Verified Lightweight Bytecode Verification

Gerwin Klein and Tobias Nipkow

Technische Universität München, Institut für Informatik
http://www.in.tum.de/~{kleing|nipkow}/

Abstract. Eva and Kristoffer Rose proposed a (sparse) annotation of Java Virtual Machine code with types to enable a one-pass verification of welltypedness. We have formalized a variant of their proposal in the theorem prover Isabelle/HOL and proved soundness and completeness.

1 Introduction

The Java Virtual Machine (*JVM*) comprises a typed assembly language, an abstract machine for executing it and the so-called *Bytecode Verifier* (*BV*) for checking the welltypedness of JVM programs. Resource-bounded JVM implementations on smart cards do not provide bytecode verification because of the relatively high space and time consumption. They either do not allow dynamic loading of JVM code at all or rely on cryptographic methods to ensure that bytecode verification has taken place off-card. In order to allow on-card verification, Eva and Kristoffer Rose [3] proposed a (sparse) annotation of Java Virtual Machine code with types to enable a one-pass verification of welltypedness. Roughly speaking, this transforms a type reconstruction problem into a type checking problem, which is easier. Based on these ideas we have extended an existing formalization of the JVM in the theorem prover Isabelle/HOL [2, 1]. In §2 we describe the general idea of bytecode verification and its formalization in Isabelle/HOL. In §3 we explain how lightweight bytecode verification works, how we formalized it and proved it correct and complete.

2 Bytecode verification

The JVM is a stack machine where each method activation has its own expression stack and local variables. The types of operands and results of bytecode instructions are fixed (modulo subtyping), whereas the type of a storage location may differ at different points in the program. Let's look at an example:

instruction	stack	local variables
Load 0	[]	[Class B, int]
Store 1	[Class A]	[Class B, unusable]
Getfield F A	[]	[Class B, Class A]
Goto -2	[Class A]	[Class B, Class A]

On the left the instructions are shown and on the right the type of the stack elements and the local variables. The type information attached to an instruction characterizes the state *before* execution of that instruction. We assume that class B is a subclass of A and that A has a field F of type A.

Execution starts with an empty stack and the two local variables hold a reference to an object of class B and an integer. The first instruction loads local variable 0, a reference to a B object, on the stack. The type information associated with following instruction may puzzle at first sight: it says that a reference to an A object is on the stack, and that the type of local variable 1 has become unusable. This means the type information has become less precise but is still correct: a B object is also an A object and an integer is now classified as unusable. The reason for these more general types is that the predecessor of the Store instruction may have either been Load 0 or Goto -2. Since there exist different execution paths to reach Store, the type information of the two paths has to be "merged". The type of the second local variable is either int or Class A, which are incompatible, i.e. the only common supertype is 'unusable'.

Bytecode verification is the process of inferring the types on the right from the instruction sequence on the left and some initial condition, and of ensuring that each instruction receives arguments of the correct type. This can be done on a per method basis because each method has fixed argument and result types. The two tables on the right are together called a *method type*, one line of the method type is called a *state type*. To simplify matters we restrict the considerations in this paper to a single method.

For theoretical investigations it has become customary to separate type inference (computation of a method type) from type checking (checking if an instruction sequence fits a method type). Type inference is usually implemented as a dataflow analysis and may require several iterations due to subtyping. We will now ignore type inference (although we have also verified it in Isabelle/HOL) and concentrate on type checking.

A first machine-checked specification of type checking for the JVM was given by Pusch [2]. Using Isabelle/HOL she connected the type checking rules with an operational semantics for the JVM by showing that execution of type correct programs is type sound, i.e. during run time each storage location contains values of the type predicted by the method type. We will now sketch some the key ingredients of the type checking specification by Nipkow *et al.* [1] that our formalization of lightweight bytecode verification builds on.

Type checking of methods is modeled by a predicate wt_method relating the instruction sequence, types of method parameters, return type, etc. with a method type φ. In essence, the definition

$$
\begin{array}{l}
\mathsf{wt\_method} :: [\mathit{jvm\_prog}, \mathit{cname}, \mathit{ty\ list}, \mathit{ty}, \mathit{nat}, \mathit{instr\ list}, \mathit{method\_type}] \rightarrow \mathit{bool} \\
\mathsf{wt\_method}\ \Gamma\ C\ pTs\ rT\ mxl\ ins\ \varphi \equiv \\
\quad \mathsf{let}\ \mathit{max\_pc} = \mathsf{length}\ \mathit{ins}\ \mathsf{in} \\
\qquad \mathit{max\_pc} < \mathsf{length}\ \varphi \wedge 0 < \mathit{max\_pc} \wedge \mathsf{wt\_start}\ \Gamma\ C\ pTs\ mxl\ \varphi \wedge \\
\qquad (\forall pc.\ pc < \mathit{max\_pc} \longrightarrow \mathsf{wt\_inst}\ (\mathit{ins}\ !\ pc)\ \Gamma\ rT\ \varphi\ \mathit{max\_pc}\ pc)
\end{array}
$$

states that, in a declaration context Γ (containing all class declarations of the program), $\mathsf{wt\_method}$ holds for an instruction sequence ins (the method body) and a method type φ when each single instruction $ins\ !\ pc$ is well typed (the Isabelle/HOL operator ! returns the nth element of a list). The predicate $\mathsf{wt\_inst}$ checking single instructions may take into account the return type rT, the current program counter pc, and the maximum program counter max_pc (the length of the instruction sequence). $\mathsf{wt\_start}$ ensures that the types on the operand stack and of the local variables are initialized correctly with regard to the class C the method is declared in, the parameters pTs of the method, and the number of local variables mxl.

$\mathsf{wt\_inst}$ is a case distinction over the instruction set. As the type checking conditions for single instructions are very similar to each other, we only take a look at an example:

$$
\begin{array}{l}
\mathsf{wt\_inst} :: [instr, jvm\_prog, ty, method\_type, nat, nat] \to bool \\
\mathsf{wt\_inst}\ (\mathsf{Load}\ idx)\ \Gamma\ rT\ \varphi\ max\_pc\ pc = \\
\quad \mathsf{let}\ (ST, LT) = \varphi\ !\ pc\ \mathsf{in} \\
\qquad pc{+}1 < max\_pc \wedge idx < \mathsf{length}\ LT \wedge \\
\qquad (\exists t.\ (LT\ !\ idx) = usable\ t \wedge \Gamma \vdash (t\ \#\ ST\ ,\ LT) \preceq_s \varphi\ !\ (pc{+}1))
\end{array}
$$

The predicate first checks some applicability conditions like $pc+1 < max\_pc$ and $idx < \mathsf{length}\ LT$, then calculates the effect of the instruction on the current state type and eventually requires that the result be compatible with the state type at the next instruction in the control flow.

The current state type consists of the stack ST and the local variables LT at position pc in the method type. Both are lists containing the types before execution of the instruction. In the Load case we require some type t other than unusable at index idx in LT. The state type of the next instruction at position $pc+1$ must correctly approximate a state type where t is on top of the stack (# is the list constructor in Isabelle/HOL). The local variables are unchanged. This correct approximation $\_ \vdash \_ \preceq_s \_$ is Java's *widen* relation lifted to state types and extended by the element unusable. We already used it informally in the example program.

3 Lightweight bytecode verification

Two things make the traditional bytecode verifier unsuitable for on-card verification: the type reconstruction algorithm itself is large and complex, and the whole method type is held in memory. Lightweight bytecode verification addresses both problems.

The need for dataflow analysis is caused by the fact that some instructions may have multiple preceding paths of execution and that the types constructed on these paths have to be merged. This can only occur at the targets of jumps. The basic idea of lightweight bytecode verification is to look what happens when we provide the result of the type reconstruction process at these points beforehand. This additional outside information is called the *certificate*. It becomes apparent that the type reconstruction is now reduced to a single linear pass over the instruction sequence: each time we would have to consider more than one path of execution, the result is already there and only needs to be checked, not constructed. The second effect is that apart from the certificate we only need constant memory: the type reconstruction can be reduced to a function that calculates the state type at $pc+1$ only from the state type at pc and the global information that is already provided from outside. After having calculated the type at $pc+1$, we can immediately forget about the one at pc.

For our example program, the situation at the start of the lightweight bytecode verification process looks like that:

instruction	stack local variables
Load 0	
Store 1	[Class A] [Class B, unusable]
Getfield F A	
Goto -2	

From that the whole method type is reconstructed in a single linear pass: The state type ([], [Class B, int]) for the Load instruction will be filled in as initialization. The state type for Store 1 is in the certificate, since Store is the target of the Goto -2 jump. The lightweight bytecode verifier calculates the effect of Load 0, i.e. ([Class B], [Class B, int]), and checks if the certificate ([Class A], [Class B, unusable]) correctly approximates this result. The types before execution of Getfield are then easily calculated from the state type and the effect of Store alone, i.e. the result is ([], [Class B, Class A]). The effect of Getfield F A also only needs the current state type and yields ([Class A], [Class B, Class A]). For the last instruction the lightweight bytecode verifier has to check if the calculated state type is correctly approximated by the jump target. We did not store this state type, but since it is a target of a jump, we have it in the certificate and only need to check if the certificate at this point correctly approximates our calculated state type. Note, that all paths of executions that entered into the merging for the state type of Store 1 were checked, but no iteration or additional memory was required.

3.1 Formalization

With that kind of process and certificate in mind, we can start a formalization of the lightweight bytecode verifier. We have two goals here: On the one hand, we want the formalization to be similar to the one of the traditional bytecode verifier, so we can easily spot commonalities and differences. On the other hand, we now not only want to model type checking, but also the simplified form of type reconstruction, i.e. we want functions, not predicates. As a solution, we write the predicates checking single instructions in a form that is similar to the traditional bytecode verifier, and that can still easily be read as a function. For example the predicate for Load

```
wtl_inst :: [instr,jvm_prog,ty,state_type,state_type,nat,nat] → bool
wtl_inst (Load idx) Γ rT s s' cert max_pc pc =
        let (ST,LT) = s in
                pc+1 < max_pc ∧ idx < length LT ∧
                (∃t. (LT ! idx) = usable t ∧ (t # ST , LT) = s')
```

can be read as a function yielding the next state type s' from the current instruction Load idx, the current state type s, the program counter pc, and the maximum program counter max_pc. Declaration context Γ, return type rT, and the certificate $cert$ are not used in the Load case. The predicate still closely mimics the corresponding wt_inst from the traditional byte code verifier: it is apparent, that we have the same applicability conditions and model the same effect the instruction has on the stack.

We now have a function that calculates the state type s' at $pc + 1$ from the state type s at pc. Iterating this process over the list of instructions we can then feed this s' as current state type to the next instruction:

```
wtl_inst_list :: [instr list,jvm_prog,ty,state_type,state_type,certificate,nat,nat] → bool
wtl_inst_list (i#is) Γ rT s0 s2 cert max_pc pc =
        (∃s1. wtl_inst_option i Γ rT s0 s1 cert max_pc pc ∧
              wtl_inst_list is Γ rT s1 s2 cert max_pc (pc+1))
```

wtl_inst_option is a simple case distinction: if there is already type information stored in the certificate at the current program counter, as for Store 1 in the example, we must not use our calculated type, but the certificate containing the merged type information instead. To ensure correctness, we still have to check, if the certificate correctly approximates the calculated state type, i.e. if the certificate really is the result of a merge of our state type with another one. Therefore we have:

```
wtl_inst_option :: [instr,jvm_prog,ty,state_type,state_type,certificate,nat,nat] → bool
wtl_inst_option i Γ rT s0 s1 cert max_pc pc ≡
    case cert!pc of
        None      → wtl_inst i Γ rT s0 s1 cert max_pc pc
      | Some s0' → (Γ ⊢ s0 ≼s s0') ∧ wtl_inst i Γ rT s0' s1 cert max_pc pc
```

3.2 Soundness

When we specify a new kind of bytecode verification we of course wish to know if this new bytecode verifier does the right thing. In our case this means: if the lightweight bytecode verifier accepts a piece of code as welltyped, the traditional bytecode verifier should accept it, too. We must also show that it is safe to rely on outside information, i.e. in the soundness proof we must not make any assumption on how the certificate was produced. So the soundness theorem is

$$\forall cert.\ \mathsf{wtl\_method}\,\Gamma\ \ldots\ cert \implies \exists\varphi.\ \mathsf{wt\_method}\,\Gamma\ \ldots\ \varphi$$

where Γ ... is shorthand for the same set of parameters, return type etc. for both judgments.

This means, that if the certificate was tampered with, the lightweight bytecode verifier either rejects the method as not welltyped, or if it does not reject, it was still able to reconstruct the method type correctly.

We prove this by constructing a φ from a successful run of the lightweight bytecode verifier and showing that this φ satisfies wt_method. φ must have the following properties: if the certificate contains a state type s at some point pc, φ contains that s at the same point pc. Otherwise, if the lightweight bytecode verifier has come to a position pc in its type reconstruction process and has calculated a current state type s, φ will contain that s at position pc.

If wtl_method holds, there clearly always is such a φ. By case distinction over all instructions we get that both bytecode verifiers compute the same effects of instructions on state types, and, because the certificate is always checked to correctly approximate the calculated state type, we get that for each instruction wt_inst holds. Thus the traditional bytecode verifier accepts.

3.3 Completeness

Of course, the trivial bytecode verifier that rejects all programs also would be correct in the sense above. Therefore we show that our lightweight bytecode verifier also is complete, i.e. that if a program is welltyped with respect to the traditional bytecode verifier, the lightweight bytecode verifier will accept the same program with an easy to obtain certificate.

How will this certificate look like? We get the information we need from the method type of a successful run of the traditional bytecode verifier. Since we want to minimize the amount of information we have to provide, we do not take the whole method type as the certificate, but only the state types at certain positions.

As in the example, the certificate should contain the type information at jump targets. Due to some simplifications in our formalization of the traditional bytecode verifier and the μJava language, this is not enough though. The first thing is, our traditional bytecode verifier does not ignore dead code, but requires instructions that can never be executed to be type correct, too. If the instruction directly after a Goto for instance is not a jump target, it can never be executed. Since the effect of Goto on the state type only tells us something about the target of the Goto, but nothing about the state type of the instruction at $pc + 1$, the lightweight bytecode verifier would have no means to construct this state type at $pc + 1$ if it wasn't in the certificate. So we also include the state types directly after Goto and Return instructions. Since dead code should be eliminated by the compiler anyway, this is not really an issue. On the other hand, it is not hard to take dead code into account and we plan to do so in the future. We also need the state type after a method invocation in some cases. This is due to the fact that we do not really model exceptions at the JVM level. In μJava, a method invocation on a class reference containing the value null is equivalent to a halt. If the bytecode verifier discovers that this class reference is always null, the instruction after that may again be dead code and we have to include it in the certificate. Again, programs produced by an optimizing compiler should not contain such cases.

So the certificate contains the targets of jumps and some rare cases, we have to include for the completeness proof, because we do not want to make any assumptions about how the code was produced.

Let make_cert be the function that produces such a certificate from an instruction sequence and a method type. Then

$$\mathsf{wt\_method}\,\Gamma\ \ldots\ ins\ \varphi\ \Longrightarrow\ \mathsf{wtl\_method}\,\Gamma\ \ldots\ ins\ (\mathsf{make\_cert}\ ins\ \varphi)$$

follows by induction over the length of the instruction sequence.

4 Conclusion

We have formalized a variant of lightweight bytecode verification for μJava and proved its soundness and completeness in Isabelle/HOL. Our formalization is comparatively easy to transform into a functional program. The completeness result is both stronger and weaker than that of [3]. Eva and Kristoffer Rose have a more complex formalization of the lightweight bytecode verifier that only needs the certificate when a type merge really produces a different type than calculated so far. Doing so could lead to a smaller type annotation of class files (although this claim would require formal proof). It does however not save space during the verification pass, since the state type at jump targets has to be saved for later checks anyway. Our completeness result on the other hand includes the simpler and easier to implement notion that (apart from artificial cases) the targets of jumps are all that is needed for linear type reconstruction.

References

1. T. Nipkow, D. v. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages ?–? IOS Press, 2000. To appear.
2. C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lect. Notes in Comp. Sci.*, pages 89–103. Springer-Verlag, 1999.
3. E. Rose and K. Rose. Lightweight bytecode verification. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.