CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2001; **0**:1–19 *Prepared using* cpeauth.cls *[Version: 2000/05/12 v2.0]*

Verified lightweight bytecode verification

Gerwin Klein and Tobias Nipkow

Technische Universität München
Department of Informatics
80290 Munich, Germany
Email: {kleing,nipkow}@in.tum.de

SUMMARY

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

KEY WORDS: Java, Bytecode Verification, Theorem Proving, Data Flow Analysis

1. Introduction

The Java Virtual Machine (*JVM*) comprises a typed assembly language, an abstract machine for executing it, and the so-called *Bytecode Verifier* (*BV*) for checking the welltypedness of JVM programs. Resource-bounded JVM implementations on smart cards do not provide bytecode verification because of the relatively high space and time consumption. They either do not allow dynamic loading of JVM code at all or rely on cryptographic methods to ensure that bytecode verification has taken place off-card. In order to allow on-card verification, Eva and Kristoffer Rose [21] proposed a (sparse) annotation of JVM code with types to enable a one-pass verification of welltypedness. Roughly speaking, this transforms a type reconstruction problem into a type checking problem, which is easier. More precisely, the type inference problem is a data flow analysis problem that requires an iterative solution, whereas the type checking problem merely needs a single pass to check consistency of the type annotations with the code. Based on these ideas we have extended an existing formalization of the JVM in Isabelle/HOL [18, 12]. Isabelle [16] is a generic theorem prover that can be instantiated with different object logics, and Isabelle/HOL [14], simply typed higher order logic, is the most widely used of these object logics. We will first describe the general idea of bytecode verification and its formalization in Isabelle/HOL. After that we explain how lightweight bytecode verification works, how we formalized it and proved it correct and complete. The full formalization is available on the web [13].

1.1. Related work

Starting with the paper by Stata and Abadi [22], there is a growing body of literature [3, 4, 19, 20, 5, 15] that tries to come to grips with the subtleties of the BV, especially *subroutines* (a JVM specific concept distinct from methods) and *object initialization*. Probably the most complete formalization to date is by Freund [2]. All of these papers formalize the BV with the help of a customized type system. In addition there are some less orthodox approaches: Jones [6] and Yelland [26] reduce bytecode verification to type checking in Haskell, Posegga and Vogt [17] reduce it to model checking.

Our work builds on a related line of research, that of embedding the formalization of the JVM and the BV in a theorem prover. A first machine-checked specification of type checking for the JVM was given by Pusch [18]. Using Isabelle/HOL she connected the type checking rules with an operational semantics for the JVM by showing that execution of type correct programs is type sound, i.e. during run time each storage location contains values of the type employed during type checking. We start from a revised version of this work [12]. Recently, Nipkow [11] has also verified a data flow analysis implementation of bytecode verification against the above type checking rules. These machine-checked formalizations do not deal with some of the subtleties of the BV, namely exception handling, object initialization, and subroutines. We come back to this point in the conclusion.

More abstractly, lightweight bytecode verification is an instance of *proof carrying code* (PCC) [10], where the "proof" (of well-typedness) is the type annotation of the JVM code. Abstracting from the specifics of the JVM, this leads to the idea of *typed assemply languages* put forward by Morrisett *et al.* [8], who describe the compilation from λ-calculus to a typed variant of a conventional RISC assembly language.

2. The bytecode verifier

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

instruction		stack	local variables	
`Load 0`	`Some (`	`[],`	`[Class B, integer]`	`)`
`Store 1`	`Some (`	`[Class A],`	`[Class B, Err]`	`)`
`Load 0`	`Some (`	`[],`	`[Class B, Class A]`	`)`
`Getfield F A`	`Some (`	`[Class B],`	`[Class B, Class A]`	`)`
`Goto -3`	`Some (`	`[Class A],`	`[Class B, Class A]`	`)`

On the left the instructions are shown and on the right the type of the stack elements and the local variables. The type information attached to an instruction characterizes the state *before* execution of that instruction. We assume that class `B` is a subclass of `A` and that `A` has a field `F` of type `A`. The `Some` before each of the type entries means that we were able to predict some type for each of the instructions. If one of the instructions had been unreachable, the type entry would have been `None`.

Execution starts with an empty stack and the two local variables hold a reference to an object of class `B` and an integer. The first instruction loads local variable 0, a reference to a `B` object, on the stack. The type information associated with the following instruction may puzzle at first sight: it says that a reference to an `A` object is on the stack, and that usage of local variable 1 may produce an error. This means the type information has become less precise but is still correct: a `B` object is also an `A` object and an integer is now classified as unusable (`Err`). The reason for these more general types is that the predecessor of the `Store` instruction may have either been `Load 0` or `Goto -3`. Since there exist different execution paths to reach `Store`, the type information of the two paths has to be "merged". The type of the second local variable is either `integer` or `Class A`, which are incompatible, i.e. the only common supertype is `Err`.

Bytecode verification is the process of inferring the types on the right from the instruction sequence on the left and some initial condition, and of ensuring that each instruction receives arguments of the correct type. This can be done on a per method basis because each method has fixed argument and result types. The righthand side of the the table is called a *method type*, one line of the method type is called a *state type*. To simplify the presentation we restrict considerations in this paper to a single method. Thus type inference is the computation of a method type from an instruction sequence, while type checking means checking that a given method type fits an instruction sequence. Lightweight bytecode verification is in between: only (crucial) bits of the method type are given, the rest is computed, but this computation is performed in a single pass over the instruction sequence, i.e. in linear time. This is in contrast to full bytecode verification, which requires an iterative computation. In this paper we concentrate on type checking and lightweight bytecode verification.

Abstractly, lightweight bytecode verification can be seen as a combination of two principles:

- Result checking: instead of computing the method type, it is merely checked that the given method type fits.
- Trading space for time: it is sufficient to store only the state type for the entry point to each basic block (a code sequence with only one entry and exit point) because the remaining state types in that block can be computed in linear time.

The same principles can be applied to any data flow analysis problem.

We will now sketch some of the key ingredients of the type checking specification by Nipkow *et al.* [12] that our formalization builds on.

2.1. Order on types

Before we can proceed to the formalization of the bytecode verifier itself we first have to define what extactly we mean when we say a state type is less precise than another one.

In our Isabelle formalization state types are values of type

$$\texttt{(ty list} \times \texttt{ty err list) option}$$

That means a state type is a datatype `option` with elements that are tuples of `ty list` (the stack) and `ty err list` (local variables). The `option` datatype over an element type `'a` is declared as:

```
datatype 'a option = None | Some 'a
```

The Isabelle notation above defines a new datatype `option`, very similar to datatypes in functional programming. It has two constructors, namely `None` with no arguments and `Some` with one argument. The datatype is polymorphic: it has a type variable `'a` which in this case is used for the type of the argument of `Some`.

We use `option` here with the meaning already mentioned in the example: instructions with state type `None` are unreachable, `Some t` indicates a reachable instruction with type information `t`.

The stack part `ty list` of a state type is a list of usual Java types, e.g. reference types for classes or primitive types like `integer`. The local variables of a method are modeled by a list with `ty err` elements. Similar to `option`, the datatype `err` is used to extend a type `'a` by one element:

```
datatype 'a err = Err | Ok 'a
```

Here the additional element is `Err` and we use it to indicate that usage of a local variable with that type may produce an error at runtime. According to the JVM Specification [7] only local variables can contain such unusable entries, the stack cannot—hence the difference.

Having defined what a state type is in Isabelle, we will now move on to an order on these state types: the notion of "less precise" or "more general" is basically the same as the supertype relation in Java. The expression `G ⊢ Class B ≼ Class A` means: class `A` is more general than class `B` in a declaration context `G` iff `A` is a superclass of `B` in that context. The relation is reflexive, transitive, and antisymmetric. Primitive types like `integer` can only be compared with themselves.

When two types have to be merged in the bytecode verifier, the result is their least common supertype. With just the usual Java types, this least common supertype may not always exist, e.g. for `integer` and some class. As in the example above we would have a situation where we cannot predict which type the entry will actually have. Because JVM instructions are monomorphic, i.e. will take either an `integer` or a reference type, but not both, we need to express that the value of this entry cannot be used. Again as already mentioned in the example, we mark unusable entries by giving them type `Err`. When we say that `Err` is the least common supertype of any two incompatible types, we say in other words that `Err` is the most general type we can assign to an entry. If we want to lift the supertype relation from Java types `ty` to the new type system with the `Err` element `ty err`, we have to treat `Err` as top element:

```
G ⊢ a' <=o a ≡ case a of
                  Err  ⇒ True
                | Ok t ⇒ (case a' of Err ⇒ False | Ok t' ⇒ G ⊢ t' ≼ t)
```

The ≡ sign means "equal by definition", whereas the = sign means usual equality on an arbitrary type (on booleans = therefore means "if and only if").

The step to lists with elements of type `ty err` is easy: when we want to compare two lists we compare them componentwise. We write this order on `ty err list` as `<=l`.

The next step is tuples: a further relation `<=s` compares a pair of stack entries and local variables. Local variables have exactly type `ty err list` so we can use our list order `<=l` on them unchanged. The stack on the other hand only has type `ty list`. In order to reuse `<=l` we lift all stack entries from `ty` to `ty err` by explicitly stating that they are usable:

```
G ⊢ (s,l) <=s (s',l') ≡ G ⊢ map Ok s <=l map Ok s' ∧ G ⊢ l <=l l'
```

With that we can finally define an order `<='` on state types. We only need to lift `<=s` to the option datatype. As with `err` we have one additional element. This time it is `None`, used to indicate that an instruction is not reachable. Contrary to the `err` case we must now treat the additional element as bottom element, not as top element. The motivation for that will become clearer when we take a closer look at the bytecode verifier itself in the next section.

```
G ⊢ s' <=' s ≡
      case s' of
        None    ⇒ True
      | Some t' ⇒ (case s of None ⇒ False | Some t ⇒ G ⊢ t' <=s t)
```

2.2. Type checking

In this paper we only treat the type checking part of the bytecode verifier. We model type checking as a predicate that determines if a method is welltyped with respect to a given method type:

```
wt_method G C pTs rT mxl ins phi ≡
let max_pc = length ins in
  0 < max_pc ∧ wt_start G C pTs mxl phi ∧
  (∀pc. pc < max_pc ⟶ wt_instr (ins!pc) G rT phi max_pc pc)
```

The predicate `wt_method` is parameterized by a declaration context `G`, the class `C` in which the method is declared, a list `pTs` of the method's parameter types, the return type `rT` declared for the method, the number of declared local variables `mxl`, the method body `ins` (a list of instructions), and finally the method type `phi` (a list of state types). With respect to these parameters a method is welltyped iff:

- the method contains at least one instruction
- the method type satisfies some start condition `wt_start`, and
- each instruction in the method is welltyped with respect to a predicate `wt_instr`. The `!` is the Isabelle operator that yields the nth element of a list. In the condition `pc < max_pc` as throughout the rest of this paper `pc` is a natural number, so we don't need an additonal `0 ≤ pc`.

The start condition may look a bit complicated at first sight:

```
wt_start G C pTs mxl phi ≡
  G ⊢ Some ([], Ok(Class C)#(map Ok pTs)@replicate mxl Err) <=' phi!0
```

The operator `#` is Isabelle's cons for lists, `@` appends two lists, and `replicate n x` produces a list with `n` entries all having the value `x`. The predicate `wt_start` ensures that the state type of instruction 0 correctly approximates a certain intitial state. It is the initial state of stack and local variables directly after invocation of the method:

- The first instruction is reachable, marked by `Some` at the beginning of the expression.
- The first component of the pair is the empty list. That means the operand stack must be empty.
- The first local variable contains the `this` pointer. It is a reference to the class `C` the method belongs to. The preceding `Ok` marks the value explicitly as usable.
- The next entries in the local variables are the parameter types of the method `pTs`. Again, with `map Ok pTs` we mark each parameter explicitly as usable.
- Finally, the declared local variables are treated: as many entries as there are declared local variables in the method are marked with `Err` as not yet initialized.

The only thing missing now is the definition of the welltypedness conditions for single instructions:

```
wt_instr i G rT phi mpc pc ≡
app i G rT (phi!pc) ∧
(∀pc′ ∈ set (succs i pc). pc′ < mpc ∧ G ⊢ step i G (phi!pc) <=′ phi!pc′)
```

Most of the work is again delegated, this time to three functions: `app` for applicability conditions, `succs` for a list of the program counters of successor instructions, and `step` for the effect the instruction will have on the state type when executed. With these functions we have: a single instruction is welltyped iff the instruction is applicable in the current state type, the program counter of each successor instruction lies within the method, and if again for each successor `pc′` the state type at `pc′` is more general than the state type we get when we execute the instruction in the current state type (`set` converts a list into a set).

With this definition we can model the bytecode verifier independently of the actual instruction set as long as we have functions `app`, `succs`, and `step` describing the poperties of instructions the bytecode verifier is interested in.

If we further demand that

```
app i G rT None = True
app i G rT (Some s) ⟶ (∃s′. step i G (Some s) = Some s′)
step i G None = None
```

we obtain from our definition of `None` as bottom element of `<=′` the following two properties for unreachable code: all instructions reachable from instruction 0 must contain entries of the form `Some s`, and no unreachable instruction can influence the welltypedness of the rest, because `G ⊢ None <=′ phi!pc′` will always yield true.

Figure 1 shows the definition of `step` for the μJava instruction set. Figure 2 shows the applicability conditions.

The definition of `step` in figure 1 builds on a `step′` and directly satisfies the conditions above. The equations for `step′` are written in pattern matching style as used in functional programming. The function is only well defined for state types matching the pattern on the left-hand side, and there are only equations for functions that have at least one successor instruction in the same method, i.e. `Return` is missing. This is sound since we know that `step` will only be needed when the list of successors is not empty, and when the instruction is applicable. The applicability conditions will ensure that the state type matches the pattern given in the definition of `step`. The proof that `step` and `app` indeed have something to do with the semantics of the μJava instruction set, and that they work together as described, is a crucial part of the type safety proof of the bytecode verifier. A description of that proof for an earlier version of the bytecode verfier can be found in [18], the detailed Isabelle proof for the current version is available from our web page [13].

Figure 1 uses some notation and functions not yet mentioned: in the first equation we encounter `val`. It is defined by `val (Ok x) = x`. The expression `LT[idx := Ok ts]` is a list update. The entry at position `idx` in the list `LT` gets the new value `Ok ts`. The value `NT` in the equation for `Aconst_null` is the Java null type, the type of the value `null`. It is approximated by any other reference type. As expected, `fst` and `snd` are selector functions returning the first resp. the second component of a pair. Function `the` is the destructor for the option datatype. It is defined by `the (Some x) = x` and (like `val`) it is underspecified, i.e. the case `the None` is left out. As a consequence one can only prove

```
step' (Load idx,  G, (ST,LT))           = (val (LT!idx)#ST,LT)
step' (Store idx, G, (ts#ST,LT))        = (ST, LT[idx:= Ok ts])
step' (Bipush i, G, (ST,LT))            = (integer#ST,LT)
step' (Aconst_null, G, (ST,LT))         = (NT#ST,LT)
step' (Getfield F C, G, (oT#ST,LT))     = (snd (the (field (G,C) F))#ST,LT)
step' (Putfield F C, G, (vT#oT#ST,LT))  = (ST,LT)
step' (New C, G, (ST,LT))               = (Class C#ST,LT)
step' (Checkcast C, G, (RefT rt#ST,LT)) = (Class C#ST,LT)
step' (Pop, G, (ts#ST,LT))              = (ST,LT)
step' (Dup, G, (ts#ST,LT))              = (ts#ts#ST,LT)
step' (Dup_x1, G, (ts1#ts2#ST,LT))      = (ts1#ts2#ts1#ST,LT)
step' (Dup_x2, G, (ts1#ts2#ts3#ST,LT))  = (ts1#ts2#ts3#ts1#ST,LT)
step' (Swap, G, (ts1#ts2#ST,LT))        = (ts2#ts1#ST,LT)
step' (IAdd, G, (integer#integer#ST,LT)) = (integer#ST,LT)
step' (Ifcmpeq b, G, (ts1#ts2#ST,LT))   = (ST,LT)
step' (Goto b, G, s)                    = s
step' (Invoke C mn fpTs, G, (ST,LT))    = (let ST' = drop (length fpTs) ST
    in  (fst (snd (the (method (G,C) (mn,fpTs))))#(tl ST'),LT))

step i G (Some s) = Some (step' (i,G,s))
step i G None     = None
```

Figure 1. Effect of instructions on a state type

interesting properties about `the` if the argument is known to be of the form `Some x`. Again as in some functional programming languages, `take` and `drop` return the first `n` elements of a list, or all but the first `n` elements. Finally there are two μJava specific lookup functions `field` and `method`. The first one yields the declared name and type of a class field, the second one gives full declaration information for methods. Both respect the structure of the class hierarchy, inheritance and visibility of names. In μJava we have not modeled access modifiers like `public` and `private`, though. The rather large `step'` equation for `Invoke` merely takes the method's parameters and the reference on which the method is invoked from the stack, and then puts the type of the return value on it in turn.

Figure 2 looks even more involved. The definition of `app'` makes use of a default equation at the bottom stating that the instruction is by default not applicable when no other equation matches. For `app'` we need some new notation, too. The predicate `is_class G C` unsurprisingly tests wether the name `C` is declared as a class in the program `G`. In the `Ifcmpeq` equation, `PrimT` and `RefT` are type constructors for primitive types and reference types in μJava. As with `step'`, the `Invoke` instruction is the largest. In its equation we use a function `rev` returning a list in reverse order, and the function `zip` that converts a pair of lists into a list of pairs. The equation states that the stack must at least contain the parameters for the method call and reference `T` on which to invoke the method. Of course, this `T` should be compatible with the class the `Invoke` instruction expects. If we want to invoke a method, we also have to look up if a method with the given name and signature exists in the program. Since

```
app' (Load idx, G, rT, (ST,LT))            = (idx < length LT ∧ LT!idx ≠ Err)
app' (Store idx, G, rT, (ts#ST, LT))       = (idx < length LT)
app' (Bipush i, G, rT, s)                  = True
app' (Aconst_null, G, rT, s)               = True
app' (Getfield F C, G, rT, (oT#ST, LT))    = (∃vT. field (G,C) F = Some (C,vT) ∧
                                              G ⊢ oT ≼ Class C ∧ is_class G C)
app' (Putfield F C, G, rT, (vT#oT#ST, LT)) = (∃vT'. field (G,C) F = Some (C,vT') ∧
                                              G ⊢ oT ≼ Class C ∧ G ⊢ vT ≼ vT' ∧
                                              is_class G C)
app' (New C, G, rT, s)                     = (is_class G C)
app' (Checkcast C, G, rT, (RefT rt#ST,LT)) = (is_class G C)
app' (Pop, G, rT, (ts#ST,LT))              = True
app' (Dup, G, rT, (ts#ST,LT))              = True
app' (Dup_x1, G, rT, (ts1#ts2#ST,LT))      = True
app' (Dup_x2, G, rT, (ts1#ts2#ts3#ST,LT))  = True
app' (Swap, G, rT, (ts1#ts2#ST,LT))        = True
app' (IAdd, G, rT, (integer#integer#ST,LT)) = True
app' (Ifcmpeq b, G, rT, (ts#ts'#ST,LT))    = ((∃p. ts = PrimT p ∧ ts' = PrimT p) ∨
                                              (∃r r'. ts = RefT r ∧ ts' = RefT r'))
app' (Goto b, G, rT, s)                    = True
app' (Return, G, rT, (T#ST,LT))            = (G ⊢ T ≼ rT)
app' (Invoke C mn fpTs, G, rT, (ST,LT))    = (length fpTs < length ST ∧
                                      (let apTs = rev (take (length fpTs) ST);
                                           T    = hd (drop (length fpTs) ST)
                                       in
                                         G ⊢ T ≼ Class C ∧
                                         method (G,C) (mn,fpTs) ≠ None ∧
                                         (∀(a,f)∈set (zip apTs fpTs). G ⊢ a ≼ f)))

app' (i,G,rT,s) = False

app i G rT s ≡ case s of None ⇒ True | Some t ⇒ app' (i,G,rT,t)
```

Figure 2. Applicability conditions

`method` already handles inheritance and visibility, it is enough to check if the lookup is successfull. The `zip` expression then checks if the actual parameters on the stack are type compatible with the expected formal parameters from the instruction.

3. The lightweight bytecode verifier

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

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

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

instruction	certificate
`Load 0`	`None`
`Store 1`	`Some ( [Class A], [Class B, Err] )`
`Load 0`	`None`
`Getfield F A`	`None`
`Goto -3`	`None`

At this point we use the option datatype with a different meaning: `None` indicates that the certificate contains no entry at this point. `Some` means that the certificate stores a state type of a reachable instruction.

From the certificate the whole method type is reconstructed in a single linear pass: The state type `Some ([], [Class B, integer])` for the `Load` instruction will be filled in as initialization. The state type for `Store 1` is in the certificate, since `Store` is the target of the `Goto -3` jump. The lightweight bytecode verifier calculates the effect of `Load 0`, i.e. `Some ([Class B], [Class B, integer])`, and checks if the certificate `Some ([Class A], [Class B, unusable])` correctly approximates this result. The types before execution of the next instructions `Load`, `Getfield` and `Goto` are then easily calculated from current state type and the effect of the instructions alone. We arrive at `Goto` with a current state type `Some ([Class A], [Class B, Class A])`. The lightweight bytecode verifier now checks if the calculated state type is correctly approximated by the jump target.

We didn't store the state type of the target, but since it is a jump target, we have an entry in the certificate at that point: we only need to check if the entry correctly approximates our calculated state type.

Note that all execution paths joining at `Store 1` were checked, but no iteration or additional memory was required.

In the terminology of data flow analysis (see e.g. [9]) the certificate records the type information at the entry points to so called *basic blocks* (and potentially additional points). This is completely standard in (global) data flow analysis where basic blocks are viewed as atomic (hence their name), and their local structure is immaterial. What is more, this view has significant advantages not just for lightweight but also for standard bytecode verification: during the iterative computation of the method type it is sufficient to store those state types that correspond to entry points of basic blocks. This is a significant reduction in space at practically no additional cost in time.

3.1. Formalization

With that kind of process and certificate in mind, we can start a formalization of the lightweight bytecode verifier. We have two goals here: on the one hand, we want the formalization to be abstract and as easy to understand as possible. On the other hand, we now not only want to model type checking, but also the simplified form of type reconstruction, i.e. we want functions, not predicates. The lightweight bytecode verifier is a functional program with a structure very similar to the predicates presented for the bytecode verifier above. We have one layer for methods, one for lists of instructions (corresponding to the ∀ quantifier in `wt_method`), and one for single instructions. The instruction layer is divided in one part for certificate checking and one part for the actual type checking.

We begin at the bottom layer with the lightweight type checking function for single instructions. Because it is easier to read and also shorter, we don't show the complete functional definition for this level, but proved the following equivalence instead:

```
(wtl_inst i G rT s cert max_pc pc = Ok s') =
 (app i G rT s ∧
  (∀pc' ∈ set (succs i pc). pc' < max_pc) ∧
  (∀pc' ∈ set (succs i pc). pc' ≠ pc+1 ⟶ G ⊢ step i G s <=' cert!pc') ∧
  s' = (if pc+1 ∈ set (succs i pc) then step i G s else cert!(pc+1)))
```

This function `wtl_inst` corresponds closely to the predicate `wt_instr` from the traditional bytecode verifier. It takes as arguments an instruction `i`, a declaration context `G`, the return type, the current state type `s`, the certificate, the maximal program counter, and the current program counter `pc`. It yields `Ok s'` where `s'` is the state type at `pc+1` when the instruction is welltyped, and `Err` when it is not. The big conjunction on the right-hand side decomposes into four parts:

- as with the traditional bytecode verifier, `wtl_inst` requires the instruction to be applicable. This is modeled by the term `app i G rT s`.
- again as in the traditional case, `∀pc'∈set (succs i pc). pc' < max_pc` ensures that all successor program counters lie within the method.
- the rest is a bit different: since there is no global method type available any more, we can only check jump targets with the certificate. Jump targets are all successors of the instruction apart from `pc+1`. So `G ⊢ step i G s <=' cert!pc'` is only tested for `pc' ≠ pc+1`.

- to calculate the state type at `pc+1` we again execute the instruction on the current state type. The problem is: that can only yield a valid result if `pc+1` is among the successors of the current instruction. If it is not, e.g. if the current instruction is a `Return` or `Goto`, we have to think of something different: if the instruction at `pc+1` is ever executed, it must have been the target of a jump. In this case we will have the information we need in the certificate and can proceed with that. If the instruction at `pc+1` is unreachable, the corresponding state type should be `None`, which is exactly what the certificate will contain in this case, too.

In `wtl_inst` we have covered almost everything we have done in the example above. There is still a difference, though: when "executing" `Store 1` in the example we did not start from the state type calculated from the `Load` before, but we used the value of the certificate instead. We also checked if the certificate correctly approximated the current state type. We achieve this behaviour in the formalization by another predicate:

```
wtl_cert i G rT s cert max_pc pc ≡
case cert!pc of
  None    ⇒ wtl_inst i G rT s cert max_pc pc
| Some s’ ⇒ if G ⊢ s <=’ (Some s’) then
                wtl_inst i G rT (Some s’) cert max_pc pc
             else Err
```

In `wtl_cert`, we first check for a current state type `s` if there is something stored in the certificate. If there is nothing, we just proceed as we would have. If there is something stored however, we first compare with the current state type, and then use the stored value instead. If the check fails, the instruction is rejected as not welltyped.

We can now write a function that calculates the state type reached after a whole list of instructions:

```
wtl_inst_list []     G rT cert max_pc pc s = Ok s
wtl_inst_list (i#is) G rT cert max_pc pc s =
                    (let s’ = wtl_cert i G rT s cert max_pc pc in
                     strict (wtl_inst_list is G rT cert max_pc (pc+1)) s’)
```

The function takes the same arguments as `wtl_inst` but now works on a list of instructions instead. An empty list of instructions does not change the state type at all. In the cons case we calculate the effect of the first instruction and pass the result on to the rest of the lift. With `strict` we can lift a function from type `’a ⇒ ’b err` to `’a err ⇒ ’b err` in the canonical way:

```
strict f x ≡ case x of Err ⇒ Err | Ok v ⇒ f v
```

Using `wtl_inst_list` it is easy to express welltypedness of a method:

```
wtl_method G C pTs rT mxl ins cert ≡
let max_pc = length ins;
    start  = (Some ([], Ok(Class C)#(map Ok pTs)@(replicate mxl Err)))
in
  0 < max_pc ∧ wtl_inst_list ins G rT cert max_pc 0 start ≠ Err
```

The arguments are the same as with `wt_method` but we have now only the certificate instead of the whole method type. The state type fed to `wtl_inst` corresponds to the situation at method invocation time as in wt_start. For welltypedness of the method we only have to demand that the result is not an error, but a legal state type.

The formalization of the lightweight bytecode verifier is again independent of the actual instruction set and builds on the same functions app, step, and succs as the traditional bytecode verifier. The main funtion is a simple linear sweep through the instruction list, and the functions for single instructions get no global type information apart from the certificate.

3.2. Soundness

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

```
wtl_method G C pTs rT mxl ins cert ⟶
(∃phi. wt_method G C pTs rT mxl ins phi)
```

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

We will now sketch the outline of the soundness proof. The detailed, machine checked Isabelle/Isar proof document is available from our website [13]. Isabelle/Isar [25] is a generic way to write Isabelle proofs in a more human readable form (as opposed to tactic scripts). The hope is that such Isabelle/Isar proof documents give the reader more insight why a property holds, and not only with which sequence of commands the prover can establish it.

We prove the soundness theorem by first describing what the method type phi should look like. To do that we can take into account all information from a successful run of the lightweight bytecode verifier. Then we show that such a phi always exists and that it satisfies wt_method.

It could be the case that there is more than one welltyping phi for any given method. For soundness we only need to show the existence of one of them. We pick a phi with the following properties:

- if the certificate contains a state type s at some point pc, phi contains that s at the same point.
- otherwise, if the lightweight bytecode verifier has processed the first pc instructions and has calculated a current state type s, phi will contain that s at position pc.

The predicate fits captures that notion:

```
fits phi is G rT s cert ≡
∀pc s'.
  pc < length is ⟶
  wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s' ⟶
  phi!pc = (case cert!pc of None ⇒ s' | Some t ⇒ cert!pc)
```

It is obvious, that such a phi always exists. Given a function calculating the entries of phi (essentially just fits written as function), Isabelle proves automatically:

```
∃phi. fits phi is G rT s cert
```

It remains to show that this `phi` satisfies `wt_method`. When we look back at the definition of `wt_method`, we find it consists mainly of the demand that `wt_instr` should hold for all instructions in the method. So we prove first:

```
wtl_inst_list is G rT cert (length is) 0 s ≠ Err ∧
fits phi is G rT s cert ⟶
(∀pc. pc < length is ⟶ wt_instr (is!pc) G rT phi (length is) pc)
```

If we have that and combine it with the fact that `wt_start` also holds (because we use the exact same term as start in `wtl_method`) we finally get that for every certificate there is a method type such that `wt_method` holds and have proved our soundness result.

The fact that `wt_instr` holds for all instructions is the main statement of our soundness proof. Figure 3 shows how this property may be established in Isabelle (the whole soundness proof is about 600 lines). We will not explain all elements of the Isar proof language we used in full formal detail (we refer the reader to [25] for that). The hope is however that the Isar language is intuitive enough to follow the chain of reasoning without being an expert on the proof tool. We will go through the proof step by step and explain the main points informally.

Figure 3 begins by stating the property to be proved as Isabelle inference rule. Then the proof starts off by assuming that all the premises hold, and by labeling them for later usage with `fits`, `pc`, and `wtl` respectively. Since our goal is to show that `wt_instr` holds, we would like to start from `wtl_inst` for the same instruction and the same state type. The assumptions on the other hand only contain the successful run on the whole method, so the next few proof steps will first establish that `wtl_inst` must have worked successfully.

From the assumptions `pc` and `fits` we can obtain state types `s'` and `s''` such that partial execution of `wtl_inst_list` up to the instruction at position `pc` yields `Ok s'` and `wtl_cert` for the instruction itself yields `Ok s''`. This must hold since the the bytecode verifier couldn't have run successfully otherwise. The command `by` tells Isabelle to prove this by applying the lemma `wtl_all` (which states the possibility of partial execution of `wtl_inst_list` in general) and the proof method `auto`. We again label the two properties for later usage.

From our assumptions `fits`, `pc` and `wtl` we now get that `phi` contains the same value as the certificate if it has the form `Some t` or, if the certificate is `None`, that `phi` contains the state type `s'` we just obtained from the partial execution. The property `c_Some` talks not only about the current position `pc` but about all positions in the method since we will need it later for a `pc'` other than `pc`. Both facts follow almost directly from the definition of `fits`.

Now we are able to show that `wtl_inst` does not return an error for the same instruction and state type we require for `wt_instr`. This property `wti` follows mainly from `wtl_cert` and the things we learned about the relationship between `phi` and `cert`. Isabelle can establish it by unfolding the definition of `wtl_cert` and by cases analysis on the `option` type and `if` expression contained in there.

If we take a look at the definition of the property we set out to prove, i.e. `wt_instr`, we see that it mainly consists of the demand that `G ⊢ step (is!pc) G (phi!pc) <=' phi!pc'` should hold for all successors `pc'`. The rest, i.e. applicability and that `pc'` should lie within the method, is neither difficult nor very interesting. Isabelle will establish it automatically at the end of the proof.

Let us concentrate on the more interesting part instead: we take an arbitrary but fixed `pc'` with `pc' ∈ set (succs (is!pc) pc)` (assumption `pc'` in figure 3). For this successor, which must

```
theorem ⟦ wtl_inst_list is G rT cert (length is) 0 s ≠ Err;  pc < length is;
         fits phi is G rT s cert ⟧ ⟹ wt_instr (is!pc) G rT phi (length is) pc
proof -
  assume fits: fits phi is G rT s cert
  assume pc: pc < length is and wtl: wtl_inst_list is G rT cert (length is) 0 s ≠ Err
  then obtain s′ s′′ where
    w: wtl_inst_list (take pc is) G rT cert (length is) 0 s = Ok s′ and
    c: wtl_cert (is!pc) G rT s′ cert (length is) pc = Ok s′′
    by - (drule wtl_all, auto)
  from fits wtl pc
  have c_Some: ∀t pc. pc < length is ∧ cert!pc = Some t ⟶ phi!pc = Some t
    by (auto dest: fits_lemma1)
  from fits wtl pc
  have c_None: cert!pc = None ⟶ phi!pc = s′ by (auto dest!: fitsD_None)
  from pc c c_None c_Some
  have wti: wtl_inst (is!pc) G rT (phi!pc) cert (length is) pc = Ok s′′
    by (auto simp add: wtl_cert_def split: if_splits option.splits)
  { fix pc′ assume pc′: pc′ ∈ set (succs (is!pc) pc)
    with wti have less: pc′ < length is by (simp add: wtl_inst_Ok)
    have G ⊢ step (is!pc) G (phi!pc) <=′ phi ! pc′
    proof (cases pc′ = pc+1)
      case False with wti pc′
      have G: G ⊢ step (is!pc) G (phi!pc) <=′ cert!pc′  by (simp add: wtl_inst_Ok)
      hence cert!pc′ = None ⟶ step (is!pc) G (phi!pc) = None by auto
      hence cert!pc′ = None ⟶ ?thesis by auto
      moreover
      { fix t assume cert!pc′ = Some t
        with less have phi!pc′ = cert!pc′ by (simp add: c_Some)
        with G have ?thesis by simp }
      ultimately show ?thesis by blast
    next
      case True with pc′ wti
      have step (is!pc) G (phi!pc) = s′′ by (simp add: wtl_inst_Ok)
      also from w c fits pc wtl
      have pc+1 < length is ⟶ G ⊢ s′′ <=′ phi!(pc+1) by (auto intro: wtl_suc_pc)
      with True less have G ⊢ s′′ <=′ phi!pc′ by auto
      finally show ?thesis .
    qed }
  with wti show ?thesis by (auto simp add: wtl_inst_Ok wt_instr_def)
qed
```

Figure 3. Part of the soundness proof in Isabelle/Isar notation

Prepared using cpeauth.cls

of course also lie within the method (fact `less` in figure 3 where `wtl_inst_Ok` is the equation for `wtl_inst` from section 3.1), we show that `G ⊢ step (is!pc) G (phi!pc) <=' phi!pc'` holds. The proof is by case distinction on the term `pc' = pc+1`.

- Looking at the definition of `wtl_inst` we see that the case `pc' ≠ pc+1`, i.e. when `pc'` is a jump target, is covered by the certificate: `G ⊢ step (is!pc) G (phi!pc) <=' cert!pc'`. If the certificate contains `None`, the result of `step (is!pc) G (phi!pc)` must also have been `None`, because `None` is the bottom element of the order `<='`. For the same reason, this `step (is!pc) G (phi!pc)` must then be smaller than `phi!pc'` which is what we wanted to show. "What we wanted to show" is abbreviated by the schematic variable `?thesis` in figure 3. It binds to the nearest enclosing property to be proved. Since the case `cert!pc' = None` is now taken care of, we can direct our attention to a `cert` of the form `Some t`: we know from the fact `c_Some` established above that it must be equal to `phi!pc'`. Hence we again arrive at `G ⊢ step (is!pc) G (phi!pc) <=' phi!pc'`. Both statements together conclude the case `pc' ≠ pc+1`.
- The second case is `pc' = pc+1`. As a first step we notice that `step (is!pc) G (phi!pc)` is just the result we obtained for `wtl_cert` in the very beginning. The next step establishes that this `s''` is smaller than `phi!(pc+1)`. In order to fit the Isabelle proof on a single page we moved this into a lemma `wtl_suc_pc` that is not shown here. It states that the result of `wtl_cert` at instruction `pc` will always be smaller than `phi` at position `pc+1`.
  The proof of the lemma again works by case distinction on the certificate, this time at position `pc+1`: if it is `None`, `fits` says that `phi!(pc+1)` is the same as `s''` and the conjecture becomes trivial because `<='` is reflexive. If the certificate contains a value, then again it is equal to `phi!(pc+1)`, and `wtl_cert` for the instruction at `pc+1` will ensure that our goal `G ⊢ s'' <=s phi!(pc+1)` holds. We relied on `wtl_cert` for the instruction at `pc+1`: such an instruction will always exists, because we have that `pc' = pc+1` is a legal successor of our current instruction.
  Combining `step (is!pc) G (phi!pc) = s''` and the result `G ⊢ s'' <=' phi!pc'` of the lemma, we arrive in one trivial step at our goal and have concluded the case `pc = pc+1`.

We now have established

```
∀pc'∈set (succs (is!pc) pc). G ⊢ step (is!pc) G (phi!pc) <=' phi!pc'
```

By unfolding the definitions of `wtl_inst` and `wt_instr`, Isabelle then automatically proves our main goal

```
wt_instr (is!pc) G rT phi (length is) pc
```

3.3. Completeness

Of course, the trivial bytecode verifier that rejects all programs also would be correct in the sense above. Therefore we show that our lightweight bytecode verifier also is complete, i.e. that if a program is welltyped with respect to the traditional bytecode verifier, the lightweight bytecode verifier will accept the same program with an easy to obtain certificate. What will this certificate look like? As in the example, we get the information we need from the method type of a successful run of the traditional bytecode verifier. Since we want to minimize the amount of information we have to provide, we do not take the whole method type as the certificate, but only the targets of jumps.

We will again only sketch the proof. See the web for details [13]. Before we prove completeness, we write a function `make_cert` that builds us a certificate from a method type `phi` and the corresponding instruction list. The certificate should contain the jump targets, so we define

```
is_target ins pc ≡
  ∃pc'. pc ≠ pc'+1 ∧ pc' < length ins ∧ pc ∈ set (succs (ins!pc') pc')

make_cert ins phi ≡
  map (λpc. if is_target ins pc then phi!pc else None) [0..length ins(]
```

The new bit of notation `[0..length ins(]` is the list of natural numbers from `0` to `length ins-1`. The function goes through a list `ins` of instructions and looks at each position if the current instruction is a jump target. If it is, the certificate gets the value of `phi`, if it is not, the certificate gets no entry.

Now the completeness theorem is:

```
wt_method G C pTs rT mxl ins phi ⟶
wtl_method G C pTs rT mxl ins (make_cert ins phi)
```

One of the key ingredients to the proof of that theorem is monotonicity of the functions `app` and `step`:

```
G ⊢ s <=' s' ∧ app i G rT s' ⟶ app i G rT s

succs i pc ≠ [] ∧ app i G rT s' ∧ G ⊢ s <=' s' ⟶
G ⊢ step i G s <=' step i G s'
```

For monotonicity of `step` we may take into account that the instruction has at least one successor and that the applicability conditions hold. Otherwise, a call of the `step` function doesn't make much sense anyway. We have proved these monotonicity theorems for the μJava instructions by case distinction over the instruction set. The proof is rather long (500 lines) and not very interesting. It contains mostly reasoning about the `<='` order. One of the most involved instructions is method invocation.

With a certificate `cert` as produced by `make_cert` above, this monotonicity carries over to `wtl_inst` and then `wtl_cert`:

```
wtl_cert i G rT s1 cert mpc pc = Ok s1' ∧ G ⊢ s2 <=' s1 ⟶
(∃s2'. wtl_cert i G rT s2 cert mpc pc = Ok s2' ∧ G ⊢ s2' <=' s1')
```

The other key ingredient is a relationship between `wt_instr` from the traditional bytecode verifier and `wtl_cert`: if `wt_instr` holds for an instruction `i` then `wtl_cert` will successfully return a valid state type `s` when fed with the same value `phi!pc` that `wt_instr` used. If we are not yet at the last instruction, this state type `s` will also satisfy `G ⊢ s <=' phi!(pc+1)`:

```
wt_instr i G rT phi mpc pc ⟶ wtl_cert i G rT (phi!pc) cert mpc pc ≠ Err

wt_instr i G rT phi mpc pc ∧
wtl_cert i G rT (phi!pc) cert mpc pc = Ok s ⟶  G ⊢ s <=' phi!(pc+1)
```

With these we can prove completeness by induction. Induction on the list of instructions does not work immediately, we have to strengthen the goal: we will show that, when the lightweight bytecode verifier has finished part of the job, i.e. has processed the first `n` instructions, it will always be able to finish successfully. Even that is not enough yet. We demand further that we will even be able to finish when we continue with any state type smaller than `phi` at the current position. So under the assumption that `wt_instr` holds for all instructions, and with a `cert` as described above, we prove the following statement by induction on the list of instructions `b` that has not yet been processed:

```
∀s. G ⊢ s <=′ phi!pc ⟶ wtl_inst_list b G rT cert (length ins) pc s ≠ Err
```

The base case, where `b` is the empty list of instructions, is trivial. In the cons case `b = i#b′` is composed of one instruction `i` and a rest list `b′`. That means the lightweight bytecode verifier has processed `n-1` instructions and we have to prove that it will be able to finish the rest of the method which contains now one more instruction. Our induction hypothesis is

```
∀s. G ⊢ s <=′ phi!(pc+1) ⟶
    wtl_inst_list b′ G rT cert (length ins) (pc+1) s ≠ Err
```

We have to show that `wtl_inst_list` gives a valid result for the list `b = i#b′` with a state type `s` satisfying `G ⊢ s <=′ phi!pc`.

To do that, we show that `wtl_cert` with `i` and `s` returns a state type `s′` smaller than `phi!(pc+1)`. We could then instantiate the induction hypothesis with this `s′`. From the fact that `i` is welltyped and that the rest list can continue with the calculated state type, we instantly have `wtl_inst_list` for the whole `b = i#b′`. Since we know that `wt_instr` holds for instruction `i`, we get that `wtl_cert` gives a valid result with `i` and `phi!pc`. We also know that this result `s′′` satisfies `G ⊢ s′′ <=′ phi!(pc+1)`. Monotonicity gives us that `wtl_cert` also has a valid result `s′` for `i` and `s`, because `G ⊢ s <=′ phi!pc`. Moreover, this `s′` satisfies `G ⊢ s′ <=′ s′′`, and since `<=′` is transitive also `G ⊢ s′ <=′ phi!(pc+1)`. Together with the induction hypothesis instantiated with `s′` we get that the cons case of the induction holds, too.

Now we have as corollary what we wanted in the first place: `wtl_inst_list` will produce no error for the whole instruction list when fed with the start term. This in turn directly implies that `wtl_method` holds.

4. Conclusion

We have presented our formalization of lightweight bytecode verification for μJava. It contains the lightweight bytecode verifier as an executable functional program for which we have proved soundness and completeness. Both theorems are with respect to our formalization of the traditional bytecode verifier, which has already been proved type safe. All proofs have been done with Isabelle/HOL, the theorems in this paper are directly generated from the Isabelle proof document.

The pure specification of the lightweight bytecode verifier alone is only about 50 lines of Isabelle definitions. The proofs of soundness and completeness however, together with all related lemmas, take up about 1500 lines of Isabelle/Isar text, and about 6 months of work for one person. The lightweight bytecode verifier builds on the existing specification of the whole μJava language. All JVM and BV related parts of μJava together consist of about 7000 lines of Isabelle definitions and proofs. This includes JVM operational semantics, type safety for the bytecode verifier, and an executable traditional bytecode verifier together with proof that it satisfies the specification presented in this paper. The source language of μJava is another 2500 lines, among which are μJava's operational semantics, wellformedness of the class hierarchy, lookup functions, and type safety of the source language. In total μJava is about 9500 lines of Isabelle code and generates about 160 pages of printed, human readable Isabelle/Isar proof document. All of μJava is publicly available as part of the official Isabelle distribtion.

In comparison to our formalization the approach of Eva and Kristoffer Rose [21] is a bit more general, but also a bit more complex. They only need the certificate when a type merge really produces a different type than expected, which leads to a smaller type annotation. It does however not save space during the verification pass, since the state type at all jump targets has to be saved for later checks anyway. Our completeness result on the other hand includes the simpler and easier to implement notion that the certificate should contain all jump targets.

The soundness theorem states that the lightweight bytecode verifier accepts only typecorrect programs, and that it is safe to rely on outside information. The completeness theorem states that the lightweight bytecode verifier will accept the same welltyped programs as the traditional bytecode verifier. Both theorems together give us that lightweight bytecode verification is functionally completely equivalent to traditional bytecode verification. The functional implementation shows that the algorithm is linear in time and constant in space. All these results together enable a secure scheme for on-card verification with Java smartcards: programs are annotated with a certificate, produced by a traditional bytecode verifier or directly by the compiler off-card. On-card verification can then take place with the efficient and compact lightweight bytecode verifier as part of the card's JVM. This scheme provides easy, seamless use for developers while maintaining all security properties from bytecode verification that we have become accustomed to. The major advantage over cryptographic methods is that no trust at all in the certifying party and authenticity of the certificate is needed.

Lightweight bytecode verification is already in industrial use as part of the KVM [23, 24], the virtual machine of the Java 2 Micro Edition. So how does the KVM deal with the features we have not dealt with here? Exception handlers are not really difficult but add clutter, which is the main reason why we have ignored them. Sun's specifications do not go into details either. The main change is that the verifier has to follow the transfer of control to exception handlers as well. Object initialization is trickier, and [24] is more explicit about it: one needs to introduce two further kinds of objects, newly created ones and partly initialized ones. Again, this should be straightforward to add to our formalization. The notorious subroutine problem [22] is solved by brute force: subroutines are inlined. This is another indication that JVM-style subroutines are more of a problem than a solution (see also [1]).

REFERENCES

1. Freund S. The costs and benefits of Java bytecode subroutines. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.
2. Freund S. *Type Systems for Object-Oriented Intermediate Languages*, PhD thesis, Stanford University, 2000.
3. Freund S, Mitchell J. A type system for object initialization in the Java bytecode language. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1998.
4. Freund S, Mitchell J. A formal framework for the Java bytecode language and verifier. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, 1999.
5. Hagiya M, Tozawa A. On a new method for dataflow analysis of Java virtual machine subroutines. In *Static Analysis (SAS'98)*, Levi G (ed.), *Lect. Notes in Comp. Sci.*, Springer, 1998; **1503**:17–32.
6. Jones M. The functions of Java bytecode. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.
7. Lindholm T, Yellin F. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

8. Morrisett G, Walker D, Crary K, Glew N. From system F to typed assembly language. In *Proc. 25th ACM Symp. Principles of Programming Languages*, ACM Press, 1998; 85–97.
9. Muchnick S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
10. Necula G. Proof-carrying code. In *Proc. 24th ACM Symp. Principles of Programming Languages*, ACM Press, 1997; 106–119.
11. Nipkow T. Verified bytecode verifiers. In *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, Honsell F (ed.), *Lect. Notes in Comp. Sci.*, Springer, 2001.
12. Nipkow T, von Oheimb D, Pusch C. μJava: Embedding a programming language in a theorem prover. In *Foundations of Secure Computation*, Bauer F, Steinbrüggen R (ed.), IOS Press, 2000; 117–144.
13. Nipkow T, von Oheimb D, Pusch C. Project Bali. http://isabelle.in.tum.de/Bali/
14. Nipkow T, Paulson L. *Isabelle/HOL. The Tutorial*. http://www.in.tum.de/ nipkow/pubs/tutorial.html [2001]
15. O'Callahn R. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th ACM Symp. Principles of Programming Languages*, ACM Press, 1999; 70–78.
16. Paulson L. *Isabelle: A Generic Theorem Prover*, Lect. Notes in Comp. Sci. vol. 828. Springer, 1994
17. Posegga J, Vogt H. Byte code verification for Java smart cards based on model checking. In *Computer Security — ESORICS 98*, Quisquater J, Deswarte Y, Meadows C, Gollmann D (ed.), *Lect. Notes in Comp. Sci.*, Springer, 1998; **1485**:175–190.
18. Pusch C. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99),* Cleaveland W (ed.), *Lect. Notes in Comp. Sci.*, Springer, 1999; **1579**:89–103.
19. Qian Z. A formal specification of Java virtual machine instructions for objects, methods and subroutines. In Formal Syntax and Semantics of Java, Alves-Foss J (ed.), *Lect. Notes in Comp. Sci.*, Springer, 1999; **1523**:271–311.
20. Qian Z. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Programming Languages and Systems*, Accepted for publication.
21. Rose E, Rose K. Lightweight bytecode verification. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.
22. Stata R, Abadi M. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*, ACM Press, 1998; 149–161.
23. Sun Microsystems. CLDC and the K Virtual Machine (KVM). http://java.sun.com/products/cldc/ [May 2000]
24. Sun Microsystems. Connected, limited device configuration. Specification version 1.0. http://java.sun.com/aboutJava/communityprocess/final/jsr030/ [May 2000]
25. Wenzel M. Isar - a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics, TPHOLs'99,* Bertot Y, Dowek G, Hirschowitz A, Paulin C, Thery L (ed.), *Lect. Notes in Comp. Sci.*, Springer, 1999; **1690**:167–183.
26. Yelland P. A compositional account of the Java virtual machine. In *Proc. 26th ACM Symp. Principles of Programming Languages*, ACM Press, 1999; 57–69.