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

Cornelia Pusch*

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

Abstract. We have formalized large parts of the Java Virtual Machine (JVM) with the theorem prover Isabelle/HOL. Based on the work by Qian[Qia98], we have then formalized a specification for a Java bytecode verifier and formally proved the soundness of the specification.

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 from Java.

The Java Virtual Machine Specification [LY96] describes a set of static and structural constraints that must hold for a program to assure safe execution, and requires that the JVM verifies itself that these constraints hold. There are several possible verification strategies, e.g.:

- Cohen [Coh97] has implemented in ACL2 a so called *defensive* JVM, where runtime checks are performed to guarantee a type-safe execution of the code.
- The Java Virtual Machine Specification [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 presented in this paper is the result of formalizing Qian's work in Isabelle/HOL, and it assumes some familiarity with both the JVM and 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.

* Research supported by DFG project *Bali*

There are several other approaches to formalize (parts of) of the JVM (see [Ber97], [BS98], [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 gives a minimal introduction to Isabelle. Section 3 describes our formalization of the JVM, in particular the representation of runtime data and the defintion 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 summarize our results and outline future work.

2 Isabelle

Isabelle 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. In the following we give an overview over the basic types and functions used in this paper. A detailed introduction to the Isabelle system can be found in [Pau94], and an online theory library is available at [Isa].

The type system is very similar to that of ML, with slight syntactic differences: function types are denoted by $\tau \Rightarrow \tau'$, 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 $\alpha \times \beta$.

Function application is preferably written in a curried style. Nevertheless, several definitions in our formalization are written in an uncurried style; this is due to restrictions of the TFL-package for well-founded recursive functions [Sli96,Sli97].

The basic types *bool*, *nat* and *int* are predefined, where the latter two are strictly distinguished. Isabelle/HOL also offers the polymorphic datatypes α *set* (with the usual set operators) and α *list*. The list constructors are [] (for the empty list) and $x\#xs$ (as 'cons' operator). We have $\mathsf{map}\ f\ xs$ to apply a function to all elements of a list, and $\mathsf{zip}\ xs\ ys$ takes two lists and returns a list of pairs. The functions $\mathsf{hd}\ xs$ and $\mathsf{tl}\ xs$ return the head resp. tail of a list. $\mathsf{length}\ xs$ computes the length of a list and $\mathsf{set}\ xs$ converts a list into a set.

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

$$\alpha\ option = \mathsf{None} \mid \mathsf{Some}\ \alpha$$

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

$$\alpha \leadsto \beta = (\alpha \Rightarrow \beta\ option)$$

'Partial function application' $f\ !!\ x$ (of *type* $[\alpha \leadsto \beta,\alpha] \Rightarrow \beta$) returns the result value if any, and an unknown value $\mathsf{arbitrary}$ else.

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

3 The Java Virtual Machine

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. We have also defined a predicate wf_classfiles that checks the well-formedness of classfiles. Due to lack of space, we cannot present the entire formalization that can be found at [NOP98] and in [Pus98]. However, we want to introduce the main ideas of our approach.

3.1 JVM Runtime Data

The JVM operates on two different types of values, primitive values and reference values. Among the primitive values, we consider only those 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:

$val = \mathsf{Intg}\ int \mid \mathsf{Addr}\ loc \mid \mathsf{Null}$

You may note that in contrast to our formalization, the Java Virtual Machine Specification [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 a (well-typed) JVM program. 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.

3.2 Operational Semantics of JVM Instructions

The Java Virtual Machine Specification [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.

Qian describes the operational semantics as a set of conditional state transition rules, where a state transition is only possible if the state conforms to certain restrictions. A statically well-typed program should then assure that a transition rule in question will be indeed applicable.

In our approach, we formalize the behaviour of a JVM instruction in terms of 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 in several groups of related instructions, describing each by a 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:

$$\begin{aligned}
\mathit{instr} \quad &= \mathsf{LAS}\ \mathit{load_and_store} \mid \mathsf{CO}\ \ \mathit{create_object} \mid \mathsf{MO}\ \ \mathit{manipulate_object}\\
&\mid \mathsf{MA}\ \ \mathit{manipulate_array} \mid \mathsf{CH}\ \ \mathit{check_object} \mid \mathsf{MI}\ \ \mathit{meth_inv}\\
&\mid \mathsf{MR}\ \ \mathit{meth_ret} \mid \mathsf{OS}\ \ \mathit{op_stack} \mid \mathsf{CB}\ \ \mathit{cond_branch}\\
&\mid \mathsf{UB}\ \ \mathit{uncond_branch}
\end{aligned}$$

Within the context of a JVM program, (i.e. set of classfiles), JVM execution consists in iterated transformation of the machine state according to the current instruction. The machine state is 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:

$$\begin{aligned}
&\mathit{frame} &&= \mathit{opstack} \times \mathit{locvars} \times \mathit{ident} \times \mathit{method_loc} \times \mathit{p_count}\\
&\mathit{jvm_state} &&= \mathit{xcpt\ option} \times \mathit{heap} \times \mathit{frame\ list}
\end{aligned}$$

If an exception has beed raised[1] or the frame stack is empty, execution terminates. 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 × jvm_state ⇒ 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)) =
  let cp = get_cpool (CFS !! cn)
  in
  Some (case (get_code CFS cn ml) ! pc of
   | MO ins ⇒   ... exec_mo ...
   ⋮
```

For example, the operational semantics of the Getfield instruction for object field access looks as follows:

```
exec_mo :: [manipulate_object,classfiles,cpool,heap,opstack,p_count]
              ⇒ (xcpt option × heap × opstack × p_count)
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)
```

[1] We do not yet treat exception handling.

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*. If it is 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 an entire JVM program 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 σ to a new state σ', where the pair (σ,σ') is in the reflexive transitive closure of successful execution steps:

$$\_ \vdash \_ \longrightarrow^* \_ \ :: [\mathit{classfiles},\mathit{jvm\_state},\mathit{jvm\_state}] \Rightarrow \mathit{bool}$$
$$CFS \vdash \sigma \longrightarrow^* \sigma' \stackrel{\text{def}}{=} (\sigma,\sigma') \in \{(s,t).\ \mathsf{exec}\ (CFS,s) = \mathsf{Some}\ t\}^*$$

4 A Specification for a Bytecode Verifier

A bytecode verifier statically derives the types of possible runtime data and checks that all instructions will get arguments of the right type. We give a specification for the bytecode verifier by formally defining the property of static well-typedness.

4.1 Static types

As Qian has pointed out in his work, the attempt to statically type-check JVM programs 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[2]. In this case, the types of the 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. In our formalization, static types are represented as values of datatype *tys*. According to Qian, we introduce several auxiliary definitions. Among the primitive types , we only consider type Integer. A reference type is either the type of the null reference (NT), or an interface of class name (IT *id* resp.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 set of reference types. (Due to restrictions to the construction of inductive

[2] Surprisingly, the typing rule for the similar working conditional expression of the Java source language reveales 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.

datatypes, we model reference type sets as lists.) The return type of methods is denoted by a value of type *tyOrVoid*, which is either a static type or Void. 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:

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

If two types are merged, the resulting supertype must cover both types. A type a covers a type a' ($a \sqsupseteq a'$), 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:

$$
\begin{array}{l}
\_ \sqsupseteq \_ :: [\mathit{any},\mathit{any}] \Rightarrow \mathit{bool} \\
\mathsf{Unusable} \qquad\qquad\quad \sqsupseteq a' \\
\mathsf{US}\ (\mathsf{PTS\ Integer}) \sqsupseteq \mathsf{US}\ (\mathsf{PTS\ Integer}) \\
\mathsf{US}\ (\mathsf{RTS}\ rs) \qquad\ \ \sqsupseteq \mathsf{US}\ (\mathsf{RTS}\ rs') = (\mathsf{set}\ rs' \subseteq \mathsf{set}\ rs)
\end{array}
$$

A *program 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 program state types:

$$
\mathit{state\_type} \qquad = \mathit{tys\ list} \times \mathit{any\ list}
$$

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

$$
\begin{array}{l}
\_ \sqsupseteq \_ :: [\mathit{state\_type},\mathit{state\_type}] \Rightarrow \mathit{bool} \\
(ST,LT) \sqsupseteq (ST',LT')' \stackrel{\mathsf{def}}{=} \mathsf{map\ US}\ ST \sqsupseteq \mathsf{map\ US}\ ST' \wedge LT \sqsupseteq LT'
\end{array}
$$

In Qian's work, a *program state type family* collects the type information for all instructions of a JVM program. We introduce two intermediate types, collecting the type information for a single method and for a set of methods defined in a single class. We have modeled the code area of a method as a list of instructions, therefore a method state is naturally a list of program state type information. A class type maps a method reference to a method type, and a program type maps a class name to a class type:

$$
\begin{array}{ll}
\mathit{method\_type} & = \mathit{state\_type\ list} \\
\mathit{class\_type} & = \mathit{method\_loc} \Rightarrow \mathit{method\_type} \\
\mathit{prog\_type} & = \mathit{ident} \Rightarrow \mathit{class\_type}
\end{array}
$$

4.2 Statical Well-typedness

Qian describes a type inference system by giving typing rules for each instruction. A program is well-typed, if there exists a program state type family that *satisfies* the typing rules for each instruction in the program.

In our work, we have chosen a more abstract definition of well-typedness by defining directly a predicate that checks, whether a program point is well-typed. Additionally, the well-typedness predicate checks several well-formedness constraints, e.g. an index to local variables must not be greater than the number of local variables.

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 holds:

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

The above 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 contain values according to the type of the current class *cn* and the parameter descriptor[3] *pd* of the current method:

$$
\begin{array}{l}
\mathsf{wt\_start} :: [\mathit{classfiles,ident,param\_desc,method\_type}] \Rightarrow \mathit{bool} \\
\mathsf{wt\_start}\ \mathit{CFS}\ \mathit{cn}\ \mathit{pd}\ \Phi \stackrel{\mathsf{def}}{=} \\
\quad \Phi\ !\ 0 \sqsupseteq ([],(\mathsf{RTS}\ [\mathsf{CT}\ \mathit{cn}])\#(\mathsf{map}\ (\mathsf{US}\ o\ (\mathsf{fd2tys}\ \mathit{CFS}))\ \mathit{pd}))
\end{array}
$$

The code array of a method must not be empty, i.e. its length must be greater than zero. A method is then 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:

[3] In JVM classfiles, type information is represented by so called *descriptors*. The function fd2tys converts a field descriptor into a static type.

$\mathsf{wt\_method} :: [\mathit{classfiles},\mathit{ident},\mathit{param\_desc},\mathit{return\_desc},\mathit{instr\ list},\mathit{method\_type}] \Rightarrow \mathit{bool}$

$\mathsf{wt\_method}\ \mathit{CFS}\ \mathit{cn}\ \mathit{pd}\ \mathit{rd}\ \mathit{ins}\ \Delta \stackrel{\mathsf{def}}{=}$
$\quad \mathsf{let}\ \mathit{cp} = \mathsf{get\_cpool}\ (\mathit{CFS}\ !!\ \mathit{cn});$
$\quad\quad \mathit{max}_{pc} = \mathsf{length}\ \mathit{ins}$
$\quad \mathsf{in}$
$\quad 0 < \mathit{max}_{pc} \wedge \mathsf{wt\_start}\ \mathit{CFS}\ \mathit{cn}\ \mathit{pd}\ \Delta\ \wedge$
$\quad \forall\ \mathit{pc}.\ \mathit{pc}{<}\mathit{max}_{pc} \longrightarrow \mathsf{wt\_instr}\ (\mathit{ins}\ !\ \mathit{pc})\ \mathit{CFS}\ \mathit{rd}\ \mathit{cp}\ \Delta\ \mathit{max}_{pc}\ \mathit{pc}$

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:

$\mathsf{wt\_class} :: [\mathit{classfiles},\mathit{ident},\mathit{class\_type}] \Rightarrow \mathit{bool}$

$\mathsf{wt\_class}\ \mathit{CFS}\ \mathit{cn}\ \Gamma \stackrel{\mathsf{def}}{=}$
$\quad \forall \mathit{ml}\ \mathit{rd}\ \mathit{ins}.\ \mathsf{get\_methods}\ (\mathit{CFS}\ !!\ \mathit{cn})\ \mathit{ml} = \mathsf{Some}\ (\mathit{rd},\mathit{ins})$
$\quad\quad\quad \longrightarrow \mathsf{wt\_method}\ \mathit{CFS}\ \mathit{cn}\ (\mathit{snd}\ \mathit{ml})\ \mathit{rd}\ \mathit{ins}\ (\Gamma\ \mathit{ml})$

A JVM program is well-typed with respect to a program type Φ, 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 a Bytecode Verifier

A bytecode verifier (or more abstract: a type system) statically determines the types of all runtime data. The soundness of a static type system is often (also in [Qia98]) described as follows: if a program is statically well-typed, then in every execution step the runtime values will be correct with respect to their static types. In our opinion, this sounds more like a soundness property of the operational semantics with respect to the static type system. We consider it more adequate to talk about the correctness of a static type with respect to a runtime value[4]. Therefore, we call a bytecode verifier sound, if the statically predicted type gives a correct approximation of a runtime value produced during execution.

In this section, we will show that any bytecode verifier satisfying our specification 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 §3.1), whereas Qian has to go through the program and assign type tag to each value

[4] In fact, this does not alter our definition, since a relation *type*$\sim$*value* can be read as 'type is correct with respect to value' or 'value is correct with respect to type'. We just prefer the first interpretation.

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]:

$\mathsf{approx\_val} :: [\mathit{classfiles,heap,val,any}] \Rightarrow \mathit{bool}$
$\mathsf{approx\_val}\ \mathit{CFS}\ \mathit{hp}\ x \qquad \mathsf{Unusable}$
$\mathsf{approx\_val}\ \mathit{CFS}\ \mathit{hp}\ (\mathsf{Intg}\ i)\ (\mathsf{US}\ (\mathsf{PTS\ Integer}))$
$\mathsf{approx\_val}\ \mathit{CFS}\ \mathit{hp}\ (\mathsf{Addr}\ a)\ (\mathsf{US}\ (\mathsf{RTS}\ rs))\ =$
$\quad \exists\ obj\ rt.\ hp\ a = \mathsf{Some}\ obj \land$
$\qquad \mathsf{fd2tys}\ \mathit{CFS}\ (\mathsf{get\_obj\_type}\ obj) = \mathsf{RTS}\ [rt] \land$
$\qquad \mathsf{wideRefsConvertible}\ (\mathit{CFS},\ rt,\ rs)$
$\mathsf{approx\_val}\ \mathit{CFS}\ \mathit{hp}\ \mathsf{Null}\ (\mathsf{US}\ (\mathsf{RTS}\ rs))$

In case of a runtime object reference Addr a, the static type approximates the runtime value by restricting it to point to objects whose type is convertible to the static type.

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

$\mathsf{approx\_loc} :: [\mathit{classfiles,heap,locvars,any\ list}] \Rightarrow \mathit{bool}$
$\mathsf{approx\_loc}\ \mathit{CFS}\ \mathit{hp}\ \mathit{loc}\ \mathit{LT} \stackrel{\mathsf{def}}{=}$
$\quad \mathsf{length}\ \mathit{loc} = \mathsf{length}\ \mathit{LT} \land \forall(\mathit{val,any}){\in}\mathsf{set}\ (\mathsf{zip}\ \mathit{loc}\ \mathit{LT}).\ \mathsf{approx\_val}\ \mathit{CFS}\ \mathit{hp}\ \mathit{val}\ \mathit{any}$

$\mathsf{approx\_stk} :: [\mathit{classfiles,heap,opstack,tys\ list}] \Rightarrow \mathit{bool}$
$\mathsf{approx\_stk}\ \mathit{CFS}\ \mathit{hp}\ \mathit{stk}\ \mathit{ST} \stackrel{\mathsf{def}}{=}$
$\quad \mathsf{length}\ \mathit{stk} = \mathsf{length}\ \mathit{ST} \land$
$\quad \forall(\mathit{val,tys}){\in}\mathsf{set}\ (\mathsf{zip}\ \mathit{stk}\ \mathit{ST}).\ \mathsf{approx\_val}\ \mathit{CFS}\ \mathit{hp}\ \mathit{val}\ (\mathsf{US}\ \mathit{tys})$

5.2 Soundness Proof

Qian states a soundness theorem saying that for a statically well-typed program, 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 reveals 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:

$\mathsf{correct\_obj} :: [\mathit{classfiles,heap,obj}] \Rightarrow \mathit{bool}$
$\mathsf{correct\_obj}\ \mathit{CFS}\ \mathit{hp}\ (\mathsf{Obj}\ \mathit{cn}\ \mathit{od}) =$
$\quad \mathsf{is\_class}\ \mathit{CFS}\ \mathit{cn} \land$
$\quad \forall \mathit{fl}\ \mathit{fd}.\ (\mathsf{get\_all\_fields}\ (\mathit{CFS,cn}))\ \mathit{fl} = \mathsf{Some}\ \mathit{fd}$
$\qquad \longrightarrow \exists \mathit{val}.\ \mathit{od}\ \mathit{fl} = \mathsf{Some}\ \mathit{val} \land \mathsf{approx\_val}\ \mathit{CFS}\ \mathit{hp}\ \mathit{val}\ (\mathsf{fd2tys}\ \mathit{CFS}\ \mathit{fd})$

$\mathsf{correct\_obj}\ \mathit{CFS}\ \mathit{hp}\ (\mathsf{Arr}\ \mathit{fd}\ \mathit{ad}\ \Rightarrow) =$
$\quad \forall \mathit{val} \in \mathsf{set}\ \mathit{ad}.\ \mathsf{approx\_val}\ \mathit{CFS}\ \mathit{hp}\ \mathit{val}\ (\mathsf{fd2tys}\ \mathit{CFS}\ \mathit{fd})$

$\mathsf{correct\_heap} :: [\mathit{classfiles},\mathit{heap}] \Rightarrow \mathit{bool}$
$\mathsf{correct\_heap}\ \mathit{CFS}\ \mathit{hp} \stackrel{\mathsf{def}}{=} \forall a\ \mathit{obj}.\ \mathit{hp}\ a = \mathsf{Some}\ \mathit{obj} \longrightarrow \mathsf{correct\_obj}\ \mathit{CFS}\ \mathit{hp}\ \mathit{obj}$

The predicate correct_frame checks whether the operand stack entries *stk* and local variables *loc* have been correctly approximated by the state type $(\mathit{ST},\mathit{LT})$. 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:

$\mathsf{correct\_frame} :: [\mathit{classfiles},\mathit{heap},\mathit{state\_type},\mathit{frame}] \Rightarrow \mathit{bool}$
$\mathsf{correct\_frame}\ \mathit{CFS}\ \mathit{hp}\ (\mathit{ST},\mathit{LT})\ (\mathit{stk},\mathit{loc},\mathit{cn},\mathit{ml},\mathit{pc}) \stackrel{\mathsf{def}}{=}$
$\quad \mathsf{approx\_stk}\ \mathit{CFS}\ \mathit{hp}\ \mathit{stk}\ \mathit{ST} \wedge \mathsf{approx\_loc}\ \mathit{CFS}\ \mathit{hp}\ \mathit{loc}\ \mathit{LT} \wedge \mathsf{is\_class}\ \mathit{CFS}\ \mathit{cn} \wedge$
$\quad \exists \mathit{rd}\ \mathit{ins}.\ \mathsf{get\_methods}\ (\mathit{CFS}\ !!\ \mathit{cn})\ \mathit{ml} = \mathsf{Some}\ (\mathit{rd},\mathit{ins}) \wedge \mathit{pc} < \mathsf{length}\ \mathit{ins}$

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:

$\mathsf{correct\_frames} :: [\mathit{classfiles},\mathit{heap},\mathit{prog\_type},\mathit{return\_desc},\mathit{method\_loc},\mathit{frame\ list}] \Rightarrow \mathit{bool}$
$\mathsf{correct\_frames}\ \mathit{CFS}\ \mathit{hp}\ \Phi\ rd_{n+1}\ ml_{n+1}\ []\ = (rd_{n+1}{=}\mathsf{V})$
$\mathsf{correct\_frames}\ \mathit{CFS}\ \mathit{hp}\ \Phi\ rd_{n+1}\ ml_{n+1}\ (f_n\#\mathit{frs}) =$
$\quad \mathsf{let}\ (\mathit{stk},\mathit{loc},\mathit{cn},\mathit{ml},\mathit{pc}) = f_n;$
$\qquad (\mathit{rd},\mathit{ins}) \quad = \mathsf{get\_methods}\ (\mathit{CFS}\ !!\ \mathit{cn})\ !!\ \mathit{ml};$
$\qquad \mathit{cp} \qquad\quad = \mathsf{get\_cpool}\ (\mathit{CFS}\ !!\ \mathit{cn});$
$\qquad (\mathit{ST},\mathit{LT}) = (\Phi\ \mathit{cn}\ \mathit{ml})\ !\ \mathit{pc}$
$\quad \mathsf{in}$
$\quad \exists \mathit{mi}\ c\ k\ l.\ \mathit{pc} = k{+}1 \wedge \mathit{ins}\ !\ k = \mathsf{MI}\ \mathit{mi} \wedge \mathsf{extract\_meth}\ \mathit{cp}\ \mathit{mi} = (c,ml_{n+1},rd_{n+1},l) \wedge$
$\quad \mathsf{correct\_frame}\ \mathit{CFS}\ \mathit{hp}\ (\mathsf{pop\_rd}\ \mathit{CFS}\ rd_{n+1}\ \mathit{ST},\ \mathit{LT})\ f_n \wedge$
$\quad \mathsf{correct\_frames}\ \mathit{CFS}\ \mathit{hp}\ \Phi\ \mathit{rd}\ \mathit{ml}\ \mathit{frs}$

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}:

$\mathsf{correct\_state} :: [\mathit{classfiles},\mathit{prog\_type},\mathit{jvm\_state}] \Rightarrow \mathit{bool}$
$\mathsf{correct\_state}\ \mathit{CFS}\ \Phi\ (\mathsf{Some}\ x,\mathit{hp},\mathit{frs})$
$\mathsf{correct\_state}\ \mathit{CFS}\ \Phi\ (\mathsf{None},\mathit{hp},[])$
$\mathsf{correct\_state}\ \mathit{CFS}\ \Phi\ (\mathsf{None},\mathit{hp},f_{n+1}\#\mathit{frs}) =$
$\quad \mathsf{let}\ (\mathit{stk},\mathit{loc},\mathit{cn},ml_{n+1},\mathit{pc}) = f_{n+1};$
$\qquad (rd_{n+1},\mathit{ins}) = \mathsf{get\_methods}\ (\mathit{CFS}\ !!\ \mathit{cn})\ !!\ ml_{n+1}$
$\quad \mathsf{in}$
$\quad \mathsf{correct\_heap}\ \mathit{CFS}\ \mathit{hp} \wedge$
$\quad \mathsf{correct\_frame}\ \mathit{CFS}\ \mathit{hp}\ ((\Phi\ \mathit{cn}\ \mathit{ml})\ !\ \mathit{pc})\ f_{n+1} \wedge$
$\quad \mathsf{correct\_frames}\ \mathit{CFS}\ \mathit{hp}\ \Phi\ rd_{n+1}\ ml_{n+1}\ \mathit{frs}$

Now we can prove the following main soundness theorem:

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

It says that for a set of well-formed classfiles *CFS* that are statically well-typed with program type Φ, program execution starting from a correct state σ leads to a correct state σ'.

Due to our definition of the operational semantics, the execution of a program is guaranteed by definition. Thus, we also can easily prove the following alternative theorem:

$$\begin{array}{l}\textsf{wf\_classfiles}\ \mathit{CFS} \wedge \textsf{wt\_classfiles}\ \mathit{CFS}\ \Phi\ \wedge \\ \textsf{correct\_state}\ \mathit{CFS}\ \Phi\ (\mathit{xp},\mathit{hp},\mathit{frs}) \wedge \mathit{xp}{=}\textsf{None} \wedge \mathit{frs}{\neq}[] \\ \Longrightarrow \exists\sigma'.\ \textsf{exec}\ (\mathit{CFS},(\mathit{xp},\mathit{hp},\mathit{frs})) = \textsf{Some}\ \sigma' \wedge \textsf{correct\_state}\ \mathit{CFS}\ \Phi\ \sigma'\end{array}$$

It points out that for every non-final correct state there exists an execution step to another correct state.

The proof for the main theorem has been carried out by induction over the transitive closure of the execution function. Then the preservation of the correctness property for a single execution step had to be shown by case distinction over the instructions.

6 Results and Further Work

The Isabelle/HOL formalization of the JVM classfile structure and the operational semantics comprises about 1000 lines, the specification of the bytecode verifier took another 500 lines. lines. The proof skripts 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 resp. method for some static type must assure that an appropriate field or method can be found at runtime.

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.

References

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

[BS98] Egon Börger and Wolfram Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. 1998. To appear in FMCS98.

[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. Available on the World Wide Web at URL `http://www.in.tum.de/~isabelle/library/`.

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

[NOP98] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. The Bali project. 1998. `http://www.in.tum.de/~isabelle/bali/`.

[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 Universitsät München, 1998.

[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.

[Sli96] Konrad Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 381–397. Springer-Verlag, 1996.

[Sli97] Konrad Slind. Derivation and use of induction schemes in higher-order logic. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 275–290. Springer-Verlag, 1997.

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