A Parametrized Polyvariant Bytecode Verifier

Ludovic Henrio & Bernard Paul Serpette

Inria Sophia-Antipolis
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
`First.Last@inria.fr`
`http://www.inria.fr/oasis/First.Last`

Résumé

This paper presents a bytecode verifier based on an abstract small-step semantics. It is described with transition rules on abstract states of a virtual machine. By construction, the bytecode verifier is polyvariant: more than one abstract state can be constructed for a given program point. We separate each abstract step into two substeps: a join step which can merge some abstract states and an operational step which computes the effect of a specific instruction on a state. We give properties needed for these two steps in order to obtain the soundness of the bytecode verifier. We define a simple abstract join step definition allowing full polymorphism of subroutines. Finally, by choosing a dedicated representation of abstract stacks, we will see that our bytecode verifier is able to perform a type check of recursion.

1. Introduction

Bytecode interpreters, and more generally virtual machines, were introduced to increase the portability of high-level languages. In the past, type safety proofs were only performed on high-level languages. During the last decade, with the introduction of the Java Virtual Machine (JVM) [11], bytecode languages have become a way of distributing platform-independent programs. It becomes necessary to verify the safety of these programs. The purpose of a bytecode verifier (BCV) is to insure type safety at load time. In order to do that, a BCV must infer local variables (registers) and operand stack types for each bytecode instruction.

Since the verification is performed at load time, a BCV should accept most valid programs, even if compilers perform clever transformations. For the JVM, **jsr/ret** instructions are the crucial points of the verification. This issue and its underlying problems have been handled using four different approaches:

- *data-flow.* This methodology is inspired by the algorithm informally described in the JVM specification [11] (chap. 4.9.2). In this approach, a single abstract state is associated with each instruction. A formal description of this algorithm is given by Stata and Abadi [23]. In their work, subroutines can be used only in strict LIFO order and are not polymorphic over manipulated locations: a subroutine that swaps the value of two local variables is not well typed when called from two different contexts. Some later works give some ad-hoc solutions in order to partially retrieve polymorphism in subroutines. [5][4][15][22].
- *bytecode transformation.* This approach was introduced for time [16] or space [10] efficiency purposes. In both cases, the LIFO order of subroutine calls must be respected.
- *typing.* The idea is to propose a polymorphic type system where continuations, introduced by the **jsr** instruction, are typed as higher-order functions. This technique, used by [12] and [8],

shows that problems with subroutines are linked to recursive types. The main difficulty of this approach is to make sure that continuations do not reference themselves.

- *polyvariance*. The general idea is to accept that several abstract states may be associated with an instruction. This approach, described in [9] (chap. 5.3), uses *contours* to separate states. These contours, when implemented as call stacks, allow subroutine bodies to be verified according to the caller chain. Polyvariance may be indirectly reached by using model checking [13] or abstract interpretation (unpublished work of Brisset, explained in [9]). More recently, Coglio [1] followed this approach. His work is very close to our and seems to be done independently at the same time.

Our work is a polyvariant approach of bytecode verification without the notion of *contours*. Indeed, contours introduce abstract domains which do not correspond to concrete ones, and the stack of subroutine calls imposes a LIFO order on the **jsr/ret** chains. We propose to separate each abstract execution step (abstract execution of an instruction on an abstract state) into a join step which may merge some abstract states and an operational step which computes the effect of a specific instruction on an abstract state. Therefore the analysis is parameterized by a join operator allowing the specification of a pure monovariant BCV (standard data-flow) to a full polyvariant BCV *à la* Brisset (all traces of the control flow are analyzed). As pointed in Leroy's overview on bytecode verification [9]: *Brisset's algorithm can be degraded by applying widening steps in order to reduce the number of states.* Our contribution can be viewed as the specification of **where** widening may occur (the join step) and **how** the widening is done (the join operator).

In our polyvariant approach, we have implemented and specified a BCV with polymorphic subroutines. Moreover, by choosing a dedicated representation of the abstract operand stack, our BCV is able to type check recursion.

In section 2, we introduce our framework by specifying dynamic and abstract domains. A key point of this specification is to properly separate the abstract semantics into two substeps. The join step specifies how states can be merged before an instruction. And the operational step describes how instructions change the internal state of the machine. We extract a property which must be verified by the abstract join operator and formally specify how abstract values (i.e. types) can be merged at a join point. Meanwhile, we extract a property needed for intra-procedural instructions (all instructions except call, return and throw) in order to obtain an easy soundness proof.

Section 3, as an illustration of our framework, focuses on subroutines. In this section, we introduce the couple of instructions **jsr/ret** and the underlying continuation types. We will show how these continuations can be used for compilation of applicative language such as Scheme.

Section 4 gives some join point rules and abstract stack specifications for a bounded stack (i.e. with no recursion). With a very simple join point rule, our BCV has the following features: (i) polymorphism with respect to local variables (ii) polymorphism with respect to the operand stack values (iii) polymorphism with respect to the operand stack size (iv) possibility of multiple uses for a continuation (v) possibility of multiple entries for a subroutine (vi) continuation calls and returns may not occur in a stack-like manner.

Section 5 shows that if abstract stacks are specified as graphs, the bytecode verifier accepts programs even in case of recursion. Abstracting stacks by graphs is a subject tackled by Horspool and Vitek [6]. They use regular expressions to represent combinations of types on the stack at different points in a PostScript program. Graphs (or equivalently regular expressions) as abstract values were also used in other abstract interpretation [2] algorithms. But generally, graphs were used to abstract relations inside the heap for aliasing analysis by Deutsch [3] , and by Sagiv, Reps and Wilhelm[17] for shape analysis.

2. Framework

2.1. Dynamic domains

We suppose that the dynamic semantics of the machine is given by a set of small-step transition rules which defines a relation $\mapsto$ on *states* ($State \mapsto State$). A state denotes, at least, the dynamic stack of method invocations. A state ($p, q, r, s \in State$) is defined as a list of *frames* ($State = Frame^*$) where a frame ($f \in Frame$) is defined as a tuple $Frame = Method, Int, Value[], Stack$. A frame ($f = m, pc, l, \sigma$) consists of the method invoked at this point, the program counter pointing to the instruction to be executed, the vector of local variable values and the operand stack. Just like lists, the $Stack$ domain is defined inductively with an empty stack ($\epsilon \in Stack$) and a constructor $\cdot$: $v \in Value \ \wedge \ \sigma \in Stack \Rightarrow v{\cdot}\sigma \in Stack$.

A method ($m \in Method$) consists of a functional type and a vector of instructions ($Method = (Type^* \to Type), Ins[]$). If $m = (ty, code)$ is a method, $m[pc]$ stands for $code[pc]$ and $\|m\|$ stands for $\|code\|$ where $\| \bullet \|$ denotes the length of lists or vectors. The tuple (m, pc), called a *program point*, denotes a specific instruction of the given method.

In order to define a join operation independently of the instruction to be executed, we split the $\mapsto$ relation into two relations on $State$: $p \mapsto s \Leftrightarrow \exists q.\ p \rhd q \lhd s$. The first relation, $\rhd$, describes how states are transferred from an instruction to another one. It is independent from the executed or analyzed instructions. This relation is somewhat obvious for the dynamic semantics: the state generated by an instruction is transferred as it is to its control-flow successor checking if the program counter is valid:

$$\frac{pc < \|m\|}{(m, pc, l, \sigma){\cdot}f^* \rhd (m, pc, l, \sigma){\cdot}f^*} \tag{1}$$

This rule may also be used for other checks such as verifying that the size of the operand stack will never grow beyond a static bound.

In contrast, the second relation, $\lhd$, describes the *real* operational semantics of an instruction; as data, the program counter is updated at this point. For example, the instruction **pop** is specified by the following rule:

$$\frac{m[pc] = \mathbf{pop}()}{(m, pc, l, v{\cdot}\sigma){\cdot}f^* \lhd (m, pc+1, l, \sigma){\cdot}f^*} \tag{2}$$

2.2. Abstract domains

To define the static semantics, each dynamic domain must be mapped onto an abstract one. The corresponding abstract domain for $Value$ is the $Type$ domain. We suppose that the abstraction function $\Downarrow : Value \to Type$ is given; $\Downarrow$ is the projection of values on types. We extend this projection on vector of local variables in the natural way: $\Downarrow(l) = \widehat{l} \Leftrightarrow \forall i < \|l\|,\ \widehat{l}[i] = \Downarrow(l[i])$. The abstraction of operand stacks ($\widehat{Stack}$) is inductively defined by $\Downarrow(\epsilon) = \widehat{\epsilon}$ and $\Downarrow(v{\cdot}s) = \Downarrow(v){\cdot}\Downarrow(s)$. But, in order to manage recursion, it is not specified if abstract stacks have a unique decomposition under the $\cdot$ constructor.

An abstract state ($\widehat{s} \in \widehat{State}$) is the abstraction of the first element of a concrete state (*current frame*). Method and program counter are abstracted by themselves (abstraction function is identity):

$$\Downarrow((m, pc, l, \sigma){\cdot}f^*) = m, pc, \Downarrow(l), \Downarrow(\sigma)$$

In order to manage sub-typing, we suppose we have a partial order ($\leq$) on $Type$. This partial order is defined on abstract local variables in the natural way: $\widehat{l} \leq \widehat{l'} \Leftrightarrow \forall i < \|\widehat{l'}\|.\ \widehat{l}[i] \leq \widehat{l'}[i]$. The definition of $\leq$ on abstract stacks is delayed until section 4 and 5. Abstract states are ordered by: $m, pc, \widehat{l}, \widehat{\sigma} \leq m, pc, \widehat{l'}, \widehat{\sigma'} \Leftrightarrow \widehat{l} \leq \widehat{l'} \ \wedge\ \widehat{\sigma} \leq \widehat{\sigma'}$. Note that two abstract states may be comparable if and only if they represent the same program point.

This partial order will be used as a simple relation and will always appear with the pattern $\Downarrow(s) \leq \widehat{s}$ where $s \in \mathit{State}$ and $\widehat{s} \in \widehat{\mathit{State}}$. Specific properties of $\leq$ are not needed until we define the semantics of particular instructions or of the abstract join operator. When necessary, we will assume the existence of a semi-lattice under Type with a least upper bound (lub) operator ($\sqcup$). This lub operator is extended to abstract local variables, abstract stacks and abstract states according to $\leq$ definitions.

The static semantics of the bytecode verifier is given by rules defining a relation $\hookrightarrow$ on abstract states. Like for concrete states, this relation is split into two relations also on $\widehat{\mathit{State}}$: $\widehat{p} \hookrightarrow \widehat{s} \Leftrightarrow \exists \widehat{q}.\ \widehat{p} \mathrel{\widehat{\triangleright}} \widehat{q} \mathrel{\widehat{\triangleleft}} \widehat{s}$. $\widehat{\triangleright}$ is the abstract join operator which may collapse some abstract states and $\widehat{\triangleleft}$ is defined by abstract rules like the following one for the instruction **pop**:

$$\frac{m[pc] = \mathbf{pop}()}{m, pc, \widehat{l}, \widehat{v}{\cdot}\widehat{\sigma} \mathrel{\widehat{\triangleleft}} m, pc+1, \widehat{l}, \widehat{\sigma}} \tag{3}$$

2.3. Properties for soundness

The soundness of the BCV is achieved by a step soundness stating that, if, from the abstraction of a concrete state p we can make one abstract step, then p can go to a state s such that the abstraction of s is also reachable from the abstraction of p. This can be depicted by the following diagram:

$$\begin{array}{ccc} p & & \\ \Downarrow_{\leq} & & \\ & & \\ \widehat{p} & \hookrightarrow & \widehat{r} \end{array} \qquad \Longrightarrow \qquad \begin{array}{ccc} p & \mapsto & s \\ \Downarrow_{\leq} & & \Downarrow_{\leq} \\ \widehat{p} & \hookrightarrow & \widehat{s} \\ \widehat{p} & \hookrightarrow & \widehat{r} \end{array}$$

Or, more formally, by the property:

$$\Downarrow(p) \leq \widehat{p} \ \wedge\ \widehat{p} \hookrightarrow \widehat{r} \Rightarrow \exists s, \widehat{s}.\ p \mapsto s \ \wedge\ \widehat{p} \hookrightarrow \widehat{s} \ \wedge\ \Downarrow(s) \leq \widehat{s}$$

If we show internal relations, the diagram splits into two parts:

$$\begin{array}{ccccc} p & & & & \\ \Downarrow_{\leq} & & & & \\ \widehat{p} & \widehat{\triangleright} & \widehat{q} & & \\ & & \widehat{q} & \widehat{\triangleleft} & \widehat{r} \end{array} \qquad \Longrightarrow \qquad \begin{array}{ccccc} p & \triangleright & p & \triangleleft & s \\ \Downarrow_{\leq} & & \Downarrow_{\leq} & & \Downarrow_{\leq} \\ \widehat{p} & \widehat{\triangleright} & \widehat{q} & \widehat{\triangleleft} & \widehat{s} \\ & & \widehat{q} & \widehat{\triangleleft} & \widehat{r} \end{array} \tag{4}$$

This highlights the two properties we need on abstract states: first the $\widehat{\triangleright}$ relation must check the validity of the control flow, i.e. correct program counter, and it must verify the property 5 which guarantees the soundness of $\widehat{\triangleright}$. This allows $\widehat{\triangleright}$ to merge several abstract states provided the property 5 is verified.

$$\Downarrow(p) \leq \widehat{p} \ \wedge\ \widehat{p} \mathrel{\widehat{\triangleright}} \widehat{q} \Rightarrow p \triangleright p \ \wedge\ \Downarrow(p) \leq \widehat{q} \tag{5}$$

Taking $\Downarrow(p) = \widehat{p}$, the property becomes $\widehat{p} \mathrel{\widehat{\triangleright}} \widehat{q} \Rightarrow \widehat{p} \leq \widehat{q}$ which is a generalization of the lub done in join points.

On the right part of the diagram 4, we need the following property for intra-procedural instructions (all instructions except call, return and throw):

$$\Downarrow(p) \leq \widehat{q} \ \wedge\ \widehat{q} \mathrel{\widehat{\triangleleft}} \widehat{r} \Rightarrow \exists s, \widehat{s}.\ p \triangleleft s \ \wedge\ \widehat{q} \mathrel{\widehat{\triangleleft}} \widehat{s} \ \wedge\ \Downarrow(s) \leq \widehat{s} \tag{6}$$

To verify this property on an instruction, we have to look at both concrete and abstract rules for this instruction. We check that, when an abstract rule can be applied, at least **one** corresponding concrete rule can be chosen such that the abstraction of the resulting concrete state is also reachable in **one** abstract step.

It is clear that the **pop** instruction is intra-procedural (rules 2 and 3). Unfortunately, some instructions may not directly verify this definition. For example, if an instruction can raise a dynamic error (such as **NullPointerException** for a **getfield** instruction), the intra-procedural property is not straightforward. Nevertheless, for these instructions, we can transform the source code in order to make dynamic tests explicit. Thus, the **throw** instruction becomes statically visible, the transformed **getfield** instruction will not have to perform specific runtime checks and then can verify the intra-procedural property.

3. Subroutines

3.1. Motivations

We have to be clear about what is the *meaning* of subroutines and how a compiler *may* and *must* use them. In Serrano's thesis [19][20], functions are classified in various category depending of results provided by a control flow analysis. One specific category contains those functions that can be *globalized* by applying a λ-lifting transformation. In this case the compiler does not generate closures, but can we avoid the λ-lifting ? This is the first aim of subroutines by sharing the local variables and the operand stack between the caller and the callee. Moreover, using the shared local variables and operand stack, the compiler may implements its own protocol for passing arguments and returning results.

But, providing a simple couple **jsr/ret** of instructions for calling to and returning from a subroutine without explicit continuation, leads to contortioned implementation where frames have to manage both an operand stack and a continuation stack, the latter used exclusively by the **jsr/ret** couple of intructions.

Thus, subroutines are handled in the JVM by the two instructions **jsr/ret**, but the instruction **jsr** creates an intra-procedural continuation, pushes it as a value onto the **operand** stack and branches to a specified instruction. The pushed continuation can be used later by a **ret** instruction in order to perform a computed branch inside the current method. Intra-procedural continuations are second-class objects: they cannot be passed as arguments or returned by function calls, they cannot be stored in the heap and thus can only appear in the frame where they are created.

Nevertheless, these continuations can be used by other instructions than **ret** (at least **load** and **store** instructions) and the virtual machine becomes much attractive for functional language compilers. This is the second aim of subroutines : allowing a compiler to use intra-procedural continuation for its own optimizations. This must include, at least for functional language compiler, a kind of continuation passing style and the possibility of recursion.

During the implementation of a new JVM backend for the Bigloo Scheme compiler [18], we had studied the opportunity to map some functions introduced by the **letrec** binder to JVM subroutines. We had not achieved this mapping mainly for bytecode verification purpose : by now we don't use at all subroutines. One main motivation of this paper is to show that bytecode verification can handle properly, without restrictions, the intra-procedural continuations and thus can be heavily used by applicative languages such as Scheme, Haskell or ML.

Finally, even if we will mimic the **jsr/ret** instructions of the JVM, our work may beneficially be used for less restrictive virtual machine.

$$\frac{m[pc] = \mathbf{jsr}(pc')}{(m, pc, l, \sigma)\cdot f^* \lhd (m, pc', l, @_{pc+1}\cdot\sigma)\cdot f^*} \qquad \frac{m[pc] = \mathbf{jsr}(pc')}{m, pc, \widehat{l}, \widehat{\sigma} \mathrel{\widehat{\lhd}} m, pc', \widehat{l}, \partial_{\{pc+1\}}\cdot\widehat{\sigma}}$$

$$\frac{m[pc] = \mathbf{ret}(i) \ \wedge \ l[i] = @_{pc'}}{(m, pc, l, \sigma)\cdot f^* \lhd (m, pc', l, \sigma)\cdot f^*} \qquad \frac{m[pc] = \mathbf{ret}(i) \ \wedge \ \widehat{l}[i] = \partial_s \ \wedge \ pc' \in s}{m, pc, \widehat{l}, \widehat{\sigma} \mathrel{\widehat{\lhd}} m, pc', \widehat{l}, \widehat{\sigma}}$$

Figure 1: Concrete and abstract execution of **jsr/ret**

3.2. Rules

The low-level continuations introduced by **jsr** are the encapsulation of code index which can be defined by a constructor $@ : Int \to \partial$, where ∂ is the common ancestor for all intra-procedural continuation types: $\forall pc.\ \Downarrow(@_{pc}) \leq \partial$.

Since a program point is a static value, i.e. known at compilation time, we can also tag an intra-procedural continuation type with a code index. To be more general, we tag continuation type by set of code indexes: $\Downarrow(@_{pc}) = \partial_{\{pc\}}$. These types form a lattice with a lub operator relying on union: $\partial_{s_1} \bigsqcup \partial_{s_2} = \partial_{s_1 \cup s_2}$ The underlying partial order is: $\partial_{s_1} \leq \partial_{s_2} \Leftrightarrow s_1 \subseteq s_2$.

This is a precise implementation of continuation types, since we never lose track of continuation targets.

These types are defined in order to obtain a direct proof that **jsr/ret** follow the property 6 (cf fig 1). With $\Downarrow(@_{pc}) = \partial_{\{pc\}}$, it is clear that **jsr** follows the intra-procedural property 6 and **ret** is also intra-procedural because $\Downarrow(@_{pc'}) \leq \partial_s \Rightarrow pc' \in s$.

4. Bounded stack

If, as in the JVM, the size of the operand stack never grows beyond a static bound N, then abstract stacks can be specified as lists of types ($\widehat{Stack} = Type^*$). Abstract stacks can be compared ($\leq$) or merged ($\sqcup$) if and only if they have the same length: $t_1{\cdot}t_2{\cdot}\ldots{\cdot}t_n{\cdot}\epsilon \ \sqcup \ t'_1{\cdot}t'_2{\cdot}\ldots{\cdot}t'_n{\cdot}\epsilon = (t_1 \ \sqcup \ t'_1){\cdot}(t_2 \ \sqcup \ t'_2){\cdot}\ldots{\cdot}(t_n \ \sqcup \ t'_n){\cdot}\epsilon$.

For a given method, the set of used types is finite, so the set of used abstract states is bounded and the fixpoint computation will terminate. We may choose not to perform any specific behavior on join point:

$$\frac{pc < \|m\| \ \wedge \ \|\widehat{\sigma}\| \leq N}{m, pc, \widehat{l}, \widehat{\sigma} \mathrel{\widehat{\rhd}} m, pc, \widehat{l}, \widehat{\sigma}} \tag{7}$$

Nevertheless, with this rule, a bytecode verifier may generate a large number of abstract states, therefore becoming expensive in time. For example, the following Java code for m:

```
B m(B x0, Object x1, ..., Object xn) {
   if P x1=x0;
   ...
   if P xn=x0;
   return(x0);
}
```

$$\frac{}{\partial_s \equiv \partial_s} \qquad \frac{t_1 \not\leq \partial \ \wedge \ t_2 \not\leq \partial}{t_1 \equiv t_2} \qquad \frac{}{\epsilon \equiv \epsilon}$$

$$\frac{\widehat{\sigma_i} \equiv \widehat{\sigma_j} \ \wedge \ t_1 \equiv t_2}{t_1 {\cdot} \widehat{\sigma_i} \equiv t_2 {\cdot} \widehat{\sigma_j}} \qquad \frac{\forall k < \|\widehat{l}\|.\widehat{l}[k] \equiv \widehat{l'}[k]}{\widehat{l} \equiv \widehat{l'}} \qquad \frac{\widehat{l} \equiv \widehat{l'} \ \wedge \ \widehat{\sigma} \equiv \widehat{\sigma'}}{m, pc, \widehat{l}, \widehat{\sigma} \equiv m, pc, \widehat{l'}, \widehat{\sigma'}}$$

Figure 2: definition of $\equiv$

produces 2^n abstract states for the return instruction. To avoid this exponential complexity, standard bytecode verifiers merge all the abstract states available before a program point. In order to do that, we can define an abstract join rule which keeps only one abstract state for each instruction:

$$\frac{pc < \|m\| \ \wedge \ \|\widehat{\sigma}\| \leq N \ \wedge \ \widehat{s} \in Before(m, pc)}{\widehat{s} \mathrel{\widehat{\triangleright}} \bigsqcup_{\widehat{s_i} \in Before(m,pc)} \widehat{s_i}} \tag{8}$$

Where $Before(m, pc)$ is the set of abstract states available before the specified program point: the set of abstract states reachable from the abstract initial state $\widehat{Init}(m)$ by zero or more abstract steps ($\hookrightarrow^*$).

$$Before(m, pc) = \{\widehat{s} = m, pc, \widehat{l}, \widehat{\sigma} / \widehat{Init}(m) \hookrightarrow^* \widehat{s}\}$$

Since, by definition of the lub operator, $\forall i.\ x_i \leq \bigsqcup_i x_i$, the abstract join property 5 is respected. But, with this rule, subroutines do not have any kind of polymorphism. If a subroutine is called in two different places, the two associated abstract states are merged at the entry point of the subroutine.

Between the identity (7) and full lub (8) rules, many join operators may be defined: any cover of $Before(m, pc)$ can be used. A cover for a set of abstract states is a function $\mathcal{H} : \wp(\widehat{State}) \rightarrow \wp(\wp(\widehat{State}))$ such that: $S = \bigcup_{S_i \in \mathcal{H}(S)} S_i$.
By extension we will note $\mathcal{H}(m, pc)$ for $\mathcal{H}(Before(m, pc))$. A general abstract join rule becomes:

$$\frac{pc < \|m\| \ \wedge \ \|\widehat{\sigma}\| \leq N \ \wedge \ E \in \mathcal{H}(m, pc) \ \wedge \ \widehat{s} \in E}{\widehat{s} \mathrel{\widehat{\triangleright}} \bigsqcup_{\widehat{s_i} \in E} \widehat{s_i}} \tag{9}$$

The full lub abstract join (rule 8) can be redefined with $\mathcal{H}_0(E) = \{E\}$ and the identity abstract join (rule 7) with $\mathcal{H}_\infty(E) = \{\{e_i\} / e_i \in E\}$.

We propose a cover of a set of abstract states with a partition based on an equivalence relation $\equiv$: $\mathcal{H}_\equiv(E) = \{E_i\}_i$ with $x \in E_i \Rightarrow (y \in E_i \Leftrightarrow x \equiv y)$. The relation $\equiv$ is defined in Figure 2 and expresses that two abstract states are equivalent if they have exactly the same continuation types (if any), in the same places within local variables and operand stack.

5. Recursion

The only way to generate arbitrary sized stack is to use recursion. In that case, we need to find a finite representation of abstract stacks. The key point is not to find a recursion point, which is hard to discover, but to highlight the *intention* of recursion by finding a duplication of a continuation.

Abstract stacks are graphs with an entry point ($\mathcal{I}$) and an exit point ($\mathcal{F}$). Nodes of the graph contain continuation types. Edges are labeled with lists of other types.

$$\widehat{\sigma} = \langle N, E \rangle \ \text{ with } \begin{cases} \{\mathcal{I}, \mathcal{F}\} \subseteq N \subseteq \{\mathcal{I}, \mathcal{F}\} \cup \partial \\ E \subseteq \{n_1 \xrightarrow{W} n_2 | n_1, n_2 \in N, W \in (Type - \partial)^*\} \end{cases}$$

The empty stack is defined as $\widehat{\epsilon} = \langle\{\mathcal{I}, \mathcal{F}\}, \{\mathcal{I} \xrightarrow{\epsilon} \mathcal{F}\}\rangle$. The concrete stacks corresponding to an abstract stack are (subtypes of) the ones which are verified by the automata corresponding to the abstract stack graph. When $\widehat{\sigma} = \langle N, E\rangle$, we define $Tail(E) = \{n_1 \xrightarrow{W} n_2 \in E | n_1 \neq \mathcal{I}\}$.

The addition of elements on the stack is divided into two cases. To add a standard type, we only need to append it to the label of the edge starting from the initial node:

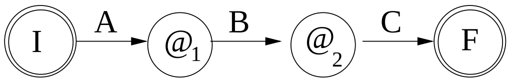
$$t{\cdot}\langle N, E\rangle = \langle N, \{\mathcal{I} \xrightarrow{t \cdot W} n | \mathcal{I} \xrightarrow{W} n \in E\} \cup Tail(E)\rangle$$

We consider that there is an intention of recursion if the same continuation type is pushed on the stack several times. Thus, to add a continuation node, we create a new node unless there is a node labeled with the same abstract destination address. After that, the set of edges is updated in order to put the continuation node just behind the initial one:

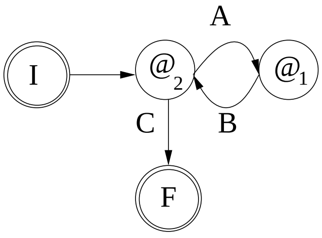
$$\partial_s{\cdot}\langle N, E\rangle = \langle N \cup \{\partial_s\}, \{\mathcal{I} \xrightarrow{\epsilon} \partial_s\} \cup \{\partial_s \xrightarrow{W} n | \mathcal{I} \xrightarrow{W} n \in E\} \cup Tail(E)\rangle$$

Note that, by construction, the initial node $\mathcal{I}$ has one and only one outgoing edge. In contrast with a simple list, these new abstract stacks do not have a unique decomposition under the $\cdot$ constructor. For example, $\partial_1{\cdot}\partial_1{\cdot}\partial_1{\cdot}\epsilon$ is equal to $\partial_1{\cdot}\partial_1{\cdot}\epsilon$. And, when we write $\widehat{\sigma} = \widehat{v}{\cdot}\widehat{\sigma'}$, more than one $\widehat{\sigma'}$ may be found. Nevertheless, when $\widehat{v}$ is not a continuation type, the decomposition is unique.

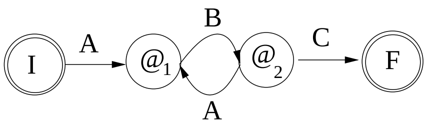
For example, consider the abstract stack $\widehat{\sigma}$ constructed by $A{\cdot}\partial_1{\cdot}B{\cdot}\partial_2{\cdot}C{\cdot}\epsilon$, the associated graph is :



Then $\partial_2{\cdot}\widehat{\sigma}$ become :



And this same abstract stack can also be obtained by $\partial_2{\cdot}\widehat{\sigma'}$ where σ' is the following graph :



The abstract stack equality is the structural equality of graphs.

The order relation between abstract stacks corresponds to the fact that a “bigger” stack must accept every concrete stacks accepted by a “smaller” one:

$$\langle N_1, E_1\rangle \leq \langle N_2, E_2\rangle \Leftrightarrow \left\{ \begin{array}{l} N_1 \subseteq N_2 \\ n \xrightarrow{W} n' \in E_1 \Rightarrow \exists n \xrightarrow{W'} n' \in E_2, W \leq W' \end{array} \right.$$

Where, in $W \leq W'$, $\leq$ is the order relation on lists of non-continuation types of the same length.

Equivalence on states is the same as defined in Figure 2 except for stack equivalence. Two abstract stacks will be equivalent if they have exactly the same continuation types linked with edges of the same length:

$$\langle N_1, E_1 \rangle \ \equiv\ \langle N_2, E_2 \rangle \Leftrightarrow \begin{cases} N_1 = N_2 \\ n_1 \overset{W}{\to} n_2 \in E_1 \Leftrightarrow n_1 \overset{W'}{\to} n_2 \in E_2 \\ n_1 \overset{W}{\to} n_2 \in E_1 \ \wedge\ n_1 \overset{W'}{\to} n_2 \in E_2 \Rightarrow \|W\| = \|W'\| \end{cases}$$

To define the lub of two states, we only redefine the operator $\sqcup$ on equivalent abstract stacks. It is not defined for every pair of graphs because, if we choose $\mathcal{H}_{\equiv}$ as partitioning function, we only need to apply it on elements of the same equivalence class. The lub of two equivalent states only needs to merge the edges which have the same length:

$$\langle N_1, E_1 \rangle \ \sqcup\ \langle N_2, E_2 \rangle = \langle N_1, \{ n_1 \overset{W'}{\to} n_2 | W' = \ \bigsqcup \{ W | n_1 \overset{W}{\to} n_2 \in E_1 \cup E_2 \} \rangle$$

If, by the abstract join rule, all the paths between two continuation types are bounded, then the set of used abstract state is also bounded. So, if all the paths between two continuation types are bounded, then the lattice of abstract stacks is finite. This fact ensures that abstract execution will always terminate and find a fix point.

6. Examples

The first example, taken from the Jbook [22], can be generated by a Java compiler and is refused by allmost all BCV [21]. The problem for monovariant BCV is that the second local variable at line 15 must be typed by ($t \ \sqcup\ int$), where t is the type of this variable at line 10 (entry of the subroutine). Hence, for type oriented BCV, both type variable and lub constructor must be added to the underlying type system. This lub constructor seems to be the missing point of O'Callahan's BCV [12].

PC	Instruction	States (before)	
0	**iload**(0)	[int $\top$ $\top$],()	
1	**ifeq**(5)	[int $\top$ $\top$],(int)	
2	**jsr**(10)	[int $\top$ $\top$],()	
3	**ldc**(1)	[int $\top$ ∂_3],()	
4	**ireturn**()	[int $\top$ ∂_3],(int)	
5	**ldc**(2)	[int $\top$ $\top$],()	
6	**istore**(1)	[int $\top$ $\top$],(int)	
7	**jsr**(10)	[int int $\top$],()	
8	**iload**(1)	[int int ∂_8],()	
9	**ireturn**()	[int int ∂_8],(int)	
10	**astore**(2)	[int $\top$ $\top$],(∂_3)	[int int $\top$],(∂_8)
11	**iload**(0)	[int $\top$ ∂_3],()	[int int ∂_8],()
12	**ifeq**(15)	[int $\top$ ∂_3],(int)	[int int ∂_8],(int)
13	**ldc**(3)	[int $\top$ ∂_3],()	[int int ∂_8],()
14	**istore**(1)	[int $\top$ ∂_3],(int)	[int int ∂_8],(int)
15	**ret**(2)	[int $\top$ ∂_3],()	[int int ∂_8],()

The second example shows how subroutines can be polymorphic with respect to local variables and operand stack. The subroutine, beginning at 6 swaps the top of the operand stack and the first local variable (at index 0). Note that this subroutine is also polymorphic with respect to the size of the operand stack, i.e. at the entry of the subroutine the operand stack must contain *at least* two elements.

PC	Instruction	States (before)	
0	**aload**(0)	[X Y $\top$],()	
1	**aload**(1)	[X Y $\top$],(X)	
2	**jsr**(6)	[X Y $\top$],(Y X)	
3	**astore**(1)	[Y Y ∂_3],(X X)	
4	**jsr**(6)	[Y X ∂_3],(X)	
5	**areturn**()	[X X ∂_5],(Y)	
6	**store**(2)	[X Y $\top$],(∂_3 Y X)	[Y X ∂_3],(∂_5 X)
7	**load**(0)	[X Y ∂_3],(Y X)	[Y X ∂_5],(X)
8	**swap**()	[X Y ∂_3],(X Y X)	[Y X ∂_5],(Y X)
9	**store**(0)	[X Y ∂_3],(Y X X)	[Y X ∂_5],(X Y)
10	**ret**(2)	[Y Y ∂_3],(X X)	[X X ∂_5],(Y)

The next example shows how **jsr** and **ret** instructions may not occur in LIFO order.

PC	INS	States (before)
0	**jsr**(2)	[$\top$ $\top$ $\top$],()
1	**return**()	[∂_1 ∂_4 ∂_7],()
2	**astore**(0)	[$\top$ $\top$ $\top$],(∂_1)
3	**jsr**(5)	[∂_1 $\top$ $\top$],()
4	**ret**(2)	[∂_1 ∂_4 ∂_7],()
5	**astore**(1)	[∂_1 $\top$ $\top$],(∂_4)
6	**jsr**(8)	[∂_1 ∂_4 $\top$],()
7	**ret**(0)	[∂_1 ∂_4 ∂_7],()
8	**astore**(2)	[∂_1 ∂_4 $\top$],(∂_7)
9	**ret**(1)	[∂_1 ∂_4 ∂_7],()

The last example shows the use of recursion. The subroutine beginning at `3` computes, in local variable 1, the factorial of the number taken in local variable 0. The instruction **jmp** branch to instruction pointed by the continuation found at the top of the operand stack.

PC	Instruction	States (before)	
0	**jsr**(3)	[int $\top$] ()	
1	**iload**(1)	[int int] ()	
2	**ireturn**()	[int int] (int)	
3	**iload**(0)	[int $\top$] (∂_1)	[int $\top$] ((∂_{14} int)$^+$ ∂_1)
4	**ifne**(8)	[int $\top$] (int ∂_1)	[int $\top$] (int (∂_{14} int)$^+$ ∂_1)
5	**ldc**(1)	[int $\top$] (∂_1)	[int $\top$] ((∂_{14} int)$^+$ ∂_1)
6	**istore**(1)	[int $\top$] (int ∂_1)	[int $\top$] (int (∂_{14} int)$^+$ ∂_1)
7	**jmp**()	[int int] (∂_1)	[int int] ((∂_{14} int)$^+$ ∂_1)
8	**iload**(0)	[int $\top$] (∂_1)	[int $\top$] ((∂_{14} int)$^+$ ∂_1)
9	**dup**()	[int $\top$] (int ∂_1)	[int $\top$] (int (∂_{14} int)$^+$ ∂_1)
10	**ldc**(1)	[int $\top$] (int int ∂_1)	[int $\top$] (int int (∂_{14} int)$^+$ ∂_1)
11	**isub**()	[int $\top$] (int int int ∂_1)	[int $\top$] (int int int (∂_{14} int)$^+$ ∂_1)
12	**istore**(0)	[int $\top$] (int int ∂_1)	[int $\top$] (int int (∂_{14} int)$^+$ ∂_1)
13	**jsr**(3)	[int $\top$] (int ∂_1)	[int $\top$] (int (∂_{14} int)$^+$ ∂_1)
14	**iload**(1)	[int int] (int ∂_1)	[int int] (int (∂_{14} int)$^+$ ∂_1)
15	**imul**()	[int int] (int int ∂_1)	[int int] (int int (∂_{14} int)$^+$ ∂_1)
16	**istore**(1)	[int int] (int ∂_1)	[int int] (int (∂_{14} int)$^+$ ∂_1)
17	**jmp**()	[int int] (∂_1)	[int int] ((∂_{14} int)$^+$ ∂_1)

7. Implementation notes

7.1. Fix-point

Using this framework, we have implemented, in Java, the verification for all the JVM instructions. The outmost loop of the bytecode verifier looks like:

```
public void verify(State s) {
  if(computed(s)) return;
  addbefore(s);
  for(StateList l1=join(before(s)); l1!=nil; l1=l1.cdr)
    for(StateList l2=next(l1.car); l2!=nil; l2=l2.cdr)
      for(StateList l3=op(l2.car); l3!=nil; l3=l3.cdr)
        verify(l3.car);
}
```

The expression `join(before(s))` computes $\mathcal{H}(m, pc)$ where m and pc are taken in the state `s`. The function `next(s)` takes the instruction pointed to by `s`, finds the number n of types needed by this instruction in the top of the abstract stack and finds all the decompositions (unique for bounded stack) $\widehat{\sigma} = t_1 \cdot t_2 \dots t_n \cdot \widehat{\sigma'}$ of the abstract stack of `s`. Therefore, the function `op(s)`,which computes the successors of `s` by the relation $\widehat{\triangleleft}$, takes its arguments in the operand stack without ambiguity.

We have implemented the three versions of the abstract join rule proposed in section 4 (i.e. $\mathcal{H}_\infty$, $\mathcal{H}_\equiv$ and $\mathcal{H}_0$). As a benchmark we took the classes computed as the closure transitivity of the relation c_1 *potentially needs* c_2 (i.e. c_2 appears in the constant pool of c_1) with **java.lang.Object** as root. For SUN JDK 1.3, we obtained 624 classes in 24 different packages, 6161 methods and 160595 instructions. As a general result we have computed the number of times that the `verify` method was called divided by the number of instructions (i.e 160595). We have also computed the number of times that the `join` method was called divided by the same number of instructions. Here are our results:

	verify/ins	join/ins
$\mathcal{H}_\infty$	1.61	1.09
$\mathcal{H}_\equiv$	1.28	1.07
$\mathcal{H}_0$	1.22	1.06

Recursive loop

The results of 1.6 for *join/ins* in Leroy's bytecode verifier [10] can be explained knowing that his algorithm is less aggressive than ours: an extra complete loop may be executed in order to check that the fix-point is reached. We have rewritten the method `verify` with this fix-point methodology and we have:

	verify/ins	join/ins
$\mathcal{H}_\infty$	1.63	1.54
$\mathcal{H}_\equiv$	1.59	1.54
$\mathcal{H}_0$	1.52	1.52

Iterative loop over a global boolean

If we add a vector of booleans in order to track exactly the instructions to be revisited, the results fall down with:

	verify/ins	join/ins
$\mathcal{H}_\infty$	1.18	1.01
$\mathcal{H}_\equiv$	1.15	1.01
$\mathcal{H}_0$	1.12	1.01

Iterative loop over a vector of booleans

N	0	1	2	3	4	5	6	7	8	9	10	12	13	14	16	20	30
$\mathcal{H}_\infty$	13	151224	7336	1106	392	133	234	5	35	9	2	9	18	9	33	9	28
$\mathcal{H}_\equiv$	13	160337	134	65	28	0	0	0	18	0	0	0	0	0	0	0	0

Figure 3: States per instruction history

Figure 3 gives a finer result with the history of the number N of abstract states associated to each instruction, i.e. the length of the last list computed by the method `join`. For example, for the abstract join point we have proposed, we found 18 instructions (all in the same method `writeObject` of the class `java.io.ObjectOutputStream`) which require 8 abstract states for 2 nested subroutines (as mentioned in [9]: *more than two nested subroutines are unheard of (for typical Java code)*). The 13 instructions which do not need any abstract state correspond to dead code. As expected, all instructions which need more than one abstract state are in exception handler code. These handlers are the only places where Java compilers introduce subroutines. Nevertheless, and even if the 1.61 general result seems to be good, the $\mathcal{H}_\infty$ abstract join point may give very large set of abstract states.

7.2. Sub-typing

Even if the types found in method and field prototypes come directly from a high level language, the type algebra used by a bytecode verifier may be different from the one that is used by the language. For example, for the JVM type system, the types **char** and **int** are equals, but **char**[] and **int**[] are different. A more complicated example occurs when the high-level type algebra, ordered by $\leq$, does not constitute a semi-lattice (multiple inheritance via interfaces) which may be mandatory for the bytecode verifier. Here is the Java example we have found in [7] [8] [9]:

```
interface I {...}
interface J {...}
class C1 implements I, J {...}
class C2 implements I, J {...}
```

In that case, the two types `C1` and `C2` have the common super-types `I` and `J` that are not comparable. The JVM type system (*JVMType*), if a lub operator is needed, must make some transformation on types found in declarations (*JType*). We have not followed the methodology of Qian [15] and Push [14] who make use of list of *JType* as *JVMType*, nor the type completion proposed by [7], but implemented two other different strategies. First, following Sun's approach, all interfaces are identical to the class type **Object**[1]; by this way we get more expressivity (more programs are accepted) but the verifier cannot statically guarantee that a reference implements a given interface. Second, we can consider that all reference types of *JVMType* are a class type of *JType* (say $C1$) annotated by the set of all interfaces it implements (say $\{I, J\}$). The lub operator can be defined by using the lub operator on classes (a class can have at most one superclass) and the intersection for interfaces:

$$A_{s_1} \sqcup B_{s_2} = (A \sqcup B)_{s_1 \cap s_2}$$

And thus,

$$A_{s_1} \leq B_{s_2} \Leftrightarrow A \leq B \ \wedge \ s_2 \subseteq s_1$$

Some programs accepted by Sun's verifier may not be accepted anymore, but the **invokeinterface** instruction does not have to make any runtime check. For the aforementioned example we have: $C1_{\{I,J\}} \sqcup C2_{\{I,J\}} = Object_{\{I,J\}}$.

[1]We have found that Sun's partial order used by the verifier is not transitive (or does not cleanly implement the antisymmetry) with **Object**[] $\leq$ **Object** $\leq$ **I**, but **Object**[] $\not\leq$ **I**, when **I** is an interface. One can check that it does not break the soundness of the verifier since every time that the transitivity is needed, say $a \leq b \leq c$, c is computed by the lub operator which, in Sun's implementation, does not have interfaces in its codomain.

All strategies, including ours, have the same goal: completion from the partial order of *JType* to the semi-lattice of *JVMType*. Except for the Sun's approach, which impose dynamic tests for the **invokeinterface** instruction, we think that all approaches are equivalent with respect to the expressivity of the verifier. Nevertheless, Qian's and Push's sub-typing relation is less defined than our. Consider the following example:

```
interface I {...}
interface J {...}
interface K {...}
class A implements I, J    {...}
class B implements I, J, K {...}
class C implements    J, K {...}
```

In Qian's and Push's type system, the lub of `A`, `B` and `C` (denoted by `(A B C)`) is not a sub-type of the lub of `A` and `B`; the inverse is true and both are sub-types of `J`. But they never have to compare these kind of types because each time the verifier needs to test $T_1 \leq T_2$, T_2 cannot be constructed by a lub (i.e. it is a singleton). In our strategy, $A_{\{I,J\}} \sqcup B_{\{I,J,K\}} \sqcup C_{\{J,K\}} = A_{\{I,J\}} \sqcup C_{\{J,K\}} = Object_{\{J\}}$. The same result is achieved by full type completion (creation of the semi-lattice before performing verification).

The problem with full type completion is that **all** classes and interfaces (and recursively their super types) needed by the method to be verified must be fetched in the disk in order to find the partial order to be completed. In fact, in our solution and in Push's and Qian's type system, the super-type of A is not needed until we have to compare it to a different type. One may consider that Push's, Qian's and our strategies are a *lazy* type completion. Maybe, the full type completion algorithm, following the Dedekind-MacNeille one, can be demand driven, but this is out of the scope of this paper.

Finally, our solution seems suitable because we cleanly separate *classes* from *interfaces*. Therefore, the specific interface instruction (**invokeinterface**) and the specific class instructions (**getfield**, **putfield**, **invokevirtual**) known where to check sub-typing.

8. Conclusion

We presented a framework for defining a bytecode verifier from a polyvariant point of view. We guarantee the soundness of the verifier provided a few properties are verified (abstract join property 5 and intra-procedural property 6). This framework can be used for any machine based on a small-step transition system. The approach we used to introduce intra-procedural continuations, is the most open and general one compared to all approaches we are aware of. All the complexity of the bytecode verification is delayed, on the one hand, to an abstract join point operator which may introduce several abstract states into each instruction and, on the other hand, to the domain of abstract stacks which may seem complicated if we use graphs, but allows recursion. In the context of the JVM with code produced by standard Java compilers, we have shown that, the overhead introduced is not significant. But, we still have to investigate the complexity of our algorithm in a more general context. For this purpose we have begun an implemention of a simple common subexpression optimizer: in a given method, all identical sequences of instructions are merged into a common subroutine, and as expected our BCV is not sensible to this kind of code transformations.

8.1. Acknowledgements

The present version version of this paper has been influenced by comments and remarks of anonymous referres. We would also like to thank Egon Börger, Robert F. Stärk, Xavier Leroy, Julien Vayssière and Fabrice Huet for their comments.

Bibliographie

[1] Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. In *Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs*, June 2002.

[2] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Language*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[3] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[4] Stephen N. Freund and John C. Mitchell. Specification and verification of java bytecode subroutines and exceptions (summary). Technical Note CS-TN-99-91, Computer Science Department, Stanford University, August 1999.

[5] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine Subroutines. In Giorgio Levi, editor, *Static Analysis, 5th International Symposium, SAS'98*, number 1503 in Lecture Notes in Computer Science, pages 17–32. Springer-Verlag, Berlin Germany, 1998.

[6] R. Nigel Horspool and Jan Vitek. Static analysis of postscript code. *Computer Languages*, 19(2):65–78, 1993.

[7] T. B. Knoblock and J. Rehof. Type elaboration and subtype completion for java bytecode. In *In 27th ACM Symposium on Principles of Programming Languages*, 2000.

[8] Christopher League, Valery Trifonov, and Zhong Shao. Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[9] Xavier Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.

[10] Xavier Leroy. On-card bytecode verification for Java card. In I. Attali and T. Jensen, editors, *Smart card programming and security, proceedings E-Smart 2001*, volume 2140 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 2001.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[12] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–78. ACM Press, 1999.

[13] J. Posegga and H. Vogt. Java bytecode verification using model checking. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.

[14] Cornelia Pusch. Proving the soundness of a java bytecode verifier specification in isabelle/HOL. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 89–103, 1999.

[15] Z. Qian. A formal specification of Java[TM] virtual machine instructions for objects, methods and subroutines. *j-LECT-NOTES-COMP-SCI*, 1523:271–??, 1999.

[16] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.

[17] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[18] Bernard Serpette and Manuel Serrano. Compiling Scheme to JVM Bytecode: a Performance Study. In *ICFP'02 — International Conference on Functional Programming*, October 4–6, 2002, Pittsburgh, PA, USA, 2002.

[19] M. Serrano. *Vers une compilation portable* et *performante des langages fonctionnels.* Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris VI), Paris, France, December 1994.

[20] M. Serrano. Control Flow Analysis: a Functional Languages Compilation Paradigm. In *10th Symposium on Applied Computing*, pages 118–122, Nashville, Tennessee, USA, February 1995.

[21] R. F. Stärk and J. Schmid. Java bytecode verification is not possible. Technical report, Department of Computer Science, ETH Zürich, Switzerland, 2000.

[22] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.

[23] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In ACM, editor, *Conference record of POPL'98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium, San Diego, California, 19–21 January 1998*, pages 149–160, New York, NY 10036, USA, 1998. ACM Press.