# Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL

Cornelia Pusch*

Institut für Informatik, Technische Universität München
80290 München, Germany
http://www.in.tum.de/~pusch/

**Abstract.** Compiled Java programs may be downloaded from the World Wide Web and be executed on any host platform that implements the Java Virtual Machine (JVM). However, in general it is impossible to check the origin of the code and trust in its correctness. Therefore standard implementations of the JVM contain a *bytecode verifier* that statically checks several security constraints before execution of the code.
We have formalized large parts of the JVM, covering the central parts of object orientation, within the theorem prover Isabelle/HOL. We have then formalized a specification for a Java bytecode verifier and formally proved its soundness. While a similar proof done with paper and pencil turned out to be incomplete, using a theorem prover like Isabelle/HOL guarantees a maximum amount of reliability.

## 1 Introduction

The Java Virtual Machine (JVM) is an abstract machine consisting of a memory architecture and an instruction set. It is part of the Java language design developed by Sun Microsystems and serves as a basis for Java implementations. However, it also can be used as intermediate platform for other programming languages, since the JVM works independently of Java. The corresponding compiler then generates architecture-independent JVM code instead of machine code for a specific host platform. This approach allows execution of compiled JVM code on any host platform that implements the JVM. However, this advantage does not come without risks. One can download any JVM code from the World Wide Web, and in general it is impossible to check the origin of the code and trust in its correctness.

The Java Virtual Machine Specification (short JVMS) [LY96] describes a set of static and structural constraints that must hold for the code to assure safe execution, and requires that the JVM itself verifies that these constraints hold. However, this is not a formal specification, and it is in the nature of informal descriptions to contain ambiguities or even inconsistencies. Our goal is to give a fully formal specification of the JVM and a bytecode verifier that overcomes

* Research supported by DFG project *Bali*.

W.R. Cleaveland (Ed.): TACAS/ETAPS'99, LNCS 1579, pp. 89–103, 1999.

this problem. We think that this work can be useful in several aspects: on the one hand it allows the formal investigation of central concepts of the JVM, such as the correctness of the bytecode verifier and compiler verification; on the other hand it may serve as reference specification that is more accurate than the informal description.

Formalizing a real life programming language is a very complex task and it is likely that an approach done with paper and pencil also will be susceptible to more or less grave errors. Therefore, tool assistance is required to reach a maximum amount of reliability. A theorem prover like Isabelle/HOL [Pau94, Isa] offers valuable support in developing consistent specifications and correct proofs.

To avoid the execution of incorrect JVM code, several verification strategies for JVM code may be used, for example:

- Cohen [Coh97] has implemented a so called *defensive* JVM using the theorem prover ACL2. In this approach runtime checks are performed to guarantee a type-safe execution of the code.
- The JVMS [LY96] describes Sun's implementation of a bytecode verifier, where most of the type-checking is done statically but several parts are delayed until runtime.
- Qian [Qia98] has developed a specification for an extended bytecode verifier, where all type-checking is done statically.

The specification of a bytecode verifier in Isabelle/HOL presented in this paper follows Qian's work. However, our formalization of the operational semantics [Pus98] has been done independently of Qian's approach. Therefore we had to deviate from Qian's work in several points to make it fit to our approach.

There are several other approaches to formalize (parts of) the JVM (see [Ber97, FM98, Gol97, HBL98, SA98]). As far as we know, our work is the first to formally prove the soundness of a bytecode verifier using a theorem prover.

The rest of the paper is organized as follows: section 2 briefly introduces Isabelle/HOL. Section 3 describes our formalization of the JVM, in particular the representation of runtime data and the definition of an operational semantics for the JVM instructions. In section 4 we introduce the notion of static well-typedness and give a formal specification for a bytecode verifier. Section 5 defines the notion of soundness for a bytecode verifier and sketches the corresponding soundness proof. In section 6 we discuss two extensions we have added to the specification, and section 7 summarizes our results and outlines future work.

## 2 Isabelle/HOL

Isabelle [Pau94, Isa] is a generic theorem prover that can be instantiated with different object logics. The formalization and proofs described in this paper are based on the instantiation for Higher Order Logic, called Isabelle/HOL. Subsequently we give an overview over the basic types and functions used in this paper.

Isabelle's type system is very similar to that of ML, with slight syntactic differences: function types are denoted by $\tau_1 \Rightarrow \tau_2$, where $\tau_1 \Rightarrow \tau_2 \Rightarrow \ldots \Rightarrow \tau_n$ may be abbreviated as $[\tau_1,\tau_2,\ldots] \Rightarrow \tau_n$. Product types are written as $\alpha \times \beta \times \gamma$.

Functions are preferably defined in a curried style (i.e. *f a b c*). Occasionally we have to define uncurried functions *f (a, b, c)*; this is due to restrictions of Isabelle's package for well-founded recursive functions.

The basic types *bool*, *nat* and *int* are predefined. Isabelle/HOL also offers the polymorphic types $\alpha$ *set* (with the usual set operators) and $\alpha$ *list*. The list constructors are [] ('nil') and *x#xs* ('cons'). The functions hd *xs* and tl *xs* return the head and tail of a list. The *i*-th list element is written *xs* ! *i*, length *xs* computes the length of a list, and set *xs* converts a list into a (finite) set. We also have map *f xs* to apply a function to all elements of a list, and zip *xs ys* takes two lists and returns a list of pairs.

Inductive datatypes can be defined by enumerating their constructors together with their argument types. For example, the predefined datatype for optional values looks as follows:

$\alpha$ *option* = None | Some $\alpha$

In Isabelle/HOL, all functions are total. Partiality can be modeled using the predefined 'map' type which is defined as follows:

$\alpha \rightsquigarrow \beta = (\alpha \Rightarrow \beta$ *option*$)$

We use the infix operator !! of type $[\alpha \rightsquigarrow \beta,\alpha] \Rightarrow \beta$ for 'partial' function application. Whenever $f\ x =$ Some $y$ then $f$ !! $x = y$. In the case of None the result will be an unknown value arbitrary, defined as $\varepsilon x.$ False (where $\varepsilon$ is Hilbert's description operator).

Throughout this paper, we write logical constants in sans serif, whereas variables and types appear in *italic*.

## 3 The Java Virtual Machine

JVM code is stored in so called *classfiles*. If the code is produced by compilation of a Java program, each Java class is translated into a separate classfile. Similar to Java classes, a JVM classfile contains information about inheritance and implementation relations, as well as field and method definitions. Method code consists of a sequence of JVM instructions (*bytecode*). The machine model of the JVM has different memory areas for runtime data: a *heap* stores runtime objects and a *frame stack* contains state information for each active method invocation. Each method frame has its own *operand stack* and *local variables* array. Similar to Java, the JVM has an exception mechanism to treat error conditions. In our formalization, we consider a set of predefined exceptions, but do not yet treat exception handling.

We have formalized large parts of the JVM, including the classfile structure and the operational semantics for a subset of JVM instructions covering the central parts of object orientation. Due to lack of space, we cannot present the entire formalization that can be found in [Pus98, NOP]. However, we introduce the main ideas of our approach.

### 3.1 JVM classfiles

The first component of a classfile consists of the *constant pool*, a kind of symbol table containing name and type information. This is followed by a flag indicating whether the classfile describes an interface or a class, several pointers to constant pool entries returning the names of the current class, its superclass and direct superinterfaces, and finally the field and method definitions:

$$\alpha\ classfile = cpool \times iflag \times idx \times idx \times idx\ list \times fields \times \alpha\ methods$$

The type for methods is parameterized over the type of the method code, which may be instantiated later. This allows us to formalize the JVM instruction set and its operational semantics in a modular way.
A predicate wf_classfiles checks the well-formedness of classfiles, e.g. the superclass and superinterface relations must be acyclic and method overriding must obey certain type restrictions.

**Example:** Consider a set of classfiles (see figure 1) consisting of class Object, as well as the classes C0, C1, C2, and Q. C0 and Q are direct subclasses of Object; C1 and C2 are both extensions of C0. Class C0 contains an integer field f0, class Q contains a method m.

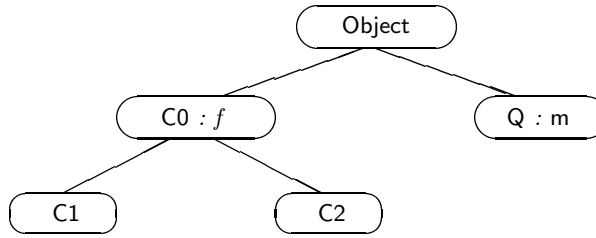


**Fig. 1.** Example class hierarchy

Figure 2 shows the contents of classfile Q. The interface flag is set to False, cpool index 1 points to the name of class Q. This information extends over two entries: the keyword Class indicates the entry type, index 9 points then to another entry containing the string Q (with keyword Utf8). The superclass index points in the same way to class name Object. The description of method m contains again two pointers. The first one returns name m, the second one points to a *type descriptor*. In our case, method m gets two arguments of type C1 and C2 and returns an integer. The code section m_code will be shown later.

### 3.2 JVM Runtime Data

The JVM operates on two different types of values, primitive values and reference values. We consider only primitive values, of type integer. The reference values are pointers to objects, the null pointer is expressed by a special null reference. The realization of object references is kept abstract: we model them by an opaque type *loc* that is not further specified. We define a datatype for JVM values as follows:

| *cpool* | cpool |
|---|---|
| *iflag* | False |
| $idx_{class}$ | 1 |
| $idx_{super}$ | 2 |
| $idx_{inter}$ | [] |
| *fields* | [] |
| *methods* | (7, 10, m_code) |

| | cpool |
|---|---|
| 1 | Class 9 |
| 2 | Class 8 |
| ⋮ | ⋮ |
| 7 | Utf8 m |
| 8 | Utf8 Object |
| 9 | Utf8 Q |
| 10 | Utf8 ([L C1,L C2],I) |
| ⋮ | ⋮ |

**Fig. 2.** Classfile for Q

$val =$ Intg $int$ | Addr $loc$ | Null

You may have noticed that in contrast to our formalization, the JVMS [LY96] does not require values to be tagged with their runtime types. However, our approach does not impose any restrictions on possible implementations, because the type information is not used to determine the operational semantics of (correct) JVM code. We use the type tags only to state and prove the correctness of the bytecode verifier, where the runtime types are checked against the static type information.

### 3.3 Operational Semantics of JVM Instructions

The JVMS [LY96] describes the operational semantics for each instruction in the context of a JVM state where several constraints hold, e.g. there must be an appropriate number of arguments on the operand stack, or the operands must be of a certain type. If the constraints are not satisfied, the behaviour of the JVM is undefined.

In our approach, we formalize the behaviour of JVM instructions with total functions. If a state does not satisfy the constraints of the current instruction, e.g. if an element should be popped from an empty operand stack, the result will be the unknown value arbitrary.

We have structured the instructions into several groups of related instructions, describing each by its own execution function. This makes the operational semantics easier to understand, since every function only works on the parameters that are needed for the corresponding group of instructions:

$instr$ = LAS $load\_and\_store$ | CO $create\_object$ | MO $manipulate\_object$
| MA $manipulate\_array$ | CH $check\_object$ | MI $meth\_inv$
| MR $meth\_ret$ | OS $op\_stack$ | CB $cond\_branch$ | UB $uncond\_branch$

Now, we can instantiate the type parameter for the code section of a classfile and introduce the following type abbreviation, describing a partial mapping from class names to classfiles:[1]

$$classfiles = ident \rightsquigarrow (instr\ list)\ classfile$$

**Example:** The code of method m is shown in figure 3. Aload $i$ loads the content of local variable $i$ onto the operand stack. Ifnull 3 compares the top operand stack element against Null and performs a conditional jump to $pc = pc+3$. Goto 2 performs an unconditional jump to $pc = pc+2$. Getfield 4 loads a field described at cpool entry 4 onto the operand stack (which is in our example integer field f0). Finally, Ireturn closes the current method invocation and returns the integer result to the calling method.

| *pc* | *instr* |
|---|---|
| 0 | Aload 1 |
| 1 | Ifnull 3 |
| 2 | Aload 1 |
| 3 | Goto 2 |
| 4 | Aload 2 |
| 5 | Getfield 4 |
| 6 | Ireturn |

**Fig. 3.** Code of method m

Execution of a JVM instruction transforms the machine state. The machine state is formalized as a triple consisting of an exception flag, an object heap, and a frame stack. For each active method invocation, there exists a frame containing its own operand stack, a list of local variables, the name of the current class, a reference to the current method, and the program counter:

$$\begin{aligned} frame &= opstack \times locvars \times ident \times method\_loc \times pc \\ jvm\_state &= xcpt\ option \times heap \times frame\ list \end{aligned}$$

If an exception has been raised or the frame stack is empty, execution terminates.[2] If the machine has not yet reached a final state, the function exec performs a single execution step: it calls an appropriate execution function (e.g. exec_mo) and incorporates the result in the new machine state. If execution has reached a final state, exec does not return a new state. This is modeled by embedding the result state in an *option* type:

exec :: $classfiles \times jvm\_state \Rightarrow jvm\_state\ option$
exec ($CFS$, (Some $xp$, $hp$, $frs$)) = None
exec ($CFS$, (None, $hp$, [])) = None
exec ($CFS$, (None, $hp$, ($stk$,$loc$,$cn$,$ml$,$pc$)#$frs$)) =
  case (get_code $CFS$ $cn$ $ml$) ! $pc$ of MO $ins$ ⇒ Some ( ... exec_mo ...) | ...

[1] We have abstracted from the size of instructions and regard the code section as a list of instructions.

[2] We do not yet treat exception handling.

For example, the operational semantics of the Getfield instruction for object field access looks like this:

```
exec_mo :: [manipulate_object,classfiles,cpool,heap,opstack,pc]
             ⇒ (xcpt option × heap × opstack × pc)
exec_mo (Getfield idx) CFS cp hp stk pc =
  let oref       = hd stk;
      (cn,od)    = get_Obj (hp !! (get_Addr oref));
      (fc,fn,fd) = extract_Fieldref cp idx;
      xp'        = if oref=Null then Some NullPointer else None
  in
  (xp' , hp , (od !! (fc,fn))#(tl stk) , pc+1)
```

*CFS* denotes a set of JVM classfiles. The operand stack *stk* is supposed to contain a reference to a class instance stored on the heap *hp*. In case of a null reference an exception is thrown. Otherwise, the referenced object contains class name *cn* and object data *od*. Index *idx* should point to a Fieldref entry in the constant pool *cp*, containing a class name *fc*, a field name *fn* and a field descriptor *fd*. The tuple (*fc*,*fn*) determines the field whose value is stored on the operand stack. Finally, the program counter *pc* is incremented.

Execution of the entire code then consists of repeated application of exec as long as the result is not None. The relation $CFS \vdash \sigma \longrightarrow^* \sigma'$ maps a given set of classfiles *CFS* and a JVM state $\sigma$ to a new state $\sigma'$, where the pair $(\sigma,\sigma')$ is in the reflexive transitive closure of successful execution steps:

```
_ ⊢ _ ⟶* _ :: [classfiles,jvm_state,jvm_state] ⇒ bool
CFS ⊢ σ ⟶* σ' ≝ (σ,σ') ∈ {(s,t). exec (CFS,s) = Some t}*
```

## 4 A Specification for a Bytecode Verifier

Standard implementations of the JVM contain a *bytecode verifier* that statically checks several security constraints before execution of the code. One main aspect of the bytecode verifier is to statically derive the types of possible runtime data and check that all instructions will get arguments of the correct type.

### 4.1 Static types

As Qian has pointed out in his work [Qia98], the attempt to statically type-check JVM code requires the introduction of reference type sets instead of single types. This is due to the fact that, as a result of a branching instruction, a program point may have multiple preceeding program points. These predecessor points are allowed to contain values of different types.[3] In this case, the types of the

[3] Surprisingly, the typing rule for the similar working conditional expression of the Java source language turns out to be more restricted (see [GJS96] and the discussion at [Typ]): it requires that the two branches yield two types where the first is a supertype of the second or vice versa.

two branches have to be merged to the first common supertype. However, the JVM allows multiple inheritance of interfaces, and therefore this supertype is not necessarily unique.

Qian defines a static type system including types representing adresses of subroutine calls and uninitialized objects. We do not yet consider these aspects of the JVM, but have added array types. Static types are represented as values of datatype *tys*. Among the primitive types, we only consider type Integer. A reference type is either the type of the null reference (NT), or an interface or class name (IT *id* or CT *id*), or an array type (AT *ts*, where *ts* contains the type of the components of the array). A static type consists then either of a primitive type or a list of reference types.[4] During bytecode verification, type information of different execution paths has to be merged. In case of incompatible types, the result becomes unusable. This is expressed by a value of type *any*, which is either a static type or Unusable. The return type of methods is denoted by a value of type *tyOrVoid*, which is either a static type or Void:

$$
\begin{array}{ll}
prim & = \mathsf{Integer} \\
ref & = \mathsf{NT} \mid \mathsf{IT}\ ident \mid \mathsf{CT}\ ident \mid \mathsf{AT}\ tys \\
tys & = \mathsf{PTS}\ prim \mid \mathsf{RTS}\ (ref\ list) \\
any & = \mathsf{Unusable} \mid \mathsf{US}\ tys \\
tyOrVoid & = \mathsf{Void} \mid \mathsf{TY}\ tys
\end{array}
$$

We abbreviate US (PTS $p$) and US (RTS $r$) by Prim $p$ and Refs $r$.

If two types are merged, the resulting supertype must *cover* both types. A type $a$ covers a type $a'$ (written $CFS \vdash a \sqsupseteq a'$), if any instruction that is applicable to all values of type $a$ is also applicable to all values of type $a'$. The predicate holds in the following cases:

$$
\begin{array}{l}
\_ \vdash \_ \sqsupseteq \_ :: [classfiles, any, any] \Rightarrow bool \\
CFS \vdash \mathsf{Unusable} \quad \sqsupseteq a' \\
CFS \vdash \mathsf{Prim\ Integer} \sqsupseteq \mathsf{Prim\ Integer} \\
CFS \vdash \mathsf{Refs}\ rs \quad \sqsupseteq \mathsf{Refs}\ rs' = (\forall r' \in \mathsf{set}\ rs'.\ \exists r \in \mathsf{set}\ rs.\ \mathsf{widenConv}\ CFS\ r'\ r)
\end{array}
$$

Qian gives a more restrictive definition identifying the covering of reference types with the superset relation. In our definition, an element of the subtype needs not be contained in the supertype, it just must be convertible to one of its elements.

A *state type* contains type information for all local variables and the operand stack of the current invocation frame at a certain program point. The local variables may contain unusable values (as a result of merging two incompatible types), whereas only usable values may be stored on the operand stack. We extend the predicate $\sqsupseteq$ in two steps to state types:

$$state\_type = tys\ list \times any\ list$$

$$
\begin{array}{l}
\_ \vdash \_ \sqsupseteq \_ :: [classfiles, any\ list, any\ list] \Rightarrow bool \\
CFS \vdash as \sqsupseteq as' \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{length}\ as = \mathsf{length}\ as' \wedge \forall (a,a') \in \mathsf{set}\ (\mathsf{zip}\ as\ as').\ CFS \vdash a \sqsupseteq a'
\end{array}
$$

[4] Due to restrictions to the construction of inductive datatypes, we model reference type sets as lists.

$\_ \vdash \_ \sqsupseteq \_ :: [classfiles, state\_type, state\_type] \Rightarrow bool$
$CFS \vdash (ST,LT) \sqsupseteq (ST',LT') \stackrel{\text{def}}{=}$
$\quad CFS \vdash \mathsf{map\ US}\ ST \sqsupseteq \mathsf{map\ US}\ ST' \wedge CFS \vdash LT \sqsupseteq LT'$

Type information for the entire code of a method is collected in a value of *method type*. A value of *class type* maps a method reference to a value of method type, and a value of *program type* maps a class name to a value of class type:

$method\_type = state\_type\ list$
$class\_type = method\_loc \Rightarrow method\_type$
$prog\_type = ident \Rightarrow class\_type$

### 4.2 Static Well-typedness

A bytecode verifier has to infer type information for each instruction and then check if the method code is well-typed. In our specification, well-typedness is checked with respect to a given type. A correct implementation of that specification must then compute a type that is well-typed according to the specification.

We define a type checking predicate that checks whether an instruction at a certain program point is well-typed with respect to a given method type. Additionally, it checks several other constraints, e.g. an index to local variables must not be greater than the number of local variables and the program counter must remain within the current method. These constraints are indispensable to carry out the soundness proof for the bytecode verifier. The type-checking predicate makes a case distinction over the instruction to be executed at the current program point. In case of Getfield, the instruction is well-typed if the following predicate holds:

$\mathsf{wt\_MO} :: [manipulate\_object, classfiles, cpool, method\_type, pc, pc] \Rightarrow bool$
$\mathsf{wt\_MO}\ (\mathsf{Getfield}\ idx)\ CFS\ cp\ \Delta\ max_{pc}\ pc =$
$\quad \mathsf{let}\ (ST,LT) = \Delta\ !\ pc;$
$\qquad (fc,fn,fd) = \mathsf{extract\_Fieldref}\ cp\ idx$
$\quad \mathsf{in}$
$\quad pc{+}1 < max_{pc} \wedge \mathsf{is\_class}\ CFS\ fc \wedge$
$\quad \mathsf{get\_fields}\ (CFS\ !!\ fc)\ (fc,fn) = \mathsf{Some}\ fd \wedge$
$\quad \exists rs\ ST'.\ ST = (\mathsf{RTS}\ rs)\ \#\ ST' \wedge$
$\qquad\qquad \mathsf{widenConv}\ CFS\ rs\ [\mathsf{CT}\ fc] \wedge$
$\qquad\qquad CFS \vdash \Delta\ !\ (pc{+}1) \sqsupseteq (fd\ \#\ ST',\ LT)$

All well-typedness predicates contain a line of the form $CFS \vdash \Delta\ !\ (pc{+}1) \sqsupseteq type$, which means that the next instruction expects a type according to *type*. Since that next instruction has possibly other predecessors, its type information is not necessarily equal to *new_type*, but rather must cover it.
The above predicate enforces that the incremented program counter $pc{+}1$ does not exceed the code length $max_{pc}$. The class *fc* must be defined and must contain a field with name *fn* according to the constant pool entry. The stack must not be empty and the top stack element must contain a reference type convertible to the type of *fc*. Finally, the next instruction must expect a type according to the field descriptor *fd* on top of the operand stack.

Similarly to the execution function exec, we define a predicate wt_instr that selects the appropriate well-typedness predicate for each group of instructions. We extend the notion of well-typedness to methods, classes, and programs: at the beginning of a method body, the operand stack must be empty, and the local variables must contain values according to the type of the current class *cn* and the parameter descriptor *pd* of the current method:

wt_start :: [*classfiles,ident,param_desc,method_type*] ⇒ *bool*
wt_start *CFS cn pd* Δ $\stackrel{\text{def}}{=}$
  *CFS* ⊢ Δ ! 0 ⊒ ([],(Refs [CT *cn*])#(map (fd2any *CFS*) *pd*))

The code array of a method must not be empty, i.e. its length must be greater than zero. A method is well-typed with respect to a method type Δ, if it is well-typed at the beginning of the method body, and if for every program point in the method body the instruction is well-typed:

wt_method :: [*classfiles,ident,param_desc,return_desc,instr list,method_type*] ⇒ *bool*
wt_method *CFS cn pd rd ins* Δ $\stackrel{\text{def}}{=}$
  let *cp* = get_cpool (*CFS* !! *cn*);
    $max_{pc}$ = length *ins*
  in
  0 < $max_{pc}$ ∧ wt_start *CFS cn pd* Δ ∧
  ∀ *pc*. *pc*<$max_{pc}$ ⟶ wt_instr (*ins* ! *pc*) *CFS rd cp* Δ $max_{pc}$ *pc*

**Example:** Method m is well-typed with respect to the method type shown in figure 4. The Getfield instruction at *pc*=5 requires an element of reference type on top of the operand stack. This may have been put there either by the Aload 1 instruction at *pc*=2 or by the Aload 2 instruction at *pc*=4. This is reflected by the static type *ST* ! 5 = [RTS [CT C0]], which covers both possibilities[5].

| *pc* | *ST* | *LT* |
|---|---|---|
| 0 | [] | [Refs [CT Q],Refs [CT C1],Refs [CT C2]] |
| 1 | [RTS [CT C1]] | [Refs [CT Q],Refs [CT C1],Refs [CT C2]] |
| 2 | [] | [Refs [CT Q],Refs [CT C1],Refs [CT C2]] |
| 3 | [RTS [CT C1]] | [Refs [CT Q],Refs [CT C1],Refs [CT C2]] |
| 4 | [] | [Refs [CT Q],Refs [CT C1],Refs [CT C2]] |
| 5 | [RTS [CT C0]] | [Refs [CT Q],Refs [CT C1],Refs [CT C2]] |
| 6 | [PTS Integer] | [Refs [CT Q],Refs [CT C1],Refs [CT C2]] |

**Fig. 4.** Static type of method m

A class is well-typed with respect to a class type Γ, if every method defined in that class is well-typed with respect to the corresponding method type:

wt_class :: [*classfiles,ident,class_type*] ⇒ *bool*
wt_class *CFS cn* Γ $\stackrel{\text{def}}{=}$
  ∀*ml rd ins*. get_methods (*CFS* !! *cn*) *ml* = Some (*rd,ins*)
    ⟶ wt_method *CFS cn* (*snd ml*) *rd ins* (Γ *ml*)

[5] *ST* ! 5 = [RTS [CT C1,CT C2]] is also a correct type.

A JVM program is well-typed with respect to a program type $\Phi$, if every defined class is well-typed with respect to the corresponding class type:

$$\mathsf{wt\_classfiles} :: [\mathit{classfiles}, \mathit{prog\_type}] \Rightarrow \mathit{bool}$$
$$\mathsf{wt\_classfiles}\ \mathit{CFS}\ \Phi \stackrel{\mathsf{def}}{=} \forall \mathit{cn}.\ \mathsf{is\_class}\ \mathit{CFS}\ \mathit{cn} \longrightarrow \mathsf{wt\_class}\ \mathit{CFS}\ \mathit{cn}\ (\Phi\ \mathit{cn})$$

## 5 Soundness of the Bytecode Verifier Specification

A bytecode verifier (or more abstract: a type system) statically determines the types of all runtime data. A type system is sound, if the statically predicted type gives a correct approximation of a runtime value produced during execution.[6]

In this section, we will show that our specification of a bytecode verifier is sound. For a concrete implementation of a bytecode verifier, it then remains to be proved that it satisfies our specification.

### 5.1 Correct Approximation of Runtime Values

In our formalization, runtime values carry some type information (see §3.2), whereas Qian has to go through the code and assign a type tag to each value depending on the instruction it has been created by. However, he only gives an informal motivation that indeed all runtime values can be associated with a tag. Therefore, our correctness relation between runtime data and static types differs from that given in [Qia98]:

$$\begin{aligned}
&\mathsf{approx_val} :: [\mathit{classfiles}, \mathit{heap}, \mathit{val}, \mathit{any}] \Rightarrow \mathit{bool}\\
&\mathsf{approx_val}\ \mathit{CFS}\ \mathit{hp}\ (\mathsf{Intg}\ i) \quad \mathit{at} = \mathit{CFS} \vdash \mathit{at} \sqsupseteq \mathsf{Prim\ Integer}\\
&\mathsf{approx_val}\ \mathit{CFS}\ \mathit{hp}\ \mathsf{Null} \quad \mathit{at} = \exists \mathit{rs}.\ \mathit{CFS} \vdash \mathit{at} \sqsupseteq \mathsf{Refs}\ \mathit{rs}\\
&\mathsf{approx_val}\ \mathit{CFS}\ \mathit{hp}\ (\mathsf{Addr}\ a)\ \mathit{at} = \exists\ \mathit{obj}.\ \mathit{hp}\ a = \mathsf{Some}\ \mathit{obj} \wedge\\
&\qquad\qquad \mathit{CFS} \vdash \mathit{at} \sqsupseteq (\mathsf{fd2any}\ \mathit{CFS}\ (\mathsf{get_obj_type}\ \mathit{obj}))
\end{aligned}$$

An integer value must have static type Integer or Unusable. The Null reference is approximated by any reference type or Unusable, and in case of an object reference Addr $a$, the corresponding object type must be a subtype of the static type.

This notion of correct approximation is extended to local variables and the operand stack:

$$\begin{aligned}
&\mathsf{approx_loc} :: [\mathit{classfiles}, \mathit{heap}, \mathit{locvars}, \mathit{any\ list}] \Rightarrow \mathit{bool}\\
&\mathsf{approx_loc}\ \mathit{CFS}\ \mathit{hp}\ \mathit{loc}\ \mathit{LT} \stackrel{\mathsf{def}}{=}\\
&\quad \mathsf{length}\ \mathit{loc} = \mathsf{length}\ \mathit{LT} \wedge \forall (\mathit{val}, \mathit{any}) \in \mathsf{set}\ (\mathsf{zip}\ \mathit{loc}\ \mathit{LT}).\ \mathsf{approx_val}\ \mathit{CFS}\ \mathit{hp}\ \mathit{val}\ \mathit{any}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{approx_stk} :: [\mathit{classfiles}, \mathit{heap}, \mathit{opstack}, \mathit{tys\ list}] \Rightarrow \mathit{bool}\\
&\mathsf{approx_stk}\ \mathit{CFS}\ \mathit{hp}\ \mathit{stk}\ \mathit{ST} \stackrel{\mathsf{def}}{=}\\
&\quad \mathsf{length}\ \mathit{stk} = \mathsf{length}\ \mathit{ST} \wedge\\
&\quad \forall (\mathit{val}, \mathit{tys}) \in \mathsf{set}\ (\mathsf{zip}\ \mathit{stk}\ \mathit{ST}).\ \mathsf{approx_val}\ \mathit{CFS}\ \mathit{hp}\ \mathit{val}\ (\mathsf{US}\ \mathit{tys})
\end{aligned}$$

[6] This is often formulated as 'runtime data must be correct with respect to its static type'. Technically, there is no difference, but we regard our view as more intuitive.

### 5.2 Soundness Proof

Qian states a soundness theorem saying that for statically well-typed bytecode, the correctness relation between runtime values and static types of the current operand stack and local variables will be preserved in every execution step. However, his proof given in [Qia97] remains sketchy, and it turnes out that the theorem cannot be proved in the given form. A stronger proof invariant has to be formulated, assuring the correctness not only of the current operand stack and local variables, but the correctness of the entire state containing all active invocation frames. In particular, the method executed in the (n+1)-th frame must correspond to a method invocation of the n-th frame.

We define several auxiliary predicates to formulate the correctness of all state components: in a *correct heap*, all objects contain correct data:

```
correct_obj :: [classfiles,heap,obj] ⇒ bool
correct_obj CFS hp (Obj cn od) =
  is_class CFS cn ∧
  ∀fl fd. (get_all_fields (CFS,cn)) fl = Some fd
        ⟶ ∃val. od fl = Some val ∧ approx_val CFS hp val (fd2any CFS fd)
correct_obj CFS hp (Arr fd ad ⇒) =
  ∀val∈set ad. approx_val CFS hp val (fd2any CFS fd)

correct_heap :: [classfiles,heap] ⇒ bool
correct_heap CFS hp ≝ ∀a obj. hp a = Some obj ⟶ correct_obj CFS hp obj
```

The predicate correct_frame checks whether the operand stack entries *stk* and local variables *loc* have been approximated correctly by the state type (*ST,LT*). Additionally, the frame itself must be well-formed, i.e. the class *cn* is defined, the method reference *ml* points to an existing method, and the program counter *pc* points to an instruction inside the method code:

```
correct_frame :: [classfiles,heap,state_type,frame] ⇒ bool
correct_frame CFS hp (ST,LT) (stk,loc,cn,ml,pc) ≝
  approx_stk CFS hp stk ST ∧ approx_loc CFS hp loc LT ∧ is_class CFS cn ∧
  ∃rd ins. get_methods (CFS !! cn) ml = Some (rd,ins) ∧ pc < length ins
```

The predicate correct_frames checks whether a method reference $ml_{n+1}$ and a return descriptor $rd_{n+1}$ (belonging to frame $f_{n+1}$) fit to the next frame $f_n$ of the frame stack. If the frame stack is empty, the method must have return type *void* (i.e. return descriptor V). If there exists a frame $f_n$, the last executed instruction must have invoked method $ml_{n+1}$ with return type $rd_{n+1}$. Besides that, $f_n$ itself must be correct. These checks are performed recursively on the remaining stack:

```
correct_frames :: [classfiles,heap,prog_type,return_desc,method_loc,frame list] ⇒ bool
correct_frames CFS hp Φ rd_{n+1} ml_{n+1} [] = (rd_{n+1}=V)
correct_frames CFS hp Φ rd_{n+1} ml_{n+1} (f_n#frs) =
  let (stk,loc,cn,ml,pc) = f_n;
      (rd,ins)    = get_methods (CFS !! cn) !! ml;
      cp          = get_cpool (CFS !! cn);
      (ST,LT) = (Φ cn ml) ! pc
```

```
  in
  ∃mi c k l. pc = k+1 ∧ ins ! k = MI mi ∧ extract_meth cp mi = (c,ml_{n+1},rd_{n+1},l) ∧
  correct_frame CFS hp (pop_rd CFS rd_{n+1} ST, LT) f_n ∧
  correct_frames CFS hp Φ rd ml frs
```

The entire state is correct, if an exception has been thrown or the frame stack is empty. In case of a nonempty frame stack, the heap must be correct, the top level frame $f_{n+1}$ must be correct, and the remaining frame *frs* must be correct with respect to the method $ml_{n+1}$ executed on the top level frame and its return descriptor $rd_{n+1}$:

```
correct_state :: [classfiles,prog_type,jvm_state] ⇒ bool
correct_state CFS Φ (Some x,hp,frs)
correct_state CFS Φ (None,hp,[])
correct_state CFS Φ (None,hp,f_{n+1}#frs) =
  let (stk,loc,cn,ml_{n+1},pc) = f_{n+1};
      (rd_{n+1},ins) = get_methods (CFS !! cn) !! ml_{n+1}
  in
  correct_heap CFS hp ∧
  correct_frame CFS hp ((Φ cn ml) ! pc) f_{n+1} ∧
  correct_frames CFS hp Φ rd_{n+1} ml_{n+1} frs
```

Now we can prove the following main soundness theorem:

$$\begin{array}{c} \mathsf{wf\_classfiles}\ \mathit{CFS} \wedge \mathsf{wt\_classfiles}\ \mathit{CFS}\ \Phi \wedge \\ \mathsf{correct\_state}\ \mathit{CFS}\ \Phi\ \sigma \wedge \mathit{CFS} \vdash \sigma \longrightarrow^* \sigma' \\ \Longrightarrow \mathsf{correct\_state}\ \mathit{CFS}\ \Phi\ \sigma' \end{array}$$

It says that for a set of well-formed classfiles *CFS* that are statically well-typed with program type $\Phi$, program execution in a correct state $\sigma$ leads to correct states $\sigma'$.[7] This means that starting from a correct initial state (invoking the main method of the executed class), all possible runtime data for a program *CFS* is correctly approximated by its static type $\Phi$. Inspecting the definitions of well-typedness and correct approximation, we are able to conclude that all required constraints will be satisfied at runtime, e.g. in case of the Getfield instruction, the top operand stack element will be a reference value Null or Addr *a*.

The proof of the main theorem has been carried out by induction over $\mathit{CFS} \vdash \sigma \longrightarrow^* \sigma'$. Then the preservation of the correctness property for a single execution step had to be shown by case distinction over the instructions.

## 6 Extensions to the Bytecode Verifier Specification

A bytecode verifier implementing our specification rejects bytecode that would not do any harm at runtime. Of course, it is not possible to build a complete static

[7] Remember that in our formalization, execution of a program is guaranteed by definition, since we modeled it using total functions.

type system, since static well-typedness is undecidable. However, we can eliminate two unnecessary restrictions in our specification: instructions that are not reachable, i.e. dead code, may be neglected during type-checking, and operand stack values may be of type Unusable if they are not used for further computation. In fact, optimizing compilers will detect dead code and eliminate it. However, bytecode may stem from other sources, e.g. may be hand-written. Besides that, we wanted to check the modularity of our proofs: a modification of our specification should not entail too much adaptions of our proof script.

Therefore, we have defined a predicate reach :: [*instr list,nat*] ⇒ *bool*. It checks whether a certain program point may be reached from the starting point. We have then replaced in our definition of wt_method the premise $pc <$ length *ins* by reach *ins pc*. Due to this, we had to adapt our proof invariant: a correct state now only contains reachable program points. We could then prove the new correctness statement by using an additional lemma, stating that any reachable state leads to another reachable state. The existing lemmas were not affected.

Our second extension, the introduction of possibly Unusable values on the operand stack, did not impose any changes to the proofs at all. It strikes positively that the formalization gets more readable, since operand stack and local variables are now treated in a uniform way, admitting both values of type *any*.

## 7 Results and Further Work

We have given a fully formal specification for the JVM and a bytecode verifier, and then formally proved the soundness of the bytecode verifier using the theorem prover Isabelle/HOL. The formalization of the JVM classfile structure and the operational semantics comprises about 1000 lines, the specification of the bytecode verifier took another 500 lines. The proof scripts contain approximately 2400 lines. It took about 6 month to develop the formalization and conduct the proof. The most complex parts of the proof concern the instructions for field access and method invocation, where the existence of a field or method for some static type must assure that an appropriate field or method can be found at runtime.

Isabelle/HOL turned out to be an adequate instrument to model real life programming languages such as Java (see also [ON98]). It is obvious that we had to make certain restrictions in this first approach to formalize the JVM. For example we do not consider the size of instructions and its operands and use instead abstract datatypes. These abstractions can be refined in further development steps of our formalization.

As next steps, we want to extend our formalization and the proof to subroutine call and object initialization. The work done by Qian [Qia98], Stata and Abadi [SA98], and Freund and Mitchell [FM98] showed that these constructs form the most complex part of bytcode verification, and therefore are worth a fully formal investigation using a theorem prover.

**Acknowledgments.** I would like to thank Tobias Nipkow, David von Oheimb, and Zhenyu Qian for helpful discussions about this topic. Thanks are also

owed to Wolfgang Naraschewski, Bernd Grobauer, Markus Wenzel, and Franz Regensburger who read a draft version of this paper.

## References

[Ber97] Peter Berstelsen. Semantics of Java Byte Code. . `http://http://www.dina.kvl.dk/~pmb/`, August 1997.

[Coh97] Richard M. Cohen. The defensive Java Virtual Machine specification. Technical report, Computational Logic Inc., 1997. Draft version.

[FM98] Stephen N. Freund and John C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Gol97] A. Goldberg. A Specification of Java Loading and Bytecode Verification. Technical report, Kestrel Institute, Palo Alto, CA, 1997.

[HBL98] Pieter Hartel, Michael Butler, and Moshe Levy. The Operational Semantics of a Java Secure Processor. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998.

[Isa] The Isabelle library. `http://www.in.tum.de/~isabelle/library/`.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[NOP] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. Project Bali. `http://www.in.tum.de/~isabelle/bali/`.

[ON98] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[Pus98] Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle. Technical Report TUM-I9816, Institut für Informatik, Technische Universität München, 1998. Available at `http://www.in.tum.de/~pusch/`.

[Qia97] Zhenyu Qian. A formal specification of Java Virtual Machine instructions. Technical report, 1997. Dept. of Comp. Sci., University of Bremen.

[Qia98] 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*, volume 1523 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, 1998. To appear.

[Typ] Types forum. `http://www.cs.indiana.edu/hyplan/pierce/types/`.


10.1007/b107031130007