Verified Bytecode Model Checkers

David Basin, Stefan Friedrich, Marek Gawkowski
{basin, friedric, gawkowsk}@informatik.uni-freiburg.de

Albert-Ludwigs-Universität Freiburg, Germany

Abstract. We have used Isabelle/HOL to formalize and prove correct an approach to bytecode verification based on model checking that we have developed for the Java Virtual Machine. Our work builds on, and extends, the formalization of the Java Virtual Machine and data flow analysis framework of Pusch and Nipkow. By building on their framework, we can reuse their results that relate the run-time behavior of programs with the existence of well-typings for the programs. Our primary extensions are to handle polyvariant data flow analysis and its realization as temporal logic model checking. Aside from establishing the correctness of our model-checking approach, our work contributes to understanding the interrelationships between classical data flow analysis and program analysis based on model checking.

1 Introduction

The security of Java and the Java Virtual Machine (JVM), and in particular of its bytecode verifier, has been the topic of considerable study. Research originally focused on abstract models of the JVM [4, 19] and more recently on machine checked proofs of the correctness of bytecode verifier models [2, 12, 13, 15] and type inference algorithms based on data flow analysis [11]. There are two research directions that are the starting point for our work. First, the work of Pusch [15] and Nipkow [11] on models and proofs in Isabelle/HOL for verifying bytecode verifiers based on data flow algorithms. Second, the work of Posegga and Vogt [14], later extended in [1], on reducing bytecode verification to model checking. These two directions are related: both are based on abstract interpretation and solving fixpoint equations. However, whereas data flow analysis computes a type for a method and checks that the method's instructions are correctly applied to data of that type, the model-checking approach is more declarative; here one formalizes the instruction applicability conditions as formulae in a temporal logic (e.g. LTL) and uses a model checker to verify that an abstraction of the method (corresponding to the abstract interpreter of the data flow approach) satisfies these applicability conditions.

In this paper we explore the interrelationships between these two research directions. We present the first machine-checked proof of the correctness of the model-checking approach to bytecode verification, and in doing so we build upon the Isabelle/HOL formalizations of the JVM [15] and the abstract verification framework that Nipkow developed for verifying data flow algorithms in [11]. This framework formalizes the notion of a well-typing for bytecode programs and proves that a bytecode verifier is correct (i.e., accepts only programs free of runtime type errors) when it accepts only programs possessing a well-typing. We will show that every bytecode program whose abstraction globally satisfies the instruction applicability conditions (which can be established by model checking) in fact possesses such a well-typing, i.e. our goal is to validate the model-checking approach by proving a theorem of the form

$$\begin{aligned} &(\mathit{abstraction}(\mathit{Method}) \models_{LTL} \Box \mathit{app\_conditions}(\mathit{Method})) \\ \Longrightarrow\ &(\exists\, \phi.\ \mathit{welltyping}(\phi, \mathit{Method}))\,. \end{aligned}$$

We achieve this by modifying and extending the framework of Pusch and Nipkow to support polyvariant data flow analysis and model checking.

Our development can be subdivided into six areas. Based on (1) *preliminary definitions* and (2) *semilattice-theoretic foundations* we define (3) the *JVM model*, which includes the JVM type system, the JVM abstract semantics, and the definition of the JVM state type. In (4), the *model-checking framework*, we define Kripke structures and traces, which we later use to formalize model checking. Afterwards, we define the translation of bytecode programs into (5) *finite transition systems*. We use the abstract semantics of JVM programs to define the transition relations and the JVM type system to build LTL formulae for model checking. Finally, we state and prove in (6) our *main theorem*.

Overall, most of the theories we use are adopted from the work of Nipkow [11], and hence our work can, to a large part, be seen as an instance of Nipkow's abstract correctness framework. The table below provides an overview of how our formalization differs from Nipkow's.

Theories	Status
JBasis (1), Type (1), Decl (1), TypeRel (1), State (1), WellType (1), Conform (1), Value(1), Semilat (2), Err (2), Opt (2), Product(2), JType (3), BVSpec (3)	unchanged
Listn (2)	the stack model is changed
JVMInstructions (1), JVMExecInstr (1), JVMExec (1) JVMType (3), Step (3),	modified due to the changes in the stack model
Semilat2 (2), Kripke (4), LTL (4), ModelChecker (5), JVM_MC (6)	new

Our original motivation for undertaking this development was a pragmatic one: we had developed a model-checking based tool [1] for verifying bytecode and we wanted to establish the correctness of the approach taken. We see our contributions, however, as more general. Our development provides insight into the relationship between monovariant and polyvariant data flow analysis and model checking, in particular, what differences are required in their formalization. Monovariant analysis associates one program state type, which contains information about the stack type and register type, to each control point (as in Sun's verifier or more generally Kildall algorithm, which was analyzed by Nipkow), or one such type per subroutine to each control point. In contrast, polyvariant analysis allows multiple program state types per control point, depending on the number of control-flow paths that lead to this control point [9]. In the formalization of Pusch and Nipkow [15, 11], monovariant data flow analysis is used, which is adequate since they do not consider subroutines and interfaces. In our approach, we use model checking, which performs a polyvariant data flow analysis. The result is not only that we can base our bytecode verification tool on standard model checkers, but also that our bytecode verifier accepts more (type-correct) programs than bytecode verifiers performing monovariant data flow analysis. For instance, our tool can successfully type check programs where, for a given program point and for two different execution paths, the operand stack has two different sizes or unused stack elements have different (and incompatible) types.

Despite switching to a polyvariant, model-checking approach, we were able to reuse a surprising amount of Nipkow's formal development, as the above table indicates. Our main change was to allow stack elements of incompatible types, which we achieved by generalizing the notion of the JVM state type. In doing so, we enlarged the set of programs that fulfill the well-typing definition for bytecode programs as formalized by Pusch [15]. An additional change was required from the model-checking side: to allow the computation of the supremum of two stacks at each program point, we had to specialize our notion of type correct program by imposing additional constraints concerning the stack size on programs. Overall, the changes we made appear generally useful. Polyvariant data flow analysis constitutes a generalization of monovariant data flow analysis and it should be possible (although we have not formally checked this) to formalize the monovariant data flow analysis using our JVM model with an accordingly modified notion of well-typing.

Finally, note that we formalize model checking not algorithmically, as done in [16], but declaratively. Consequently our formalization of model checking is independent of the implemented model-checking algorithm. In our work with Posegga [1], for instance, we implemented the backends both for symbolic and for explicit state model checkers. As our correctness theorem here is a statement about the correctness of the model-checking approach to bytecode verification it is valid for both these backends. Hence the plural in our title “verified bytecode model checkers.”

2 Background

We present background concepts necessary for our formalization. The sections 2.1, 2.2, and 2.4–2.6 describe (unmodified) parts of Nipkow’s formalization and are summarized here for the sake of completeness.

2.1 Basic types

We employ basic types and definitions of Isabelle/HOL. Types include *bool*, *nat*, *int*, and the polymorphic types α *set* and α *list*. We employ a number of standard functions on (cons) lists including a conversion function set from lists to sets, infix operators # and @ to build and concatenate lists, and a function size to denote the length of a list. $xs!i$ denotes the i-th element of a list xs and $xs[i := x]$ overwrites i-th element of xs with x. Finally, we use Isabelle/HOL records to build tuples and functional images over sets: $(\!| \mathsf{a} :: \alpha,\ \mathsf{b} :: \beta |\!)$ denotes the record type containing the components a and b of types α and β respectively and $\mathsf{f}\,`\,A$ denotes the image of the function f over the set A.

2.2 Partial orders and semilattices

A partial order is a binary predicate of type $\alpha\ ord = \alpha \to \alpha \to bool$. We write $x \leq_r y$ for $r\ x\ y$ and $x <_r y$ for $x \leq_r y \wedge x \neq y$. $r :: \alpha\ ord$ is a **partial order** iff r is reflexive, antisymmetric und transitive. We formalize this using the predicate $\mathsf{order} :: \alpha\ ord \to bool$. $\top :: \alpha$ is the **top** element with respect to the partial order r iff the predicate $\mathsf{top} :: \alpha\ ord \to \alpha \to bool$, defined by $\mathsf{top}\ r\ T \;\equiv\; \forall x. x \leq_r T$ holds. Given the types $\alpha\ binop \;=\; \alpha \to \alpha \to \alpha$ and $\alpha\ sl \;=\; \alpha\ set \times \alpha\ ord \times \alpha\ binop$ and the supremum notation $x +_f y = f\ x\ y$, we say that $(A,\ r,\ f) :: \alpha\ sl$ is a (**supremum**) **semilattice** iff the predicate $\mathsf{semilat} :: \alpha\ sl \to bool$ holds

$$\begin{aligned}\mathsf{semilat}(A,\ r,\ f) \equiv\ & \mathsf{order}\ r\ \wedge\ \mathsf{closed}\ A\ f\ \wedge \\ & (\forall x\, y \in A.\ x \leq_r x +_f y)\ \wedge\ (\forall x\, y \in A.\ y \leq_r x +_f y)\ \wedge \\ & (\forall x\, y\, z \in A.\ x \leq_r z\ \wedge\ y \leq_r z \longrightarrow x +_f y \leq_r z),\end{aligned}$$

where $\mathsf{closed}\ A\ f \;\equiv\; \forall x\, y \in A.\ x +_f y \in A$.

2.3 Least upper bounds of sets

The above definitions are for reasoning about the supremum of binary operators $f :: \alpha\ binop$. We define a new theory, Semilat2 to reason about the supremum of sets.

To build the suprema over sets, we define the function $\mathsf{lift\_sup} :: \alpha\, sl \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$, written $\mathsf{lift\_sup}\,(A,\, r,\, f)\, T\, x\, y \;=\; x \uplus_{(A,\, r,\, f),T}\; y$, that lifts the binary operator $f :: \alpha\ binop$ over the type α:

$$\begin{array}{l} x \uplus_{(A,\, r,\, f),\, T}\; y \equiv \texttt{if}\ (\mathsf{semilat}\,(A,\, r,\, f)\ \wedge\ \mathsf{top}\, r\, T)\ \texttt{then} \\ \qquad\qquad\qquad \texttt{if}\ (x \in A\ \wedge y \in A)\texttt{then}\ (x\ +_f\ y)\ \texttt{else}\ T \\ \qquad\quad \texttt{else}\ \mathsf{arbitrary} \end{array}$$

To support reasoning about the least upper bounds of sets, we introduce the **bottom** element $B :: \alpha$, which is defined by $\mathsf{bottom}\, r\, B \;\equiv\; \forall\, x.\, B \leq_r x$. We use the Isabelle/HOL function $\mathsf{fold} :: (\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta\ set \rightarrow \alpha$ to build the least upper bounds of sets and in the following we write $\bigsqcup_{(A,\, r,\, f),\, T,\, B}\ A'$ for $\mathsf{fold}\,(\lambda\, x\, y.\ x\ \uplus_{(A,\, r,\, f),\, T}\ \ y)\, B\, A'$. We have shown that $\bigsqcup_{(A,\, r,\, f),\, T,\, B}\ A'$ is a least upper bound over A'.

Lemma 1. *Given* $\mathsf{semilat}\,(A,\, r,\, f)$, $\mathsf{finite}\, A$, $T \in A$, $\mathsf{top}\, r\, T$, $B \in A$, $\mathsf{bottom}\, r\, B$ *and given an arbitrary function* g, *where* $g\, B \;=\; B$, $\forall\, x \in A.\, g\, x \in A$, *and* $\forall\, x\, y \in A.\, g\,(x +_f\, y)\ \leq_r (g\, x\ +_f\ g\, y)$, *then*

$$(g\,`\,A'' \subseteq A') \longrightarrow g\,(\textstyle\bigsqcup_{(A,\, r,\, f),\, T,\, B}\ A'')\ \leq_r\ (\textstyle\bigsqcup_{(A,\, r,\, f),\, T,\, B}\ A')$$

2.4 The error type and *err*-semilattices

The theory Err defines an error element to model the situation where the supremum of two elements does not exist. We introduce both a datatype and a corresponding construction on sets:

$$\textbf{datatype}\ \alpha\ err \equiv \mathsf{Err} \mid \mathsf{OK}\, \alpha \qquad\qquad \mathsf{err}\, A \equiv \{\mathsf{Err}\} \cup \{\mathsf{OK}\, a \mid a \in A\}$$

Orderings r on α can be lifted to $\alpha\ err$ by making Err the top element:

$$\mathsf{le}\ r\ \ (\mathsf{OK}\, x)\ \ (\mathsf{OK}\, y) = x\ \ \leq_r\ \ y$$

We now employ the following lifting function

$$\begin{array}{l} \mathsf{lift2}\ ::\ (\alpha \rightarrow \beta \rightarrow \gamma\ err) \rightarrow \alpha\ err \rightarrow \beta\ err \rightarrow \gamma\ err \\ \mathsf{lift2}\ f\ (\mathsf{OK}\, x)\ (\mathsf{OK}\, y) = f\ x\ y \\ \mathsf{lift2}\ f\ \_\qquad\quad \_\qquad\quad = \mathsf{Err} \end{array}$$

to define a new notion of an *err*-semilattice, which is a variation of a semilattice with a top element. It suffices to say how the ordering and the supremum are defined over non-top elements and hence we represent a semilattice with top element Err as a triple of type *esl*: $\alpha\ esl \;=\; \alpha\ set\ \times\ \alpha\ ord\ \times\ \alpha\ ebinop$, where $\alpha\ ebinop \;=\; \alpha\ \rightarrow\ \alpha\ \rightarrow\ \alpha\ err$. We introduce also conversion functions between the types *sl* and *esl*:

$$\begin{array}{ll} \mathsf{esl}\ ::\ \alpha\, sl\ \rightarrow\ \alpha\ esl & \mathsf{sl}\ ::\ \alpha\ esl\ \rightarrow\ \alpha\ err\ sl \\ \mathsf{esl}\,(A,\, r,\, f)\ =\ (A,\, r,\, \lambda\, x\, y.\, \mathsf{OK}\,(f\ x\ y)) & \mathsf{sl}\,(A,\, r,\, f)\ =\ (\mathsf{err}\, A,\, \mathsf{le}\, r,\, \mathsf{lift2}\, f) \end{array}$$

Finally we define $L :: \alpha\, esl$ to be an *err*-**semilattice** iff $\mathsf{sl}\, L$ is a semilattice. It follows that $\mathsf{esl}\ L$ is an *err*-semilattice if L is a semilattice.

2.5 The option type

Theory Opt introduces the type *option* and the set opt as duals to the type *err* and the set err:

$$\textbf{datatype}\ \alpha\ option \equiv \mathsf{None} \mid \mathsf{Some}\, \alpha \qquad \mathsf{opt}\, A \equiv \{\mathsf{None}\} \cup \{\mathsf{Some}\, a \mid a \in A\}$$

The theory also defines an ordering where None is the bottom element and a supremum operation:

$$\mathsf{le}\ r\ (\mathsf{Some}\ x)\ (\mathsf{Some}\ y) = x \le_r y \qquad \mathsf{sup}\ f\ (\mathsf{Some}\ x)\ (\mathsf{Some}\ y) = \mathsf{Some}(f\ x\ y)$$

Note that the function $\mathsf{esl}\,(A,\ r,\ f)\ =\ (\mathsf{opt}\ A,\ \mathsf{le}\ r,\ \mathsf{sup}\ f)$ maps *err*-semilattices to *err*-semilattices.

2.6 Products

Theory $\mathsf{Product}$ provides what is known as the *coalesced* product, where the top elements of both components are identified.

$$\begin{array}{l}
\mathsf{esl} :: \alpha\ esl \to \beta\ esl \to (\alpha \times \beta)\ esl \\
\mathsf{esl}\,(A,\ r_A,\ f_A)\,(B,\ r_B,\ f_B) = (A \times B,\ \mathsf{le}\,r_A\,r_B,\ \mathsf{sup}\ f_A\,f_B) \\
\mathsf{sup} :: \alpha\ ebinop \to \beta\ ebinop \to (\alpha \times \beta)\ ebinop \\
\mathsf{sup}\,f\,g = \lambda\,(a_1,\ b_1)\,(a_2,\ b_2).\ \mathsf{Err.sup}\ (\lambda\,x\,y.\,(x,\ y))\ (a_1 +_f a_2)\ (b_1 +_g b_2)
\end{array}$$

The ordering function $\mathsf{le} :: \alpha\ ord \to \beta\ ord \to (\alpha \times \beta)\ ord$ is defined as expected.

Note that $\times$ is used both on the type and set level. Nipkow has shown that if both L_1 and L_2 are *err*-semilattices, so is $\mathsf{esl}\,L_1\,L_2$.

2.7 Lists of fixed length

For our application we must model the JVM stack differently from Nipkow. Our formalization is modified to allow the polyvariant analysis and thus requires associating multiple stack-register types to each control point within the program.

To facilitate this modification, we define the theory Listn, of fixed length lists over a given set. In HOL, this is formalized as a set rather than a type:

$$\mathsf{list}\,n\ A\ \equiv\ \{xs \mid \mathsf{size}\ xs = n\ \wedge\ \mathsf{set}\ xs \subseteq A\}$$

This set can be turned into a semilattice in a componentwise manner, essentially viewing it as an n-fold Cartesian product:

$$\begin{array}{ll}
\mathsf{sl} :: nat \to \alpha\ sl \to \alpha\ list\ sl & \mathsf{le} :: \alpha\ ord \to \alpha\ list\ ord \\
\mathsf{sl}\,n\ (A,\ r,\ f) = (\mathsf{list}\,n\ A,\ \mathsf{le}\,r,\ \mathsf{map2}\ f) & \mathsf{le}\,r\ =\ \mathsf{list_all2}\ (\lambda\,x\,y.\,x \le_r y)
\end{array}$$

Here the auxiliary functions $\mathsf{map2} :: (\alpha \to \beta \to \gamma) \to \alpha\ list \to \beta\ list \to \gamma\ list$ and $\mathsf{list\_all2} :: (\alpha \to \beta \to bool) \to \alpha\ list \to \beta\ list \to bool$ are defined as expected. We write $xs \le_{[r]} ys$ for $xs \le_{(\mathsf{le}\ r)} ys$ and $xs +_{[f]} ys$ for $xs +_{(\mathsf{map2}\ f)} ys$. Observe that if L is a semilattice then so is $\mathsf{sl}\,n\ L$.

To combine lists of different lengths, we use the following function:

$$\begin{array}{l}
\mathsf{sup} :: (\alpha \to \beta \to \gamma\ err) \to \alpha\ list \to \beta\ list \to \gamma\ list\ err \\
\mathsf{sup}\,f\ xs\ ys\ =\ \texttt{if}\ (\mathsf{size}\ xs\ =\ \mathsf{size}\ ys)\ \texttt{then}\ \mathsf{OK}\ (\mathsf{map2}\ f\ xs\ ys)\ \texttt{else}\ \mathsf{Err}
\end{array}$$

Note that in our JVM formalization, the supremum of two lists xs and ys of equal length returns a result of the form $\mathsf{OK}\ zs$ with $zs!i\ =\ \mathsf{Err}$ in the case that the supremum of two corresponding elements $xs!i$ and $ys!i$ equals Err. This differs from sup in Nipkow's formalization, which returns Err in this case. Below we present the *err*-semilattice $\mathsf{upto\_esl}$ of all lists up to a specified length:

$$\begin{array}{l}
\mathsf{upto_esl} :: nat \to \alpha\ sl \to \alpha\ list\ esl \\
\mathsf{upto_esl}\,n\ =\ \lambda\,(A,\ r,\ f).\ (\cup_{i \le n}\ \mathsf{list}\ i\ A,\ \mathsf{le}\,r,\ \mathsf{sup}\ f)
\end{array}$$

We have shown that if L is a semilattice then $\mathsf{upto\_esl}\,n\ L$ is an *err*-semilattice.

3 The JVM model

In this section we show how the JVM can be formalized for the purpose of polyvariant data flow analysis. In Section 3.1, our formalization adopts, unchanged, Nipkow's formalization of the JVM type system. Using this formalization, we define our modified program state type and construct a semilattice whose carrier set consists of elements of this type. Based on the modified notion of the program state type, we redefine, in Section 3.2, the syntax and abstract semantics of JVM programs and consequently also redefine the definitions of a JVM abstract execution function and of well-typed methods in Section 3.3.

3.1 Type system and well-formedness

The theory Types defines the types of our JVM. Our machine supports the void type, integers, null references, and class types (based on the type *cname* of class names):

$$\mathbf{datatype}\ ty\ \equiv\ \mathsf{Void} \mid \mathsf{Integer} \mid \mathsf{Boolean} \mid \mathsf{NullT} \mid \mathsf{Class}\ cname$$

The theory Decl defines class declarations and programs. Based on the type *mname* of method names and *vname* of variable names, we model a JVM program $\mathsf{P} :: (cname \times (cname \times (vname \times ty)\,list \times ((mname \times ty\,list) \times ty \times \gamma)\,list))\,list$ as a list of class files. Each class file records its class name, the name of its super class, a list of field declarations, and a list of method declarations. The type *cname* is assumed to have a distinguished element Object. Our program formalization gives rise to a subclass relation subcls1 and subclasses induce a subtype relation $\mathsf{subtype} :: \gamma\ prog \to ty \to ty \to bool$, where $\gamma\ prog$ is a type association defined as follows:

$$\begin{aligned}
\gamma\ mdecl &= (mname \times ty\ list) \times ty \times \gamma\\
\gamma\ class &= cname \times (vname \times ty)\ list \times \gamma\ mdecl\ list\\
\gamma\ cdecl &= cname \times \gamma\ class\\
\gamma\ prog &= \gamma\ cdecl\ list
\end{aligned}$$

Corresponding to the subtype P relation we have supremum on types $\mathsf{sup} :: ty \to ty \to ty\ err$. As abbreviations, we define $\mathsf{types}\ P = \{\tau \mid \mathsf{is\_type}\ P\ \tau\}$ and $\tau_1 \sqsubseteq_P \tau_2 = \tau_1 \leq_{\mathsf{subtype}\ P} \tau_2$. Below, we use the predicate $\mathsf{is\_class}\ P\ C$ to express that the class C is in the program P.

Well-formedness of JVM programs is defined by context conditions that can be checked statically prior to bytecode verification. We formalize this using a predicate $\mathsf{wf\_prog} :: \gamma\ wf\_mb \to \gamma\ prog \to bool$, where $\gamma\ wf\_mb = \gamma\ prog \to cname \to \gamma\ mdecl \to bool$. The predicate wf_mb expresses that a method $m :: \gamma\ mdecl$ in class C from program P is a well-typing. Informally, $\mathsf{wf\_prog}\ wf\_mb\ P$ means that: $\mathsf{subcls1}\ P$ is univalent (i.e. $\mathsf{subcls1}\ P$ represents a single inheritance hierarchy) and acyclic, and both $(\mathsf{subcls1}\ P)^{-1}$ and $(\mathsf{subtype}\ P)^{-1}$ are well-founded. The following lemma holds for all well-formed programs P:

Lemma 2. $\forall\ wf\_mb\ P.$
$\mathsf{wf\_prog}\ wf\_mb\ P \longrightarrow \mathsf{semilat}\,(\mathsf{sl}\,(\mathsf{types}\ P,\ \mathsf{subtype}\ P,\ \mathsf{sup}\ P)) \wedge \mathsf{finite}\,(\mathsf{types}\ P)$

We will use the semilattice $\mathsf{sl}\,(\mathsf{types}\ P,\ \mathsf{subtype}\ P,\ \mathsf{sup}\ P)$ to construct a semilattice with the carrier set of program states.

The JVM is a stack machine where each activation record consists of a stack for expression evaluation and a list of local variables (called **registers**). The abstract semantics, which operates with types as opposed to values, records the type of each stack element and each register. At different program points, a register may hold incompatible types, e.g. an integer or a reference, depending on the computation path that leads to that point. This facilitates the reuse of registers and is modeled by the HOL type *ty err*, where $\mathsf{OK}\ \tau$ represents the type τ and Err represents the inconsistent type. In our JVM formalization, the elements of the stack can also be reused. Thus the configurations of our abstract JVM are pairs of lists, which model an expression stack and a list of registers:

$$state\_type \;=\; ty\ err\ list\ \times\ ty\ err\ list$$

Note that this type differs from the type *ty list* $\times$ *ty err list* presented in [11], where the stack only holds the values that can actually be used.

We now define the type of the program state as *state_type option err*, where OK None indicates a program point that is not reachable (dead code), OK (Some s) is a normal configuration s, and Err is an error. In the following, we use the type association $state\ =\ state\_type\ option\ err$. Turning *state* into a semilattice structure is easy because all of its constituent types are (*err*-)semilattices. The set of operand stacks forms the carrier set of an *err*-semilattice because the supremum of stacks of different size is Err; the set of lists of registers forms the carrier set of a semilattice because the number of registers is fixed:

$$\begin{array}{ll}
\mathsf{stk_esl} & ::\ \gamma\ prog \to nat \to ty\ err\ list\ esl \\
\mathsf{stk_esl}\ P\ maxs & \equiv \mathsf{upto_esl}\ maxs\ (\mathsf{sl}\ (\mathsf{types}\ P,\ \mathsf{subtype}\ P,\ \mathsf{sup}\ P)) \\
\mathsf{reg_sl} & ::\ \gamma\ prog \to nat \to ty\ err\ list\ sl \\
\mathsf{reg_sl}\ P\ maxr & \equiv \mathsf{Listn.sl}\ maxr\ (\mathsf{sl}\ (\mathsf{types}\ P,\ \mathsf{subtype}\ P,\ \mathsf{sup}\ P))
\end{array}$$

Since any error on the stack must be propagated, the stack and registers are combined in a coalesced product using Product.esl and then embedded into *option* and *err* to form the semilattice sl:

$$\begin{array}{ll}
\mathsf{sl} & ::\ \gamma\ prog \to nat \to nat \to state\ sl \\
\mathsf{sl}\ P\ maxs\ maxr & \equiv \mathsf{Err.sl}\ (\mathsf{Opt.esl}\ (\mathsf{Product.esl}\ (\mathsf{stk_esl}\ P\ maxs)\ (\mathsf{Err.esl}\ \mathsf{reg_sl}\ P\ maxr)))
\end{array}$$

In the following we abbreviate sl P $maxs$ $maxr$ as sl, states P $maxs$ $maxr$ as states, le P $maxs$ $maxr$ as le, and sup P $maxs$ $maxr$ as sup. Using the properties about (*err*-) semilattices, it is easy to prove the following lemma:

Lemma 3.
(1). $\forall\, P\ maxs\ maxr.\ \mathsf{sl} = (\mathsf{states},\ \mathsf{le},\ \mathsf{sup})$,
(2). $\forall\, wf\_mb\ P\ maxs\ maxr.\ (\mathsf{wf\_prog}\ wf\_mb\ P) \longrightarrow \mathsf{semilat}\ (\mathsf{states},\ \mathsf{le},\ \mathsf{sup})$,
(3). $\forall\, wf\_mb\ P\ maxs\ maxr.\ (\mathsf{wf\_prog}\ wf\_mb\ P) \longrightarrow \mathsf{finite}\ (\mathsf{states})$,
(4). $\forall\, P\ maxs\ maxr.\ \mathsf{Err} \in \mathsf{states}$,
(5). $\forall\, P\ maxs\ maxr.\ \mathsf{top\ le\ Err}$,
(6). $\forall\, P\ maxs\ maxr.\ (\mathsf{OK\ None}) \in \mathsf{states}$, *and*
(7). $\forall\, P\ maxs\ maxr.\ \mathsf{bottom\ le}\ (\mathsf{OK\ None})$.

3.2 Program syntax and abstract semantics

The theory JVMInstructions defines the JVM instruction set. In our JVM formalization, the polymorphic instructions Load, Store, and CmpEq are replaced with instructions that have their counterparts in the Sun JVM instruction set, i.e. one such instruction for each base type (see the explanation at the end of Section 3.3).

$$
\begin{array}{llll}
\quad\textbf{datatype}\ \mathit{instr}\ = & \mathsf{iLoad}\ \mathit{nat} & \mid \mathsf{bLoad}\ \mathit{nat} & \mid \mathsf{aLoad}\ \mathit{nat} \\
\mid \mathsf{Invoke}\ \mathit{cname}\ \mathit{mname} & \mid \mathsf{iStore}\ \mathit{nat} & \mid \mathsf{bStore}\ \mathit{nat} & \mid \mathsf{aStore}\ \mathit{nat} \\
\mid \mathsf{Getfield}\ \mathit{vname}\ \mathit{cname} & \mid \mathsf{iIfcmpeq}\ \mathit{nat} & \mid \mathsf{LitPush}\ \mathit{val} & \mid \mathsf{Dup} \\
\mid \mathsf{Putfield}\ \mathit{vname}\ \mathit{cname} & \mid \mathsf{bIfcmpeq}\ \mathit{nat} & \mid \mathsf{New}\ \mathit{cname} & \mid \mathsf{Dup\_x1} \\
\mid \mathsf{Checkcast}\ \mathit{cname} & \mid \mathsf{aIfcmpeq}\ \mathit{nat} & \mid \mathsf{Return} & \mid \mathsf{Dup\_x2} \\
\mid \mathsf{Pop} & \mid \mathsf{Swap} & \mid \mathsf{IAdd} & \mid \mathsf{Goto}\ \mathit{int}
\end{array}
$$

We instantiate the polymorphic type of program γ *prog* using the type $nat \times nat \times instr\ list$, which reflects that the bytecode methods contain information about the class file attributes `max_stack` and `max_locals`, and the type of the method body. We model the type-level execution of a single instruction with $\mathsf{step}' :: instr \times jvm\_prog \times state\_type \times state\_type$, where $jvm\_prog = (nat \times nat \times instr\ list)\ list$. Below we show the definition of step' for selected instructions:

$$
\begin{array}{lcl}
\mathsf{step}'\ (\mathsf{iLoad}\ n,\ P,\ (st, reg)) & = & ((reg!n)\#st, reg) \\
\mathsf{step}'\ (\mathsf{iStore}\ n,\ P,\ (\tau_1\#st_1, reg)) & = & (st_1, reg[n := \tau_1]) \\
\mathsf{step}'\ (\mathsf{iAdd},\ P,\ ((\mathsf{OK\ Integer})\#(\mathsf{OK\ Integer})\#st_1, reg)) & = & ((\mathsf{OK\ Integer})\#st_1, reg) \\
\mathsf{step}'\ (\mathsf{Putfield}\ fn\ C,\ P,\ (\tau_v\#\tau_o\#st_1, reg)) & = & (st_1, reg) \\
\mathsf{step}'\ (\mathsf{iIfcmpeq}\ b,\ P,\ (\tau_1\#\tau_2\#st_1, reg)) & = & (st_1, reg) \\
\mathsf{step}'\ (\mathsf{Return},\ P,\ (st, reg)) & = & (st, reg)
\end{array}
$$

Note that the execution of the Return instruction is modeled by a self-loop. This will be useful when we model the traces of the transition system as a set of infinite sequences of program states. Finite sequences can occur only in ill-formed programs, where an instruction has no successor. The main omissions in our model are the same in [11]: both build on the type safety proof of Pusch [15], which does not cover exceptions, object initialization, and `jsr`/`ret` instructions.

3.3 Bytecode verifier specification

Now we define a predicate $\mathsf{app}' :: instr \times jvm\_prog \times nat \times ty \times state\_type \to bool$, which expresses the applicability of the function step' to a given configuration $(st, reg) :: state\_type$. We use this predicate later to formalize bytecode verification as a model-checking problem.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{app}'\ (i,\ P,\ maxs,\ rT,\ (st,\ reg)) \equiv \mathtt{case}\ i\ \mathtt{of}} \\
\ \ (\mathsf{iLoad}\ n) & \Rightarrow (n < \mathsf{size}\ st) \wedge (reg!n\ =\ (\mathsf{OK\ Integer})) \wedge (\mathsf{size}\ st < maxs) \\
\mid (\mathsf{iStore}\ n) & \Rightarrow \exists\,\tau\ st_1.\ (n < \mathsf{size}\ st) \wedge (st = \tau\#st_1) \wedge (\tau = (\mathsf{OK\ Integer})) \\
\mid \mathsf{iAdd} & \Rightarrow \exists\, st_1.\ st = (\mathsf{OK\ Integer})\#(\mathsf{OK\ Integer})\#\mathsf{st}_1 \\
\mid (\mathsf{Putfield}\ fn\ C) & \Rightarrow \exists\,\tau_v\ \tau_o\ \tau_f\ st_2.\ st = (\mathsf{OK}\ \tau_v)\#(\mathsf{OK}\ \tau_o)\#\mathsf{st}_2 \wedge (\mathsf{is\_class}\ P\ C) \wedge \\
 & \quad (\mathsf{field}\ (P, C)\ fn\ =\ \mathsf{Some}\ (C,\ \tau_f)) \wedge \tau_v \sqsubseteq \tau_f \wedge \tau_0 \sqsubseteq \mathsf{Class}\ C \\
\mid (\mathsf{iIfcmpeq}\ b) & \Rightarrow \exists\, st_2.\ st = (\mathsf{OK\ Integer})\#(\mathsf{OK\ Integer})\#\mathsf{st}_2 \\
\mid \mathsf{Return} & \Rightarrow \exists\,\tau\ \mathsf{st}_1.\ st = \tau\#st_1 \wedge \tau \sqsubseteq \mathsf{rT}
\end{array}
$$

Further, we introduce a successor function $\mathsf{succs} :: instr \to nat \to nat\ list$, which computes the possible successor instructions of a program point with respect to a given instruction.

$$
\begin{array}{l}
\mathsf{succs}\ i\ p \equiv \mathtt{case}\ i\ \mathtt{of}\ \mathsf{Return} \Rightarrow [p] \mid \mathsf{Goto}\ b \Rightarrow [p+b] \mid \mathsf{iIfcmpeq}\ b \Rightarrow [p+1,\ p+b] \\
\quad \mid \mathsf{bIfcmpeq}\ b \Rightarrow [p+1,\ p+b] \mid \mathsf{iIfcmpeq}\ b \Rightarrow [p+1,\ p+b] \mid \_ \Rightarrow [p+1]
\end{array}
$$

We use succs to construct the transition function of a finite transition system. To reason about the boundedness of succs, we define the predicate $\mathsf{bounded} :: (nat \to nat\ list) \to list \to bool$, where $\mathsf{bounded}\ f\ n\ \equiv\ \forall\, p < n.\ \forall\, q \in \mathsf{set}\ (f\ p).\ q < n$.

We can now define a well-typedness condition for bytecode methods. The predicate wt_method formalizes that, for a given program P, class C, method parameter list pTs, method return type rT, maximal stack size $maxs$, and maximal register index $maxr$, a bytecode instruction sequence bs is well-typed with respect to a given method type ϕ.

$$method\_type \;=\; state\_type\ option\ list$$

$$\begin{array}{l}
\mathsf{le_state_opt} :: jvm_prog \rightarrow state_type\ option \rightarrow state_type\ option \rightarrow bool \\
\mathsf{le_state_opt}\ P \;\equiv\; \mathsf{Opt.le}\ (\lambda\,(st_1,\ reg_1)\ (st_2,\ reg_2).\ st_1 \le_{[\mathsf{Err.le\ (subtype}\ P)]}\ st_2 \wedge reg_1 \le_{[\mathsf{Err.le\ (subtype}\ P)]}\ reg_2)
\end{array}$$

$$\begin{array}{l}
\mathsf{step} :: instr \rightarrow jvm_prog \rightarrow state_type\ option \rightarrow state_type\ option \\
\mathsf{step}\ i\ P \;\equiv\; \lambda\, i\, P\, s.\, \mathtt{case}\ s\ \mathtt{of}\ \mathsf{None} \;\Rightarrow\; \mathsf{None} \mid \mathsf{Some}\ s' \;\Rightarrow\; \mathsf{step}'\ (i,\ P,\ s')
\end{array}$$

$$\begin{array}{l}
\mathsf{app} :: instr \rightarrow jvm_prog \rightarrow nat \rightarrow ty \rightarrow state_type\ option \rightarrow bool \\
\mathsf{app}\ i\ P\ maxs\ rT\ s \;\equiv\; \mathtt{case}\ s\ \mathtt{of}\ \mathsf{None} \;\Rightarrow\; \mathtt{True} \mid \mathsf{Some}\ s' \;\Rightarrow\; \mathsf{app}'\ (i,\ P,\ maxs,\ rT,\ s')
\end{array}$$

$$\begin{array}{l}
\mathsf{wt_method} :: jvm_prog \rightarrow cname \rightarrow ty\ list \rightarrow ty \rightarrow nat \rightarrow nat \rightarrow instr\ list \rightarrow method_type \rightarrow bool \\
\mathsf{wt_method}\ P\ C\ pTs\ rT\ maxs\ maxl\ bs\ \phi \;\equiv \\
(0 < \mathsf{size}\ bs) \wedge \\
(\mathsf{Some}\ ([],\ (\mathsf{OK}\ (\mathsf{Class}\ C))\#(\mathsf{map\ OK}\ pTs)@(\mathsf{replicate}\ maxl\ \mathsf{Err})))\ \le_{\mathsf{le_state_opt}}\ \phi!0 \wedge \\
(\forall\, pc.\, pc < \mathsf{size}\ bs \;\longrightarrow \\
\quad (\mathsf{app}\ (bs!pc)\ P\ maxs\ rT\ (\phi!pc)) \wedge \\
\quad (\forall\, pc' \in \mathsf{set}\ (\mathsf{succs}\ (bs!pc)\ pc).\ (pc' < \mathsf{size}\ bs) \;\wedge\; (\mathsf{step}\ (bs!pc)\ P\ (\phi!pc)\ \le_{\mathsf{le_state_opt}}\ \phi!pc'))
\end{array}$$

This definition of a well-typed method is in the style of that given by Pusch [15].

The last element of our JVM model is the definition of the abstract execution function exec:

$$\begin{array}{l}
\mathsf{exec} :: jvm_prog \rightarrow nat \rightarrow ty \rightarrow instr\ list \rightarrow nat \rightarrow state \rightarrow state \\
\mathsf{exec}\ P\ maxs\ rT\ bs\ pc \;\equiv\; \lambda\, s.\, \mathtt{case}\ s\ \mathtt{of}\ \mathsf{Err} \;\Rightarrow\; \mathsf{Err} \mid \mathsf{OK}\ s' \;\Rightarrow \\
\qquad \mathtt{if}\ \mathsf{app}\ (bs!pc)\ P\ maxs\ rT\ s'\ \mathtt{then}\ \mathsf{OK}\ (\mathsf{step}\ (bs!pc)\ P\ s')\ \mathtt{else}\ \mathsf{Err}
\end{array}$$

Abbreviating exec $P\ maxs\ rT\ bs\ pc$ as exec, we now prove:

Lemma 4. $\forall\, wf\_mb\ P\ maxs\ maxr\ bs\ pc.\ \forall\, s_1\ s_2 \in \mathsf{states}.$
$(\mathsf{wf\_prog}\ wf\_mb\ P \wedge \mathsf{semilat\ sl}) \longrightarrow \mathsf{exec}\ (s_1\ +_{\mathsf{sup}}\ s_2)\ \le_{\mathsf{le}}\ (\mathsf{exec}\ s_1)\ +_{\mathsf{sup}}\ (\mathsf{exec}\ s_2)$

This lemma states a "semi-homomorphism" property of the exec function with respect to the le relation. To prove it we must show that it holds in our formalization of the JVM that if two arbitrary program states $x,\, y \in A$ are well-typed with respect to an instruction at a given program point, then so is $x +_f y$. This would have been impossible to prove using the Nipkow's formalization of the JVM with polymorphic instructions; hence we have replaced the polymorphic instructions in the JVM instruction set with collections of monomorphic ones.

4 Model checking

Bytecode verification by model checking is based on the idea that the bytecode of a method represents a transition system which, when suitably abstracted (e.g. by replacing data with their types) is finite and that one can uniformly produce, from bytecode, a temporal logic formula that holds iff the bytecode is well-typed with respect to a given type system. In [1] we presented a system based on this idea. For a bytecode method M our system generates a transition system K (in the input language of either the Spin or the Smv model checker) and a correctness property ψ and uses the model checker to prove that ψ is globally invariant (i.e. $\Box\,\psi$ or $AG\,\psi$ depending on the model checker).

Here we verify the correctness of this approach. Namely we first formalize the translation of a bytecode method M to the transition system K; second we formalize the way in which ψ is generated, and third we formalize what it means for ψ to globally hold for K, i.e. $K \models \Box\psi$. We show that when model checking succeeds then there exists a type ϕ that is a well-typing for the method M.

4.1 Kripke structures and method abstraction

Kripke structures. In the following we define Kripke structures, which we use to formalize the abstract transition system of a method. A Kripke structure K consists of a non-empty set of states, a set of initial states, which is a subset of the states of K, and a transition relation next over the states of K. A trace of K is an infinite sequence of states such that the first state is an initial state and successive states are in the transition relation of K. A state is reachable in K if it is contained in a trace of K. We also define a suffix function on traces, which is needed to define the semantics of LTL-formulas. These definitions are standard and their formalization in Isabelle/HOL is straightforward.

$$\alpha\ \mathit{kripke} = (\!|\ \mathsf{states} :: \alpha\ \mathit{set}, \mathsf{init} :: \alpha\ \mathit{set}, \mathsf{next} :: (\alpha \times \alpha)\ \mathit{set}\ |\!)$$

$$\mathsf{is\_kripke}\ ::\ \alpha\ \mathit{kripke} \to \mathit{bool}$$
$$\mathsf{is\_kripke}\ K\ \equiv\ \mathsf{states}\ K \neq \emptyset \wedge \mathsf{init}\ K \subseteq \mathsf{states}\ K \wedge \mathsf{next}\ K \subseteq \mathsf{states}\ K \times \mathsf{states}\ K$$

$$\alpha\ \mathit{trace} = \mathit{nat} \to \alpha$$

$$\mathsf{is\_trace}\ ::\ \alpha\ \mathit{kripke} \to \alpha\ \mathit{trace} \to \mathit{bool}$$
$$\mathsf{is\_trace}\ K\ t\ \equiv\ t\ 0 \in \mathsf{init}\ K \wedge \forall i.\ (t\ i,\ t\ (\mathsf{Suc}\ i)) \in \mathsf{next}\ K$$

$$\mathsf{traces}\ ::\ \alpha\ \mathit{kripke} \to \alpha\ \mathit{trace}\ \mathit{set}$$
$$\mathsf{traces}\ K\ \equiv\ \{t \mid \mathsf{is\_trace}\ K\ t\}$$

$$\mathsf{reachable}\ ::\ \alpha\ \mathit{kripke} \to \alpha \to \mathit{bool}$$
$$\mathsf{reachable}\ K\ q\ \equiv\ \exists\, t\, i.\ \mathsf{is\_trace}\ K\ t \wedge q = t\ i$$

$$\mathsf{suffix}\ ::\ \alpha\ \mathit{trace} \to \mathit{nat} \to \alpha\ \mathit{trace}$$
$$\mathsf{suffix}\ t\ i\ \equiv\ \lambda j.\ t\ (i + j)$$

Method abstraction. With the above definitions at hand, we can formalize the abstraction of bytecode methods as a finite Kripke system over the type $\mathit{abs\_state} = \mathit{nat} \times \mathit{state\_type}$. We generate the transition system using the function abs_method, which for a given program P, class name C, method parameter list pTs, return type rT, maximal register index maxr, and bytecode instruction sequence bs, yields a Kripke structure of type $\mathit{abs\_state}\ \mathit{kripke}$.

$$\mathsf{abs\_method} :: \mathit{jvm\_prog} \to \mathit{cname} \to \mathit{ty} \to \mathit{list} \to \mathit{ty} \to \mathit{nat} \to \mathit{instr}\ \mathit{list} \to \mathit{abs\_state}\ \mathit{kripke}$$
$$\begin{aligned}&\mathsf{abs\_method}\ P\ C\ pTs\ rT\ \mathit{maxr}\ bs\ \equiv\\ &\qquad (\!|\ \mathsf{states}\ =\ \mathsf{UNIV},\\ &\qquad\quad \mathsf{init}\ =\ \mathsf{abs\_init}\ C\ pTs\ \mathit{maxr},\\ &\qquad\quad \mathsf{next}\ =\ (\textstyle\bigcup_{pc \in \{p.\, p < (\mathsf{size}\ bs)\}} \mathsf{abs\_instr}\ (bs!pc)\ P\ pc)\ |\!)\end{aligned}$$

The set of states of the transition system is modeled by the set UNIV of all elements of type $\mathit{abs\_state}$. The initial abstract state set abs_init contains one element, which models the method entry where the program counter is set to 0 and the stack is empty. At the method entry, the list of local variables contains the this reference OK (Class C), the methods parameters map OK pTs, and maxr uninitialized registers replicate maxr Err.

$$\mathsf{abs\_init}\ ::\ \mathit{cname}\ \to\ \mathit{ty}\ \mathit{list}\ \to\ \mathit{nat}\ \to\ \mathit{abs\_state}\ \mathit{set}$$
$$\mathsf{abs\_init}\ C\ pTs\ \mathit{maxr}\ \equiv\ \{\,(0,\ ([],\ (\mathsf{OK}\ (\mathsf{Class}\ C))\#((\mathsf{map}\ \mathsf{OK}\ pTs))@(\mathsf{replicate}\ \mathit{maxr}\ \mathsf{Err})))\,\}$$

The relation next is generated by the function $\mathsf{abs\_instr}$, which uses step' and succs, which together make up the transition relation of our transition system. As we have shown in a lemma, the next relation is finite because both the type system and the number of storage locations (stack and local variables) of the abstracted method are finite.

$$\begin{array}{l}\mathsf{abs\_instr} :: \mathit{instr} \rightarrow \mathit{jvm\_prog} \rightarrow \mathit{nat} \rightarrow (\mathit{abs\_state} * \mathit{abs\_state})\ \mathit{set}\\ \mathsf{abs\_instr}\ i\ P\ pc \equiv \{((pc', q), (pc'', q')) \mid pc'' \in \mathsf{set}\,(\mathsf{succs}\ i\ pc') \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad (q' = \mathsf{step}'\,(i, P, q)) \wedge pc = pc'\}\end{array}$$

4.2 Temporal logic and applicability conditions

Temporal logic. The syntax of LTL formulae is given by the datatype $\alpha\ ltl$.

$$\begin{array}{l}\textbf{datatype}\ \alpha\ ltl \equiv \mathsf{Tr} \mid \mathsf{Atom}\,(\alpha \rightarrow bool) \mid \mathsf{Neg}\,(\alpha\ ltl) \mid \mathsf{Conj}\,(\alpha\ ltl)\,(\alpha\ ltl)\\ \qquad\qquad\qquad \mid \mathsf{Next}\,(\alpha\ ltl) \mid \mathsf{Until}\,(\alpha\ ltl)\,(\alpha\ ltl)\end{array}$$

As is standard, the modalities eventually, $\Diamond :: \alpha\ ltl \rightarrow \alpha\ ltl$, and globally, $\Box :: \alpha\ ltl \rightarrow \alpha\ ltl$, can be defined as syntactic sugar.

$$\Diamond f \equiv \mathsf{Until}\ \mathsf{Tr}\ f \qquad \Box f \equiv \mathsf{Neg}\,(\Diamond\,(\mathsf{Neg}\ f))$$

We give the semantics of LTL formulae using a satisfiability predicate

$$\_ \models \_ :: \alpha\ trace \rightarrow \alpha\ kripke \rightarrow \alpha\ ltl \rightarrow bool$$

where

$$\begin{array}{lcl}t \models \mathsf{Tr} & = & \mathsf{True}\\ t \models \mathsf{Atom}\ p & = & p\,(t\,0)\\ t \models \mathsf{Neg}\ f & = & \neg\,(t \models f)\\ t \models \mathsf{Conj}\ f\ g & = & (t \models f) \wedge (t \models g)\\ t \models \mathsf{Next}\ f & = & \mathsf{suffix}\ t\ 1 \models f\\ t \models \mathsf{Until}\ f\ g & = & \exists j.\,(\mathsf{suffix}\ t\ j \models g) \wedge\\ & & \quad (\forall i.\, i < j \longrightarrow \mathsf{suffix}\ t\ i \models f)\end{array}$$

Furthermore, we say that property f is globally satisfied in the transition system K if it is satisfied for all traces of K.

$$\begin{array}{l}\_ \models \_ :: \alpha\ kripke \rightarrow \alpha\ ltl \rightarrow bool\\ K \models f \equiv \forall t \in \mathsf{traces}\ K.\ t \models f\end{array}$$

Applicability conditions. The applicability conditions are expressed by an LTL formula of the form $\Box\,\mathsf{Atom}\ f$. We extract the formula f from the bytecode, using the functions $\mathsf{spc\_instr}$ and $\mathsf{spc\_method}$.

$$\begin{array}{l}\mathsf{spc\_instr} :: instr \rightarrow jvm\_prog \rightarrow nat \rightarrow nat \rightarrow nat \rightarrow ty \rightarrow abs\_state \rightarrow bool\\ \mathsf{spc\_instr}\ i\ P\ pc'\ h\ maxs\ rT \equiv\\ \qquad \lambda\,(pc, q).\ pc = pc' \longrightarrow ((\mathsf{size}\,(fst\ q)) = h) \wedge (\mathsf{app}'\,(i, P, maxs, rT, q))\\ \mathsf{spc\_method} :: jvm\_prog \rightarrow nat\ list \rightarrow ty \rightarrow nat \rightarrow instr\ list \rightarrow abs\_state \rightarrow bool\\ \mathsf{spc\_method}\ P\ hs\ rT\ maxs\ bs\ q \equiv\\ \qquad \forall\, pc < (\mathsf{size}\ bs).\ (\mathsf{spc\_instr}\ (bs!pc)\ P\ pc\ (hs!pc)\ maxs\ rT\ q)\end{array}$$

The function $\mathsf{spc\_instr}$ expresses the instruction applicability condition for a program state at a program point pc'. Note that besides the conditions formalized in app', we also require that, in all states associated with this program point, the stack has a fixed predefined size h. We employ this to fit the polyvariant model-checking approach into the monovariant framework. In order to prove that there is a state type, we must show that the supremum of all possible stack types associated with the program point is different from the error element Err, and hence we require this restriction.

The function spc_method builds f as the conjunction of the applicability conditions of all program points in the bytecode bs (the list hs contains, for each program point in bs, a number that specifies the stack size at that program point). The resulting formula expresses the well-typedness condition for a given program state and hence, $K \models \Box \mathsf{Atom} f$ formalizes, for a Kripke structure K resulting from a method abstraction, the global type safety of the method. This formula, together with the additional constraints explained below, makes up the definition of a bytecode-modelchecked method, which is formalized using the predicate bcm_method.

$$
\begin{array}{l}
\mathsf{bcm\_method} :: \mathit{jvm\_prog} \rightarrow \mathit{cname} \rightarrow \mathit{ty\ list} \rightarrow \mathit{ty} \rightarrow \mathit{nat} \rightarrow \mathit{nat} \rightarrow \mathit{instr\ list} \rightarrow \mathit{bool} \\
\mathsf{bcm\_method}\ P\ C\ pTs\ rT\ maxr\ maxs\ bs\ \equiv\ \exists\, hs. \\
\quad (\mathsf{abs\_method}\ P\ C\ pTs\ rT\ maxr\ bs)\ \models\ \Box\ (\mathsf{Atom}\ (\mathsf{spc\_method}\ P\ hs\ rT\ maxs\ bs)) \wedge \\
\quad \mathsf{size}\ hs\ =\ \mathsf{size}\ bs\ \wedge \\
\quad (\mathsf{traces}\ (\mathsf{abs\_method}\ P\ C\ pTs\ rT\ maxr\ bs))\ \neq\ \{\}\ \wedge \\
\quad (\forall\, (x,\, y)\ \in\ (\mathsf{next}\ (\mathsf{abs\_method}\ P\ C\ pTs\ rT\ maxr\ bs)). \\
\quad\quad \exists\, t\, i.\ (\mathsf{is\_trace}\ (\mathsf{abs\_method}\ G\ C\ pTs\ rT\ maxr\ bs)\ t)\ \wedge\ (t\, i,\, t\, (\mathsf{Suc}\ i))\ =\ (x,\, y)) \wedge \\
\quad 0\ <\ (\mathsf{size}\ bs)\ \wedge \\
\quad \mathsf{bounded}\ (\lambda\, n.\ \mathsf{succs}\ (bs!n)\ n)\ (\mathsf{size}\ bs)
\end{array}
$$

The first conjunct expresses the global applicability conditions for the given bytecode bs. The second conjunct requires that a stack size is specified for every program point. The third conjunct states that the transition system is nonempty, i.e. that there exists at least one non-trivial model for a given LTL formula. The fourth conjunct characterizes a liveness property of the abstract interpreter. We use this assumption to show that a given program point pc, which is reachable in finitely many steps, is also contained in an (infinite) trace $t \in \mathit{traces}\, K$. The last two conjuncts express well-formedness properties of the bytecode, namely, that the list of instructions is not empty and that all instructions have a successor.

4.3 Main theorem and proof

Our main theorem, stating the correctness of bytecode verification by model checking can now be given.

Theorem 1.

$$
\begin{array}{l}
(\mathsf{wf\_prog}\ \mathit{wf\_mb}\ P)\ \wedge\ (\mathsf{is\_class}\ P\ C)\ \wedge\ (\forall\, x \in \mathsf{set}\ pTs.\ \mathsf{is\_type}\ P\ x)\ \longrightarrow \\
(\mathsf{bcm\_method}\ P\ C\ pTs\ rT\ maxr\ maxs\ bs) \longrightarrow \exists\, \phi.\ \mathsf{wt\_method}\ P\ C\ pTs\ rT\ maxs\ maxr\ bs\ \phi
\end{array}
$$

This theorem states the correctness only for the bytecode bs for a single method, of a single class C, of a program P; however, it can easily be extended to a more general statement for the entire program P. We have actually proved in Isabelle the more general statement but, to avoid unnecessary complexity, we restrict ourselves here to this simpler statement.

This theorem has four assumptions: (A1) the program P is well-formed; (A2) the class C belongs to the program P; (A3) the method signature is well-formed (i.e., all method parameters have declared types); and (A4) that the bytecode bs is successfully model-checked. Under these assumptions, the conclusion states that bs has a well-typing given by the method type ϕ. The impact of this theorem is that the existence of such a well-typing implies that bs is free of runtime type errors.

In the following, to increase readability, we abbreviate $\mathsf{abs\_method}\ P\ C\ pTs\ rT\ maxr\ bs$ as $\mathsf{abs\_method}$ and $\mathsf{bcm\_method}\ P\ C\ pTs\ rT\ maxr\ maxs\ bs$ as $\mathsf{bcm\_method}$.

Proof intuition. Our proof of the existence of a method type ϕ is constructive and builds ϕ from the transition system K pointwise for each program point pc as follows. First, we define a function $\_\ \mathsf{at}\ \_ :: (\alpha \times \beta)\ kripke \to \alpha \to \beta\ set$, where $K\ \mathsf{at}\ pc \equiv \{\, a \mid \mathsf{reachable}\, K\, (pc, a)\,\}$ denotes the program configurations reachable on any trace that belong to the program point pc. Second, we build the supremum sup_{pc} of the set $K\ \mathsf{at}\ pc$. The third step uses the fact (which must be proved) that each supremum sup_{pc} is of the form $\mathsf{OK}\ x_{pc}$ (i.e., it is not Err) and hence x_{pc} is the method type at the program point pc. In essence, we perform a data flow analysis here in our proof to build a method type.

The bulk of our proof is to show that this method type is actually a well-typing for the model checked bytecode. The main challenge here is to show that each method instruction is well-typed with respect to the instruction's state type (applicability) and that the successors of an instruction are well-typed with respect to the state type resulting from the execution of the instruction (stability). We establish this by induction on the set $K\ \mathsf{at}\ pc$; this corresponds to an induction over all traces that contribute to the state set associated with the pcth program point. Since model checking enforces applicability and thus stability for every trace, and since exec is semi-continuous in the sense that

$$\mathsf{exec}\ (\textstyle\bigsqcup_{\mathsf{sl}}\ (\mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,\mathsf{abs\_method\ at}\ pc)) \le_{\mathsf{le}} \textstyle\bigsqcup_{\mathsf{sl}}\ (\mathsf{exec}\,\text{`}\,(\mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,(\mathsf{abs\_method\ at}\ pc))),$$

then the pointwise supremum of the state types gives rise to a method type that is a well-typing.

Some proof details. We now describe the construction of the method type more formally and afterwards the proof that it is a well-typing.

The first two steps of the construction are formalized using the function $\mathsf{ok\_method\_type} :: jvm\_prog \to cname \to ty\ list \to ty \to nat \to nat \to instrlist \to nat \to state$, where we abbreviate $\bigsqcup_{\mathsf{sl,Err,OK(None)}}$ as $\bigsqcup_{\mathsf{sl}}$.

$$\begin{array}{l}
\mathsf{ok_method_type}\ P\ C\ pTs\ rT\ maxr\ maxs\ bs\ pc\ \equiv \\
\quad \mathtt{let}\ sl = \mathsf{sl}\ P\ maxs\ (1 + (\mathsf{size}\ pTs) + maxr); \\
\quad\quad K = \mathsf{abs_method}\ P\ C\ pTs\ rT\ maxr\ bs; \\
\quad \mathtt{in}\ \bigsqcup_{sl}\ \mathsf{OK}\,\text{`}\,\mathsf{Some}\,\text{`}\,(K\ \mathsf{at}\ pc)
\end{array}$$

Here, the supremum function over sets, $\bigsqcup_{sl}$, builds from K at pc the supremum sup_{pc} in the semilattice sl. To increase readability, we will use $\mathsf{ok\_method\_type}$ as an abbreviation for $\mathsf{ok\_method\_type}\ P\ C\ pTs\ rT\ maxr\ maxs\ bs$.

For the third step, we extract the method type using the function $\mathsf{ok\_val} :: \alpha\ err \to \alpha$, where $\mathsf{ok\_val}\, e = \mathtt{case}\ e\ \mathtt{of}\ (\mathsf{OK}\, x) \Rightarrow x \mid \_ \Rightarrow \mathsf{arbitrary}$. Thus the method type at program point pc can be computed by the function $\mathsf{method\_type} \equiv \mathsf{ok\_val} \circ \mathsf{ok\_method\_type}$ and we chose as the overall method type

$$\phi = \ \mathsf{map}\ (\mathsf{method\_type}\, P\ C\ pTs\ rT\ maxr\ maxs\ bs)\ [0, \ldots, \mathsf{size}\ bs - 1]\ .$$

Before explaining why ϕ is a well-typing, we state two lemmata central to the proof. As mentioned above, the correctness of our construction requires that for each program point pc, there is a state type x such that $\mathsf{ok\_method\_type}\ pc = \mathsf{OK}\ x$. The following lemma states this, and moreover that this type is applicable at program point pc.

Lemma 5. ***Well-typed supremum***

$$0 < \mathsf{size}\; bs \;\wedge\; pc < \mathsf{size}\; bs \;\wedge\; \mathsf{wf\_prog}\; wf\_mb\; P \;\wedge\; \mathsf{is\_class}\; P\; C \;\wedge\; (\forall x \in \mathsf{set}\; pTs.\; \mathsf{is\_type}\; P\; x) \;\wedge$$
$$\mathsf{size}\; hs \;=\; \mathsf{size}\; bs \;\wedge\; \mathsf{bounded}\; (\lambda n.\; \mathsf{succs}\; (bs!n)\; n)\; (\mathsf{size}\; bs) \;\wedge\; (\mathsf{abs\_method} \models\; \Box\; \mathsf{Atom}\; \mathsf{spc\_method})$$
$$\longrightarrow \exists x.\; (\mathsf{ok\_method\_type}\; pc = \mathsf{OK}\; x) \;\wedge\; (x = \mathsf{None} \vee (\exists\, st\; reg.\; x = \mathsf{Some}\,(st,\; reg) \;\wedge$$
$$\mathsf{app}'\; (bs!pc,\; P,\; maxs,\; rT,\; (st,\; reg)) \;\wedge\; (\mathsf{size}\; st \;=\; hs!pc)))$$

We prove this lemma by induction on the set of program states that belong to the program point pc. This proof uses the fact that if a property holds globally in a Kripke structure then it holds for every reachable state of the structure. The step case requires the following lemma, which states that when two states (st_1, reg_1) and (st_2, reg_2) of the type *state_type* satisfy the predicate app' for the instruction $bs!pc$ and the sizes of their stacks are equal, then the supremum of these states also satisfies the applicability condition for this instruction and the size of supremum's stack is the size of the stack of the two states. Here we abbreviate lift JType.sup P as sup'.

Lemma 6. ***Well-typed induction step***

$$pc < (\mathsf{size}\; bs) \;\wedge\; \mathsf{semilat}\; \mathsf{sl} \;\wedge\; \mathsf{size}\; hs = \mathsf{size}\; bs \longrightarrow$$
$$\mathsf{size}\; st_1 = hs!pc \;\wedge\; \mathsf{app}'\; (bs!pc,\; P,\; maxs,\; rT,\; (st_1, reg_1)) \;\wedge$$
$$\mathsf{size}\; st_2 = hs!pc \;\wedge\; \mathsf{app}'\; (bs!pc,\; P,\; maxs,\; rT,\; (st_2, reg_2)) \longrightarrow$$
$$\mathsf{size}\; (st_1 +_{[\mathsf{sup}']}\; st_2) = hs!pc \;\wedge\; \mathsf{app}'\; (bs!pc,\; P,\; maxs,\; rT,\; (st_1 +_{[\mathsf{sup}']}\; st_2,\; reg_1 +_{[\mathsf{sup}']}\; reg_2))$$

We now sketch the proof that ϕ is a well-typing. To simplify notation, we abbreviate the initial state type $([], (\mathsf{OK}(\mathsf{Class} C))\#\; (\mathsf{map} \mathsf{OK}\, pTs)@(\mathsf{replicate}\, maxl\, \mathsf{Err}))$ as (st_0, reg_0).

Following the definition of wt_method, we must show, that under the four assumptions (A1)–(A4), three properties hold: First, the method body must contain at least one instruction, i.e. $0 < \mathsf{size}\; bs$. This follows directly from the definition of bcm_method.

Second, the start of the method must be well-typed, that is $\mathsf{Some}\,(st_0,\; reg_0) \leq_{\mathsf{le\_state\_opt}} \phi!0$. Since the set of traces is not empty, the initial state type $\mathsf{Some}\,(st_0,\; reg_0)$ is contained in the set $\mathsf{Some}\,`\,(\mathsf{abs\_method}\; \mathsf{at}\; 0)$ and hence it follows that

$$\mathsf{OK}\,(\mathsf{Some}\,(st_0, reg_0)) \leq_{\mathsf{le}} \bigsqcup\nolimits_{\mathsf{sl}}\; (\mathsf{OK}\,`\,\mathsf{Some}\,`\,(\mathsf{abs\_method}\; \mathsf{at}\; 0))\,,$$

which is (by the definition of ok_method_type) the same as $\mathsf{OK}\,(\mathsf{Some}\,(st_0, reg_0)) \leq_{\mathsf{le}} \mathsf{ok\_method\_type}\; 0$. By lemma 5 we know that the right-hand side of the inequality is an OK value and thus we can strip off OK yielding $\mathsf{Some}\,(st_0,\; reg_0) \leq_{\mathsf{le\_state\_opt}} \mathsf{ok\_val}(\mathsf{ok\_method\_type}\; 0)$, which is (by the choice of ϕ and the definition of ok_method_type) the desired result.

Finally, we must show that all instructions of the method bs are well-typed, i.e.,

$$\begin{aligned} &\forall (pc < \mathsf{size}\; bs).\; \forall\, pc' \in \mathsf{set}\,(\mathsf{succs}\; (bs!pc)\; pc). \\ &\quad pc' < \mathsf{size}\; bs \;\wedge && \text{(boundedness)} \\ &\quad \mathsf{app}\; (bs!pc)\; P\; maxs\; rT\; (\phi!pc) \;\wedge && \text{(applicability)} \\ &\quad \mathsf{step}\; (bs!pc)\; P\; (\phi!pc) \;\leq_{\mathsf{le\_state\_opt}}\; (\phi!pc') && \text{(stability)} \end{aligned}$$

This splits into three subgoals (boundedness, applicability, and stability). For all three we fix a pc, where $pc < \mathsf{size}\; bs$ and a successor pc' of pc. Boundedness holds trivially, since from bcm_method it follows that succs is bounded.

To show the applicability of the instruction at the program point pc, by the definition of method_type, we must show that

$$\mathsf{app}\; (bs!pc)\; P\; maxs\; rT\; (\mathsf{ok\_val}\; (\mathsf{ok\_method\_type}\; pc))\,,$$

which we prove by case analysis. The first case is when $\mathsf{ok\_method\_type}\ pc$ is $\mathsf{OK\ None}$, which in our formalization means that pc is not reachable (i.e. there is dead code). Then applicability holds by the definition of app. The second case is when the transition system $\mathsf{abs\_method}$ generates program configurations that belong to the program point pc. We must then prove (according to the definition of app)

$$\mathsf{app}'\,(bs!pc,\ P,\ maxs,\ rT,\ (st,\ reg))\,,$$

which follows from Lemma 5. This lemma also guarantees that the third case, where $\mathsf{ok\_method\_type}\ pc$ is Err, does not occur.

The majority of the work is in showing stability. Due to the choice of ϕ and the definition of $\mathsf{method\_type}$ we must prove

$$\mathsf{step}\ (bs!pc)\ P\ (\mathsf{ok\_val}\ (\mathsf{ok\_method\_type}\ pc))\ \ \leq_{\mathsf{le\_state\_opt}}\ (\mathsf{ok\_val}\ (\mathsf{ok\_method\_type}\ pc'))\,.$$

By the definition of $\leq_{\mathsf{le}}$ it suffices to show the inequality

$$\mathsf{OK}(\mathsf{step}\ (bs!pc)\ P\ (\mathsf{ok\_val}\ (\mathsf{ok\_method\_type}\ pc)))\ \ \leq_{\mathsf{le}}\ \mathsf{OK}(\mathsf{ok\_val}\ (\mathsf{ok\_method\_type}\ pc'))$$

Lemma 5 states the applicability of $\mathsf{ok\_val}\ (\mathsf{ok\_method\_type}\ pc)$ to the instruction at program point pc on the left-hand side of the inequality and hence, by the definition of exec, we can reduce our problem to

$$\mathsf{exec}(\mathsf{OK}(\mathsf{ok\_val}\ (\mathsf{ok\_method\_type}\ pc)))\ \ \leq_{\mathsf{le}}\ \mathsf{OK}(\mathsf{ok\_val}\ (\mathsf{ok\_method\_type}\ pc'))\,.$$

Moreover, this lemma allows us to conclude that $\mathsf{ok\_method\_type}$ delivers OK values for pc and pc' and thus the argument of exec is equal to $\mathsf{ok\_method\_type}\ pc$ and the right-hand side of the inequality is equal to $\mathsf{ok\_method\_type}\ pc'$. By unfolding the definition of $\mathsf{ok\_method\_type}$, the inequality simplifies to

$$\mathsf{exec}\ (\bigsqcup_{\mathsf{sl}}\ (\mathsf{OK}\ `\ \mathsf{Some}\ `\ (\mathsf{abs\_method\ at}\ pc)))\ \leq_{\mathsf{le}}\ \bigsqcup_{\mathsf{sl}}\ (\mathsf{OK}\ `\ \mathsf{Some}\ `\ (\mathsf{abs\_method\ at}\ pc'))\,.\qquad (*)$$

In Lemma 1 we proved that if a function g is a semi-homomorphism and $g\ `\ A'' \subseteq A'$, then $g\ (\bigsqcup_{\mathsf{sl}} A'') \leq_{\mathsf{le}} \bigsqcup_{\mathsf{sl}} A'$. Inequation (*) is an instance of the conclusion of this lemma. We can prove the (corresponding instance of the) first premise of this lemma, showing that exec is a semi-homomorphism, using Lemma 4. Thus, it suffices to prove the (corresponding instance of the) second premise, i.e, to prove

$$(\mathsf{exec}\ `\ (\mathsf{OK}\ `\ \mathsf{Some}\ `\ (\mathsf{abs\_method\ at}\ pc))) \subseteq (\mathsf{OK}\ `\ \mathsf{Some}\ `\ (\mathsf{abs\_method\ at}\ pc'))\,.$$

We assume for an arbitrary state type $(st,\ reg)$ that $(st,\ reg) \in \mathsf{abs\_method\ at}\ pc$ and prove that

$$\exists (st',\ reg') \in \mathsf{abs\_method\ at}\ pc'.\ \mathsf{exec}(\mathsf{OK}(\mathsf{Some}(st,\ reg))) = \mathsf{OK}(\mathsf{Some}\ (st',\ reg'))\,.$$

From this assumption it follows that $(pc, (st,\ reg))$ is a reachable state of $\mathsf{abs\_method}$, which together with (A4) entails that the applicability conditions hold for $(st,\ reg)$. Hence, by the definition of exec, we can reduce our goal to

$$\exists (st',\ reg') \in \mathsf{abs\_methodat}pc'.\ \mathsf{OK}(\mathsf{step}\ (bs!pc)\ P\ \mathsf{Some}(st,\ reg))) = \mathsf{OK}(\mathsf{Some}\ (st',\ reg'))$$

and further, by the definition of step and by stripping off OK, to

$$\exists (st',\ reg') \in \mathsf{abs\_method\ at}\ pc'.\ \mathsf{step}'\ (bs!pc, P, (st, reg)) = (st', reg')\,.$$

However, this is equivalent to

$$\exists (st',\ reg') \in \mathsf{abs\_method\ at}\ pc'.\ ((st, reg),\ (st', reg')) \in \mathsf{next}(\mathsf{abs\_method})\,,$$

which follows directly from the liveness property that is part of (A4).

5 Conclusion

We have formalized and proved in Isabelle/HOL the correctness of our approach to model checking based bytecode verification. We were fortunate in that we could build on the framework of Pusch and Nipkow. As such, our work also constitutes a fairly large scale example of theory reuse, and the generality of their formalisms, in particular Nipkow's verification framework, played a major role in this regard. As mentioned in the introduction, the changes we made to the verification framework appear generally useful. There are some loose ends remaining though as future work. First, we would like to formalize the monovariant approach in our framework. Second, our approach to polyvariant analysis required slight changes in the stack component of the state type and this requires a new proof that programs possessing a well-typing are type sound. This should be straightforward but also remains as future work. Finally, our approach supports polyvariant analysis in that it admits bytecode with incompatible types at different program points. However, for each program point, our formalization requires the stack size to be a constant. As a result, our current formalization constitutes a midpoint, in terms of the set of programs accepted, between Nipkow's formalization and our implementation. It is trivial to implement the requirement on the stack size as a model-checked property in our implementation; however, it would be more interesting to lift this requirement in our Isabelle model, for example, by further generalizing the notion of a state type to be a set of types.

References

1. D. Basin, S. Friedrich, M. J. Gawkowski, J. Posegga. Bytecode Model Checking: An Experimental Analysis. In *9th International SPIN Workshop on Model Checking of Software*, 2002, volume 2318 of *LNCS*, pages 42–59, Grenoble. Springer-Verlag, 2002.
2. Y. Bertot. A Coq formalization of a type checker for object initialization in the Java Virtual Machine. Technical Report RR-4047, INRIA, Nov. 2000.
3. A. Coglio, A. Goldberg, and Z. Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *Proc. DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. 2*, pages 403–410. IEEE Computer Society Press, 2000.
4. S. Freund and J. Mitchell. A type system for object initialisation in the Java bytecode language. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1998.
5. S. N. Freund and J. C. Mitchell. A formal framework for the java bytecode language and verifier. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1999.
6. A. Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conf. Computer and Communications Security*, 1998.
7. M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java virtual machine subroutines. In G. Levi, editor. *Static Analysis (SAS'98)*, volume 1503 of *LNCS*, pages 17-32. Springer-Verlag, 1998.
8. G. A. Kildall. A unified approach to global program optimization. In *Proc. ACM Symp. Principles of Programming Languages*, pages 194-206, 1973.
9. X. Leroy. Java Bytecode Verification: An Overview. In G. Berry, H. Comon, and A. Finkel, editors. *CAV 2001*, *LNCS*, pages 265-285. Springer-Verlag, 2001.
10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
11. T. Nipkow. Verified Bytecode Verifiers. In *Foundations of Software Science and Computation Structure (FOSSACS'01)*, pages 347–363, Springer-Verlag, 2001.
12. T. Nipkow and D. v. Oheimb. Java$_{ligth}$ is type-safe – definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161-170, 1998.

13. T. Nipkow, D. v. Oheimb, C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 117-144. IOS Press, 2000.
14. J. Posegga and H. Vogt. Java bytecode verification using model checking. In *Workshop Fundamental Underspinnings of Java*, 1998.
15. 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 1597 of *LNCS*, pages 89-103. Springer-Verlag, 1999.
16. D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL'98*, pages 38-48. ACM Press 1998.
17. Z. Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 271-311. Springer-Verlag, 1999.
18. Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Programming Languages and Systems*, 22(4):638-672, 2000.
19. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc 25th ACM Symp. Principles of Programming Languages*, pages 149-161. ACM Press, 1998.