section.

4 Bytecode verification

Java programs introduce local variables by variable declaration statements as for example the variable `i` in `Test1` and `Test2`. The statement declares the name and the type of the new variable. The types of local variables are known at the source code level. Additionally, there may be an initial value as for example the string `"hello"` in the program `Test3`.

The compiled Java program is a sequence of bytecode instructions where the control flow of Java is explicit in the virtual machine instructions. For example, the compiler uses a conditional jump instruction for a while loop. On the bytecode level, variables are local registers (denoted by natural numbers). The types of local registers are not declared. Furthermore, a compiler may assign to different variables the same register. For example, it could use the same registers for variables declared in two disjoint block statements.

One of the main goals of bytecode verification is to ensure that each local register contains a value of the proper type when it is accessed. For this purpose the bytecode verifier assigns to each instruction the types of local registers known to have a value at that instruction. When an instruction accesses a local register,

```
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 3. Bytecode verification of `Test1` fails.

the verifier checks whether the local register has a type and whether the type is compatible with the type expected at the instruction. If an instruction loads a register and the register has no type, then the program is rejected. If the register has a type and the type is not compatible, then the program is rejected, too.

The verifier starts at the first instruction where the registers that have a type are exactly the arguments of the method (including the `this` argument). The types of these registers are the declared types of the method arguments. The verifier propagates the type information to all direct successors. If different instructions have the same successor, then the type information at the successor instruction is the type information which is common to all predecessors. For example, if the variable x has no type before the statement

$$\texttt{if}\ (e)\ x = \texttt{true}$$

then it has no type after the statement, because there is a path in the above statement where x is not assigned. More precisely, a variable x has type t at instruction i, if on every execution path from the first instruction to i, variable x has type t.

Since the values of local registers are loaded onto the operand stack and values on the operand stack are stored into local registers, the length of the operand stack and the types of the operands have to be considered by the bytecode verifier, too. The bytecode verifier therefore tries to assign to each instruction a type frame (stack map) consisting of the types of the operands on the operand stack and the types of the local registers known to have a value at that instruction.

Bytecode verification is simple if there are no subroutines. In case of subroutines, the above algorithm has to be adapted. Consider the type frames for the method `m1` listed in Table 3. The subroutine starting at label `S` is called from two different positions. At the first '`jsr S`' the register `3` is of type `int`, whereas the the register `2` has no type (it is unusable). At the second '`jsr S`' the register `2` is of type `int`, whereas the register `3` has no type. Since the bytecode verifier scans a subroutine only once, it has to merge the two type frames at label `S`. At label `S` therefore only the register `1` (which corresponds to the argument `b` of the method `m1`) has a type. The registers `2` and `3` are unusable. Since there is a path from label `S` to the '`ret 4`' instruction where the register `2` is not assigned (when `b` evaluates to false), register `2` has no type at the '`ret 4`' instruction at label `B`.

The `iload_3` and the '`goto C`' instruction are successors of the '`ret 4`' instruction, because they follow directly a '`jsr S`'. The verifier returns to those instructions the types of the registers at the `ret 4` instruction which are modified by the subroutine (none in our case). The types of registers which are not modified by the subroutine are propagated from the preceding '`jsr S`' instruction. Modified by the subroutine means assigned in the code for the finally block. Hence, the register `2` in our example is modified by the subroutine because there is a `istore_2` instruction between `S` and `B`.

The type frame from a `ret`(x) instruction is propagated to the successors as follows. Let i be the index of the `ret`(x) instruction and let j be a call to the subroutine. Hence $j+1$ is a successor of i. We take the type frame at i restricted to those registers modified by the subroutine and the type frame at j restricted to registers which are not modified by the subroutine and propagate it to $j+1$.

Since in our example register `2` is modified by the subroutine, it is not propagated from the second '`jsr S`' instruction to the '`goto C`' instruction. Hence, at label `C`, the register `2` is unusable and the check for the `iload_2` instruction fails. In the eyes of the verifier, register `2` could have no value at label `C` and therefore the verifier rejects the program reporting that register `2` contains a wrong type (the type unusable).

5 Restricting the rules of definite assignment

As mentioned above one of the main tasks of the bytecode verifier is to ensure that no local register can be accessed before it is assigned a value. The verifier also has to make sure that execution never falls off the bottom of the code array of a method [11, §4.8.2].

Therefore, already the Java compiler has to ensure the same two properties on the source code level. Every Java compiler must carry out a specific conservative flow analysis to make sure that, for every access of a local variable, the local variable is definitely assigned before the access; otherwise a compile-time error must occur [4, §16]. Each local variable must have a definitely assigned value when any access of its value occurs. An access to its value consists of using

```
int m2(boolean b) {     iload_1   ()      {1:int}
int i;                  ifeq A    (int)   {1:int}
L: { 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}
       if (b)           ireturn   (int)   {1:int,3:int}
         break L;    A: iconst_2  ()      {1:int}
     } finally {        istore_2  (int)   {1:int}
       if (b)           iload_1   ()      {1:int,2:int}
         i = 3;         ifeq B    (int)   {1:int,2:int}
     }                  jsr S     ()      {1:int,2:int}
     i = 4;             goto E    ()      {1:int} // 2 mod.by S
   }                 B: jsr S     ()      {1:int,2:int}
   return i;            goto D    ()      {1:int} // 2 mod.by S
}                    S: astore 4  (ra(S)) {1:int}
                        iload_1   ()      {1:int,4:ra(S)}
                        ifeq C    (int)   {1:int,4:ra(S)}
                        iconst_3  ()      {1:int,4:ra(S)}
                        istore_2  (int)   {1:int,4:ra(S)}
                     C: ret 4     ()      {1:int,4:ra(S)}
                     D: iconst_4  ()      {1:int}
                        istore_2  (int)   {1:int}
                     E: iload_2   ()      {1:int}
                        ireturn   // 2 contains wrong type
```

Table 4. Bytecode verification of `Test2` fails.

the identifier of the variable occurring anywhere in an expression except as the left-hand operand of the simple assignment operator `=`.

In addition, every Java compiler must carry out a conservative flow analysis to make sure that all statements are reachable and that it is not possible to drop off the end of a method body [4, §14.20]. A compile-time error occurs, if the body of a method can complete normally.

It is clear that conservative flow analysis on the source code level and the static analysis of the bytecode verifier must be related. Otherwise too many compiled Java programs would be rejected by the verifier. Therefore we propose that the "rules of definite assignment" in [4, §16] are restricted for the try-finally statement and for the labeled statement such that the programs in Table 1 are no longer valid Java programs.

The rules of definite assignment for the try-finally statement in [4, §16.2.14] are: A variable V is definitely assigned after the try-finally statement iff at least one of the following is true:

- V is definitely assigned after the try block, or
- V is definitely assigned after the finally block.

Hence, in program `Test1`, the variable `i` is definitely assigned after the try-finally block, because it is definitely assigned after the try block.

We propose to restrict the rule as follows: A variable V is definitely assigned after the try-finally statement iff at least one of the following is true:

- V is definitely assigned after the try block *and* there is no assignment to V in the finally block, or
- V is definitely assigned after the finally block.

Then, in program `Test1`, the variable `i` is no longer definitely assigned after the try-finally block, because there is the assignment '`i = 3`' in the finally block. The Java compiler would reject program `Test1` under the restricted rules of definite assignment, since the variable `i` may not be initialized in the '`return i`' statement at the end of the method. An error message would force a programmer to definitely assign a value to variable `i` *inside* the finally block.

The rules of definite assignment for the labeled statement in [4, §16.2.5] are: A variable V is definitely assigned after a labeled statement '*lab*: *stm*' (where *lab* is a label) iff all of the following are true:

- V is definitely assigned after *stm*, and
- V is definitely assigned before every '`break` *lab*' statement that may exit the labeled statement '*lab*: *stm*'.

Unfortunately the term 'may exit' is not defined in the JLS [4] and different compilers interpret it differently (see [16] for a possible interpretation). The term 'may exit', however, is not the problem.

Let us consider the labeled statement '`L`: *stm*' in program `Test2` in Table 1. The variable `i` is definitely assigned after *stm* because of the assignment '`i = 4`' at the end of *stm*. The variable `i` is also definitely assigned *before* the '`break L`' statement inside *stm* because of the preceding assignment '`i = 2`'. Hence, the variable `i` is definitely assigned after the labeled statement '`L`: *stm*' and the compiler concludes that it is initialized when it is used in the '`return i`' statement at the end of the method.

In the eyes of the verifier (Table 4) the variable `i` is possibly not initialized at the end of the method. The '`break L`' statement is compiled as a '`goto E`' instruction that jumps to the end of the code of the labeled statement. A '`jsr S`' instruction is inserted immediately before the '`goto E`' instruction, since the code for the finally block has to be executed before the break can be done. Although the variable `i` is modified by the subroutine, it has no type at the end of the subroutine, since the subroutine is also called before the '`return 1`' statement, where `i` has no value. Hence, the '`goto E`' instruction which corresponds to the '`break L`' does not propagate the type for the variable `i` and `i` is therefore unusable at the end of the method.

We propose to restrict the rule as follows: A variable V is definitely assigned after a labeled statement '*lab*: *stm*' iff all of the following are true:

- V is definitely assigned after *stm*, and

- V is definitely assigned before every '`break` *lab*' statement that may exit the labeled statement '*lab*: *stm*', and
- V is definitely assigned after every '`try` $block_1$ `finally` $block_2$' statement inside *stm* such that $block_1$ contains a '`break` *lab*' statement that may exit the labeled statement '*lab*: *stm*'.

Then, in program `Test2`, the variable `i` is no longer definitely assigned after the labeled statement, because it is not definitely assigned after the try-finally statement inside the labeled statement.

6 The main result

The two restrictions of the rules of definite assignment (explained in the previous section) are sufficient to prove the following theorem:

Theorem 1. *If a Java compiler satisfies the constraints of Part II in [16], 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 [16] 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 [11, §7.13].

In the proof of the theorem we use the framework of Abstract State Machines (see [5,1]). 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 [16]).

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 [11, §4.8.2]. Hence, they are type-safe and do not corrupt the state of the JVM at run-time. Moreover, it is shown that a method is accepted by the bytecode verifier if, and only if, there exists a bytecode type assignment for it. In fact, if there exists a bytecode type assignment for a method, then the verifier computes a principal type assignment for it.

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 main theorem therefore relates the static analysis on the Java source code level (rules of definite assignment) to the static analysis on the bytecode level (bytecode verification).

In a first attempt to prove the theorem we found the examples `Test1` and `Test2` in Table 1. After restricting the rules of definite assignment the proof went through. The proof is rather involved, since already simple lemmas about reachability properties of the generated code require many cases due to the complexity of the full Java programming language. Although the compiler is defined recursively on the structure of expressions and statement, the theorem cannot be proved by a simple induction, since the notion of bytecode type assignment is a global notion that depends on the whole code sequence for a method body.

7 Why sets of reference types?

For the definition of *bytecode type assignment* it is important that one works with *finite sets of reference types*. Why? Consider the example program `Test7` in Table 5 (similar examples are considered in [9]). The method `m7` contains a local variable `x` of type `Comparable`. Since the class `Integer` as well as class `String` implement the interface `Comparable`, both arguments `i` and `s` of the method can be assigned to the variable `x` in the body.

In the corresponding bytecode in Table 5, the type of `x` is not known. When the verifier reaches label `B` during its static analysis, the variable `x` can contain an `Integer` or a `String`. Since both types are reference types, the verifier tries to merge the two types. The JVM specification says in [11, §4.9.2] that 'the merged type of two references is the first common superclass of the two types and that such a reference type always exists because the type `Object` is a superclass of all class and interface types.' In our example the first common superclass of `Integer` and `String` is the class `Object`. Hence, the type assigned to variable `x` at label `B` is `Object`. The method invocation at the next instruction, however, requires an argument of type `Comparable` and so the verification process fails.

Sun's verifier does not reject the bytecode. Instead it inserts an additional run-time check for invocations of interface methods. According to the JVM specification [11] an `IncompatibleClassChangeError` is thrown at run-time, if the target object of an interface method invocation does not implement the interface.[1] In fact, Sun's verifier makes even a stronger assumption:

Each class implements *each* possible interface.

This can be seen by compiling class `Test8` in Table 6 and afterwards changing and re-compiling class `A` such that it does not no longer implement the interface `I` and does not contain a method `m`. The program is still accepted by the JDK 1.3 verifier and the result of the execution on the virtual machine is the message `"done"`. The only explanation for this behavior is that the verifier allows the assignment of an object of type `A` to the static field `f` of type `I` even if class `A` is completely unrelated to the interface `I`. As a consequence, for verified bytecode, it is no longer true that at run-time each field contains a value which is compatible with the declared (compile-time) type of the field. It could be possible that the declared type of the field is an interface and the value at

[1] The additional run-time check for `invokeinterface` was not required in the first edition of the JVM specification. Therefore, it seems that Sun adapted the JVM specification to their implementation of the bytecode verifier.

```
class Test7 {
  void m7(Integer i, String s) {
    Comparable x;
    if (i != null)
      x = i;
    else
      x = s;
    n7(x);
  }
  void n7(Comparable x) { }
}
```

```
    aload i
    ifnull A
    aload i
    astore x
    goto B
A:  aload s
    astore x
B:  aload_0
    aload x  // Type of x?
    invokevirtual n7(Comparable)
    return
```

Table 5. What is the type of variable x in the bytecode?

```
class Test8 {
  static I f;
  public static void main(String[] argv) {
    f = new A();
    System.out.println("done");
  }
}
```

```
interface I { void m(); }
class A implements I {
  public void m() { }
}
```

Table 6. Sun's assumption: Each class implements each interface.

```
public class Test9 {
  public static void run(J j) {
    j.m();
  }
  public static void main(String[] argv) {
    B b = new B();
    run(b);
    Object o = new Object();
    // run(o);
  }
}
```

```
interface J {
  void m();
}
class B implements J {
  public void m() { }
}
```

Table 7. HotSpot Virtual Machine Error.

Count	Description	General information
7 192	Classes	140 seconds verification time (without
21 191	Methods	checking static constraints) on
1 915 585	Instructions	400 Mhz Intel II CPU.
3 669	Unreachable instructions	
14 298	Multiply visited instructions	
1 081	`jsr` instructions	4 880 instructions need sets of reference
424	`ret` instructions	types and 10 classes (18 instructions)
12 201	`invokeinterface` instructions	would be rejected without them.

Table 8. Verification of the Java Runtime Environment.

run-time could then be any object. If the field is used as a target object of an `invokeinterface` instruction, one has to check at run-time that the class of the object does implement the interface.

Sun's assumption that each class implements each interface is not in the spirit of Java's notion of compatibility. Moreover, it seems that the HotSpot compiler has problems with optimizing away the additional run-time check for `invokeinterface`. This can be seen by compiling class `Test9` in Table 7 and inserting the commented method call `run(o)` in the corresponding bytecode. Although variable `o` is of type `Object` and the `run`-method requires an object implementing the interface `J`, Sun's verifier accepts this bytecode (see discussion above). The method `run` is invoked twice and in the `run` body the method `J/m` is called with `invokeinterface`. It seems that the HotSpot compiler checks only once that the instance `j` implements the interface `J`. In the second call, however, the variable `j` does not implement `J` and when running this example we get the following error message:

```
# HotSpot Virtual Machine Error, Unexpected Signal 11
# Please report this error at
# http://java.sun.com/cgi-bin/bugreport.cgi
# Error ID: 4F533F4C494E55580E43505005BC
# Problematic Thread: prio=1 tid=0x804dbb8 nid=0x18e5 runnable
```

Class `Test9` illustrates that the `invokeinterface` instruction must be checked on each call.

A better solution is to allow *sets of reference types* in the bytecode verification process. The type of variable `x` at label `B` in Table 5 is then the set $\{\texttt{Integer}, \texttt{String}\}$. The meaning of this type is 'either `Integer` or `String`'. At the method invocation in the next instruction, the verifier has just to check that each element of the set of reference types assigned to `x` is a subtype of `Comparable`. No additional run-time checks are needed. The modified program `Test9` is already rejected by the verifier (cf. [16]).

Working with sets of reference types is not expensive, since in practice, not so many instructions require sets with more than one reference type. Table 8 (created with the bytecode verifier of [16]) illustrates that the complete Java

Runtime Environment consists of about seven thousand classes containing about two million instructions. For only 4 880 instructions (0.25%) sets with more than one reference types are needed. On the other hand there are about 12 200 `invokeinterface` instructions which must be checked on each call without sets of reference types. When using the first common superclass in merging (instead of sets of reference types), then 18 of 4 880 instructions would be rejected by our verifier.

References

1. E. Börger and J. Huggins. Abstract State Machines 1988–1998: Commented ASM Bibliography. *Bulletin Europ. Assoc. Theoret. Comp. Science*, 64:105–127, 1998. Updated bibliography available at `http://www.eecs.umich.edu/gasm`.
2. S. N. Freund and J. C. Mitchell. Specification and verification of Java bytecode subroutines and exceptions. Technical Report CS-TN-99-91, Stanford University, 1999.
3. S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.
4. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(tm) Language Specification*. Addison Wesley, second edition, 2000.
5. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1993.
6. E. Haase. Justice: An implementation of a free class file verifier for Java. Technical report, Institut für Informatik, Freie Universität Berlin, 2001. `http://bcel.sourceforge.net/justice/`.
7. P. H. Hartel and L. Moreau. Formalising the safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 2001. To appear.
8. L. Henrio and B.P. Serpette. A framework for bytecode verifiers: Application to intra-procedural continuations. Technical report, Inria Sphia-Antipolis, 2001.
9. T. B. Knoblock and J. Rehof. Type elaboration and subtype completion for Java bytecode. *ACM Transactions on Programming Languages and Systems,*, 23(2):243–272, 2001.
10. X. Leroy. On-card bytecode verification for Java Card. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security (E-smart 2001)*, pages 150–164. Springer-Verlag, Lecture Notes in Computer Science 2140, 2001.
11. T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification*. Addison Wesley, second edition, 1999.
12. Sun Microsystems. Release notes, Java(tm) 2 SDK, standard edition, version 1.2.2, verify error workaround, 2000. `http://java.sun.com/products/jdk/1.2/changes.html\#verify`.
13. R. O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, pages 70–78, 1998.
14. Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.
15. E.G. Sirer, S. McDirmid, and B. Bershad. Kimera: A Java system security architecture. `http://kimera.cs.washington.edu/`, 1997.

16. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation.* Springer-Verlag, 2001.
17. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.