Formalizing a JVML Verifier for Initialization in a Theorem Prover

Yves Bertot

INRIA Sophia Antipolis
2004 Route des Lucioles
06902 Sophia Antipolis Cedex, France
Yves.Bertot@inria.fr

1 Introduction

The byte-code verifier is advertised as a key component of the security and safety strategy for the Java language, making it possible to use and exchange Java programs without fearing too much damage due to erroneous programs or malignant program providers. As Java is likely to become one of the languages used to embed programs in all kinds of appliances or computer-based applications, it becomes important to verify that the claim of safety is justified.

We worked on a type system proposed in [7] to enforce a discipline for object initialization in the Java Virtual Machine Language and implemented it in the Coq [5] proof and specification language. We first produced mechanically checked proofs of the theorems in [7] and then we constructed a functional implementation of a byte-code verifier. We have a mechanical proof that this byte-code verifier only accepts programs that have a safe behavior with respect to initialization. Thanks to the extraction mechanism provided in Coq [17], we obtain a program in CAML that can be directly executed on sample programs.

A safe behavior with respect to initialization means that the fields of any object cannot be accessed before this object initialized. To represent this, the authors of [7] distinguish between uninitialized objects, created by a `new` instruction and initialized objects. Initialization is represented by an `init` instruction that replaces an uninitialized object with a new initialized object. Access to fields is represented abstractly by a `use` instruction, which operates only if the operand is an initialized object. Checking that initialization is properly respected means checking that `use` is never called with the main operand being an unitialized object.

There are two parts in this work. The first part simply consists in the mechanical verifications of the claims appearing in [7]. This relies on a comparison between operational semantics rules and typing rules. In terms of manpower involved, this only required around three weeks of work. This shows that proof tools are now powerful enough to be used to provide mechanical verifications of theoretical ideas in programming language semantics (especially when semantic descriptions are given as sets of inference rules).

The second part consists in producing a program, not described in [7], that satisfies the requirements described there. To develop this program, we have

G. Berry, H. Comon, and A. Finkel (Eds.): CAV 2001, LNCS 2102, pp. 14–24, 2001.

analyzed the various constraints that should be satisfied for each instruction and how these constraints could be implemented using a unification algorithm. In all, the experiments that the proof tool can also be used as a programming tool, with the advantage that logical reasoning can be performed on program units even before they are integrated in a completely functioning context.

This paper is a short abstract of a paper published as an INRIA research report under the title *A Coq formalization of a Type Checker for Object Initialization in the Java Virtual Machine* [2].

1.1 Related Work

Several teams around the world have been working on verifying formally that the properties of the Java language and its implementation suite make it a reasonably safe language. Some of the work done is based on pen-and-paper proofs that the principles of the language are correct, see for instance [22,6,14].

Closer to our concerns are the teams that use mechanical tools to verify the properties established about the formal descriptions of the language. A very active team in this field is the Bali team at University of Munich who is working on a comprehensive study of the Java language, its properties and its implementation [15,13,19] using the Isabelle proof system [18]. Other work has been done with the formal method B and the associated tools [3], at Kestrel Institute using Specware [8,20], or in Nijmegen [9,10] using both PVS [16] and Isabelle.

2 Formalizing the Language and Type System

2.1 Data-Types

The formalization we studied is based on a very abstract and simplified description of the Java Virtual Machine language. The various data-types manipulated in the programs are represented by abstract sets: `ADDR` for addresses in programs, `VAR` for variable names, `integer` for numeral values, `T` for classes.

The type of classes being left abstract, the way objects of a given class are constructed is also left abstract. We will actually rely on the minimal assumptions that there is a family of types representing the values in each class, such a family is represented by a function from `T` to the type of data-types, written in Coq as the following parameter to the specification.

```
Parameter object_value:T -> Set.
```

Since the objective of this study is initialization, there is a distinct family of sets for uninitialized object of a given class:

```
Parameter uninitialized_value: T -> Set.
```

We also assume the existence of test equality functions for uninitialized objects.

With all this, we express that the set of values manipulated by the abstract Java virtual machine is the disjoint sum of the set of integers, the set of object values and the set of uninitialized object values, with the following definition:

```
Inductive value: Set :=
     int_val: integer ->value
    | obj: (t:T) (object_value t) ->value
    | un: (t:T) (a:ADDR) (uninitialized_value t) ->value.
```

This inductive definition shows that a value can be in one of three cases, represented by three *constructors*. The first constructor, `int_val` expresses that a value can be an integer, the second constructor, `obj` expresses that a value can be an initialized object in some class `T`, the third constructor, `un`, expresses that a value can be an unitialized object for some class `T`, and that this value is also tagged with an address. Note that this definition uses a feature called *dependent types*: viewed as a function, the constructor `obj` takes a first argument `t` in `T` and a second argument whose type depends on the first one: this type must be `object_value T`.

The formal description of the Java Virtual Machine language as used in [7] boils down to a 10 constructor inductive type in the same manner. We named this type `jvmli`.

2.2 Operational Semantics

In [7], the operational semantics are given as a set of inference rules which describe the constraints that must hold between input and output data for each possible instruction. We handle judgments of the form

$$P \vdash \langle pc, f, s\rangle \rightarrow \langle pc', f', s'\rangle.$$

These judgments must be read as *for program P, one step of execution starting from the program counter pc, variable value description f, and stack s returns the new program counter pc', the new variable value description f', and stack s'*. Stacks are simply represented as finite lists of objects of type `value`. to represent memory, we use functions written f, f', from variable names (type `VAR`) to values (type `value`).

For instance, the language has an instruction `load` that fetches a value in memory and places it on the stack. This is expressed with this inference rule:

$$\frac{P[pc] = \texttt{load}\ x}{P \vdash \langle pc, f, s\rangle \rightarrow \langle pc+1, f, f[x]\cdot s\rangle}$$

Special attention must be paid to the way initialization works. Initializing an object means performing a side-effect on this object and all references to the object should view this side-effect. Thus, if several references of the same object have been copied in the memory then all these references should perceive the effect of initialization.

This is expressed with two rules. First the `new` instruction always creates an object that is different from all objects already present in memory (in this rule $\overline{A}^{\sigma}$ is a short notation for (`uninitialized_value` σ)) .

$$\frac{P[pc] = \texttt{new}\ \sigma \quad a \in \overline{A}^{\sigma} \quad Unused(a, f, s)}{P \vdash \langle pc, f, s\rangle \rightarrow \langle pc+1, f, a\cdot s\rangle}$$

Second, all occurrences (i.e., references) of an object in memory are modified when it is initialized (A^σ is a short notation for `object_value` σ, $[a'/a]f$ is the function that maps x to $f(x)$ if $f(x) \neq a$ and to a' if $f(x) = a'$, and $[a'/a]s$ is the same stack as s except that instances of a have been replaced with a').

$$\frac{P[pc] = \texttt{init}\ \sigma \quad a \in A^{\sigma,pc} \quad a' \in A^\sigma \quad Unused(a', f, s)}{P \vdash \langle pc, f, a \cdot s\rangle \rightarrow \langle pc + 1, [a'/a]f, [a'/a]s\rangle}$$

All these rules are easily expressed as constructors for inductive propositions, that are a common features in many modern proof tools.

2.3 Type System

In [7] the authors propose a set of typing rules for `jvmli` programs. This type system is based on the existence of a representation of all types for the stack and the variables at all lines in the program. It handles judgments of the form

$$F, S, i \vdash P$$

meaning *the type information for variables in F and for stacks in S is consistent with line i of program P*. The variable `F` actually represents a function over addresses, such that $F(i)$ is a function over variable names, associating variable names to types (we will write F_i instead of $F(i)$). Similarly S represents a function over addresses, where S_i is a stack of types. Consistency between type information with the program expresses that the relations between types of memory locations correspond to the actual instruction found at that address in the program. It also involves relations between types at line i at types at all lines where the control will be transfered after execution of this instruction.

For instance, the typing rule for `load` expresses that the type information at the line $i + 1$ must indicate that some data has been added on the stack when compared with the type information at line i.

$$\frac{\begin{array}{c} P[i] = \texttt{load}\ x \\ F_{i+1} = F_i \\ S_{i+1} = F_i(x) \cdot S_i \\ i + 1 \in Dom(P) \end{array}}{F, S, i \vdash P}$$

For `new` and `init` there are a few details that change with respect to the operational semantics. While the operational semantics required that the object added on top of the stack by `new` should be unused in the memory, the type system also requires that the type corresponding to this data should be unused. For `init`, the operational semantics requires that the new initialized value should be unused but this premise has no counterpart in the typing rule. Still the typing rule for `init` requires that all instances of the uninitialized type found on top of the stack should be replaced by an initialized type. In these rules we use σ to

denote the type of initialized objects and σ_i to denote the types of uninitialized objects created at address i.

$$\frac{\begin{array}{c}P[i] = \texttt{new}\ \sigma \\ F_{i+1} = F_i \\ S_{i+1} = \sigma_i \cdot S_i \\ \sigma_i \notin S_i \\ \forall x. F_i(x) \neq \sigma_i\end{array}}{F, S, i \vdash P} \qquad \frac{\begin{array}{c}P[i] = \texttt{init}\ \sigma \\ F_{i+1} = [\sigma/\sigma_j]F_i \\ S_i = \sigma_i \cdot \alpha \\ S_{i+1} = [\sigma/\sigma_j]\alpha\end{array}}{F, S, i \vdash P}$$

A singularity of this description is that all inference rules have the same conclusion, so that the proof procedures that usually handle these operational semantics descriptions [4] failed to be useful in this study.

3 Consistency of the Type System

The main theorem in [7] is a soundness theorem, saying that once we have proved that a program is well-typed this program will behave in a sound manner. Here this decomposes in a one-step soundness theorem: if a program P is well-typed at address i with respect to type information given by F and S, then executing this instruction from a state that is consistent with F and S at address i should return a new state that is also consistent with F and S at the address given by the new program counter.

This proof of soundness is pretty easy to perform, since the operational semantics rule and the typing rule are so close. However special attention must be paid to the problem of initialization because of the use of substitution. At the operational level, initialization works by substituting all instances of the uninitialized object with an initialized instance. At the type-system level, the same operation is performed, but how are we going to ensure that exactly the same location will be modified in both substitutions?

The solution to this problem is introduced in [7] under the form of a predicate `ConsistentInit` which basically expresses that whenever two locations have the same uninitialized type, then these locations contain the same value. In other terms, although there may be several values with the same uninitialized type, we can reason as if there was only one, because two different values will never occur at the same time in memory. This ensures that the substitutions in the operational rule for `init` and in the typing rule for `init` modify the memory in a consistent way.

The theorem of soundness is then expressed not only in terms of type consistency between the state and the type information at address i, but also in terms of the `ConsistentInit` property. We also have to prove that this `ConsistentInit` property is invariant through the execution of all instructions. Proving this invariant represents a large part of the extra work imposed by initialization. A more detailed presentation of the proof is given in the extended version of this paper [2]. We also proved a progress theorem that expresses that if the state is coherent with some type information and the instruction at address

i is not a `halt` instruction then execution can progress at address i. The two theorems can be used to express that program execution for well-typed programs progresses without errors until it reaches a `halt` statement.

4 Constructing an Effective Verifier

The type system does not correspond to an algorithm, since it assumes that the values F and S have been provided. An effective byte-code verifier has to construct this information. According to the approach advocated in [7], one should first produce this data, possibly using unsafe techniques, and then use the type verification described in the previous section to verify that the program is well-typed according to that information. The approach we study in this section is different: we attempt to construct F and S is such a way that the program is sure to be well-typed if the construction succeeds: it is no longer necessary to check the program after the construction.

In [12], T. Nipkow advocates the construction of the type information as the computation of a fix-point using Kildall's algorithm [11]. We have used a similar technique, based on traversing the control flow graph of the program and finding a least upper bound in a lattice. The lattice structure we have used is the lattice structure that underlies unification algorithms and we have, in fact, re-used a unification algorithm package that was already provided in the user libraries of the proof system [21]. However, the general approach of Nipkow was not followed faithfully, because the constraints we need to ensure are not completely stable with respect to the order used as a basis for unification. As a result, we still need to perform a verification pass after the data has been constructed.

4.1 Decomposing Typing Rules into Constraints

The typing constraints imposed for each instruction can be decomposed into more primitive constraints. We have isolated 8 such kinds of constraints. To explain the semantics of these constraints, we have a concept of *typing states*, with an order between typing states, $\preceq$. Typing states are usually denoted with variables of the form t, t'. We have a function `add_constraint'` to add constraints.

1. (`tc_all_vars` i j). This one expresses that the types of variables at lines i and j have to be the same.
2. (`tc_stack` i j). This one expresses that the types in stacks at lines i and j have to be the same.
3. (`tc_top` i τ). This one expresses that the type on top of the stack at line i has to be the type τ.
4. (`tc_pop` i j). This one expresses that the stack at line j has one less element than the stack at line i
5. (`tc_push` i j x). This one expresses that the stack at line j is the same as the stack at line i where a type has been added, this type being the type of variable x at line i.

6. (`tc_push_type` i j τ). This one is like the previous one except that the type is given in the constraint.
7. (`tc_store` i x j). This one expresses that the variables at line j have the same type as at line i, except for the variable x, which receives at line j the type that is on top of the stack at line i, also the stack at line j is the same as the stack at line i with the top element removed.
8. (`tc_init` i j σ). This one expresses most of the specific constraints that are required for the instruction `init`. Its semantics is more complicated to describe and it is actually expressed with three properties. The first property expresses that the stack must have an uninitialized type on top:

   `tc_init_stack_exists`:
   $\forall i, j, t, \sigma.$
   (`add_constraint'` (`tc_init` i j σ) t) = (`Some` t) $\Rightarrow$
   (`stack_defined` t i) $\Rightarrow$
   $\exists k, \alpha. \quad S_t(i) = \sigma_k \cdot \alpha$

   The second property expresses that all variables that referred to that uninitialized type at line i must be updated with a new initialized type at line j (we define a function *subst* on functions from `VAR` to types to represent the substitution operation).

   `tc_init_frame`:
   $\forall i, j, k, t, \sigma, \alpha.$
   (`add_constraint'` (`tc_init` i j σ) t) = (`Some` t) $\Rightarrow$
   $S_t(i) = \sigma_k \cdot \alpha \Rightarrow$
   $F_t(j) = (subst\ F_t(i)\ \sigma_k\ \sigma)$

   The third property expresses the same thing for stacks (we also have a function *subst_stk* to represent the substitution operation on stack).

   `tc_init_stack`:
   $\forall i, j, k, t, \sigma, \alpha.$
   (`add_constraint'` (`tc_init` i j σ) t) = (`Some` t) $\Rightarrow$
   $S_t(i) = \sigma_k \cdot \alpha \Rightarrow$
   $S_t(j) = (subst\_stk\ \alpha\ \sigma_k\ \sigma)$
   In these statements, the predicate (`stack_defined` t i) expresses that even though the state t may be incomplete, it is necessary that it already contains enough information to know the height of the stack at line i.

The constraints for each instruction are expressed by composing several primitive constraints. For instance, the constraints for (`load` x) at line i are the following ones:

(`tc_all_vars` i $(i+1)$) (`tc_push` i $(i+1)$ x)

A special case is the instruction (`new` σ), which creates an uninitialized value, i.e., a value of type σ_i if we are at address i. We associate to this instruction the following constraints:

(`tc_all_vars` i $(i+1)$) (`tc_push_type` i $(i+1)$ σ_i)

These constraints do not express the requirement that the type of the uninitialized object, σ_i, must not already be present in the memory at line i (this was expressed by the predicate *Unused* in the typing rule).

We use an order $\preceq$ between typing information states, such that $t \preceq t'$ means that t' contains strictly more information about types than t. In fact, this order is simply the instantiation order as used in unification theory. All constraints, except the constraint `tc_init` are preserved through $\preceq$. The requirements for initialization and for creating a new unitialized object are not preserved, this explains why we have to depart from Kildall's algorithm.

4.2 Relying on Unification

We use unifiable terms to represent successive states of the verifier and unifiable terms to represent constraints. Applying a constraint c to a type state t is implemented as applying the most general unifier of c and t to t to obtain a new state t'. Let us call *cumulative constraints* constraints of the kinds 1 to 7 in the enumeration above. These are the constraints that are stable with respect to the order $\preceq$.

The fragments F and S of the typing state actually are bi-dimensional arrays. For F the lines of the arrays are indexed with addresses, while the columns correspond to variables. For S the lines are also indexed with addresses, and each line gives the type of the stack at the corresponding address. When representing these notions as unifiable terms, they can be encoded as lists of lists.

The unifiable terms are composed of variables and terms of the form

$$f_i(t_1, \ldots, t_k)$$

for a certain number of function operators f_i. These operators are as follows:

- `fcons` and `fnil` are the constructors for lists.
- `fint` is the operator for the type of integer values, (`otype` σ) is used for types of initialized objects of class σ, and (`utype` σ i) for types of unitialized objects created at address i.

The initial typing state is the pair of a variable, to express that nothing is known about F in the beginning and a term of the form $\texttt{fcons}(\texttt{fnil}, X)$ to express that we know that the stack at line 0 is the empty stack and that we do not know anything about the stack on other lines yet.

The unifiable terms corresponding to cumulative constraints are easily expressed as iterations of the basic function operators. To make this practical we define a few functions to represent these iterations. For instance, we construct the term

$$\texttt{fcons}(X_{k+1}, \ldots \texttt{fcons}(X_{k+j-1}(\texttt{fcons}(t, X_{k+j})) \ldots)$$

by a calling a function (`place_one_list` t j $(k+1)$). The third argument, $k+1$, is used to shift the indices of the variables occurring at places $1, \ldots, j-1$ in the list. This term represents a list where the j^{th} element is constrained by t

and all other elements are left unconstrained (even the length of the list is not constrained much, it only needs to be greater than j).

Similarly, (`mk_two_list` t_1 t_2 gap i k) will construct the list whose length has to be greater than $i + gap$ and whose elements at ranks i and $i + gap$ are constrained by t_1 and t_2 respectively (here again the last argument, k is used to shift the indices of all the extra variables inserted in the list).

We do not describe the encoding of all constraints, but we can already express the encoding of the constraint (`tc_push` i j x) (when $j > i$):

$$\lceil(\texttt{tc\_push}\ i\ j\ x)\rceil = \\ (\texttt{place\_one\_list}\ (\texttt{place\_one\_list}\ X_k\ x\ k+2)\ i\ (k+x+2)), \\ (\texttt{mk\_two\_list}\ X_{k+1}\ (\texttt{fcons}\ X_k\ X_{k+1})\ (j-i)\ i\ (k+x+i+2))$$

Proving that the constraints are faithfully represented by the unifiable terms we associate to them requires that we show how functions like `place_one_list` behave with respect to some interpretation functions, mostly based on some form of `nth` function to return the n^{th} element of a list. For instance, if `F`,`S` represent the typing state, knowing the type of variable x at line i simply requires that we compute the unifiable term given by (`nth` (`nth` `F` i) x).

4.3 A Two Pass Algorithm

The algorithm performs a first pass where all cumulative constraints are applied to the initial typing state to obtain preliminary information about all types of variables and stacks at all lines. The constraint `tc_init` for initializations is also applied, even though we know that it will be necessary to re-check that the constraint is still satisfied for the final state.

The second pass does not modify the typing state anymore. It simply verifies that the final typing state does satisfy the restrictive constraint imposed by instructions `new` and `init`. For (`new` σ) at line i, it means verifying that the type σ_i occurs nowhere in the variables or the stack at line i. For (`init` σ) it means checking again that the unifiable term $\lceil(\texttt{tc\_init}\ i\ (i+1)\ \sigma)\rceil$ unifies with the final state.

5 Conclusion

The extraction mechanism of Coq makes it possible to derive from this proof development a program that will run on simple examples. This program is very likely to be unpractical: no attention has been paid to the inherent complexity of the verification mechanism. At every iteration we construct terms whose size is proportional to the line number being verified: in this sense the algorithm complexity is already sure to be more than quadratic.

Still, even if the exact representation of the typing state and constraints are likely to change to obtain a more usable verifier, we believe that the decomposition of its implementation and certification in the various phases presented in this paper is likely to remain relevant. These phases are:

1. Proving the soundness of a type system that uses data not in the program,
2. Proving that a program can build the missing data and ensure the typing constraints,
3. Setting aside the constraints that may not be preserved through the refinements occurring each time a line is processed,
4. Traverse the program according to its control flow graph,

With a broader perspective, this development of a certified byte-code verifier shows that very recent investigations into the semantics of programming languages can be completely mechanized using modern mechanical proof tools. The work presented here took only two months to mechanize completely and the part of this work that consisted in mechanizing the results found in [7] took between one and two weeks. This is also an example of using a type-theory based proof system as a programming language in the domain of program analysis tools, with all the benefits of the expressive type system to facilitate low-error programming and re-use of other programs and data-structures, as we did with the unification algorithm of [21]. Future development on this work will lead to more efficient, but still certified, implementations of this algorithm and and integration in a more complete implementation such as the one provided in [1].

References

1. G. Barthe, G. Dufay, L. Jakubiec, S. Melo de Sousa, and B. Serpette. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *LNCS*, pages 302–319. Springer-Verlag, 2001.
2. Yves Bertot. A coq formalization of a type checker for object initialization in the java virtual machine. Research Report RR-4047, INRIA, 2000.
3. Ludovic Casset and Jean-Louis Lanet. How to formally specify the java byte code semantics using the b method. In *proceedings of the Workshop on Formal Techniques for Java Programs at ECOOP 99*, June 1999.
4. Christina Cornes and Delphine Terrasse. Automatizing inversion of inductive predicates in coq. In *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
5. Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. *The Coq Proof Assistant User's Guide*. INRIA, May 1993. Version 5.8.
6. Stephen N. Freund and John C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, November 1999.
7. Stephen N. Freund and John C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems*, September 2000.
8. A. Goldberg. A specification of Java loading and bytecode verification. In *Proceedings of 5th ACM Conference on Computer and Communication Security*, 1998.
9. Ulrich Hensel, Marieke Huisman, Bart Jacobs, and Hendrik Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In *Proceedings of European Symposium on Programming (ESOP '98)*, volume 1381 of *LNCS*, pages 105–121. Springer-Verlag, March 1998.

10. Marieke Huisman. *Java program verification in Higher-order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
11. G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.
12. Tobias Nipkow. Verified bytecode verifiers. unpublished, available at URL `http://www.in.tum.de/~nipkow/pubs/bcv2.html`, 2000.
13. Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.
14. R. O'Callahn. A simple, comprehensive type system for java bytecode subroutines. In *ACM Symposium on Principles of Programming Languages*, pages 70–78. ACM Press, 1999.
15. David von Oheimb and Tobias Nipkow. Machine checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998. To appear.
16. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, feb 1995.
17. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
18. Lawrence C. Paulson and Tobias Nipkow. *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
19. Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages p. 89–103. Springer-Verlag, 1999.
20. Z. Qian. A formal specification of Java Virtual machine instructions for objects, methods, and subroutines. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
21. Joseph Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs, September 1992. (In french), available at URL `http://coq.inria.fr/contribs/unification.html`.
22. Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 149–160. ACM Press, January 1998.