Fixing the Java Bytecode Verifier by a Suitable Type Domain

Roberto Barbuti
Luca Tesei
Dipartimento di Informatica
Università di Pisa
Corso Italia 40
56100 Pisa, Italy
{barbuti,tesei}@di.unipi.it

Cinzia Bernardeschi
Nicoletta De Francesco
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
Via Diotisalvi 2
56100 Pisa, Italy
{c.bernardeschi,n.defrancesco}@iet.unipi.it

ABSTRACT

The Java Virtual Machine embodies a verifier which performs a set of checks on bytecode programs before their execution. The verifier performs a data-flow analysis applied to a type-level abstract interpretation of the code. The current implementations of the bytecode verifier present a significant problem: there are legal Java programs which are correctly compiled into a bytecode that is rejected by the verifier. Also the more powerful verification techniques proposed in several papers suffer from the same problem. In this paper we propose to enhance the bytecode verifier to accept such programs, maintaining the efficiency of current implementations. The enhanced version is based on a domain of types which is more expressive than the one used in standard verification.

1. INTRODUCTION

Java programs are compiled into an intermediate language which is interpreted by the Java Virtual Machine (JVM) [7]. The intermediate language, which, following other authors, we call JVML, is a language of bytecode instructions.

Since JVML programs can be loaded from the network, security problems may arise. For this reason, JVM embodies a bytecode verifier which performs a set of checks on JVML programs before their execution. The aim of these checks is to prevent the execution of malicious or wrong code which could corrupt the integrity of the host. The verifier performs a data-flow analysis applied to a type-level abstract interpretation of the JVM.

Given the importance of the bytecode verifier, a lot of research efforts have been dedicated both to its formalization and to study extensions able to accept larger classes of correct programs than the standard verifier does [4, 5, 6, 8, 9, 10, 12].

Despite these efforts, the bytecode verifier and its extensions still present a significant problem: there are legal Java programs which are correctly compiled into a bytecode that is rejected by the verifier. This problem has been pointed out by Stärk and Schmid in [11], where examples of these programs are reported. Stärk and Schmid propose to restrict the rules for detecting legal programs of the Java compiler such that these programs are no longer allowed.

In this paper we propose to enhance the bytecode verifier to accept such programs, which are actually correct and not dangerous. The verifier is defined by abstract rules based on a domain of types which is more expressive than the one used in standard verification. The domain is powerful enough to certify a class of programs that is is strictly larger than the one accepted by current implementations of the verifier. We show how the proposed enhanced verifier correctly accepts the programs presented in [11].

It is important to notice that the proposed extension of the verifier maintains the efficiency of current implementations. In fact the algorithm executed by the verifier is the same: in particular subroutines are examined only once also if they have multiple calling points. The improvement is essentially due to the type domain we define.

2. BYTECODE VERIFICATION AND THE PROBLEM OF SUBROUTINES

The result of the compilation of a Java program is a set of *class files*. A class file is generated by the Java Compiler from each class definition of the program. It is composed of the declaration of the class and of a JVML bytecode for each method of the class. The instructions of JVML are typed: for example `iload` loads an integer onto the operand stack, while `aload` loads an address. In Figure 1 we show a restricted set of JVML instructions. Our aim is to address the problem raised by the use of subroutines (see Section 2) and it can be fully analysed using this fragment.

We assume that each compiled method is a (finite) sequence of instructions labeled by $0, 1, 2, \ldots K-1$. Moreover, $N \in I\!N$ is the number of local registers required for the method, `max_stack_height` $\in I\!N$ is the maximum height that the

`iconst` c	Push constant c with type `int` onto the stack
`inc`	Increment the integer value on top of the stack
`iload` x	Push the integer value of register x onto the stack
`istore` x	Pop off the stack an integer value and store it into local register x
`astore` x	Pop off the stack an address and store it into local register x
`ifeq` L	Pop a value off the stack and if the value is equal to 0, branch to L
`goto` L	Jump to L
`ireturn`	Pop an integer value on top of the stack and return it
`jsr` L	Jump to address L and push the address of the following instruction onto the stack
`ret` x	Jump to the address stored in register x

Figure 1: Instruction set

			Memory	Stack	Modified
int m1(boolean b) {		iload 1	{1:int, 2:$\top$, 3:$\top$, 4:$\top$ }	()	
int i;		ifeq A	{1:int, 2:$\top$, 3:$\top$, 4:$\top$}	(int)	
try {		iconst 1	{1:int, 2:$\top$, 3:$\top$, 4:$\top$}	()	
if (b)		istore 3	{1:int, 2:$\top$, 3:$\top$, 4:$\top$}	(int)	
return 1;		jsr L	{1:int, 2:$\top$, 3:int, 4:$\top$}	()	
i = 2;		iload 3	{1:int, 2:$\top$, 3:int, 4:$\top$}	()	
} finally {		ireturn	{1:int, 2:$\top$, 3:int, 4:$\top$}	(int)	
if (b)	A:	iconst 2	{1:int, 2:$\top$, 3:$\top$, 4:$\top$}	()	
i = 3;		istore 2	{1:int, 2:$\top$, 3:$\top$, 4:$\top$}	(int)	
}		jsr L	{1:int, 2:int, 3:$\top$, 4:$\top$}	()	
return i;		goto C	{1:int, 2:$\top$, 3:$\top$, 4:$\top$}	()	
}	L:	astore 4	{1:int, 2:$\top$, 3:$\top$, 4:$\top$ }	(ret(L))	
		iload 1	{1:int, 2:$\top$, 3:$\top$, 4:ret(L)}	()	{4}
		ifeq B	{1:int, 2:$\top$, 3:$\top$, 4:ret(L)}	(int)	{4}
		iconst 3	{1:int, 2:$\top$, 3:$\top$, 4:ret(L)}	()	{4}
		istore 2	{1:int, 2:$\top$, 3:$\top$, 4:ret(L)}	(int)	{4}
	B:	ret 4	{1:int, 2:$\top$, 3:$\top$, 4:ret(L)}	()	{2, 4}
	C:	iload 2	{1:int, 2:$\top$, 3:$\top$, 4:$\top$}	()	
		ireturn	error	error	

Figure 2: An example of try-finally compilation and verification

stack can have during the execution of the method, and $m \in I\!N$ is the number of the parameters of the method. These three constants can be calculated statically by the compiler.

The bytecode is subject to a static analysis called bytecode verification. A Java bytecode verification algorithm is presented in [7]. Almost all existing bytecode verifiers implement this algorithm. It performs a data-flow analysis applied to a type-level abstract interpretation of the virtual machine. The types form a domain, where the basic types (e.g. `int`, `address`, $\cdots$) are unrelated and $\top$ is the top element. The class types are related as specified by the class hierarchy. In this domain $\top$ represents either the type of an undefined register (not yet assigned) or an incorrect type.

The abstract interpreter executes JVM instructions operating over types instead of values. The goal of the verification is to assign to each instruction i a mapping M_i from local registers to types and a mapping St_i from the elements in the stack to types. These mappings represent the state $S_i = (M_i, St_i)$ in which the instruction i is performed, thus S_i is the state at the program point i. For each instruction there is a rule that specifies the correct states in which such an instruction can be executed and the state after its execution. For example, a `iload` x instruction requires a non-full stack and the `int` type associated to register x, and its effect is to push `int` onto the stack. Note that a register can have different types at different program points, but it must be assured that the state after an instruction must be compatible with the state required by the successor instruction(s): for example, the state after the execution of an `ifeq` L instruction at address i must be compatible with S_{i+1} and S_L.

The rules are used in a standard fixpoint iteration using a worklist algorithm: an instruction i is taken from the worklist and the states at the successor program points are computed. If the computed state for a successor program point j changes (either the state at i was not yet computed or the already computed state differs), j is added to the worklist. The fixpoint is reached when the worklist becomes empty. Initially, the worklist contains only the first instruction of the bytecode. The initial stack and register types represent the state on method entrance: the stack is empty and the type of the registers corresponding to the parameters are set as specified by the signature of the method. The other registers hold the undefined type $\top$.

Instructions that represent a merge point between control paths, i.e. having more than one predecessor in the control flow graph, have a particular treatment. The state at a program point of this kind is taken to be the least upper bound of the states after all predecessor instructions. If, for example, register x has type `int` on a path and type $\top$ on another path, the type of x at the merge point is $\top$. The

$$\texttt{iconst}\quad \frac{\begin{array}{c}P(i) = \texttt{iconst}\ c,\ \ S_i = (M_i, St_i),\ \ |St_i| < \texttt{max\_stack\_height}\\ i+1 \in \{0,1,\ldots,K-1\}\end{array}}{S[S_{i+1} := S_{i+1} \sqcup (M_i, \texttt{int}.St_i)]}$$

$$\texttt{inc}\quad \frac{P(i) = \texttt{inc},\ \ S_i = (M_i, St_i),\ \ St_i = \texttt{int}.SHomet'\ \ i+1 \in \{0,1,\ldots,K-1\}}{S[S_{i+1} := S_{i+1} \sqcup (M_i, \texttt{int}.St')]}$$

$$\texttt{iload}\quad \frac{\begin{array}{c}P(i) = \texttt{iload}\ x,\ \ S_i = (M_i, St_i),\ \ |St_i| < \texttt{max\_stack\_height},\\ x \in \{1,2,\ldots,N\},\ \ M_i(x) = \texttt{int}\ \ i+1 \in \{0,1,\ldots,K-1\}\end{array}}{S[S_{i+1} := S_{i+1} \sqcup (M_i, \texttt{int}.St_i)]}$$

$$\texttt{istore}\quad \frac{\begin{array}{c}P(i) = \texttt{istore}\ x,\ \ S_i = (M_i, St_i),\ \ St_i = \texttt{int}.St'\\ i+1 \in \{0,1,\ldots,K-1\}\end{array}}{S[S_{i+1} := S_{i+1} \sqcup (M_i[x := \texttt{int}], St')]}$$

$$\texttt{astore}\quad \frac{\begin{array}{c}P(i) = \texttt{astore}\ x,\ \ S_i = (M_i, St_i),\ \ St_i = \texttt{ret}(L).St'\\ i+1 \in \{0,1,\ldots,K-1\}\end{array}}{S[S_{i+1} := S_{i+1} \sqcup (M_i[x := \texttt{ret}(L)], St')]}$$

$$\texttt{ifeq}\quad \frac{\begin{array}{c}P(i) = \texttt{ifeq}\ L,\ \ S_i = (M_i, St_i),\\ St_i = \texttt{int}.St',\ \ i+1, L \in \{0,1,\ldots,K-1\}\end{array}}{S[S_L := S_L \sqcup (M_i, St'),\ \ S_{i+1} := S_{i+1} \sqcup (M_i, St')]}$$

$$\texttt{goto}\quad \frac{P(i) = \texttt{goto}\ L,\ \ S_i = (M_i, St_i),\ \ L \in \{0,1,\ldots,K-1\}}{S[S_L := S_L \sqcup (M_i, St_i)]}$$

$$\texttt{jsr}\quad \frac{\begin{array}{c}P(i) = \texttt{jsr}Home\ L,\ \ S_i = (M_i, St_i),\\ |St_i| < \texttt{max\_stack\_height},\ \ L \in \{0,1,\ldots,K-1\}\end{array}}{S[S_L := S_L \sqcup (M_i, \texttt{ret}(L).St_i)]}$$

$$\texttt{ret}\quad \frac{\begin{array}{c}P(i) = \texttt{ret}\ x,\ \ S_i = (M_i, St_i),\ \ x \in \{1,2,\ldots,N\},\\ M_i(x) = \texttt{ret}(L),\ \ \texttt{Static\_Return\_Points}(L) = \{\ell_1, \ell_2, \ldots, \ell_p\}\end{array}}{\begin{array}{c}S[S_{\ell_1} := S_{\ell_1} \sqcup (S_{\ell_1 - 1} \rhd (M_i[x := \top], St_i)),\ \ S_{\ell_2} := S_{\ell_2} \sqcup (S_{\ell_2 - 1} \rhd (M_i[x := \top], St_i)),\\ \cdots\ \ S_{\ell_p} := S_{\ell_p} \sqcup (S_{\ell_p - 1} \rhd (M_i[x := \top], St_i))]\end{array}}$$

$$\texttt{ireturn}\quad \frac{P(i) = \texttt{ireturn},\ \ S_i = (M_i, St_i),\ \ St_i = \texttt{int}.St'}{S}$$

Figure 3: The rules of the abstract interpreter

least upper bound of stacks and memories is done pointwise. Note that the pointwise least upper bound between stacks requires that the stacks have the same height. Otherwise there is a type error.

Subroutines in JVML are code fragments that can be called from several points inside the code of a method. The instruction to call a subroutine is `jsr` L and the one to return from the subroutine is `ret` x. They are used to compile try-finally java constructs. Subroutines execute in the same activation record of the method, and therefore can access the local registers.

Subroutines complicate significantly bytecode verification by data-flow analysis. For efficiency reasons, it is required that the body of the subroutine is checked only once and not separately for the different calling points. Let a subroutine start at address L. The code of the subroutine is statically analysed to find the return points[1]. Each return point is the instruction following a `jsr` L instruction. The state at the first instruction of the subroutine (i.e. the state S_L) is obtained by merging all the states after the `jsr` L instructions. Now consider an occurrence of `jsr` L at address i and its successor instruction at $i+1$. Consider the state $S_i = (M_i, St_i)$ at i and the state $S_{ret} = (M_{ret}, St_{ret})$ after the execution of the instruction `ret` x returning from the subroutine. The state at the program point $i+1$, S_{i+1}, is computed as follows: $St_{i+1} = St_{ret}$ and $M_{i+1}(x) = \begin{cases} M_{ret}(x) & \text{if } x \text{ is modified by the subroutine} \\ M_i(x) & \text{otherwise} \end{cases}$

This solution generates a problem. Figure 2 shows a Java method `m1` which is accepted by the Sun's java compiler, but its compiled bytecode (shown in the second column) is rejected by the verifier, since it fails to type the instructions [11]. The figure also shows the typing assignment to the instructions produced by the Sun's verifier and the registers modified by the subroutine. Note that in the figure address types are represented by $\texttt{ret}(L)$: the type of return addresses from the subroutine L. The subroutine starting at address L is called from two `jsr` instructions: at the first `jsr` register `3` has type `int` and register `2` has type $\top$, while, at the second call, register `2` has type `int` and register `3` has type $\top$. Thus the body of the subroutine is abstractly executed starting from a memory which is the least

[1]Note that since JVML bytecode is unstructured, subroutines are not syntactically delimited, thus methods have been defined to identify the code of a subroutine. We do not consider a particular one, being our type domain compatible with all of them.

$\triangleright$	unt	int	ret(L)	unt_int	unt_ret(L)	$\top$
unt	unt	int	ret(L)	unt_int	unt_ret(L)	$\top$
int	int	int	ret(L)	int	$\top$	$\top$
ret(L)	ret(L)	int	ret(L)	$\top$	ret(L)	$\top$
unt_int	unt_int	int	ret(L)	unt_int	$\top$	$\top$
unt_ret(L)	unt_ret(L)	int	ret(L)	$\top$	unt_ret(L)	$\top$
$\top$	$\top$	int	ret(L)	$\top$	$\top$	$\top$

Figure 4: Full definition of the $\triangleright$ operator.

			Memory	Stack	Modified
int m1(boolean b) {		iload 1	{1:int, 2:unt, 3:unt, 4:unt}	()	
int i;		ifeq A	{1:int, 2:unt, 3:unt, 4:unt}	(int)	
try {		iconst 1	{1:int, 2:unt, 3:unt, 4:unt}	()	
if (b)		istore 3	{1:int, 2:unt, 3:unt, 4:unt}	(int)	
return 1;		jsr L	{1:int, 2:unt, 3:int, 4:unt}	()	
i = 2;		iload 3	{1:int, 2:unt_int, 3:int, 4:$\top$}	()	
} finally {		ireturn	{1:int, 2:unt_int, 3:int, 4:$\top$}	(int)	
if (b)	A:	iconst 2	{1:int, 2:unt, 3:unt, 4:unt}	()	
i = 3;		istore 2	{1:int, 2:unt, 3:unt, 4:unt}	(int)	
}		jsr L	{1:int, 2:int 3:unt, 4:unt}	()	
return i;		goto C	{1:int, 2:int, 3:unt, 4:$\top$}	()	
}	L:	astore 4	{1:int, 2:unt_int, 3:unt_int, 4:unt}	(ret(L))	
		iload 1	{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}	()	{4}
		ifeq B	{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}	(int)	{4}
		iconst 3	{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}	()	{4}
		istore 2	{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}	(int)	{4}
	B:	ret 4	{1:int, 2:unt_int, 3:unt_int, 4:ret(L)}	()	{2, 4}
	C:	iload 2	{1:int, 2:int, 3:unt, 4:$\top$}	()	
		ireturn	{1:int, 2:int, 3:unt, 4:$\top$}	(int)	

Figure 5: An example of verification using our abstract interpreter

upper bound of the two calling points, that is $M_L(1) = \texttt{int}$, $M_L(2) = M_L(3) = M_L(4) = \top$. Note that there is a path from L to the last instruction of the subroutine B: `ret 4` where register `2` is not modified and another path from L to B in which `2` is assigned an integer value. Hence, at instruction B the paths are merged and register `2` holds the type $\texttt{int} \sqcup \top = \top$. Since `2` belongs to the registers modified by the subroutine, it is assigned $\top$ in the memory for each instruction following the calls. Thus it holds $\top$ at instruction `goto` C and also at instruction at address C, where an error is signaled, since $\top$ cannot be loaded. For `3` there is no problem, since it is not modified by the subroutine, and thus it has type `int` at the first return point (where it is used). Actually the code could be considered correct since `2` holds `int` at the first `jsr` and in the subroutine it is assigned `int` on one branch and it is not modified on the other one. Thus the type of register `2` after the call could be correctly `int` .

The problem is that the standard verifier does not distinguish between a typing incompatible with `int` at all (e.g. an address) and a typing compatible with it. The domain of types seems to be too poor: the type $\top$ for a register x at the program instruction i represents three situations that could be handled in different ways: 1) a type of an unset register; 2) an error, when $\top$ derives from the least upper bound of two incompatible types; 3) a type of a register which, at a merge point i, is assigned on some program paths and is untouched on the other ones. In the next section we propose a solution where the domain of types is extended to contain a different type for each different situation. These types behave in a different way when they are composed with the return types from the subroutines.

3. A MORE PRECISE VERIFICATION

Figure 6 shows the type domain of the abstract interpreter on which the enhanced verifier is based. The bottom represents a completely undefined type. The type `unt` abstracts the values that have not been initialized, but that belong to the memory of an instruction which has already been considered in a fixpoint iteration. It is used to represent situation 1) described in the preceding section. $\top$ represents a contradictory type (resulting from the merge of incompatible types). It corresponds to situation 2). For each Type `T`, the type `unt_T` means that the register, at a merge point, has been assigned with a value of type `T` on some program paths and has not been touched on the other paths (see situation 3)). For simplicity, in the figure we have shown only one return address type, `ret`(L) (that is, we consider only a subroutine). If we have more than one return address type (corresponding to have several subroutines), for each one of them there is, in the domain, a different `ret`(L) and `unt_ret`(L) type.

The operator $\sqcup$ used in the rules of the abstract interpreter is the least upper bound operator of the new domain applied to memories and stacks pointwise.

The initial state is defined only on the program point 0, the address of the first instruction. In the initial state the first m local registers of M_0 are initialized to the actual types of the parameters of the method. The remaining registers are set to `unt`. The stack St_0 is the empty stack. We use the usual notation $(s.St)$ for a stack with top element s and the remaining part St. The execution proceeds following the worklist fixpoint algorithm.

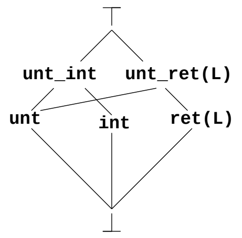

Figure 6: The abstract type domain.

There is a rule for each instruction of the program: it operates on the pair (M_i, St_i) at the program point i producing the pair (M_l, St_l) at the program point l, where l is the label of a successor instruction of i (taking into account jumps, of course). As usual, at any iteration an instruction i is selected from the working list, and the corresponding rule is applied to the current state at the program point i, (M_i, St_i). It yields a new state S' on which the process iterates. The iteration stops when a fixpoint is reached, i.e. the working list is empty. It is important to remark that, in order to check the subroutines only once, the rules must be applied in a suitable order, how the standard verifier does. Note that we use the notation $S[S_i := S_i^{new}]$ to denote a new state obtained by S by updating the state at the program point i by S_i^{new}. Let P be a function assigning to each number $i \in \{0, 1, \ldots, K-1\}$ the i-th instruction. A rule is activated if $P(i)$ equals the instruction handled by the rule and all other premises are satisfied.

The rules of the abstract interpreter are shown in Figure 3. `iconst` pushes a constant value onto the stack. The abstract interpreter generates the stack of the successor instruction $i+1$ by pushing the type `int` onto the stack; it is required that the push does not overflow the stack. In the subsequent instruction (with index $i+1$) memory and stack are assigned to the least upper bound of the old values and the ones coming from the abstract execution of i. Note that if the state S_{i+1} at the program point $i+1$ does not exist yet, $S_{i+1} \sqcup (M_i, \texttt{int}.St_i)$ simply results in $(M_i, \texttt{int}.St_i)$. `inc` requires a stack with top element `int`. `iload` x requires that the register x has type `int`, and pushes `int` onto the stack. `istore` x and `astore` x store an integer or an address, respectively, from the stack to a register. `ifeq` L requires an integer on the stack. The abstract interpreter executes both branches of conditionals and updates both successor program points (L and $i+1$). Both `goto` L and `jsr` L jump to L, thus the program point L is updated; moreover, `jsr` L pushes onto the stack the type of the return address, i.e. $\texttt{ret}(L)$. The execution of `ret` x modifies the memory and the stack of all the instructions immediately following the various `jsr` L instructions. The set of the addresses of such instructions, which we call $\texttt{Static\_Return\_Points}(L)$, can be computed statically. The memory and the stack of each instruction at these addresses is calculated from the memory and the stack both of the correspondent calling instruction and of the `ret` itself, using the $\triangleright$ operator. We define $(M, St) \triangleright (M', St') = (M \triangleright M', St')$ where $M \triangleright M'$ is obtained by applying pointwise the operator $\triangleright$ only to registers which have been modified in the subroutine. Figure 2 shows the definition of the $\triangleright$ operator in the abstract type domain. Each entry of the table is the resulting of the type on the row $\triangleright$ the type on the column. Note that the operator is not commutative. The type $\tau_1 \triangleright \tau_2$ is the type that a modified register, having type τ_1 at the calling `jsr` instruction and type τ_2 at the `ret` program point, should have, after the `ret`, to achieve a correct typing. We remark that $\texttt{int} \triangleright \texttt{unt\_int} = \texttt{int}$ and this fact allows solving the subroutine problem, as explained before. If we have two types, $\texttt{ret}(L_1)$ and $\texttt{ret}(L_2)$, they are considered incompatible, hence, for example, it would be $\texttt{ret}(L_1) \triangleright \texttt{unt\_ret}(L_2) = \top$.

If, at an iteration, there is no applicable rule, then the algorithm stops and signals an error. Formally, we should add the value *error* as a possible state and add rules to propagate the error. Moreover we should consider also a state following the `ireturn` instruction which records whether an error occurred on it. For the sake of clarity we leave the handling of errors implicit.

In Figure 5 we show how our rules correctly assign types to the program on which the verifier fails.

4. CONCLUSIONS

In [1] we have completely developed an abstract interpretation approach to verification, formally proving that the interpreter we define here is an abstract semantics based on a concrete collecting semantics of Java bytecode. There we state the usual correctness results of abstract interpretations [2, 3].

We remark that we do not propose a more powerful verification technique, like as in [5, 9], allowing to certify recursive or polymorphic subroutines.We only show how to fix a problem of the verifier by introducing a richer type system, and maintaining the standard verification algorithm, with the same complexity. We remark that other extensions do not resolve the subroutine problem.

5. REFERENCES

[1] Barbuti, R., Bernardeschi, C., De Francesco, N. and Tesei, L. Enhancing the Java Bytecode Verifier by Abstract Interpretation. Internal Report Dipartimento di Informatica, Università di Pisa, 2002.

[2] Cousot, P. and Cousot, R. Abstract Interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In Proc. of POPL'77. ACM Press, 238–258, 1977

[3] Cousot, P. and Cousot, R. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, **82**(1):43–57, 1979.

[4] Freund, S. N. and Mitchell, J. C. A Formal Framework for the Java Bytecode Language and Verifier. ACM Conference on Object-Oriented Programming Systems, Languages & Applications, 1999.

[5] Hagiya, M. and Tozawa, A. On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines. Static Analysis Symposium '98, LNCS 1503, Springer, 1998.

[6] Leroy, X. Java bytecode verification: an overview. In Proceedings of CAV'01, Springer LNCS 2102, 2001.

[7] Lindholm, T. and Yellin, F. The Java Virtual Machine Specification. The Java Series, Addison-Wesley, 1999.

[8] Nipkow, T. Verified Bytecode Verifiers. FOSSACS 2001, LNCS 2030, Springer, 2001.

[9] O'Callahan, R. A Simple, Comprehensive Type System for Java Bytecode Subroutines. ACM Symposium on Principles of Programming Languages, 1999.

[10] Qian, Z. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems* **22**(4):638–672, 2000.

[11] Stärk, R. F. and Schmid, J. The problem of bytecode verification in current implementations of the JVM. Technical Report, Department of Computer Science, ETH Zürich, 2000.

[12] Stata, R. and Abadi, M. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, **21**(1):90–137, 1999.